# EXECUTION MACHINE FOR SYNCHRONOUS LANGUAGES

**Charles ANDRE** **Hédi BOUFAÏED**

Laboratoire I3S, Université de Nice/CNRS
41, Bd Napoléon III
06041 NICE cedex, France
**andre@i3s.unice.fr**
Tel: +33 497 258 257
Fax: +33 493 212 054

**ABSTRACT**

Control-dominated systems, like controllers, are reactive systems often subject to real-time constraints. The programming style adopted for these applications is rather special: event- or interrupt-driven programs involving complex coordination. Imperative synchronous languages like Esterel have been introduced to cope with these applications.
We have developed an environment to deal with control-dominated systems. The user expresses the expected behavior of the controller using a synchronous formalism. Given this description and a configuration (inputs, outputs, interaction policies, …), a dedicated "execution machine" is generated.
This paper presents the outline of this approach based on a the synchronous paradigm, and explains the role and the architecture of the execution machine.

**Keywords**: synchronous programming, execution machine, control, implementation.

**NOMENCLATURE**

We use `Courier` as the font for programs and program objects. **`Boldface Courier`** font denotes file formats and extensions.

**INTRODUCTION**

Critical control systems such as avionics, life monitoring, and automatic control applications are becoming increasingly complex. Their implementations have evolved from mechanical devices, to electronic components and then to embedded computer systems. Designing such an application is now relevant to "Software Engineering". These systems are highly reactive and subject to real time constraints. Their behavior should be fully predictable under any circumstances. So, there is a clear demand for

- Well-adapted programming languages,
- Powerful validation tools (tests and proofs),
- Efficient and dependable implementations.

Reactivity (ability to respond to any significant stimulus) and real-time constraints (ability to respond in time) give rise to difficult programming issues.

**Synchronous Programming**

The "synchronous languages", introduced in the seminal paper of Benveniste and Berry (1991), cope with these kinds of problems. The synchronous approach to reactive and real-time system programming offers several advantages detailed in the Halbwachs's book (1993):

- Multi-style programming: either declarative, or imperative.
- Textual (languages), graphical (various charts), or mixed descriptions.
- Mathematical semantics based on the synchronous hypotheses.

If correct, a synchronous program or chart that fully respects the synchronous hypotheses can be compiled into a semantically equivalent description written in a common format called **`dc`**. The heart of a **`dc`** description is a set of Boolean equations: Boolean expressions trigger execution of classical data processing. Industrial compilers and development platforms are now available. The interactive simulation with source-level debugging is an efficient way to check reactions to various scenarios. Because of their formally defined semantics, synchronous programs lend themselves to formal proofs of properties. Safety properties, which are often critical, are the simplest to prove, even on real-world applications (intensive use of BDD (Binary Decision Diagrams) computations on Boolean automata).

Several different implementations can be derived from the Boolean equation system associated with a synchronous program. In this paper, we focus on "software implementations" (i.e., using a classical language like C, C++, …).

### Input / Output Handling

The above mentioned solutions apply well to the "heart"of reactive real-time systems (i.e., control and data processing). Few tools in synchronous development environments deal with the actual interactions with the external world. And yet, this is a key point in controller design. Programs that manage (real-time) inputs/outputs are known to be various, specific, hardware-dependent, and of little reusability. ROOM methodology introduced by Selic and al. (1994) and more recently "Real Time UML" proposed by Douglass (1998), advocate the use of objects in real-time system programming. Encapsulation of data and behavior leads to a more intuitive and powerful abstraction of acquisition and actuation. We have partially adopted this point of view.

### Execution Machine

A (real-time) controller is both a "reactive kernel" and an "interface driver". An effective, efficient, and dependable cooperation between the reactive code and the environment to be controlled needs special supports. We call "execution machine" for a synchronous program an executable architecture that supports this cooperation.

The main functionalities of an execution machine are

1. Acquisition from sensors and construction of the input image of the process to be controlled,
2. Execution of reactions specified by the synchronous program or chart,
3. Actuation from the output image generated by the reaction.

Of course, all these operations must be done in a timely manner, and the overall behavior must be consistent with the synchronous hypotheses.

In this paper, we report on our experience in building execution machines for controllers programmed in the Esterel synchronous language. A detailed presentation of this language and its environment, written by Berry (1997), is available on the web. We propose a general, generic, and flexible architecture for execution machines. An object-oriented approach has been adopted. The underlying programming language is C++.

The use of Esterel entails a fourth functionality for the execution machine: the asynchronous task management. The reason is that Esterel introduces first class objects (tasks) for lasting actions (i.e., actions whose duration cannot be considered as negligible). These tasks run concurrently with the synchronous control.

### Paper Organization

In a first section, we briefly comment the synchronous hypotheses. A simplified example of control illustrates the imperative synchronous programming style. This example points out what is part of a synchronous program and what have to be provided. The issue of "lasting actions" is also evoked. The second section is devoted to the architecture and the role of the execution machine. Its implementation is described in the third section. Finally, we illustrate the design of a controller for an ATM (Automatic Teller Machine) using our approach and available tools.

## SYNCHRONOUS PROGRAMMING

### Synchronous Hypotheses

A synchronous program expresses the "reaction" that must be done in response to stimuli. In real-world systems, because of concurrency, reactions may result in intricate overlapping actions. The synchronous approach considers simplified interactions (synchronous hypotheses):

- Inputs and outputs are manipulated as ``vectors'' of signals, i.e., their status and value do not change during a reaction.
- Computations take no time (i.e., internal computations are 0-duration).
- Information exchanges rely on instantaneous broadcasting.

From the programmer point of view, a synchronous program instantaneously reacts to external events. Another noteworthy feature introduced by the Esterel language is the extensive use of "preemption", which is a first class concept in this language.

Thanks to the simplifying hypotheses underlying the synchronous paradigm, the parallel composition defined in a synchronous language is fully deterministic. Another consequence is that sequence, concurrency, and preemption are orthogonal concepts. They can be nested at any level, in any order. The resulting behavior is perfectly defined. For all these reasons synchronous formalisms are very good in expressing complex reactive behaviors.

The programmer may choose either a declarative or an imperative style. Which one to adopt is a matter of convenience. Most reactive applications involve both data handling and control handling. Since our applications are control-dominated, in what follows, we adopt the imperative style.

### An Example of Control

The module below illustrates the Esterel programming style. This module is a control-loop. A program may be composed of many such modules and other modules that coordinate their activities.

`Control_loop` applies a classical regulation algorithm (PID = Proportional-Integral-Differential) at each occurrence of Sample. The regulation takes place as soon as `Start` occurs and is aborted by `Stop`.

```
1 module Control_loop:
2  type SigType;
3  function PID(SigType):SigType;
4   input Start,Stop,Sample:SigType;
5   output Cmd:SigType,RegON;
6   await Start;
7   abort
8     every Sample do
9       emit Cmd(PID(?Sample))
11    end every
11    ||
12    sustain RegON
13   when Stop
14 end module
```

Lines 2-5 constitute the declarative part, while lines 6-13 express the behavior. At line 2, `SigType` is a user-defined type, and `PID` (line 3) is a user-defined function. The type and the function body are not part of the synchronous program and they must be supplied by some general purpose language (usually C). With respect to the synchronous program, a function is an abstract action whose duration is 0 and a type is abstract: it is used only for type-checking. The reactive part is almost self-explanatory. Just notice that `?Sample` stands for the current value conveyed by signal `Sample`.

This module is generic. In a digital discrete control application, there are several instances of this module. The run statement is used to create new instances. Optionally, interface items (types, functions, inputs, outputs and others not shown in this example) can be renamed:

```
  run TempReg/Control_loop [
    type float/SigType;
    signal Temperature/Sample,
           Heater/Cmd ]
```

This statement instantiates a temperature regulation loop. Real numbers (`float`) are used in the computation. `Temperature` is the signal from the temperature sensor, `Heater` refers to an actuator. Other signals not renamed are left unchanged. Giving a new name to the module (`TempReg`) is optional but useful in debugging.

### Lasting Actions

As previously stated, the execution of a function is supposed to be instantaneous. So are procedure executions. This assumption is obviously unrealistic for some treatments (e.g., large matrix inversion, robot motion, …). Tasks were introduced in Esterel to deal with "lasting actions". A task executes asynchronously with respect to the program. In a first approximation, an Esterel program launches a task by emitting, to the environment, a request for "starting" this task. The task, then, executes in this environment regardless of the synchronous program. When the task terminates, it sends a "return" signal to the Esterel program. This signal unblocks the thread that was awaiting this termination.

This is a simplified view. Preemption makes the matter more difficult to deal with. A task may be suspended or aborted. For the synchronous program an aborted task no longer exists. And yet, in the environment, the actual task may still be running. It is the responsibility of the execution machine to ensure that the synchronous program receives only consistent return signals. This treatment must be transparent to the user.

## EXECUTION MACHINE ARCHITECTURE

### Goals

Due to the synchronous hypotheses, introduced in the previous section, complex reactive behaviors can be expressed in clean and precise terms. This idealization of real-world systems is conceptually very useful. But, is this abstract view suitable for actual implementations? The execution machine is our response to bridge the gap between the ideal control expressed by the synchronous formalism and its implementation.

Basically, an execution machine should be a "good" approximation of the ideal infinitely-fast machine of the synchronous paradigm. This is necessary but not sufficient to address the problem of reactive system implementation. In our solution, the execution machine has four main missions:

1. To execute reactions so that the input / output behavior be consistent with the one described by the synchronous program,
2. To handle incoming and outgoing flows of information in real-time,
3. To manage asynchronous treatments (lasting actions) concurrently with the control,

4. To preserve safety brought by the synchronous approach.

In what follows, we explain how to meet these objectives.

**Reaction**

(“Mission 1” of the execution machine).

The first issue is that the synchronous program considers a logical time, whereas the execution machine is subject to the physical time. This implies a discretisation of time. The execution machine proceeds through a series of non-overlapping executions. Each execution characterizes one instant and must reflect a reaction of the synchronous model. The “beginning of an instant” must be chosen with care, according to the dynamics of the system to be controlled.

An execution at a given instant is, of course, non instantaneous. In order to ensure an input/output behavior in accordance with the model, the changes in status and signals must be atomic.

An execution proceeds in three sequential steps:

1. Get a “snapshot” of the input signals (input image),
2. Perform the reaction,
3. Generate a fresh image of the output signals.

Most programmable logical controllers run their programs this way. Working on steady signals (images) instead of on-fly signals is necessary to avoid critical races in sequential evolutions. Moreover, “images” simulate the input and output vectors used by the synchronous model. The difference with the synchronous reaction is that the output signals are available only at the end of the execution.

Physical time must be considered as both a date and a duration. The duration of an execution must be negligible with respect to the smallest time-constant of the system to be controlled. This is a good approximation to the 0-duration of the synchronous reactions. If this condition is not met, the execution machine will be unable to monitor and/or control the application.

Non-overlapping atomic executions imply that two successive instants of reaction are at least separated by the duration of an execution. We have implemented two “activation policies”:

- Periodic activation: The period of activation must be greater than the worst execution duration. This solution is easy to implement but not always satisfactory for reactive systems with numerous sporadic events.
- Event-triggered activation: When the machine is idle, as soon as a change occurs in the environment, a new execution is launched. This policy seems to match perfectly the philosophy of reactive systems. However, if the machine is running when a triggering event occurs, the new execution must be postponed.

**Input/Output**

This subsection addresses ``Mission 2" of the execution machine. At the model level, since reactions are instantaneous, no input can change during the reaction. This is not the case for the execution machine. As explained before, inputs (seen by the execution machine) are steady during the execution of a reaction, while actual inputs may change in the environment. Input handlers implement the two facets of an input signal. Output handlers play the same role for output signals.

**Generic handlers**

Handlers are generic (arbitrary types and parameters). Input handlers support two acquisition strategies:

- On-fly: The handler samples the signal when needed,
- Interrupt-driven: Changes in the environment cause updating of the information contained in the handler.

**Task Management**

Up to now, the execution machine seems to be idle most of the time and busy only during reactions and input/output handling. This is not true when there are tasks in the synchronous program.

In Esterel, a task, say `T`, is declared by

`task T(`*ref-arg*`) (`*val-arg*`);`

where *ref-arg* is a list of reference arguments, *val-arg* is a list of value arguments. The statement that executes a task is the exec statement. It has the form

`exec T(`*ref-par*`) (`*val-par*`) return R;`

where `R` is the identifier of a return signal. A return signal is a special signal emitted when the associated task terminates.

The third mission of the execution machine is to control the interactions between the synchronous core and the (Esterel) tasks. The activities involved in this management are:

- Starting a task: When an `exec T` ... starts, it signals to the execution machine that a fresh instance of `T` should start with parameters passed by references and by value. The execution machine forwards this information to the underlying real-time operating system (RTOS).
- Killing a task: An `exec T`... statement can be aborted by the synchronous program. With respect to the program, the instance of the aborted task does not exist any more. For the

RTOS this task is still alive. The execution machine has to solve this discrepancy.

- Suspending a task: This case is similar to the previous one, but the task is only temporarily inhibited.
- Filtering return signals: More generally, because of the asynchronism between the synchronous core and the RTOS, there may exist several active instances of a unique Esterel task in the RTOS, although at most one instance is logically running for the program. The execution machine filters possible return signals so that only significant returns reach the program. This is a non-trivial treatment that requires dynamic generation of task references.

**Modular Architecture**

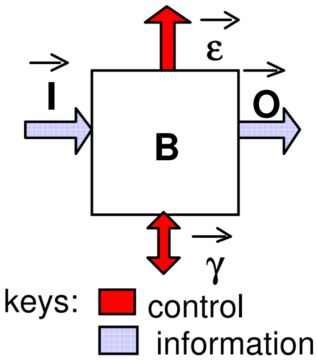


Fig.1 Reactive box.

The execution machine is recursively made of modules. A module is a “reactive box” (Fig.1), the brick of our construction. A reactive box has an incoming information flow (**I**), an outgoing information flow (**O**), and two control flows (the command $\gamma$ and the exception flows $\varepsilon$).
Under the control of $\gamma$, the reactive box generates reactions **O** from stimuli **I**. **B** is the behavior (relationship between sequences of **I** and sequences of **O**). $\varepsilon$ is optional. It is used to report particular situations.

**Execution Machine as a Reactive Box**

The execution machine is a reactive box (Fig.2). It is composed of three reactive boxes: InModule, Synchronous Process, OutModule, and two controllers: Sequencer, Observer.

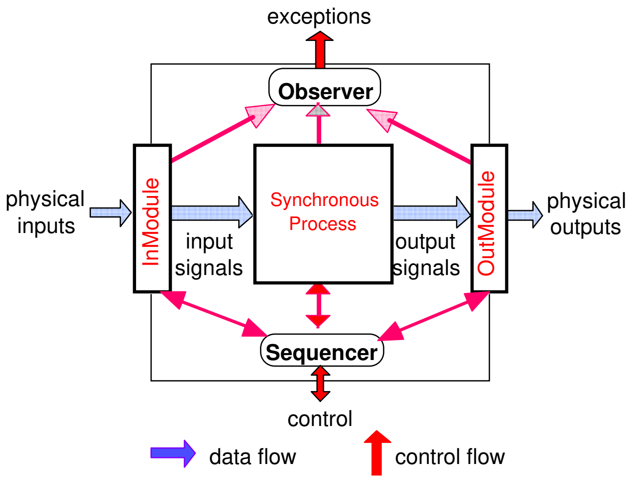


Fig.2 Execution Machine.

On this figure information flows from left to right, while control flows upward. The leftmost information flow is made of “physical signals” (sensor values, operator's commands). The input module (InModule) derives logical signals (the sort of input signals of the synchronous program) from the physical signals. Outputs of the synchronous process represents the sort of the output signals of the synchronous program. The flow generated by the output module (OutModule) is directed to actuators and operator's displays.

**In and Out Modules**

The InModule and the OutModule are themselves refined. Their structures are symmetrical (see Fig.3). Only the InModule will be detailed.
$IH_k$ is an (input) handler module. It captures physical signals and gives logical signals. The Event Builder constructs the “input event” for the current reaction of the synchronous process. In the simplest case, building the input event is just aggregating the various logical signals. The situation is not so easy when the synchronous program contains assertions on signals. In Esterel, such assertions are called “relations”. The programmer may have declared

```
relation A # B;
```

which means that signals `A` and `B` are exclusive. The Esterel compiler takes account of this relation to generate optimized code. At run-time, if it happens that `A` and `B` are simultaneously present, then an exception is generated on $\varepsilon$ and the Event Builder must take a decision. The type of decision (ignore both, give priority to either `A` or `B`, delay for one instant the occurrence of either `A` or `B`, …) is a strategy parameter passed at the instantiation of the Event Builder. Note that a sequencer and an observer are also present in the module.

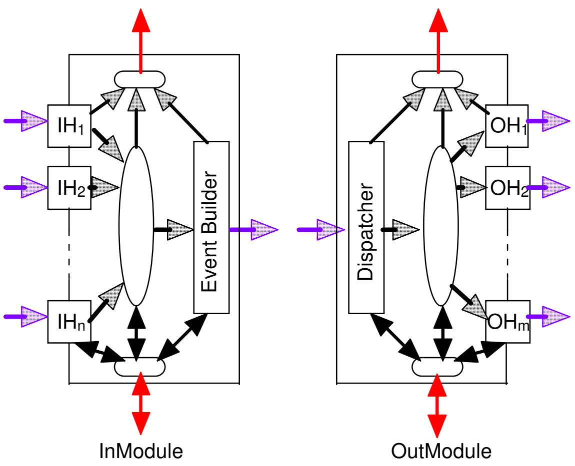


Fig.3 In and Out Modules.

**Handler Modules**

A handler is a terminal reactive box (i.e., with no further refinement). Input handler modules differ on the way they capture physical signals (sampling, interrupt-triggered) and they derive logical signals. Derivation possibilities are numerous. We just mention two typical examples:

- Rising Edge Detection: The physical input is a 2-valued sensor; The logical signal is a pure signal whose presence denotes a change from 0 to 1 of the sensor value.
- Tcl-Tk "sensor": The "physical" signal is produced by a Tcl-Tk widget. This widget contains an entry in which the user types a number. The content of this entry is validated by a "carriage-return". The derivation consists in interpreting the entered string as an integer.

Tcl-Tk sensors or actuators are very useful in human-machine interface applications and also in debugging.

## IMPLEMENTATION

For portability, reusability and maintainability reasons, we have adopted an object-oriented approach, but it is transparent to the user. Figure 4 outlines the compilation chain. The user has to provide:

- A reactive program that expresses the behavior of the controller to be developed,
- The transformational part of the application (types, functions, procedures, tasks, …)
- A configuration of the execution machine.

The reactive program is usually written in Esterel. The transformational part is written in C or C++. The configuration is given by an "execution machine description file" (format **`xmdf`**). This description contains

- The lists of physical signals (sensors, actuators);
- The instances of handlers. (the **`libIO`** library offers predefined classes of handlers, this library may be extended by the user);
- The instances of the Event Builder and the Dispatcher;
- The synchronous process as a synchronous object (see below);
- The interconnections between these objects.

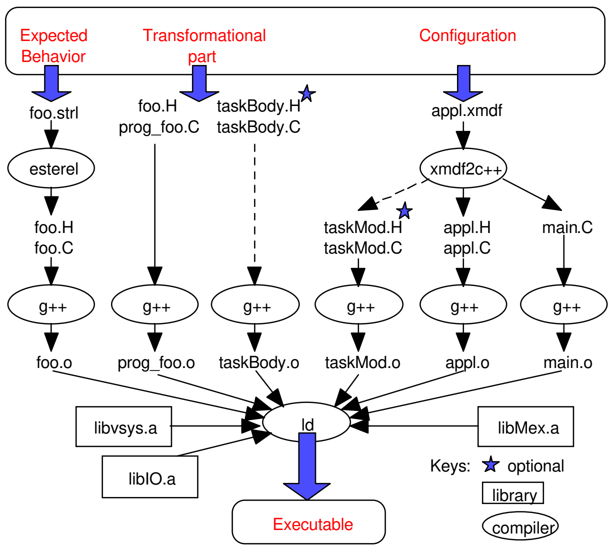


Fig.4 Compilation Chain.

Several run-time modules from the execution machine library (**`libMex`**) and the virtual machine library (**`libvsys`**) are linked with the application-dependent modules.

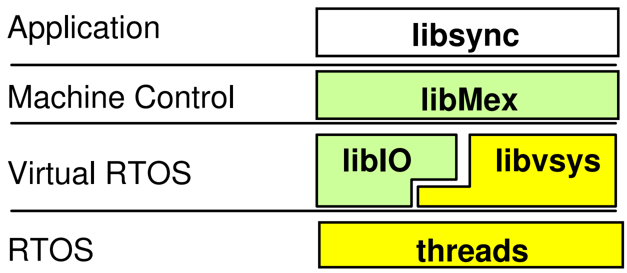


Fig.5 Layers.

Logically, an execution machine is composed of four layers (see Fig.5). Objects of a layer call services of the immediately below layer.

- The RTOS layer: It is the real-time operating system running on the target. Solaris 2 was used for our implementations.
- The virtual RTOS layer: (application-independent but RTOS-dependent). Classes of **`libvsys`** behave like a virtual RTOS. Classes of **`libIO`** provide input / output facilities.

- The machine control layer: (application-independent). Classes of **`libMex`** perform coordination of reactive boxes.
- The application layer: is made of the application-dependent objects.

### Synchronous Objects

Object technology is standard. What is not standard is the combination of synchronous and object programming. We briefly present the "synchronous objects" introduced by Boulanger (1993) to address this problem. His idea was to encapsulate the synchronous code into objects and then manipulate these objects as classical ones.
The synchronous code is a compiled version of a synchronous description. A synchronous class is associated with each synchronous code. All the synchronous classes derive from an abstract class named "`Synchronous`" that defines the basic protocol of any synchronous object. The behavior (dynamic model in Objet Modeling Technology) of the objects of a class is defined in an unambiguous way by the associated synchronous description. This description is more precise and more flexible than statecharts adopted in object-oriented approaches to reactive systems like ROOM or real-time UML.
The Synchronous class has virtual methods to access to interface signals (e.g.,`resetOutputs()`, `setInputs()`). The actual code of these methods is application-dependent. The `react` method deserves a special attention: It allows the object to react according to the synchronous semantics. `react()` disables all output signals (`resetOuputs()`), updates all input signals (`setInputs()`), and calls `activate()`, a method that performs the reaction (internal state and output signals updating).
A synchronous object can communicate with another synchronous object either synchronously or asynchronously. A synchronous object can only communicate asynchronously with a regular object. Synchronous communication imposes that objects share the same notion of instant. Instances of the `Clock` class capture this notion. Each clock determines a scheduling of the objects it manages, so that interconnection of synchronous objects may have a "synchronous" behavior.

### Classes

Fig.6 shows relationships between classes, using the OMT notation. `execMachine` is an aggregation of

- An `InputModule` and an `OutputModule` for interfacing,
- A `Synchronous` for the reaction itself,
- An `execManager` for tasks.

Note that `InputModule` and `OutputModule` derive from `Synchronous`, which is the cornerstone of our implementation.

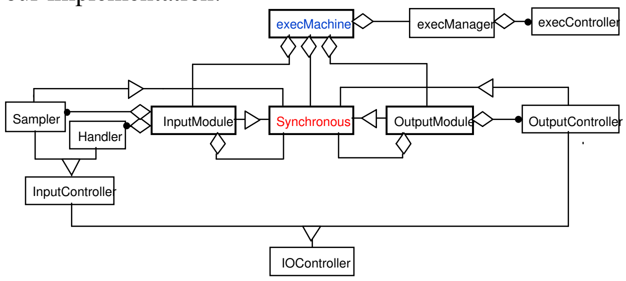


Fig.6 Classes.

## A COMPREHENSIVE SESSION

We choose to present the outlines of the design of the controller of an ATM. It is a usual application, the specification of which is known, in its broad lines, to every body. The ATM exhibits reactive behaviors and involves various interfaces. It is a "soft real-time" system (no harsh time constraints) with strict safety properties (undesirable situations should never occur) The same example has been studied by Rumbaugh (1991) using OMT. In what follows, we explain how our synchronous approach has been applied to the design of an ATM's controller. More details are available in Boufaïed's thesis (1998). In comparison with OMT, we go further in expressing behaviors, interfacing and validation.

### Abstraction

Like OMT, ROOM and real-time UML, we start with scenarios and collaboration diagrams to describe the interactions between the ATM and its environment. Here, the environment is composed of the customer and the bank (a remote computer that manages accounts). From the scenarios we identify events and expected behaviors. Events are abstracted to signals (e.g., the action of asking for an amount of money is associated with an input signal `amount`, valued by integers; the event of keeping a card in case of fraudulent use is associated with the pure output signal `CARDKept`; …). Sometimes, signals are not the best abstraction. For instance, the connection between the ATM and the bank is better expressed by a task taking parameters and returning at the completion of the connection. The expected behavior of the ATM has been expressed by a 1-page synchronous chart (a graphical form of Esterel). This model appears to be very convenient for expressing preemptions. In the ATM behavior specification the cancellation of a transaction is a delicate issue. Pressing the `Cancel` button must be ignored during the money delivery or after three consecutive erroneous confidential codes. A synchronous chart can clearly specify such behaviors.

**Transformational Part**

In the ATM program, we use a user's defined type: `Card`. This type could have been defined as a structure in C. Since access to the fields of the record are necessary, a C++ class has been developed. This is a standard implementation of an abstract data type. More interesting is the choice of tasks. As already mentioned, `connect` is a task dedicated to the connection phase between the ATM and the host; `disconnect` is the corresponding task for the disconnection. `getAndCheckCode` is a more specialized task. Its prototype is

```
task getAndCheckCode()(integer);
```

The value argument is the card number. The invocation of this task is

```
exec getAndCheckCode()(ncard)return
                  CodeOK;
```

where `CodeOK` is a Boolean return signal. The task gets the code from the keyboard without echo, then calls a Perl's script that applies the Unix password encoding algorithm, and finally compares the result with the expected code. The task returns with true if and only if the code is correct. This task does a complex job that surely takes time! This is a convincing example of the usefulness of tasks.

**Interactive Simulation**

A scenario is a sequence of events. An Esterel program covers lots of possible scenarios. An interactive simulation of the program allows the designer to check his/her model against expected behaviors. XES (X Esterel Simultor) is a tool, provided with the Esterel compiler, dedicated to the interactive execution of an Esterel program. XES uses Tcl-Tk interfaces and allows for source-level debugging. Moreover, sequences of stimuli can be recorded and replayed. The phase of interactive simulation reveals unexpected behaviors early in the design process and therefore is almost unavoidable. With our environment, the designer may select Tcl-Tk-based handlers to make his/her simulation more realistic.

**Formal Validation**

Interactive simulation does not pretend to be exhaustive. Since safety properties must be ensured under any circumstances, we need a way to cover all possible evolutions. The XEVE model checker, designed by Bouali (1997) performs such analyses. XEVE performs a symbolic execution of the program and can say for this program whether a signal is potentially emitted (i.e., there exists a reachable state in which the signal is present) or not.

The task of the designer is to associate an observer with each safety property to check. This observer, which is a reactive module, emits a distinguished signal in case of violation of the property to be proved. A property is satisfied if and only if XEVE finds that the violation signal is never emitted by the (controller) program composed in parallel with the associated observer.

Remark: These observers are conceptual and used only in debugging. They do not appear in the final code, contrary to the observers in reactive boxes that monitor the execution.

In our environment, we also propose simplified models of handler behaviors. They can be composed with the controller in order to establish “end to end” safety properties (i.e., properties between physical input signals and physical output signals). Due to the explosion of the state space, the model checking may become untractable. Fortunately, for the ATM (about $10^6$ states) we stay within the capabilities of the current version of XEVE.

**Actual Implementation**

When the synchronous program has successfully got through the interactive simulation and the property checking phases, the designer has to adapt its input/output modules. Actual handlers are then instantiated and the application is recompiled. **`libIO`** supports serial/parallel ports, and sockets. The synchronous program has not to be modified since its interface to the input/output modules is unchanged. “Logical” handlers used in simulation and “physical” handlers are different implementations of the same classes.

## CONCLUSION

The synchronous paradigm was introduced in the mid-80’s and has been developed in the 90’s. It is now a credible approach to real-time reactive system programming.

Theoretical issues (semantics, validation) have been intensively studied. In order to transfer the synchronous technology from academic research to the industrial world, compilation techniques have been tremendously improved and languages made more user-friendly. Commercial supports for synchronous languages are now available. The European project called “Synchron” aims at developing a common platform for synchronous programming. The concept of execution machine, presented in this paper, is a contribution to this objective. The role of an execution machine for synchronous programs is to reduce the gap between the “ideal” synchronous world and the real-world. We have adopted a pragmatic approach that combines asynchronous and asynchronous programming. An execution machine is made of cooperating objects, is programmed in a widely used (asynchronous) language and relies on a standard real-time operating system. This is, however, transparent to the user. The user has

- To express the expected behavior in a synchronous formalism,
- To program the transformational part of the application in a sequential language like C or C++,
- And to configure the application (i.e., to give the relationships between physical and logical signals).

The architecture of an execution machine is modular. The building block is the reactive box. A reactive box is characterized by its inputs, its outputs, and its behavior. Predefined generic reactive boxes are available for input and output processing. The coordination of the reactive boxes is ensured by controllers automatically instantiated during the compilation.

Our execution machine have been developed for controller implementations. It could be applied, as well, to not real-time but highly reactive applications like Human-Machine Interfaces.

## ACKNOWLEDGMENTS

This research has been supported by the CENT, contract 94-1B-111.

## REFERENCES

Benveniste A., and Berry G., 1991, "The Synchronous Approach to Reactive and Real-Time Systems", *Proceedings of the IEEE*, 79(9):1270-1282, September 1991.

Berry G., 1997, "The Esterel v5 Language Primer", not yet published, available on the web, www.inria.fr/meije/esterel, INRIA, Sophia Antipolis, 1997

Bouali A., 1997, "XEVE: an Esterel Verification Environment", Technical Report, CMA-ENSMP, Sophia Antipolis.

Boufaïed H., 1998, "Machines d'exécution pour langages synchrones", PhD thesis, Université de Nice, November.

Boulanger F., 1993, "Intégration de modules synchrones dans une programmation par objets", PhD Thesis, Supélec/Université Paris-Sud (Orsay), December.

Douglass B.P., 1998, "Real-Time UML: Developing Efficient Objects for Embedded Systems", Object Technology Series, Addison-Wesley, Reading, Massachussetts.

Halbwachs N., 1993, "Synchronous Programming of Reactive Systems", Kluwer Academic Publishers, Amsterdam.

Rumbaugh J, Blaha M., Premerlani W., Eddy F., and Lorensen W., 1991, "Object-Oriented Modeling and Design", Prentice-Hall, Englewood Cliffs.

Selic B., Gullekson G.,, and Ward P., 1994, "Real-Time Object-Oriented Modeling", John Wiley Publishing.