Software Implementation of Synchronous Programs

Charles André* Frédéric Boulanger† Alain Girault‡

* I3S Lab. University of Nice-Sophia Antipolis/CNRS, France. Tel: +33 4 92 94 27 40. Email: andre@unice.fr
† SUPÉLEC, Service Informatique, France. Tel: +33 1 69 85 14 84. Email: Frederic.Boulanger@supelec.fr
‡ INRIA Rhône-Alpes, BIP project, France. Tel: +33 4 76 61 53 51. Email: Alain.Girault@inrialpes.fr

Abstract

Synchronous languages allow a high level, concurrent, and deterministic description the behavior of reactive systems. Thus, they can be used advantageously for the programming of embedded control systems. The runtime requirements of synchronous code are light, but several critical properties must be fulfilled.

In this paper, we address the problem of the software implementation of synchronous programs. After a brief introduction to reactive systems, this paper formalizes the notion of "execution machine" for synchronous code. Then, a generic architecture for centralized execution machines is introduced. Finally, several effective implementations are presented.

1 Introduction

1.1 Reactive Systems

Reactive systems are computer systems that react continuously to their environment, at a speed determined by the latter [16]. This class of systems contrasts with *transformational systems* and *interactive systems*. Transformational systems are classical programs whose inputs are available at the beginning of their execution, and which deliver their outputs when terminating: for instance compilers. Interactive systems are programs which react continuously to their environment, but at their own speed: for instance operating systems. Among reactive systems are most of the industrial real-time systems: control, supervision, and signal-processing systems. These systems must meet the following requirements:

1. **Safety requirements.** This is perhaps their most important feature since these systems are often critical ones. For instance, the consequences of a software error in an aircraft automatic pilot or in a nuclear plant controller are disastrous. Therefore these systems require rigorous design methods and languages as well as formal verification and validation of their behavior.
2. **Temporal requirements.** This concerns both the input rate and the input/output response time. To check their satisfaction on the implementation, it is necessary to know bounds on the execution time of each computation as well as on the maximal input rate.
3. **Concurrency requirements.** It is convenient and natural to design such systems as sets of components that cooperate to achieve the intended behavior. Here we distinguish between the *specification* parallelism and the *execution* parallelism. The latter is sometimes required by the implementation, while the former helps the programmer in specifying his/her system clearly and concisely.
4. **Determinism.** These systems, or at least their most critical parts, always react the same way to the same inputs. This property makes their design, analysis and debugging easier. It must therefore be preserved by the implementation.

A programming language well suited to the design of reactive systems should therefore be parallel and deterministic, and allow formal behavioral and temporal verification.

1.2 The Synchronous Approach

Synchronous languages have been introduced in the 80's to make the programming of reactive systems easier [6]. The purpose of these languages is to give the designer ideal temporal primitives, thus reducing the chance of programming misconceptions. Instead of the interleaving

paradigm, they are based on the simultaneity principle: all parallel activities share the same discrete time scale. Concretely, this means that the parallel statement $a||b$ is viewed as the "package" ab where a and b are simultaneous. Each activity can then be dated on the discrete time scale; this has the following advantages:

- Time reasoning is made simpler.
- Interleaving-based non-determinism disappears, which makes program debugging, testing, and validating easier.

Concerning the implementation, the idea is to project this discrete time scale onto the physical time. As the scale is discrete, *nothing* occurs between two consecutive instants: everything must happen as if the processor running the program were infinitely fast. This is the *synchrony hypothesis*.

Of course, such an infinitely fast processor does not exist, but it suffices that any input be treated before the next one. In order to verify this condition, one only needs to know the minimal input period, and an upper bound on the execution time of the object program. For this purpose, synchronous languages have deliberately restricted themselves to programs that can be compiled into a finite deterministic interpreted automaton, a control structure whose transitions are deterministic sequential programs operating on a finite memory. Each transition, whose execution time is statically computable, corresponds to the system reaction to an input.

There are numerous languages based upon the synchrony hypothesis: ESTEREL [7], LUSTRE [14], SIGNAL [17], STATECHARTS [15], SML [11], SYNCCHARTS [1], ARGOS [18], and SR [13].

Synchronous languages have recently seen a tremendous interest from leading companies developing automatic control software for critical applications, such as SCHNEIDER ELECTRIC, DASSAULT AVIATION, AÉROSPATIALE, SNECMA, CADENCE, TEXAS INSTRUMENTS, THOMSON,... For instance, LUSTRE is used to develop the control software for nuclear plants [5] and AIRBUS planes [**?**]. ESTEREL is used to develop DSP chips for mobile phones [**?**], to design and verify DVD chips, and to program the flight control software of RAFALE fighters [**?**]. And SIGNAL is used to develop digital controllers for airplane engines. The key advantage pointed by these companies is that the synchronous approach has a rigorous mathematical semantics which allows the programmers to develop critical software faster and better.

Finally, all synchronous languages can import and manipulate external objects (constants, variables, procedures, and functions), specified in a *host language*, e.g., C, ADA,... The compiling model adopted for the various synchronous languages consists then in compiling the source program towards an intermediate format where parallelism, preemptions, local communications, and so on, have been transformed into sequential deterministic code. This intermediate format consists of several tables and a control part. The tables describe the input/output signals, the constants, the types, the variables, and so on. The control part is either a deterministic finite state automaton (the internal OC format), or a system of Boolean equations with registers (the internal DC or SSC format). In both cases, the intermediate code program is compiled into a *transformational function* in the host language.

1.3 Problem Statement

When executing synchronous programs, one must deal with the big difference between the program and its environment. Indeed, the program is synchronous while its environment is intrinsically asynchronous, i.e., its evolutions *are not* governed by the synchrony hypothesis.

As we have said, a reactive system must react continuously to its environment, at a speed imposed by the latter. Concretely, the program communicates with its environment through input/output signals. Input and output signals are respectively sensed and emitted by the program. We distinguish two kinds of sensors, and accordingly two kinds of inputs:

- *State sensors*: They measure a physical value either continuous (e.g. the temperature) or discrete (e.g. an on/off limit switch). They give the current state of the physical value, sampled in the case of a continuous one.
- *Event sensors*: They measure both the state changes of a physical value (e.g., moving above a threshold) and the discrete events (e.g., an alarm). A state change is by essence discrete, fleeting, and must therefore be expected specifically in order to be observed. We include in this part messages possibly coming from other subsystems. In the case of a

large scale system, the designers often divide it into several subsystems that are programmed separately. Thus, each subsystem receives inputs from the environment as well as from other subsystems, via a local bus (CAN, VAN, FIP, home made bus, and so on). This was the case of the CO3N4 nuclear plant controller made by SCHNEIDER ELECTRIC.

The program is synchronous. From the implementation point of view, this means that it transforms instantaneously a tuple of inputs into a tuple of outputs. An *instant* of the synchronous program corresponds therefore to the reception of a new input tuple, the reaction to these inputs, and the emission of a new output tuple. As a consequence:

- the inputs of a same instant are synchronous since they belong to the same tuple,
- the outputs are synchronous with the inputs since the reaction of the program is instantaneous.

It follows an intrinsic mismatch between the synchronous program and the asynchronous environment. Any implementation, be it software or hardware, must solve this discrepancy, through a *synchronous/asynchronous interface*, whose precise purpose remains to be stated.

1.4 Paper Outline

We address in this paper the problem of implementing synchronous programs. There are two ways of implementing such programs: either software or hardware. We focus here on their software implementation. We formalize our problem in Section 2 by studying the interactions between the program, the interface, and the environment. Then we present in Section 3 some practical implementations, before concluding in Section 4.

Implementations can be either centralized, or distributed. This presentation focuses on the former. The distribution of synchronous programs raises other issues that are beyond the scope of this paper.

2 Formalization

2.1 The Execution Machine

The purpose of the execution machine is to actually execute a synchronous program in an asynchronous environment, that is, to observe the current state of the environment (sensing phase), to decide what to do (execution phase), and to act upon the environment (acting phase). Figure 1 states these interactions and emphasizes the necessary input and output treatments.

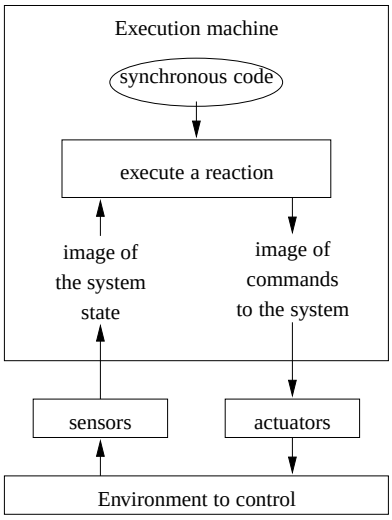

Figure 1: Interactions between the execution machine and the environment

We define an execution machine as the combination of a reactive machine, a transformational machine, and a controller [2]:

- The reactive machine is made of the object program obtained after compiling the synchronous program, the interface functions for inputs and outputs, and the run-time specific to the target processor.
- The transformational machine implements the constants, types, procedures, and functions external to the synchronous program in the chosen host language (for instance C).
- The controller coordinates everything together.

As said in Section 1.2, a synchronous program is compiled into a function in the host language. This function, which belongs to the above reactive machine is itself *transformational and not reactive*. This means that it must be explicitly invoked, possibly with inputs, and that it terminates, possibly with some results. The role of the execution machine is precisely to give a reactive behavior to this transformational function. To this end, the controller must trigger the reactions of the program to make it reactive to its environment. Hence the controller must include an *execution loop* in charge of invoking the transformational function. Each invocation corresponds to an

instant of the synchronous program. We present in the next section two strategies for this execution loop.

Finally, the transformational function has strictly speaking neither inputs nor outputs. The program inputs are implicit in the sense that they are updated by dedicated functions. Concretely, to each input signal corresponds a function in charge of updating the value of the signal (except if the signal is pure) and marking the signal as present. These update functions must be invoked by the execution machine. Concerning the program outputs, they are explicitly emitted by output functions invoked by the transformational function. These output functions must be written by the programmer.

It is clear that no execution machine, no matter how fast, can react in zero time. This fact may seem redhibitory for the execution of synchronous programs. The remaining of this paper shows how to remove this obstacle and achieve synchrony in non zero time.

2.2 The Execution Loop

We distinguish two models for the execution loop: the *general* model and the *periodic* model:

- In the general model, each input event triggers a new reaction of the program.
- In the periodic model, the program reactions are triggered at each "tick" of a real-time periodic clock.

Here is the code in each case:

- General model:

```
for_each event
        read more inputs
        compute next state
        emit outputs
end_for_each
```

- Periodic model:

```
for_each tick
        read inputs
        compute next state
        emit outputs
end_for_each
```

In each case, several tasks must be taken into account besides the program:

- In the general model:
  - the sensors for the inputs coming from the environment,
  - the local bus for inputs coming from the other subsystems.
- In the periodic model:
  - the real-time periodic clock,
  - the sensors for the inputs coming from the environment,
  - the local bus for inputs coming from the other subsystems.

For commodity reasons, we call such tasks *interface tasks*. Each interface task is executed concurrently with the program, and has a higher priority. Their implementation will be explained in the sequel.

Each interface task, except the real-time periodic clock, invokes the update function of the corresponding input. It is important to distinguish between the sensor reading task and the sensor itself. As we have seen in Section 1.3, continuous inputs are sampled by a sensor. This sampling can be periodic, triggered by the program (polling), or even triggered by the sensor itself (smart sensor). Concerning the discrete inputs, we have said that they are fleeting and must be expected specifically: this is exactly the purpose of the interface tasks executed concurrently with the program.

All these tasks can interrupt the execution loop, which raises two problems: the consistency between inputs and the validity of the synchrony hypothesis. We will study these two problems in the following sections.

Finally, let us mention the fact that the most commonly used model in industry applications is the periodic one. It is for instance the case of the nuclear plant controller CO3N4 of SCHNEIDER ELECTRIC, as well as the flight controller of the AIRBUS A340 of AÉROSPATIALE.

2.3 Consistency Between Inputs

The problem of the consistency between inputs in a given instant comes from the possibility for a given input to be updated *during* the program transition, that is during the `compute next state` phase of the execution loop. For instance, during a reaction, an ESTEREL program may read twice the value of an input signal. If, due to an interruption of the interface task of this input sensor, the input value is updated, there will be an inconsistency.

Programmable logic controllers already encounter this problem. In order to prevent the risk of a value change

during their reaction, they set all the input values at the beginning of a reaction and keep them during the whole reaction.

We adopt a similar solution for execution machines. All the inputs received since the previous instant are memorized into buffers, and then, during the reaction each signal is read at most once from the buffers. Inputs are thus read exclusively during the `read more inputs` or `read inputs` phase of the execution loop. Besides, the program has a vector of buffers, one for each of its inputs. Each buffer contains a value of the entry type and a Boolean telling whether or not the input has been received since the previous instant. As a result, the inputs are received by the program in the following way:

- When an interface task interrupts the execution loop because a new input has been received, it writes this value in the corresponding buffer and sets the Boolean to `true`. Any further interruption of the same interface task writes a new value in the buffer, overwrites the previous value, and lets the Boolean to `true`. If the considered input is continuous, then the loss of the overwritten value makes sense since it is preferable to work with the newest value. If the considered value is discrete, then the loss of the overwritten value means the loss of an event: we address this problem in the following section.
- When the program runs the `read more inputs` or `read inputs` phase, it scans the buffer vector, and for each Boolean set to `true`, it invokes the corresponding update function with the value of the buffer. At the same time, all the Booleans are set to `false`. This scanning of the buffer vector must be executed within a critical section, so as to be impossible to interrupt. It is the only part of the execution loop that must be so.

Figure 2 illustrates this behavior. Here, the function `MODULE_I_X` is the update function of the input signal `X`. Arrows represent the control, not the data flow.

Figure 2: The `read inputs` phase of the execution loop

2.4 Validity of the Synchrony Hypothesis

Validating the synchrony hypothesis means proving that the program is *faster* than its environment. This is a *physical* interpretation of the *ideal* notion of instantaneity. This property ensures in particular that no input event can be *lost*. The importance of this property comes from the fact that some input events are fleeting. Without the interface tasks mechanism presented in Section 2.3, in order to prove that the program is faster than its environment, it would require to prove that the reaction time of the program is systematically lower than the time lag between any two successive input events. In any case, it is not possible to establish necessary conditions that validate the synchrony hypothesis. The conditions that we establish in this section are thus *sufficient conditions*.

Let us define formally the program reaction time as well as the input clocks:

- The program *basic reaction time* is the maximal time for running the sequential code obtained after compiling the program for the target processor. Since this code is sequential and deterministic, it is possible to find an upper bound of this time from the characteristics of the target processor. This upper bound is what we call the basic reaction time.
- The program *total reaction time* is the sum of the basic reaction time plus the execution time of all interface tasks during one period of the real-time clock.
- The *clock* of an input, be it discrete or continuous, is the infinite sequence of the instants when the input events occur.
- The *minimal period* of an input is the minimal period of its clock.

In order to compute the execution time of all interface tasks, it is necessary to know the minimal period of each input. These frequencies must therefore be given in the system specification.

Within the periodic model, it suffices to satisfy two conditions to be certain that the software implementation satisfies the synchrony hypothesis. The first one is that the real-time clock period be greater than the program total reaction time: this ensures that the program has enough time to run between two successive ticks of the real-time clock. The second one is that the smallest of all the inputs minimal periods be strictly greater than the real-time clock period: this ensures that no input event is lost[1].

[1] In order to determine by what margin the smallest of all the inputs minimal periods must be strictly greater than the real-time clock period, it is actually necessary to take into account the characteristics of the sensor hardware: time needed to prepare and maintain the sensed value, minimal time between two successive acquisitions,...

Within the general model, a first approach consists in requiring that the minimal period of the *union* of all input clocks be greater than the program basic reaction time. The union clock is the infinite sequence of the instants of *all* the input events. A first relaxation consists in excluding the continuous inputs from the clocks union, and thus in triggering a sampling of each continuous input during the `read more inputs` phase of the execution loop. Still with this approach, the period of the union clock can be very small. A second relaxation consists in considering the infinite sequence of time slots where *only one* input event occurs. The minimal period of this union clock is greater.

In conclusion, the general case is much more constraining to validate than the periodic case. This is one of the reasons why the periodic model is the most employed in industry.

3 Practical Implementations

3.1 Introduction

We present in this section some techniques for implementing synchronous execution machines. Besides the formal aspects seen in Section 2, these techniques allow the taking into account of the practical aspects of the implementation, that depend on the programming model, the hardware environment, and the context where the synchronous program is used.

First, we explain how to satisfy the constraints established in Section 2. It consists of a finer description level where we describe usable mechanisms and techniques. We also draw the attention on possible problems. Such dysfunctions must be considered as warnings to the reader willing to design its own synchronous execution machine. Finally, we present some effective implementations, restricted to centralized solutions.

Concerning the practical implementations, few detailed documents are available. The documentation of ESTEREL–V5 [8], given along with the compiler distribution, includes low level informations on the C/ESTEREL interface. Of course, these only concern ESTEREL. Yet, while the problem of the synchronous execution machine is not specifically treated there, this documentation is very useful for designers.

3.2 Architecture of an Execution Machine

The architectural description makes it possible to understand what are the main functional components of the execution machine, and their interactions.

3.2.1 Information Flows

An execution machine is a reactive system whose purpose is to react to incoming information by generating output information. This role has been explained in Section 2.1. These information flows have to be *controlled*: dedicated control signals are in charge of that. Possible dysfunctions of the execution machine are indicated by exceptions signals.

3.2.2 Control

The set of control signals includes:

- An input signal *begin of instant* (`BoI`), which is compulsory. Its occurrence triggers a new reaction of the execution machine.
- An output signal *end of instant* (`EoI`), which is also compulsory. This signal is emitted by the execution machine, after the output image has been updated. With respect to the environment, the occurrence of (`EoI`) indicates the end of the current reaction.
- Optional control signals, which are used for fine control of the execution machine. They are especially useful for hierarchical execution machines. They can stop, suspend, resume, and re-initialize the execution.

In this section, we consider only the first two signals.

The respective dates of occurrence of `BoI` and `EoI` must be such that the synchrony hypothesis is satisfied (see Section 2.3). The simplest case is the periodic activation: `BoI` signals are periodically emitted by an external clock. Of course, the `EoI` associated with a `BoI` must be emitted *before* the end of the clock period.

3.2.3 Monitoring

Observers can be used to monitor the execution machine. In the case of an abnormal behavior of the machine (not of the program), an exception signal is emitted.

These exception signals are, above all, warnings sent to the user of the execution machine. Clearly, raising an exception signal means that the implementation *is no longer*

running under the synchronous hypothesis. The user *must* be kept informed of this problem.

In more sophisticated execution machines (e.g., fault-tolerant execution machines), exception signals can be handled by a higher-level execution machine. The upper machine can then force actions in the lower machine through the optional above-mentioned control signals. The user must be cautious with this kind of "control loop" in execution machines: the handling of an exception may cause the execution machine to violate timing constraints. The cure will be worse than the disease!

Below, we list some typical dysfunctions; this list is not exhaustive:

1. *Violation of a relation*. Suppose that the user has declared in his/her program that `A` and `B` are two exclusive signals (i.e., never simultaneously present). This assertion may be violated during a reaction. The reason for this violation may be either a lack of knowledge about the environment, or a sensor failure. The latter is a chance event that can be detected only while the system is operating. The former is a misconception and should be avoided by rigorous design methodologies. In both cases, since violations may lead to unpredictable executions, the execution machine *must not ignore this violation*. A possible conservative strategy is to "filter" faulty signals, so that only *acceptable* events are considered for executions. There exist several filtering techniques; none is fully satisfactory. Whatever the recovering politic adopted, all violations must be reported by the execution machine.

2. *Lasting transition*. An execution machine can arm a watchdog at each beginning of a reaction. If the transition is not terminated before the deadline, an exception is raised. This exception can be due to a transient overloading of the system or to errors in the user's program. The latter is often due to execution of the transformational parts of the program (e.g., calls to external functions or procedures). It is the responsibility of the designer to ensure that external transformational parts of his/her program have a bounded and known duration. When this property cannot be guaranteed, asynchronous executions must be considered for this data processing (see below the notion of "task" in ESTEREL).

3. *Data overwriting*. Observers can be attached to acquisitions and actuations. Overwriting a value means that the application is no longer run in real-time.

3.2.4 Structure

The various functionalities of the execution machine can be assigned to dedicated modules (Figure 3). Dashed lines are flows of control, whereas solid lines are data flows. `I` is the input tuple presented to the *synchronous kernel*. The kernel computes the reaction and generates the output tuple `O`.

The *controller* ensures the correct synchronous behavior: atomic reaction and bounded reaction time. In its simplest form, the controller is a sequencer whose behavior can be expressed by the following ESTEREL-like pseudo-code:

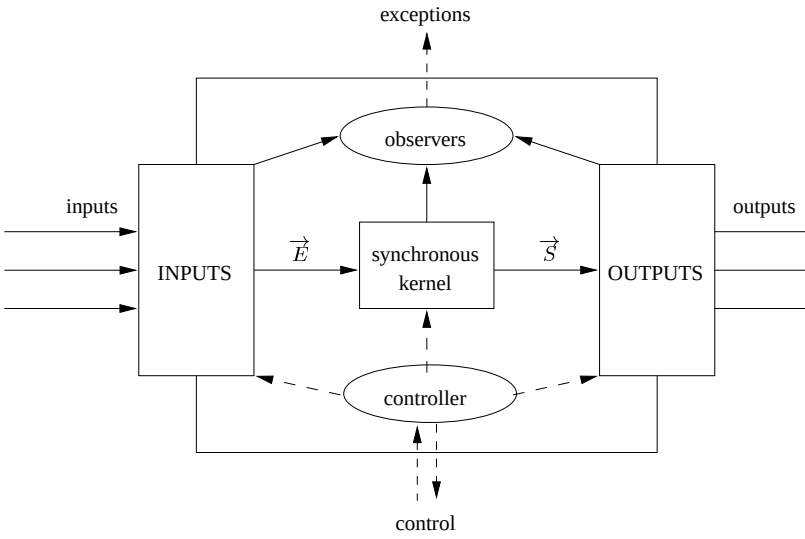

Figure 3: Execution Machine: Structure

```
initialization
every BoI do
     read inputs; build I;
     react;
     build O; write outputs;
end_every
```

This pseudo-code is compatible with the one presented in Section 2.2. Auxiliary variables have been introduced

and some phases refined. For instance, the `read inputs` phase of the execution loop is refined into a subphase of input acquisition (`read inputs`) and a subphase of input tuple construction (`build I`).

3.2.5 Inputs / Outputs

The consistency of inputs has been analyzed in Section 2.3. Modules `Input` and `Output` in Figure 3 make the necessary interfacing between the synchronous kernel and the environment. They are, themselves, reactive systems with their own control flows and data flows. Figure 4 shows a possible refinement of the input module.

- Modules `A` are interface tasks described in Section 2.2. They may be interruption handlers or peripheral drivers. A signal `R` (for "Reading") triggers the sending of a value `a`.
- This information is consumed by an optional filtering module `F` that produces signals (with the synchronous language meaning of this word). These filtering modules are useful for imperative synchronous languages since they give greater importance to events instead of values. Consider for instance an ESTEREL program. Let `a` be the logical level 0 or 1 at a push button. When pressed, the button changes from 0 to 1. Now, suppose the ESTEREL program has a pure input signal[2] called `Button_Pressed`. In this case, the filtering module will generate signal `Button_Pressed` at instant k, if and only if, `a` was 0 at instant $k-1$ and 1 at instant k (i.e., the Boolean expression $\overline{a_{k-1}} \wedge a_k$). For a declarative language like LUSTRE, this "edge detection" would have been done by the program itself.
- The `tuple builder` module consumes possibly filtered signals and generates the current input tuple `I`. This generator, in the simplest cases, does a concatenation of signals. In the case of a relation violation, it can also perform extra filtering operations.

[2] In ESTEREL, a pure signal is a signal that conveys no value. Only its presence or absence is of interest.

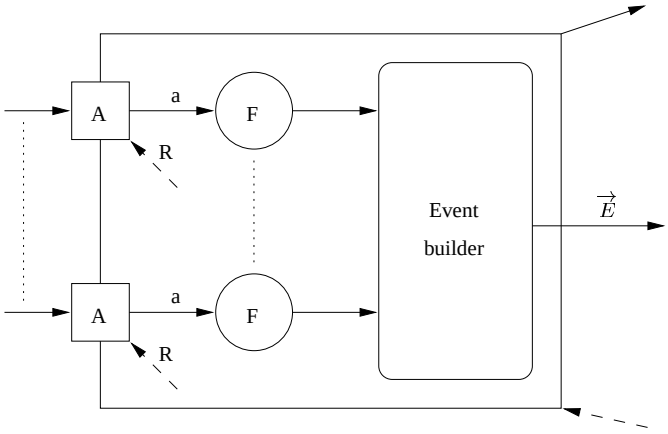

Figure 4: Execution Machine: `Input` Module

3.2.6 Asynchronous Execution: ESTEREL Tasks

Among synchronous languages, ESTEREL is the only one to support *lasting activities*, through the *asynchronous task* mechanism. Since this mechanism interferes with the synchronous/asynchronous interface, we address it specifically in this section. Contrary to functions or procedures that are supposed to take no time (synchrony hypothesis), an ESTEREL task may have any non null duration. A task can perform heavy data processing or activities not directly controlled by the synchronous program (e.g., moving a robot). The body of the task is executed asynchronously with respect to the synchronous program. Interactions with the ESTEREL program are very limited. Without entering into details:

- The task is launched by an `exec` statement;
- When the task terminates, a `return` signal is sent to the synchronous kernel;
- In order to respect the semantics of the language, when a task is executed within the scope of an abort or a suspend, the asynchronous task has to be killed, suspended, and resumed under the control of the synchronous kernel.

All these events exchanged by the synchronous kernel and the asynchronous tasks, are also controlled by the execution machine. See [9] for a detailed description of task execution and possible solutions.

3.2.7 Some Implementations

An execution machine can be small yet very efficient. This is the case for micro-controller-based implementations. For instance, an execution machine for ESTEREL programs has been implemented on the Harris' RTX2000 micro-controller [3]. Implementations on PC usually relies on some real-time operating system (RTOS). The authors have developed applications running under RTC (Real-Time Craft) and CHORUS [4]. More generic machines, but for soft real-time applications are presented in the Boufaïed's thesis [9]. With these machines, easy configuration of inputs/ouputs and module reuses, are the main concern.

The next section develops the implementation of a centralized execution machine composed of several "reactive machines".

3.3 Centralized Execution Machines

As seen in Section 2.1, an execution machine is composed of a reactive machine, a transformational machine and of a controller that coordinates their operation. A *centralized* execution machine is an execution machine with only one controller. This controller manages the synchronous code, input and output operations, and the transformational code. A *distributed* execution machine has several controllers that work together for synchronously executing several reactive machines.

The centralized execution machine is the simplest to implement since it has global control over input, output and synchronous code. Two cases arise:

- the execution machine has only one reactive machine: it must provide it with a clock and inputs, and must drive its outputs to the outer world (see the previous subsection);
- the synchronous code is composed of several reactive machines: it must provide them with a mechanism for communicating synchronously.

The second case is the most general and encompasses the first one. It allows, with some restrictions, to link several synchronous modules that were compiled separately. We discuss in this section the case of several reactive machines.

There are two limitations when using several reactive machines (for instance, several separately compiled ESTEREL modules or LUSTRE nodes). The first limitation is that instantaneous communication loops between reactive machines are *forbidden*. Such loops can be handled by the synchronous compiler since it knows the internal details of each module and is able to determine whether the loops are causal or not, and if so, to compute the behavior of the synchronous system. However, from the point of view of an execution machine, a reactive machine is a black box, and it is not possible to know if an instantaneous communication loop between several reactive machines is causal without more information about the internals of the boxes. The second limitation is that the topology of the connections between the reactive machines must be *static*, that is it is not possible to create dynamically new reactive machines or new connections. We address first the basic case (no instantaneous loops and no dynamic reconfigurations), before relaxing these two limitations in Sections 3.3.6 and 3.3.7.

3.3.1 Logical Instants

A logical instant is defined to be the reaction of the execution machine to a tuple. This leads to the following:

- At the beginning of a logical instant, every signal has the same value and is in the same state for all the reactive machines.
- At the beginning of a logical instant, each reactive machine is in a completely determined state. There is no state transition during a logical instant, only the computation of the next state of the machine.

3.3.2 Sequential Execution of Reactive Machines

When there is no instantaneous communication loop between the synchronous compilation units, there always exists a partial order induced by the dependencies between the corresponding reactive machines. Thus, the execution machine is able to chose an activation schedule that is compatible with this partial order. The execution machine must also propagate the signals that were emitted during the reaction of a reactive machine so that they are seen in the same instant by the reactive machines that follow it in the schedule.

The schedule is determined once for all the instants since the connections between the machines do not change. At each instant, the execution machine sends the `BoI` control signal to each reactive machine so that they

are all in the same logical instant. When the machines are ready to process the new instant, the execution machine activates them according to the schedule. The activation of a reactive machine consists of three steps: build its input tuple, compute its state for the next instant, and build its output tuple. Last, when all the reactive machines have been activated, the execution machine sends them the `EoI` control signal that marks the end of the instant.

The input and output tuples are built during the activations because some inputs of a machine may be produced by another machine which is activated earlier in the schedule, so the input tuple of each machine cannot be built at the beginning of the instant.

The inputs and outputs of the system must be handled separately because, to preserve the synchronous semantics, the reactive machines must have the same view of the outer world: if a signal is present at a given instant, it must be present at this same instant for all the reactive machines. Therefore, we cannot allow each reactive machine to sample the inputs of the system during its activation.

3.3.3 Input / Output Machines

One method of ensuring the consistency of the inputs for all the reactive machines in a system while preserving this execution model, is to use reactive machines to handle the inputs and the outputs of the system. From the point of view of the execution machine, an input reactive machine has only outputs, so its does not depend on any other machine and will be executed at the beginning of the schedule. Conversely, an output reactive machine has only inputs, so no other machine depends on it and it will be at the end of the schedule.

These input/output machines are a way to implement the `Input` and `Output` modules of Figure 3. Although they are reactive, these machines are seldom written in a synchronous language but rather in the implementation language of the execution machine. They are similar to drivers in an operating system: they provide an abstract view of the environment of the synchronous system in the form of tuples.

3.3.4 Behavior of a Reactive Machine

The behavior of a reactive machine can be seen as made of three phases: processing the beginning of the instant, activation, and processing the end of the instant.

Processing the beginning of the instant is generally a matter of setting the outputs absent. But for an input machine, the outputs are set according to the data coming from the interface tasks (`A` modules in Figure 4). Similarly, a "delay" reactive machine will set its output according to the input it has received at the previous instant.

During the activation, the machine builds its input tuple, then determines its next state and the output tuple. For an input machine, the output tuple was built when processing the beginning of the instant. However, such a machine may compute its next state if its method for generating tuples depends on the data it receives from the interface tasks.

In most cases, processing the end of the instant is merely setting the state of the machine to the state that was computed during the activation. For output machines, processing the end of the instant consists in propagating the output of the system to the outer world.

3.3.5 Iterating Reactive Machines

If the execution machine knows some information concerning the dependencies between the outputs and the inputs of the reactive machines, and if these machines are able to compute some of their outputs without knowing all their inputs, instantaneous communication loops between reactive machines may be allowed. For this, we need to consider two kinds of reactive machines:

- *strict* machines, that need to know all their input to be able to compute any of their outputs,
- *non-strict* machines, that may compute some of their outputs without knowing all their inputs.

When all reactive machines are strict, instantaneous loops are forbidden, and the execution machine uses sequential execution as discussed above.

When some reactive machines are non-strict, they are allowed to appear in instantaneous loops. Edwards proved that if these machines are *monotonic*, that is if they compute more of their outputs when they are provided with more of their inputs, the behavior of the synchronous system is the unique fixed point reached by iterating the reaction of the machines [13]. Moreover, the number of iterations and the sequence of the activations in each iteration can be statically determined from the topology of the synchronous system.

The key idea is to consider the tuple of all inputs `i` and outputs `o` of the system, and to consider the system as a

function that produces a new tuple `(i,o)` from the one it receives, as shown on Figure 5:

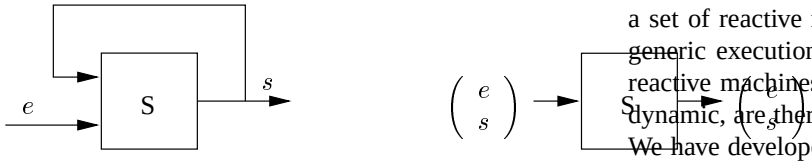

Figure 5: System with a loop : equivalent tuple system

Let us represent a not yet determined signal by $\perp$, a present signal by P and an absent signal by A. These values are partially ordered, the corresponding partial order being shown in Figure 6(a). Such a partial order can be extended to two signals, as shown in Figure 6(b): according to this order, (A, P) is more determined than $(\perp, P)$, but (A, P) may not be compared to $(P, \perp)$. By generalizing to n signals, it is possible to sort the possible values of the `(i,o)` tuple from the less determined to the most determined.

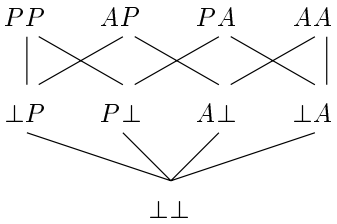

Figure 6: (a) Partial order on a single signal tuple (b) Partial order on a two signal tuple

When reactive machines are considered as functions that compute signal tuples, the condition for the existence and uniqueness of the fixed point is that these functions are monotonic for the partial order on signal tuples. This property merely ensures that only undetermined signals may change during a partial reaction of the machine. It implies also that the value of a valued signal cannot be changed once it is determined.

The main difference between this execution model and the sequential model is that the output tuple is built in several steps, and the next state of a machine cannot be computed before the end of the instant.

This iterative execution model is used in the "Synchronous Reactive" (SR) and "Synchronous Reactive C Code Generation" (SRCGC) domains of the PTOLEMY Classic[3] system developed at the EECS department of the University of California at Berkeley.

[3] http://ptolemy.eecs.berkeley.edu

3.3.6 Generic Execution Machines and Synchronous Objects

An execution machine may be designed specifically for a set of reactive machines, but it is possible to design a generic execution machine that may execute any set of reactive machines with known properties: is the system dynamic, are there instantaneous communication loops... We have developed in [10] such a generic execution machine.

This scheme requires a standard interface for the reactive machines so that the execution machine may manage them without knowing their internal details: we need an abstract notion of a reactive machine.

Object oriented languages allow the definition of abstract entities and the refinement of their behavior for more concrete entities. Once we have defined the abstract reactive machine as a class, we are able to implement a particular reactive machine as a subclass. Such subclasses are named "synchronous classes", and instances of these classes are "synchronous objects" [10].

Any synchronous class must be able to process the beginning of the instant, the activation, and the end of the instant. It may be useful to be able to get the list of signals and the dependencies between outputs and inputs for such a class. Each synchronous class implements these services according to its intended behavior, but what is important is that the execution machine does not need to know the details: it is enough for it to know that a synchronous objet will answer its request to process the beginning of the instant for instance.

The execution machine is a class library that provides the reactive machines with everything they need to run: scheduling, definition of the logical instants, communications between reactive machines, and input/output.

3.3.7 Dynamic Synchronous Behaviors

A generic execution machine may allow the creation and/or destruction of reactive machines, as well as changes in the interconnection of the machines during their execution. This allows synchronous systems to be *dynamic*: their reaction to an input tuple can lead to a reconfiguration of the system. Dynamic reconfiguration may be used to switch from a full featured system to a basic system in case a failure makes some resource unavailable.

However, a dynamic synchronous system is still a synchronous system, so it cannot change *during* an instant. Therefore, the execution machine must record the requests for changes and process them between the end of the current instant and the beginning of the next instant. Such changes may invalidate the schedule, so the execution machine must compute a new schedule each time it processes reconfiguration requests between two instants. It may then discover that the reconfiguration leads to an invalid system for which no schedule can be found. Such a case should be signaled through the exception mechanism discussed earlier in Section 3.2.3.

4 Conclusion

We have addressed in this paper the problem of executing a synchronous program in an intrinsically asynchronous environment. The main issue concerns the satisfaction of the synchrony hypothesis by the implementation. We have proposed an implementable model, called *execution machines*, to solve this problem. The purpose of an execution machine is to ensure that the synchronous program performs atomic reactions and meets the imposed real-time constraints.

We have shown that for control oriented applications, centralized solutions can be achieved easily and efficiently. We have stated several constraints on the rate of inputs and the reaction time of the synchronous program that must be satisfied by the implementation.

For applications where the real-time constraints are less strict, we have presented more sophisticated solutions that allow, for instance, the execution of several synchronous modules that have been compiled separately, even in the presence of instantaneous communication loops and dynamic reconfiguration.

This presentation does not pretend to be exhaustive. For instance, the class of distributed implementations of synchronous programs has not been discussed. There exist such implementations of distributed execution machines. The interested readers may refer to already published papers [12, 19].

References

[1] C. André. Representation and analysis of reactive behaviors: A synchronous approach. In *CESA'96*, Lille, France, July 1996. IEEE-SMC.

[2] C. André and M.-A. Péraldi. Effective implementation of ESTEREL programs. In *5th Euromicro Workshop on Real-Time Systems*, Oulu, Finland, June 1993.

[3] C. André and M-A. Péraldi. Predictability of a RTX2000-based implementation. *Real-Time System*, 10(3):223–244, May 1996.

[4] C. André, A. Ressouche, and J-M. Tanzi. Combining special purpose and general purpose languages in real-time programming. In *Workshop on Programming Languages for Real Time Industrial Applications*, Madrid (Spain), December 1998. IEEE.

[5] J.-L. Bergerand and E. Pilaud. SAGA: A software development environment for dependability in automatic control. In *SAFECOMP'88*. Pergamon Press, 1988.

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

[7] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[8] G. Berry and the ESTEREL Team. *The* ESTEREL*-V5 Documentation*. CMA/INRIA, Sophia-Antipolis, France, 1998. Available at http://www.esterel.org.

[9] H. Boufaïed. *Machines d'exécution pour langages synchrones*. PhD Thesis, Université de Nice-Sophia Antipolis, November 1998.

[10] F. Boulanger. *Intégration de Modules Synchrones dans la Programmation par Objets*. PhD Thesis, Université Paris XI – Orsay, Orsay, France, 1993.

[11] M.C. Browne and E.M. Clarke. SML: A high-level language for the design and verification of finite state machines. In *International Working Conference from HDL Descriptions to Guaranteed Correct Circuit Designs*, Grenoble, France, September 1986. IFIP.

[12] P. Caspi and A. Girault. Execution of distributed reactive systems. In S. Haridi, K. Ali, and P. Magnusson, editors, *1st International Conference on Parallel Processing, EURO-PAR'95*, volume 966 of *LNCS*, pages 15–26, Stockholm, Sweden, August 1995. Springer-Verlag.

[13] S. Edwards. *The Specification and Execution of Heterogeneous Synchronous Reactive System*. PhD Thesis, UC Berkeley, Berkeley, CA, 1997.

[14] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[15] D. Harel. STATECHARTS: A visual approach to complex systems. *Science of Computer Programming*, 8(3), 1987.

[16] D. Harel and A. Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems, NATO*. Springer-Verlag, 1985.

[17] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[18] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In W.R. Cleaveland, editor, *3rd International Conference on Concurrency Theory, CONCUR'92*, volume 630 of *LNCS*, pages 550–564, Stony Brook, USA, August 1992. Springer-Verlag.

[19] Y. Sorel. Massively parallel computing systems with real time constraints, the "algorithm architecture adequation" methodology. In *Massively Parallel Computing Systems Conference*, Ischia, Italy, May 1994.