# Objects and Synchronous Programming

**Charles ANDRÉ[1], Frédéric BOULANGER[2],**
**Marie-Agnès PÉRALDI[1,3], Jean-Paul RIGAULT[1,4],**
**Guy VIDAL-NAQUET[2]**

*[1] Laboratoire Informatique, Signaux, Systèmes (I3S) – Université de Nice Sophia Antipolis and CNRS – 41 bd Napoléon III – 06041 NICE Cedex, France.*

*[2] Laboratoire de Recherche en Informatique – Université Paris-Sud and SUPÉLEC – SUPÉLEC – Plateau de Moulon – 91192 GIF SUR YVETTE Cedex, France.*

*[3] Laboratoire d'Informatique Technique – École Polytechnique Fédérale de Lausanne – IN-Ecublens – CH-1015 LAUSANNE, Switzerland.*

*[4] École Supérieure en Sciences Informatiques (ESSI) – Université de Nice Sophia Antipolis – BP 145 – 06903 SOPHIA ANTIPOLIS Cedex, France.*

ABSTRACT: *Clear structure, support for abstraction, reuse, and easy evolution, these are the striking features of the object-oriented approach. Formal description of the reactive behavior, making it possible to prove logical correctness, is the essence of the synchronous paradigm. This paper proposes to combine these two approaches. An introductory example presents the notion of a* synchronous object*. Then, various issues related to objects and synchrony are addressed. Finally, we report on our progress in building a complete design and programming environment. Editors, compilers, simulators, and interfaces towards model checkers are integrated within this environment, which should contribute to better software quality in the field of real-time.*

KEY WORDS: *object-oriented, synchronous programming, reactive systems, reactive objects, real time systems, synchronous objects.*

## 1. Introduction

*The object* paradigm is well adapted to *complex system* programming**.** Its main advantage is the improvement in program architecture. This improvement is obtained first by a modular decomposition based on real entities used by the application (the

objects) and, second, by a clear separation between the (abstract) interface for *using* an object and the details about its *implementation* (internal design). Powerful mechanisms such as aggregation, inheritance, polymorphism allow to benefit from this decomposition. As a consequence, the system is endowed with extensibility and reusability, thus favoring evolution and maintenance.

So far, object-oriented programming has been applied mostly to *transformational systems* that is, systems that compute at their own (internal) speed. Opposite to these, there exist *reactive systems* [HAR 85], which have permanent interactions with their environment at a rate imposed by the environment itself. Most reactive systems are also *real time* systems (i.e., their reactions are temporally constrained).

*Synchronous programming* [BEN 91] was introduced as a solution to reactive programming requirements. Synchronous languages are founded on simple and rigorous mathematical semantics. Hence, one can build efficient and safe compilers. In addition, the *logical correctness* of programs can be proved formally. However, synchronous languages are mainly used to describe the control part of (reactive) systems. For other aspects of programming (data representation and transformation) they use classical (algorithmic) languages. Moreover, for an effective execution of the code generated by a synchronous language compiler, the user must provide an *execution machine* [AND 93].

Our objective is to combine the *synchronous* and *object-oriented* approaches in order to design *complex reactive systems.* The combination may impact several phases of the software life cycle: requirement analysis, preliminary and detailed design, implementation, testing, verification and validation...

This paper is organized as follows:

- We first recall the characteristics of the object-oriented approach.
- We then introduce an example: the modeling of the different operating modes of a cassette player/recorder. This example is composed of classical objects and of controllers modeled by synchronous objects.
- In part three, we present the *synchronous object* paradigm.
- The fourth part deals with *communication* between objects.
- And finally, we give an overview of a *software workbench* designed for programming with synchronous objects.

## 2. Characteristics of the Object-Oriented Approach

Using the object-oriented approach means considering the system to be developed as a collection of discrete cooperating objects. These objects are subsystem *abstractions* integrating data (*state*) and operations on these data (*methods*). Data are *encapsulated*, they cannot be accessed directly but only through a call to a method of the corresponding object. Hence, each object is responsible for its own data, that is for its own state and thus, for its own behavior. The list of methods for a given object constitutes the *interface to manipulate* (to use) the object; that is the only information that other (non-related) objects have to know about.

Object-oriented analysis and design methods (e.g., OMT) usually distinguish

three models [RUM 91]:

1. The *object model*, properly speaking, often termed also as the *class model*[1], which describes the system objects, their usage interface, and the (static) relationships that exist between objects;

2. The *dynamic model* (or *behavioral model*), which specifies the (dynamic) interactions between objects on the one hand, and the reactive behavior of individual objects on the other hand;

3. The *functional model,* which deals with data transformation and computation.

One of the major attractions of the object-oriented approach is its power of abstraction: it makes it possible to express *what* changes in the system without having to describe *how* these changes occur. Adding or removing objects is rather easy. Similarly, modifying an object locally is straightforward, provided that the modification disturbs neither the interface of the object nor the relationships of the object with other objects. Thus, incremental development (adding objects), reusability, and maintainability (local modifications) are among the main benefits of the approach.

## 3. An Example

Our aim is to model and simulate the operations carried out by a play-record cassette[2]. The device is composed of two decks (`Deck1` is play-only, and `Deck2` is play-record). Playback and recording operations offer sophisticated options involving sequences of simpler operations (e.g., continuous play from deck to deck, auto-reverse copy, blank skipping...).

We model this cassette recorder as a collection of *cooperating reactive objects*. Some objects are imposed by the application. So are sensors (push button, limit switches...) and actuators (motors, magnetic heads, LEDs...). A cassette can be conveniently abstracted by a class, which captures both the attributes of the tape and its access methods.

Contrary to approaches like SA/RT, which consider *controllers* apart from data flows, we assimilate controllers to objects. Of course, they are highly reactive objects. ROOM [SEL 94] adopts a similar point of view, where controllers are instances of “actors” (active objects).

Through this example, we will stress two major issues:

– class modeling (interest and limitations),

– dynamic behavior representation.

[1] Indeed, most object-oriented analysis and design methods are class-based; a class represents the properties common to a set of similar objects that is, it represents the *type* of an object. Then models use classes instead of individual objects.

[2] This example is inspired from an unpublished experiment performed by Frédéric BOUSSINOT with Reactive Objects [BOU 95].

In this kind of application, functional aspects are limited and will be ignored here.

### 3.1. *The Class Model*

Figure 1 presents a class model *à la* OMT for one of the application objects: the magnetic tape. In fact, there are two sorts of magnetic tapes: the tape properly speaking (`Passive_Tape`) and the tape once it has been loaded into the deck (`Tape_in_Deck`). The first object is rather passive: its only dynamic property is the

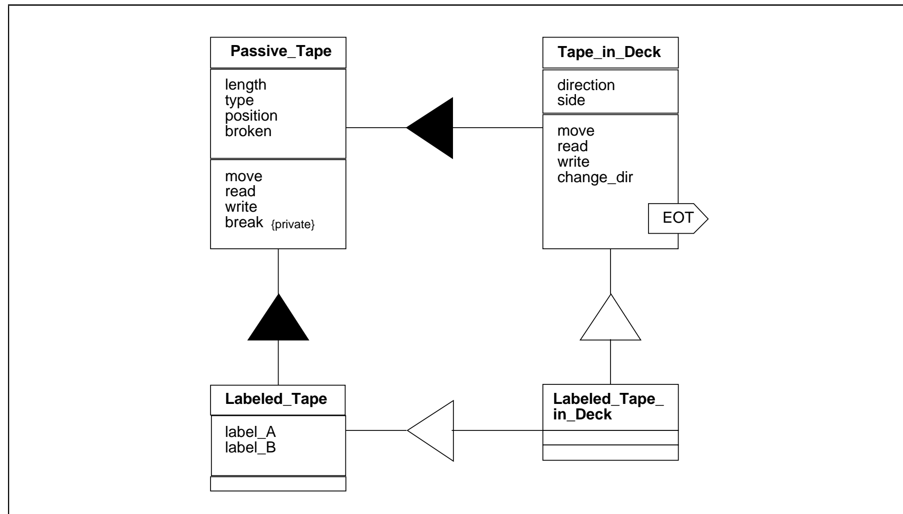



**Figure 1.** *Class (object) model of a magnetic tape.*

possibility to break. The second object, the tape inserted into a deck, enriches both the data structure and the behavior of the first: selection of the side and of the motion direction, handling of the "end of tape" (`eot`) event[3]. Inheritance (denoted by a triangle in OMT) is suited to represent such a specialization. It is then possible to derive other kinds of tape, like the ones which have a label (`Labeled_Tape`) or even the tapes that spontaneously vanish into smoke (the "Mission Impossible" tapes!). As indicated on figure 1, multiple inheritance makes it possible to represent these new tapes once inserted into the deck[4].

[3] This "spontaneous" event emission is not directly representable in an OMT class model. Thus we have enriched the OMT notation, as can be seen on figure 1.

[4] Note the use of the OMT "black triangles" representing *virtual* multiple inheritance as in C++: there will be only one copy of the base class (here `Passive_Tape`) in the derived classes, whatever the number of the inheritance paths happens to be.

When it comes to representing the deck itself, the situation gets a little trickier. Of course, we wish to consider a play-record deck as a specialization (a derivative) of a play-only deck. The components of a deck (controller, motor(s), head(s)) may be represented with an aggregation as in figure 2. Note that the association between the

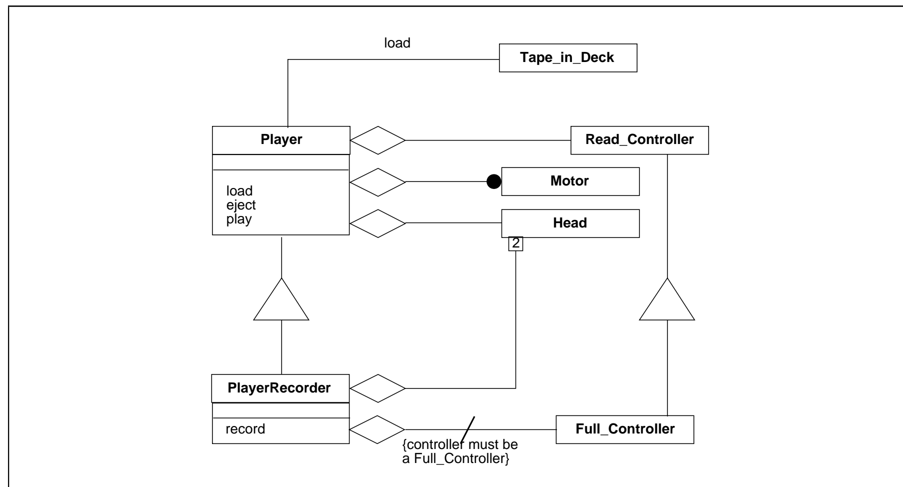



**Figure 2.** *Object model of the deck using aggregation.*

deck and the tape is *not* an aggregation, but a simple (abstract) association (`load`). Indeed the tape and the deck are perfectly independent objects; each of them may live on its own.

However, there is a problem with the model in figure 2. If a simple controller (`Read_Controller`) is sufficient for a read-only deck, the full deck must accommodate a more complete controller (`Full_Controller`). This situation is not directly representable in OMT: in figure 2, we represented it with a *constraint* on a *derived* aggregation. Another possible model is to derive the deck from the controller, as in figure 3. This may appear somewhat illogical, since a deck *is not* a (specialization of a) controller. Hence, we use *private* inheritance (that we represent by a small bar crossing the association). One can note the elegant symmetry of the scheme: multiple inheritance makes it possible to correctly represent the structure of the play-record deck compared to the play-only one.

Here, we meet a classical situation (and a huge matter of discussion) in the object-oriented approach: inheritance may be used both with the semantics of *subtyping* or as a simple means to *share implementation* (code). The discussion about these issues is beyond the scope of this paper.

To complete the model, the player-recorder itself is merely the aggregation of two decks: `Deck1` is an instance of class `Player`, whereas `Deck2` is an instance of `PlayerRecorder`.

The OMT-like models do not show precisely the exchange of messages and

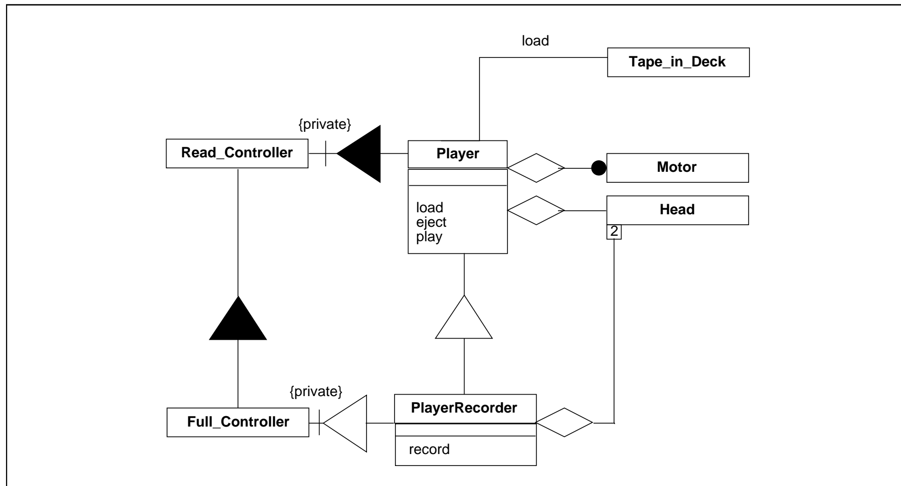



**Figure 3.** *Object model of the deck using (multiple) inheritance.*

events between objects[5]. We thus introduced a new type of diagram to make these dynamic links explicit (figure 4). On these diagrams boxes represent objects. Input ports ( ) and output ports ( ) represent methods. Connections are directed from an input port to an output one; they represent point-to-point communications. The aspects related to communications will be described in section 5. Note that in order to keep the diagram legible, several signals may be grouped together to form "buses". This is the case for output signals from object `keyboard` (`play...record`) which are also found as input to `Controller`.

### 3.2. *The Dynamic Model*

The behavior of each object is expressed by the "dynamic model". Usually, graphical representations are used (State graphs, STATECHARTS in OMT, ROOM-charts in ROOM...). We adopt the SYNCCHARTS [AND 96-1], a new model derived from STATECHARTS [HAR 85] and Argos [MAR 91]. The SYNCCHARTS are state-based models suited to reactive behavior modeling. They support hierarchy of states, orthogonality, information broadcasting and preemptions. On the last point, SYNC-CHARTS definitely surpass STATECHARTS: they make a clear distinction between

[5] This is somewhat unfair. Classical OMT dynamic model includes the description of the exchange of messages through the use of event-trace diagrams, but this description is not global to the system, it is scenario-based. Also forthcoming versions of OMT may include a system-wide description of messages and events exchange.

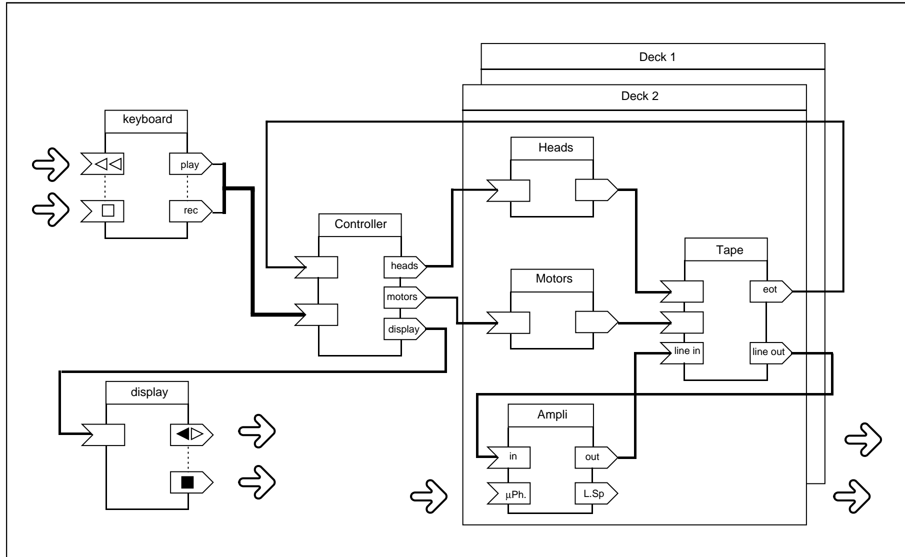



**Figure 4.** *Connections between the application objects.*

various kinds of preemptions (abortion or suspension, weak or strong, see [BER 92]).

A detailed presentation of the SYNCCHARTS is beyond the scope of this paper. We shall only comment two examples of synccharts. They are sufficient to introduce several key features of the model.

The syncchart in figure 5 expresses the behavior of the basic play-record controller. Two macro states, named `Normal_M_Ctrl` and `Basic_Ctrl`, have concurrent evolutions. The dotted line in figure 5 separates the two orthogonal components. The behavior of the motor control (`Normal_M_Ctrl` macrostate) is given in figure 6.

An arc between two states stands for abortion: as soon as the event associated with the arc occurs, the source state is exited and the target state is entered. An ordinary arc denotes a *weak abortion*: the preempted macrostate may execute its "last wishes" before being killed. An arc with a small circle at its origin denotes a *strong abortion*: the preempted state is not allowed to execute any reaction. Note that both weak and strong abortions are instantaneous and synchronous with the occurrence of the triggering event.

The behavior of a syncchart is fully *deterministic*. If several events can abort a macrostate, a priority order must be given (by small integer numbers next to the transition). In `Basic_Ctrl`, event `rec` is given the highest priority, event `rewind` the lowest.

A synchronous model, like SYNCCHARTS, relies on the notion of an *instant*. There is a clear distinction between strictly future and present-or-future occurrences. The symbol "`#`", which prefixes some events, stands for the latter case (e.g., `#rec` in the `Basic_Ctrl_State` means that if `rec` is present when entering this macrostate,

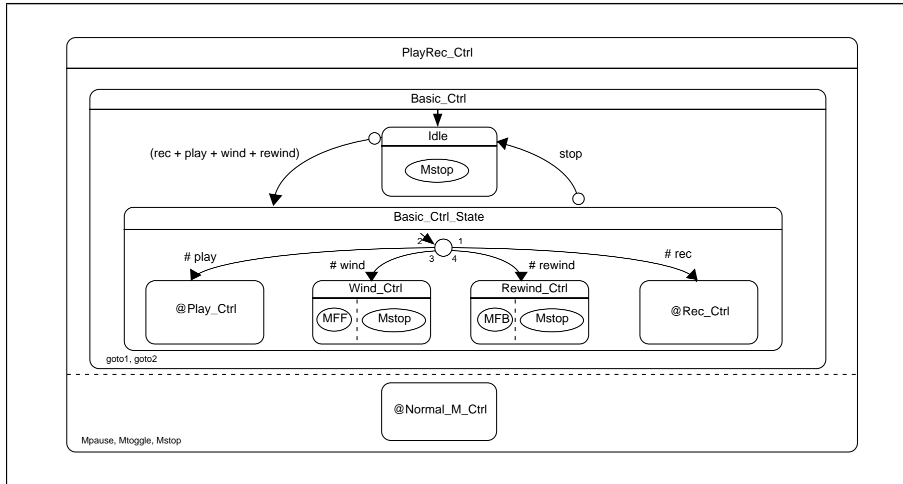



**Figure 5.** *Control of the play-record deck.*

then the component state `Rec_Ctrl` must be entered immediately). The absence of prefix # requires that only a strictly future occurrence be considered.

The evolution of a syncchart is controlled by triggering events (*event-driven* systems). Conversely, a syncchart can act upon its environment (output events). In SYNCCHARTS as in Moore machines, *outputs* can be associated with states: a signal identifier written in an ellipse (a state) indicates that this signal is emitted whenever the state is active. For example, when the state `Wind_Ctrl` is active, `MFF` (Motor Fast Forward) is sustained, so that a tape can be wound. A syncchart can also emit signals while a transition is taken, as in a Mealy machine. This possibility is not shown in the given examples.

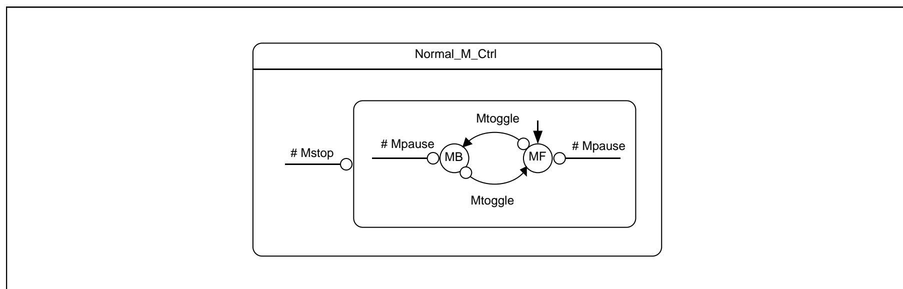



**Figure 6.** *Motor control.*

An emitted signal may be hidden from the environment and used for internal pur-

poses (synchronization). Signals Mpause, Mstop, Mtoggle are *local* to the macro state PlayRec_Ctrl. When in the Idle state, Basic_Ctrl emits the local signal Mstop that influences the behavior of the motor controller. Since a signal may convey a value, local (valued) signals are also used for internal communication.

In figure 6, special arcs are drawn: they have no source state, they target a state, and they end by a small circle head. We call them *suspension* arcs. When the event associated with a suspension arc is present, the target state is "frozen", just as if time did not flow with respect to the internal evolutions of the state (e.g., when Mstop is present, the nested state graph can neither change state, nor emit signals). A suspension can be immediate (e.g. Mstop in Normal_M_Ctrl), or strictly future (e.g. Mtoggle).

To sum up, the SYNCCHARTS are suitable for modeling sophisticated reactive behaviors. They promote preemption as a first class concept, so that normal as well as abnormal behaviors can be easily specified. Moreover, SYNCCHARTS rely on a formal semantics [AND 96-1] and they can be compiled into equivalent ESTEREL programs (program 1 is an ESTEREL program equivalent to the Normal_M_Ctrl

```
module Normal_M_Ctrl:
  input Mstop, Mpause, Mtoggle;
  output MF, MB;
  suspend
    loop
      do
        suspend
          sustain MF
        when immediate Mpause
      watching Mtoggle;
      do
        suspend
          sustain MB
        when immediate Mpause
      watching Mtoggle
    end loop
  when immediate Mstop
end module
```

**Program 1.** *Example of translation of a SYNCCHART to ESTEREL.*

syncchart of figure 6). Thus, SYNCCHARTS may be seen as a graphical variant of the ESTEREL language.

## 4. Synchronous Objects

As we saw in section 2, the object-oriented approach is a powerful abstraction

tool. It allows us to say *what* changes without necessarily specifying *how* it changes. The synchronous approach gives a similar level of abstraction for the specification of the behavior (*when* modules interact).

In the synchronous model, signals are abstractions of communications. They are instantaneously broadcast throughout the system. Thus they constitute the medium through which concurrent subsystems communicate. Arbitrarily complex interactions may be represented easily. The synchronous composition of the subsystems can be defined rigorously and has interesting properties such as the determinism of the parallel composition.

As the tape deck example shows, an application may use both transformational and synchronous reactive entities. The object-oriented approach permits the integration of both kinds of elements *by encapsulating the synchronous code into objects*. Synchronous modules are well suited to this encapsulation since they communicate only through their signals and thus can be considered as black boxes.

So, the first step is to turn a synchronous module into a class that we call a *synchronous class*. Instances of this class have the same reactive behavior as the original synchronous module. All the synchronous classes derive from an abstract class[6] named `Synchronous` that defines the basic protocol of any synchronous object. With this basic protocol we can define a synchronous execution machine that p*reserves the synchronous semantics of the modules* in the object-oriented language.

This allows synchronous objects to be interconnected and to communicate synchronously. The behavior of such a network of synchronous objects is identical to the behavior of a synchronous program built with the corresponding modules. Hence, synchronous class libraries can be used to build programs without needing a synchronous language compiler, and so, without needing the source code of the modules. Of course, this separate compilation has limitations: the directed interconnection graph of the objects must be acyclic, except when a non-instantaneous dependency is explicitly stated, such as for a delay. Indeed, checking the causality of communication loops requires the knowledge of the semantics of the modules—we have to "open the black box".

## 5. Communication between Objects

Regular objects communicate through messages. Synchronous objects are objects but they have a synchronous reactive part which communicates only through signals. So we have to define new protocols to transport signals within messages between synchronous objects, and to make signals accessible through messages for the communication between synchronous objects and regular objects.

---

[6] An abstract class is a class with methods the implementation of which is deferred to derived classes. Such a class cannot be instantiated. It defines an interface subset shared by all its subclasses.

### 5.1. *Synchronous Communication and Clocks*

Synchronous communication between objects implies that the objects which communicate together *share the same notion of an instant*. The execution machine implements this notion with the `Clock` class. Several instances of this class (that is several “clocks”) may be used in the same program in order to process events that occur at different time scales. Each clock determines a scheduling of the objects it manages according to their connections and makes them react in an order that satisfies their dependencies. All the objects managed by the same clock constitute the *clock domain*.

Contrary to ESTEREL in which synchronous communication is expressed by “name sharing” (all the signals that share the same name have the same value), synchronous communication between synchronous objects is a point to point communication, and several input signals can be connected to a same output signal. The reason is the following: when two objects are instantiated from the same class, their signals bear the same name although they correspond to independent entities; hence, they must *not* be connected. Therefore, the name sharing scheme cannot be used. Thus, input signals have a *connection method* that takes an output signal as argument and allows the explicit declaration of the connections between synchronous objects.

This point to point connection scheme makes it possible to use the type checking mechanisms of C++ [STR 86] to ensure that an input signal can only be connected to an output signal of the same type.

### 5.2. *Asynchronous Communication*

Synchronous objects must be able to communicate with regular objects or with synchronous objects belonging to another clock domain. So we need a mechanism for asynchronous communication. Communications leaving a clock domain are not really an issue since one can read the value of the output signals of a synchronous object without disturbing it. On the other hand, communications entering a clock domain require *a synchronization phase*: we must build a synchronous event from one or several asynchronous events.

Turning asynchronous events into synchronous events is not a trivial task. It is carried out by *interface objects* [BOU 93]. These special synchronous objects are sensitive to asynchronous events that activate some of their methods, and they produce synchronous events for the other objects related to the same clock. Such objects may be written in C++ or in a synchronous language. In the latter case, nothing should be assumed about the simultaneity of the inputs since this notion has no meaning for asynchronous events.

### 5.3. *Dynamicity*

We have seen that synchronous objects can be interconnected and can communicate synchronously under the control of a clock. For such a group of objects (a clock domain), it is always possible to write an equivalent program in a synchronous lan-

guage, and to use proof tools to check some properties. The object-oriented approach brings the reusability and facilitates the integration. But once a synchronous module has been translated into a C++ class, nothing forbids to dynamically create or destroy instances of this class during the execution of a program. For this, we must be able to dynamically change the connection graph, disconnecting deleted objects and connecting newly created ones.

We then reach a higher order of description since an event not only triggers the reaction of an object but may trigger a "clock reaction"—a modification of the equivalent synchronous program.

The main drawback of dynamicity is the lack of a synchronous program equivalent to a dynamic clock. This forbids the use of classical proof tools to check properties on the behavior of this clock. But dynamicity allows us to express synchronous behaviors that cannot be expressed in a synchronous language while still benefiting from the synchronous approach for the development of the components of the system and for the control of dynamicity.

Dynamicity may also be used in systems where the number of objects to manage is not known beforehand. Examples of use of dynamicity are presented in the thesis of Frédéric BOULANGER [BOU 93].

## 6. Software Workbench

To support our approach to object-oriented synchronous programming, we are developing a platform that integrates several software applications dedicated to synchronous programming (editors, compilers, simulators, code generators, and proof systems). Figure. 7 presents the main components of our platform described in detail in another paper [AND 96-2].

### 6.1. *Editors*

The description of a synchronous object can be either textual or graphical. Classical textual editors are used for C++, ESTEREL, MDLC [BOU 93] programs. On the other hand, we have developed our own graphical editors for GRAFCET [AND 94] and SYNCCHARTS.

### 6.2. *Compilers*

*Synchronous modules* can be specified by synchronous languages (LUSTRE or ESTEREL) as well as by synchronous formalisms (GRAFCET or SYNCCHARTS). Compiling a synchronous module is a complex task, which involves various compilers (see Figure. 7). LUSTRE, ESTEREL, GRAFCET, SYNCCHARTS compilers yield an intermediate common code known as the OC code. OCC++ [BOU 93], another compiler, then turns the OC file to a C++ synchronous class. The reactive synchronous code is embedded within the synchronous class. It is accessible through the standard interface that we have developed for synchronous objects.

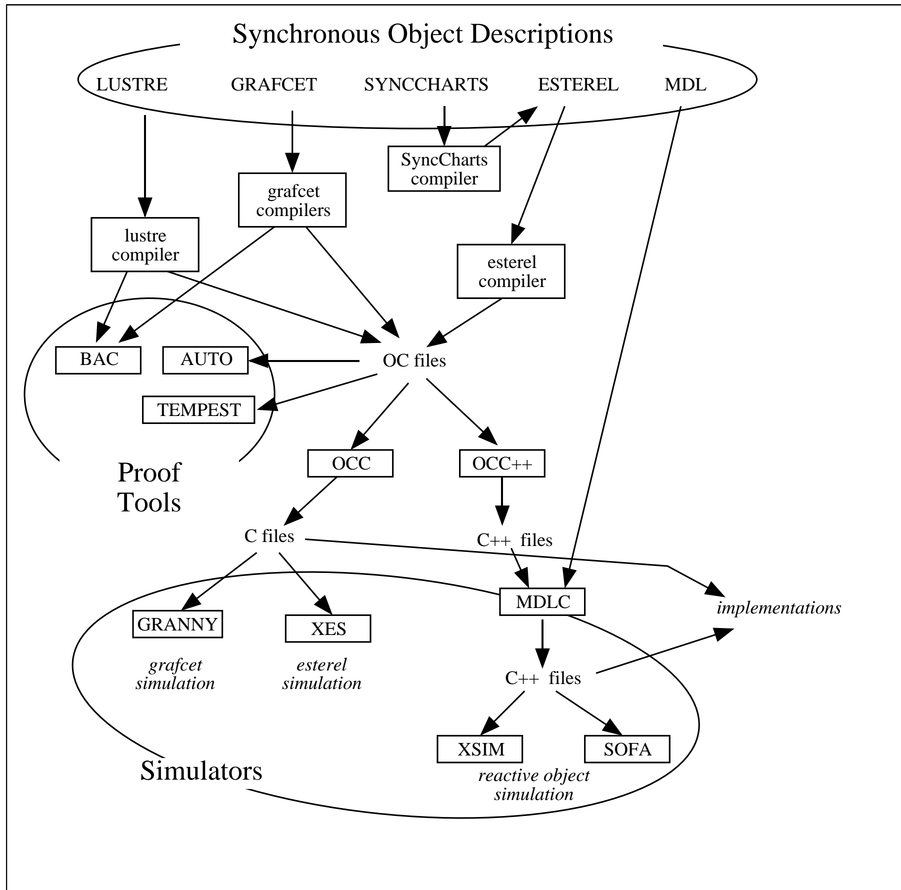



**Figure 7.** *The Synchronous Object Platform.*

OCC++ can also generate a module description file (interface, source file...).

MDLC (Module Description Language Compiler) supports the creation of new classes by composition and derivation. The Module Description Language (MDL) allows us to define new modules by static connections between and derivations from existing modules. A derived class inherits the behavior of its super-class and may have additional input or output signals. MDLC produces a C++ synchronous class from an MDL description.

A library of classes, called LIBSYNC, provides the execution environment necessary for synchronous classes generated by OCC++ or MDLC.

### 6.3. *Simulators*

At each stage of the development, reactive behaviors can be tested. Simulators, automatically generated from synchronous objects, make it easy to analyze reactions to stimuli. Their graphical interfaces developed with Tcl-Tk [OUS 94] offer friendly animation.

A deeper insight in the program dynamics is brought about by interactive simulations with visualization of the execution at the source level (backward-mapping to source). It is all the more useful since representations describe concurrent executions. Such simulations can be performed on ESTEREL programs (XES simulator) and GRAFCET (GRANNY simulator) [AND 96-3].

### 6.4. *Proof Tools*

A synchronous program written in LUSTRE, ESTEREL, GRAFCET, SYNCCHARTS can be compiled into a finite automaton or a system of equations. On these representations, proof tools can be applied. Automata are analyzed by AUTO [DES 89], MEC [ARN 94], or TEMPEST [JAG 95]. Boolean equation systems are by-products of Boolean LUSTRE programs and GRAFCET. The safety properties of such systems are efficiently tested by a specific tool: the BAC (Boolean Automaton Checker) model checker [HAL 94] [AND 96-3].

## 7. Conclusion

Combining the object-oriented approach and reactive system modeling is an emerging domain of research. A few works have started to explore this promising field (ROOM method [SEL 94], O-Charts [HAR 94], Reactive Objects [BOU 95]). This paper aims at presenting a new pioneering contribution.

The first advantage of the synchronous object approach is to make it easier to model complex reactive systems. The object view relies on well-known analysis and design methods (e.g., industrial standards like OMT) and the behavior description benefits from the rigorous semantics of the synchronous model.

In order to gain the most from this method, it is essential to have a complete development platform. At present, this objective is partially fulfilled. In particular, tools for code generation are available, the target language being another industrial standard (C++).

This environment could not be complete without simulators and automatic links to formal proof tools. For this purpose, we provide interfaces with several model checkers.

We are convinced that the synchronous object approach will be the basis for numerous further developments and will contribute to improving the software quality indispensable for critical real time systems.

## Bibliography

[AND 93] ANDRÉ C. and PÉRALDI M.A., "Effective implementation of ESTEREL programs", 5th EUROMICRO Workshop on Real-Time Systems, Oulu (Finland), June 1993.

[AND 94] ANDRÉ C. and GAFFÉ D., "Coopération GRAFCET/ESTEREL", Colloque AGI'94, Poitiers (France), June 1994.

[AND 96-1] ANDRÉ C., "Representation and Analysis of Reactive Behaviors: A Synchronous Approach", Symposium on Discrete Events and Manufacturing Systems, CESA'96 IMACS, July 9-12, 1996, Lille (France), p. 19-29.

[AND 96-2] ANDRÉ C., BOUFAÏED H., GAFFÉ D., and MARMORAT J.P., "Environnement pour la programmation synchrone des systèmes réactifs", Actes des Conférences RTS&ES'96, 10-12 janvier 1996, Paris (France), p. 27-41.

[AND 96-3] . ANDRÉ C. and GAFFÉ D., "Proving Properties of GRAFCET with Synchronous Tools", Symposium on Discrete Events and Manufacturing Systems, CESA'96 IMACS, July 9-12, 1996, Lille (France), p. 777-782.

[ARN 94] ARNOLD A., BEGAY D., and CRUBILLÉ P., "Construction and Analysis of Transition Systems with MEC", volume 3 , AMAST Series in Computing, World Scientific, Singapore, 1994.

[BEN 91] BENVENISTE A. and BERRY G., "The synchronous approach to reactive and real-time systems", *Proceeding of the IEEE*, 79(9):1270–1282, September 1991.

[BER 92] BERRY G., "Preemption in concurrent systems", *Proc FSTTCS*, Lecture notes in Computer Science, 761:72–93, 1992.

[BOU 93] BOULANGER F., Intégration de Modules Synchrones dans la Programmation par Objets, Doctoral thesis, SUPÉLEC / Université de Paris-sud, December 1993.

[BOU 94] BOULANGER F., DELEBECQUE H., and G. VIDAL-NAQUET G., "Intégration de Modules Synchrones dans un Cycle de Développement par Objets", *Actes des Conférences RTS'94*, 11–14 janvier 1994, Paris (France), p. 245–260.

[BOU 95] BOUSSINOT F., DOUMENC G., and STEFANI J.B., "Reactive Objects", Research Report 26-64, INRIA, Sophia-Antipolis (France), October 1995.

[DES 89] DE SIMONE R. and VERGAMINI D., "Aboard AUTO", Technical Report 111, INRIA, October 1989.

[HAL 94] HALBWACHS N., . "BAC: A boolean automaton checker", Technical report, VERIMAG, Montbonnot (France), February 1994.

[HAR 85] HAREL D. and PNUELI A., "On the development of reactive systems in logic and models of concurrent systems", *NATO ASI Series*, K.R Apt Ed., Springer-Verlag, 13:477–498, 1985.

[HAR 94] HAREL D.and GERY E., "Executable Objects Modeling with Statecharts", Technical Report CS94-20, Weizmann Institute of Science, September 1994 (rev. August 1995).

[JAG 95] JAGADEESAN L.J., PUCHOL C., and von OLNHAUSEN J., "Safety Property Verification of ESTEREL Programs and Applications to Telecommunication Software", Conference on Computer Aided Verification (CAV'95), Liège (Belgium), July 1995.

[MAR 91] MARANINCHI F., "The ARGOS language: Graphical Representation of Automata and Description of Reactive Systems", Proc. IEEE Intl. Conf. on Visual Languages, 1991, Kobe (Japan)..

[OUS 94] OUSTERHOUT J.K.., *Tcl and the Tk Toolkit*, Professional Computing Series. Addison-Wesley, 1994.

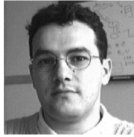
[RUM 91] RUMBAUGH J., BLAHA M., PREMERLANI W., EDDY F., and LORENSEN W., *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, 1991.

[SEL 94] SELIC B., GULLEKSON G., and WARD P.T., *Real-Time Object-Oriented Modeling*, John Wiley Publ., 1994.

[STR 86] STROUSTRUP B., *The C++ Programming Language*, 2nd Edition, Addison-Wesley Publishing Company, 1991.

## About the Authors

**Charles J. ANDRÉ** received his "Doctorat d'État" in Electrical Engineering from the University of Nice (France) in 1981. Since 1983, he has been a professor of Automatic Control and Computer Science, and since 1989, the Head of the Electrical Engineering Department of the University of Nice. He also heads the SPORTS research team in the I3S laboratory. The topic of his thesis was the theory of Petri nets. His application domain has evolved to real-time systems and especially to their synchronous programming.



**Frédéric BOULANGER** received his Engineering degree from SUPÉLEC (1989) and his doctorate degree from the University of Paris XI Orsay (1993). He is presently a member of the scientific staff in the Computer Science Department of the "École Supérieure d'Électricité" in Gif-sur-Yvette, and a member of the Computer Science Laboratory at the University of Paris XI Orsay. His scientific interests include the use of the object-oriented approach for the integration of several formalisms (among which the synchronous approach), and the use of the PTOLEMY environment together with the synchronous model.

**Marie-Agnès PÉRALDI** received her doctoral degree from the University of Nice Sophia Antipolis (France) in 1993. The topic of her thesis was the synchronous approach to the design of real-time systems. From December 1994 to September 1995, she was on a post-doctoral position at the Swiss Federal Institute of Technology (Lausanne), in the Industrial Engineering Laboratory (LIT). On September 1995, she joined the I3S laboratory where she is currently "Maître de conférences" in the SPORTS team. Her current research interests are in the field of real-time local area networks and distributed control systems.

**Jean-Paul RIGAULT** is an alumnus of the "École des Mines de Paris", from which he received his engineering degree in 1972. After holding several teaching and research positions at the "École des Mines", he is currently a professor at the University of Nice Sophia Antipolis and director of ESSI, an engineering school in Computer Science. He was among the very first members of the team who designed and developed the ESTEREL synchronous language. His interests are in the field of object-oriented software engineering, the C++ language, and real-time reactive systems.

**Guy VIDAL-NAQUET** received his "Doctorat d'État" in 1981 from the University of Paris VI. He is currently a professor at the University of Paris XI and at the "École Supérieure d'Électricité". He is a member of the Computer Science Laboratory at the University of Paris XI Orsay. His scientific interests include real-time and distributed systems and formal methods for specification and analysis.