An Object Execution Model for Reactive Modules with a C++ Implementation

Frédéric Boulanger, Guy Vidal-Naquet

École Supérieure d'Électricté and LRI - Université de Paris-Sud Orsay

Abstract

We present an execution model that allows reactive modules that follow the synchronous paradigm (all computations are ended before a new flow of data arrive) to be represented by objects that communicate synchronously or asynchronously. This model is intended to be used in the object-oriented design of complex applications that mix reactive and transformational parts. The reactive parts may be developed in any synchronous language, or even implemented in hardware. Our model allows synchronous communication between separately compiled parts, when there is no dependency loop. The object model allows dynamic creation and destruction of objects. This is supported by our model, along with dynamic changes in the synchronous communication network. This leads to the notion of reaction of a clock (a set of synchronous objects that communicate synchronously). A clock reacts to signals at instant t by producing outputs at t and determining what is going to be its state at t+1 (with a new set of objects and a new communication network). The nature of asynchronous communications is intentionally not imposed in order to be adapted to each application domain. This model is implemented by two tools and a C++class library. The first tool, *occ++*, translates oc code (automaton description) into a C++class. The second, *MDLC*, compiles descriptions of composite modules written in the *Module Description Language*. A library provides the execution machine for synchronous objects. The implementation works on any Unix machine, and has been ported to the real-time kernel VxWorks. This paper does not provide a methodology for dealing with reactive components, but presents tools for the application of Object Oriented methodologies, or object based software platforms, to systems with reactive components. Our system was integrated in the PTOLEMY environment which is used for the simulation and the prototyping of heterogeneous systems.

1 Introduction and Motivation.

Complex systems usually have components of radically different natures:
• Reactive components, that deal with the environment, receive input signals, produce output signals, and maintain a coherence relation between inputs and outputs. These components must often have real time and safety properties, and communicate with each other in a synchronous or asynchronous way.

• Transformational components, that work without interaction with other components during their computation.

Examples of such systems can be found in the fields of supervision, transportation, telecommunication, autonomous robots, self-tuning control systems.

During the last decade, the *synchronous approach* has been developed for the programming of reactive modules [BEN&BER-91]. It relies on the *synchronous hypothesis*: reaction is instantaneous (in reality sufficiently fast). This leads to a model and to programming languages that treat time in a rigorous way, and where parallelism and determinism are not antinomous.
Informally, the execution of a program written in a synchronous language is like the functioning of an automaton with outputs: outputs are instantaneously determined, and the next state is computed.

An advantage of the synchronous approach is that proofs can be obtained from the source code, after an automatic translation into mathematical language [BEN&BER-91],and [HAL-93]).

There are actually two drawbacks to using the synchronous approach: 1) the code is monolithic, i.e. it is not possible to compile parts of a system separately and assemble them and; 2) parallelization is therefore difficult.

The work we present here is aimed at smoothing the integration of these modules in complex systems, by allowing them to be viewed as objects. This will allow developers to take advantage of the features of object oriented programming techniques, like class libraries, inheritance, methodology for structuration, dynamic creation of objects. We give two examples where Object Oriented technologies could be applied with reactive components:
• *Structuration* enables us to name a whole set of components with a single name. These components may be reactive or transformational. In Fig.1 the

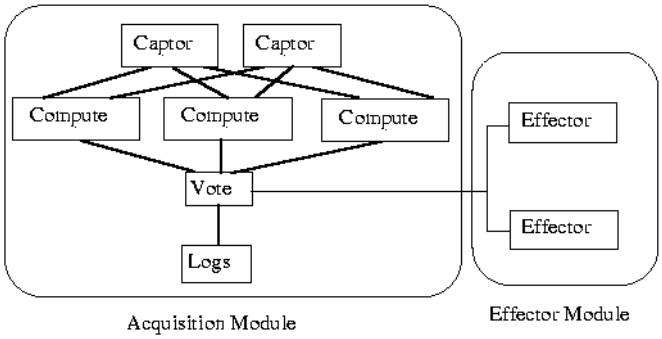

Figure 1: Structuration of Objects

acquisition module has two captors , which send their data in a synchronous way to three computation modules, whose results go through a voting module. The result of the vote is logged (asynchronously) on a data base, and is also transmitted to two effectors. One would like to see the acquisition module and the effector module as whole entities, i.e. objects.

• *Dynamicity* which means in this context that objects can be created at run-time, and connections between objects can be changed. In the following example of a controller, an object is in charge of adjusting a valve. If the pressure and/or the temperature becomes too high, a "crisis" module is substituted, and controls the valve and a safety valve.

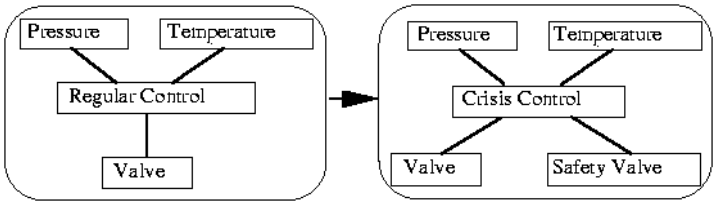

Figure 2: Dynamicity of Objects

In this scheme, the object *Regular Control* is destroyed, and the objects *Crisis Control* and *Safety Valve* are created. *Pressure* and *Temperature* are connected to *Crisis Control*. In another possible scheme, *Crisis Control* and *Safety Valve* would have already been present, only the connections would have changed.

2 Execution model.

We now present an execution model for synchronous modules. This model describes our basic objects: synchronous classes, instances of a class, clocks and basic operations: creation, and destruction of an object. This model will be used as a reference for implementation, i.e. every constraint and property given in this section will have to be satisfied by any implementation. Formal definitions can be found in [BO&VI-94]

2.1 Synchronous Classes.

A synchronous class $\mathcal{C}$ is defined by :1) a set of *input and output signals*. Signals have typed values and can be present or absent; 2) a set of *states*, with an initial state; 3) a *reaction function*: at instant t, given a state and the values of input signals, the reaction function computes the presence and values of output signals at instant *t*, and the state at instant *t+1*, (in accordance to the synchronous model).

We distinguish a special subset of outputs called input-independent, which depends only on the state of the object, and not on the values of the input signals. The presence, and the value of an input-independent signal can be evaluated independently of the inputs.

2.2 Synchronous Objects and Clocks

An object is an instantiation of a class $\mathcal{C}$, and is identified within the class by a name η, it is unambiguously designated by the identifier $\eta \bullet \mathcal{C}$. An object reacts in the way described by the reaction function of its class.

A clock represents the evolution of a set of (synchronous) objects, it is determined by a sequence of instants. At a given instant a clock is characterized by: 1) a set of input signals, and a set of output signals; 2) the set of objects on it, and the state they are in; 3) the connection between the input and output signals of the objects on the clock; 4) the values and the presence of the different signals.

A clock is not related to a physical measure of time, but with a sequence of events. A clock has to satisfy the following constraints which are not properties, in the sense that they cannot be deduced from the model itself, but are mandatory for any implementation of the model.

1. An object belongs to only one clock.
2. The instants when an object belongs to a clock form an interval, i.e. an object cannot disappear and reappear
3. At a given instant, two connected signals are of the same type, have the same value, and are both present or absent
4. For a given object $\eta \bullet \mathcal{C}$ in a state σ, the presence and values of output signal *at instant t*, and *the state at instant t+1* are given by the reaction function of the class $\mathcal{C}$.
5. Let the *precedence relation* $\prec$ on objects, be defined by $\mathcal{O}_1 \prec \mathcal{O}_2$ when there exits a non input-independent output signal of $\mathcal{O}_1$ connected to an input signal of $\mathcal{O}_2$. The transitive closure of $\prec$ is a partial order.

Condition 4 means that for an object $\mathcal{O}$ the output signals at instant t and the state of $\mathcal{O}$ at instant t+1 are computed instantaneously by the reaction function.

The following figure gives an example of a clock at a given instant.

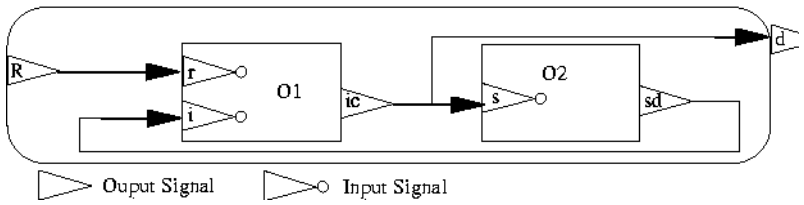

Figure 3: Connected Objects on a Clock

Note that `R` is an output signal (produced by the outside world), and `d` an input signal (that is used by the outside world)

Condition 5 expresses that connections like in Fig. 3 can happen, only if `ic` or `sd` is input independent.

One can show that if conditions 3,4,5 hold, then at an instant n, given the states of the objects, and the values of the input-signals of the clock, one can

determine unambiguously all output signals at instant n, and states at instant n+1.

If in Fig.3 object $\prime_1$ is a counter, which increments a number at each instant of a clock, and the signal r resets the counter to 0. The object $\mathcal{O}_2$ breaks the causality loop if it introduces a delay: at instant n, it stores the value i of the input signal in the state si for instant n+1. The module $\mathcal{O}_1$ produces as output 0 when the signal restarts is present, and n+1,where n is the value of input signal i, otherwise.

2.3 Dynamicity.

There are two types of dynamicity in our model:
Dynamicity of connections, and dynamicity of processes.
Since we view objects as black boxes, without analyzing their semantics, we can only describe dynamicity in the model, without saying why it occurs.

The destruction of a synchronous object $\mathcal{O}$ at instant n, has the following consequence for object $\mathcal{O}'$ which depend on $\mathcal{O}$:
If the input signal s of $\mathcal{O}'$ is connected to an output signal of $\mathcal{O}$, at instant n+1, s is not present, but has the same value it had at instant n, unless it is connected to an output signal of an object that exists at instant n+1 (the signal keeps the same value, but is absent, this is compatible with the approach in synchronous programming, that says that a signal keeps the same value unless it is explicitly modified).

Dynamicity of objects is needed when the number of objects in the system is not known, for example in traffic control, but can be used in other cases (see example of Fig. 2)

The instantiation of synchronous object $\mathcal{O}$ of a class $\mathcal{C}$ at instant n, takes effect at instant n+1.

A connection is requested at instant n, and effective at instant n+1. Dynamicity of connections has a very low cost, and may be useful to change the configuration of the system according to some conditions. Dynamicity of objects is a little more time consuming, but is sometime necessary. For example, some systems do not specify an upper limit on the number of potential managed objects. There is of course a limit, imposed by the size of the available memory, and by the time constraints. The allocation of memory has to be decided at run-time, because a static allocation would prevent an optimal use of memory. Even when the maximum number of objects is known, one may want to reuse memory when some objects are not used.

2.4 Execution machine for clocks.

For a clock, the *execution machine* [AND&PER - 93] at instant n will do the following: 1) Process requests of creation and destruction of objects; 2) Process requests of (dis)connection; 3) verify property 5; 4) schedules an order of reaction for the modules that satisfies dependency relations (there exists one, from the above result); 5) evaluates the state of each object existing on the clock;

5) make each object react: Determine the output signal at instant n, and the state n+1, which can be done due to property 5.

3 Composite synchronous classes.

In this section we describe informally two ways for building new classes.

Composition: It is possible to build new classes whose instances are composed of interconnected instances of existing classes. The formal definitions are not given here but can be deduced easily from the definitions of the input, output, state function and evaluation function of a clock.
For example, from the classes of $\mathcal{O}_1$ and $\mathcal{O}_2$, we could make a class COUNTER, as described in Fig. 3.

Inheritance: We keep the notion that when a class $\mathcal{A}$ inherits from a class $\mathcal{B}$, the behavior of $\mathcal{A}$ is a refinement of the behavior of $\mathcal{B}$. Since we consider here a synchronous class as a black box, we cannot state properties of behavior, but structural properties:

A synchronous class $\mathcal{B}$ inherits from a synchronous class $\mathcal{A}$ implies that the interface of class $\mathcal{B}$ contains the interface of class $\mathcal{A}$.
A way to derive a class $\mathcal{B}$ from a class $\mathcal{A}$ is to filter the inputs and the outputs of an instance of $\mathcal{A}$ with two additional objects.
A constraint that is not imposed by the model, but that seems useful with respect to the general discussion on inheritance, is that if all the signals not in $\mathcal{A}$ are absent, then the reaction function of class $\mathcal{B}$ is the same as the reaction of class $\mathcal{A}$. A typical example of inheritance is the construction of an ABS brake from a usual brake.

4 Realizations.

Synchronous classes are described with synchronous languages, [HAL-93], and compiled into an intermediate language OC. This allows us to use all synchronous languages that can produce OC code, without the task of compiling synchronous languages. An OC program is basically a finite automata with outputs. The basic components of our system are: 1)*OC++*, which translates an OC program into a class; 2) the *Module Description Language* (MDL) with which we can build classes through composition and inheritance; 3) *MDLC* a tool that produces a synchronous class from a description file in MDL.
A user who does not need dynamicity can create composite classes, for which scheduling will be determined at compile time by MDLC.

The communication between synchronous objects on two different clocks, or between a synchronous object and a usual object is possible in an asynchronous way. Like in PTOLEMY nothing is imposed by the model, in order to let as much freedom as possible to the user.
Among possible type of communication we have: treatment of all events, in the order they came (a missed event is an error); treatment of all events (a given percentage of missed events is not an error); treatment of the last event .

Different policies are implemented with classes whose objects are associated with the inputs and outputs signals of clocks. These objects are the equivalent of the device drivers of an operating system. Their design require a precise specification of how asynchronous events must be translated into synchronous events, i.e. how synchronous objects perceives the asynchronous world.

A library of synchronous classes provides the execution machine for synchronous objects and useful classes, like classes for interfacing with files, keyboard, graphical interfaces. With our system it is possible to combine separately compiled synchronous objects which react like one synchronous object, and thus to avoid a combinatorial explosion on the number of states. There is also an implementation on the real time kernel VxWorks,

Developed at the University of Berkeley, the PTOLEMY system stands as an object-oriented framework that supports various execution models, or *domains* [BUC&al -94]. We are currently working on a translator from the output of Lustre and Esterel compilers to the Ptolemy language. The target domains may be *DE* (Discrete Events) and *SDF* (Synchronous Data Flow). Since both *DE* and *SDF* do not match the reactive synchronous execution model, we will develop a *SREC* (Synchronous Reactive Execution and Communication) domain which will be another target for our translator. Such a domain is not intended to replace synchronous compilers for building synchronous reactive systems, but to provide support for the execution of synchronous modules in Ptolemy. We have studied the interface of the SR domain to other domains (especially DE and SDF) to determine the meaning of communications between them. The goal is to allow the use of synchronous reactive modules to control the behavior of data-flow processes. This may ease the co-design of special purpose hardware and of the software that drives it, since the border between hardware and software can be adjusted through simulation before choosing an implementation.

References

[AND&PER - 93] C. André and M.A. Peraldi: Effective Implementation of ESTEREL Programs. *Workshop on Real-Time Systems, Euromicro'93*, Oulu (Finland), June 1993

[BEN&BER-91] A. Benveniste, G. Berry: The Synchronous Approach to Reactive and Real-Time Systems. *Proceedings of the IEEE*, vol 4, April 1994

[BO&VI-94] F. Boulanger, G. Vidal-Naquet: Integration of Synchronous Modules in an Object Oriented Language. *in Information Systems, Correctness and Reusability* pp.279-291, Word Scientific ,1995.

[BUC&al -94] J.T. Buck, S. Ha, E.A. Lee, D.G. Messerschmitt: PTOLEMY , a Framework for Simulating and Prototyping Heteregeneous Systems. *International Journal of Computer Simulation*, 19(2):87-152, November 1992.

[HAL-93] N. Halbwachs: Synchronous Programming of Reactive Systems: *Kuwer Academic Press*,1993.