Integration of Synchronous Modules in an Object Oriented Language

Guy Vidal-Naquet
Laboratoire de Recherche en Informatique
Université de Paris-Sud — Centre d'Orsay
91400 Orsay, France
E-mail: Guy.Vidal-Naquet@supelec.fr

and

Frédéric Boulanger
École Supérieure d'Électricité — Service Informatique
Plateau de Moulon, 91192 Gif-sur-Yvette Cedex, France
E-mail: Frederic.Boulanger@supelec.fr

ABSTRACT

This paper presents a model and tools to embed synchronous reactive modules in an object-oriented language. Embedded modules behave and may communicate synchronously, may be created or destroyed, and their interconnection may change dynamically at run-time. They are able to communicate asynchronously with other parts, synchronous or asynchronous, of the application.New synchronous classes can be created either by static composition of instances of existing classes, or by inheritance.

1. Introduction and motivation

Complex systems have components of radically different natures:

- Reactive components, that have to deal with the environment, receive input signals, produce output signals, and basically have to maintain a coherence relation between input and output signals. These components often must have properties dealing with real time and safety, and can communicate between each other in a synchronous or asynchronous way.
- Transformational components, that compute values without interaction.

Examples of such systems are supervision systems (for transportation or telecommunications), autonomous robots, self-tuning control systems.

During the last decade, a new approach, called "synchronous" has been developed for the programming of reactive modules [2]. It relies on the hypothesis that the reaction is instantaneous (in reality sufficiently fast), which gives rise to a model and programming languages that allow to treat time in a rigorous way and to marry parallelism and determinism. Informally, the execution of a program written in a synchronous language is like the functioning of an automaton with outputs: the

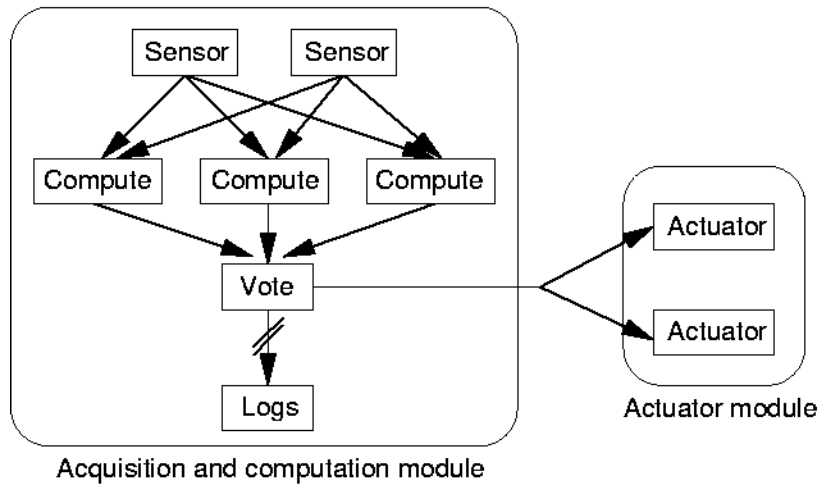

Fig. 1. Structuration

outputs are determined (instantaneously), and the next state is computed. One of the main advantages of the synchronous approach is that proof of properties can be done from the source code, after an automatic translation into mathematical language [3 6]. The execution of code written in a synchronous language requires an execution machine [1], in order to transform the input from the outside into signals that can be treated, and symmetrically, transform output signals into data acceptable by the outside. There is actually a drawback for using the synchronous approach: code is monolithic, i.e. it is not possible to compile the parts of a system separately and assemble them, 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 *regular* objects. This will allow to take advantage of the features of object oriented programming techniques, like class libraries, inheritance, methodology for structuration, dynamic creation of objects. This paper does not provide a methodology for dealing with reactive components, but provides tools for the application of OO methodologies or object based software platforms. For example, our system was easily integrated in a few days, in the Ptolemy environment of the University of Berkeley which is used for the simulation and the prototyping of heterogeneous systems [4].

We give two examples where OO could be applied:

Structuration enables to name a whole set of components with a single name, these components may be reactive or transformational.

On figure 1, the acquisition module has two sensors (e.g. for pressure and temperature), 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

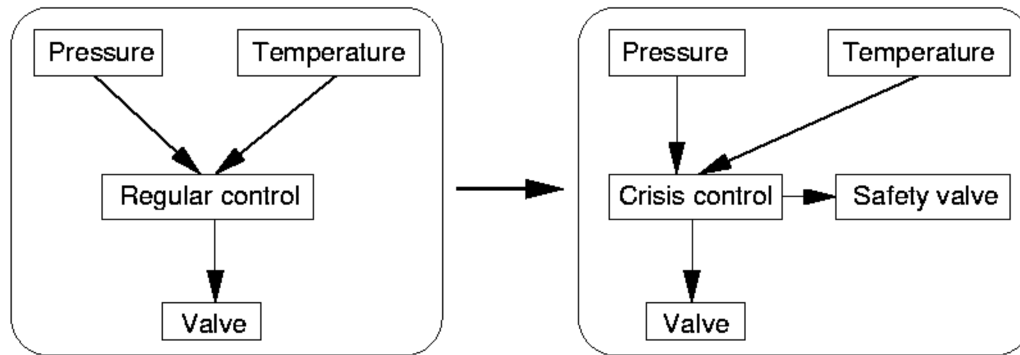

Fig. 2. Dynamicity of objects

logged on a data base, which can be done asynchronously (this is shown by // between Vote and Logs) and is also transmitted to two actuators. One would like to see the acquisition module and the actuator module as 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 example of figure 2, a controller is in charge of adjusting a valve, according to a schedule. If the pressure and/or the temperature becomes too high, another "crisis" module is substituted, and controls the valve, and a safety valve, with a possibly shorter sampling period.

In this scheme, the object "Regular Control" is destroyed, and the objects "Crisis Control" and "Safety Valve" are created. "Pressure" and "Temperature" are then connected to "Crisis Control".

In another possible scheme, shown on figure 3, the modules "Crisis Control" and "Safety Valve" would have already been present, and the connections would have changed on the fly. The advantage is a faster switch between the two configurations, but more memory is used because the modules exist all the time. From an external point of view, the behavior is the same with both schemes.

2. Operational model

We now present an operational model for the execution of synchronous modules. This model will be used as a reference for implementation. It gives the basic objects we use, and also the properties that have to be satisfied by any implementation. We use the following notations: If f is a function from A into B, and A' is a subset of A, $f\lfloor A'$ will denote the restriction of f to A'.

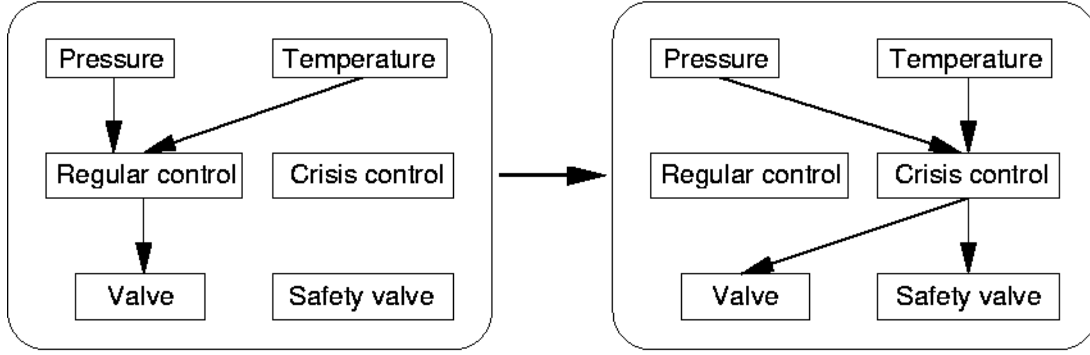

Fig. 3. Dynamicity of links

For a nuplet, $U = \langle e_1, e_2, \ldots, e_n \rangle$, $\quad U_{/e_i}$ will denote the component e_i of U.

2.1. Synchronous classes

A synchronous class defines the input and output signals, the set of states, and the reaction to inputs.
Let τ be a finite set of types
$\mathcal{B}$ the set of booleans $\{T, F\}$
$\mathcal{D}_t$ a set of values of type $t, t \in \tau$
$\mathcal{V} = \bigcup_{t \in \tau} \mathcal{D}_t$ the set of possible values

Definition 1 *A class* **C** *of synchronous objects is defined by:*

$$\mathbf{C} = \langle s_{i_C}, s_{o_C}, \Gamma_C, \Sigma_C, \sigma_0, \mathcal{R}_C \rangle$$

s_{i_C} *and* s_{o_C} *are two finite sets of input and output signals of* **C***, the types of the signals are given by the function:*

$$\Gamma_C : s_{i_C} \cup s_{o_C} \longrightarrow \tau$$

The set of states Σ_C *is a (possibly infinite) subset of* $\mathbf{N}$*,* σ_0 *is the initial state.*
$\mathcal{R}_C$ *is the reaction function:*

$$\mathcal{R}_C : (\mathcal{V} \times \mathcal{B})^{s_{i_C}} \times \Sigma_C \longrightarrow (\mathcal{V} \times \mathcal{B})^{s_{o_C}} \times \Sigma_C$$

$\mathcal{B}$ *is used to indicate if a signal is present or not.*

We distinguish a special subset of outputs, intuitively output signals whose value depends only on the state of the object, and not on the value of the input signals. More precisely, $s \in s_{o_C}$ is said to be input-independent when

$$\forall v, v_1 \in (\mathcal{V} \times \mathcal{B})^{s_{i_C}}, v', v'_1 \in (\mathcal{V} \times \mathcal{B})^{s_{o_C}}, \sigma, \sigma_1, \sigma'_1 \in \Sigma_C,$$

$$\left.\begin{array}{rcl}\mathcal{R}_C(v,\sigma) & = & (v',\sigma') \\ \mathcal{R}_C(v_1,\sigma) & = & (v'_1,\sigma'_1)\end{array}\right| \Longrightarrow v'(s) = v'_1(s)$$

The notion of input-independent signal is important because, as we will see later, it allows to break 'causality loops'.

2.2. Synchronous objects

An object is an instantiation of a class, and is identified by a name, it reacts in the way described by the reaction function of its class. More precisely: Let $\mathcal{A}$ be an alphabet, the set of possible names $\mathcal{N}$ is the set $\mathcal{A}^+$

Definition 2 *The instance* $\mathbf{O}$ *of class* $\mathbf{C} = \langle s_{i_C}, s_{o_C}, \Gamma_C, \Sigma_C, \mathcal{R}_C\rangle$ *with name* $\eta \in \mathcal{N}$*, is defined by*

$$\mathbf{O} = \langle \eta, s_i, s_o, \Gamma, \Sigma, \mathcal{R}\rangle$$

where $s_i = \eta \bullet s_{i_C}, s_o = \eta \bullet s_{o_C}, \Sigma = \Sigma_C$
$\forall s \in s_i \cup s_o, \Gamma(\eta \bullet s) = \Gamma_C(s)$
$\mathcal{R}$ *is the reaction function of* $\mathbf{O}$ *defined from* $\mathcal{R}_C$ *by renaming all signals* s *to* $\eta \bullet s$*.*

2.3. Clocks

A clock represents the evolution of a set of (synchronous) objects which react synchronously. It is a sequence of instants, and at each instant, the clock is characterized by:

- the set of objects on it, and the state they are in,
- the connection between the input and output signals of the objects on the clock,
- the values and the presence of the different signals.

Let $\mathcal{C}$ be the set of all synchronous classes (which cannot be created at run-time). We can define the set $\mathcal{M}$ of all possible objects. $\mathcal{M}$ is isomorph to $(\mathcal{N} \times \mathcal{C})$. We note $P_f(\mathcal{M})$ the set of finite subsets of $\mathcal{M}$.

Definition 3 *A clock* $\mathbf{H}$ *is a sequence of seven-uplets*

$$\mathbf{H} = (\mathbf{H}_n)_{n\in\mathbf{N}} = (\langle S_I, S_O, T, \Gamma, \mathcal{S}, \mathcal{L}, \mathcal{E}\rangle_n)_{n\in\mathbf{N}}$$

$T \in P_f(\mathcal{M})$ *is the set of objects on clock* $\mathbf{H}$ *at instant* n*,*
S_I, S_O *are the finite sets of input and output signals at instant* n*,*
Γ *is the typing function of* $S_I \cup S_O$ *i.e.* $\Gamma : S_I \cup S_O \longrightarrow \tau$.
S_I*,* S_O*, can be viewed as interface signals between the clock and the outside world.*

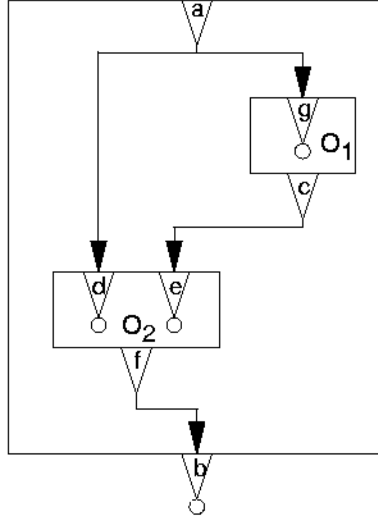

Fig. 4. Connection function

$S_{I_{H_n}} = S_I \cup \bigcup_{\mathbf{O} \in \mathbf{H}_{n/T}} \mathbf{O}_{/S_i}$ *is the set of all input signals in* $\mathbf{H}$ *at instant* n

$S_{O_{H_n}} = S_O \cup \bigcup_{\mathbf{O} \in \mathbf{H}_{n/T}} \mathbf{O}_{/S_o}$ *is the set of all output signals in* $\mathbf{H}$ *at instant* n

$S_{H_n} = S_{I_{H_n}} \cup S_{O_{H_n}}$ *is the set of all signals in* $\mathbf{H}$ *at instant* n.

S_I and S_O can change with n, so Γ changes because its domain changes.

A clock is furthermore characterized at instant n by the functions:

State evolution $\mathcal{S} : T \longrightarrow \mathbf{N}$. For each $O \in T, \quad \mathcal{S}(O) \in O_{/\Sigma}$,

Connection (or link) $\mathcal{L} : S_{I_{H_n}} \longrightarrow S_{O_{H_n}}$ (An input signal is linked to just one output signals, but several input signals can be linked to the same output signal),

Evaluation $\mathcal{E} : S_{H_n} \longrightarrow \mathcal{V} \times \mathcal{B}$. The evaluation function indicates with a boolean if a signal is present or not, and gives the values of the signals. Note that a signal has a value even when it is not present.

Figure 4 gives an example of connections, where ▽ are output signals, and ▽ are input signals. On this figure, the clock is characterized by:

$\mathbf{H}_{n/T} = \{O1, O2\}, \mathbf{H}_{n/S_I} = \{b\}, \mathbf{H}_{n/S_O} = a$

$S_{I_{H_n}} = \{b, O_1 \bullet g, O_2 \bullet d, O_2 \bullet e\}, S_{O_{H_n}} = \{a, O_1 \bullet c, O_2 \bullet f\}$

$\mathbf{H}_{n/\mathcal{L}}(O_1 \bullet g) = a, \mathbf{H}_{n/\mathcal{L}}(O_2 \bullet d) = a, \mathbf{H}_{n/\mathcal{L}}(O_2 \bullet e) = O_1 \bullet c, \mathbf{H}_{n/\mathcal{L}}(b) = O_2 \bullet f$

2.4. Constraints on clocks

A system may contain several clocks. Objects that communicate synchronously must be on the same clock.

In the following list, the constraints are not properties that can be deduced, but are properties that are compulsory for any implementation of the model.

1. An object cannot belong to more than one clock,
2. $\forall i, j \in \mathbf{N}, O \in H_{i/T}, O \in H_{j/T} \Longrightarrow \forall k \in [i, j], O \in H_{k/T}$. The instants when an object belongs to the clock form an interval of $\mathbf{N}$,
3. let $\prec_{H_n}$ be the precedence relation defined on the objects of H_n by:

   $$O \prec_{H_n} O' \Longleftrightarrow \exists s \in O_{/S_o}, \exists s' \in O'_{/S_i} \quad / \quad H_{n/\mathcal{L}}(s') = s$$

   and s is not input-independent.

   The transitive closure $\dot{\prec}$ of $\prec$ is a partial order,
4. At any given instant, two connected signals are of the same type, have the same value, and are both present or absent. More formally:

   $$H_{n/\mathcal{L}}(s) = s' \Longrightarrow H_{n/\mathcal{E}}(s) = H_{n/\mathcal{E}}(s')$$

5. For any object O in $H_{n/T}$ and $H_{n+1/T}$, with reaction function $\mathcal{R}_C$, input signals $\eta \bullet s_i$, and output signals $\eta \bullet s_o$, if $\mathcal{R}_C(H_{n/\mathcal{E}}\lfloor(\eta \bullet s_i), H_{n/\mathcal{S}}(O)) = (v, \sigma)$, then $H_{n/\mathcal{E}}\lfloor(\eta \bullet s_o) = v$, and if $O \in H_{n+1/T}$ then $H_{n+1/\mathcal{S}}(O) = \sigma$.

Condition 5 means that the value of the output signals of an object O at instant n (i.e. the restriction of the evaluation function to these signals) are computed 'instantaneously' from the value of the inputs of O through the reaction function, and that the reaction function computes the state of the object at instant n+1.

Condition 1 is necessary to determine what is an instant for an object.

Condition 2 is necessary to allow the computation at instant n−1 of the state of an object at instant n.

Theorem 1 *If condition 3 is satisfied, then there exists a unique evaluation function and a unique state evolution function that satisfy 4 and 5*

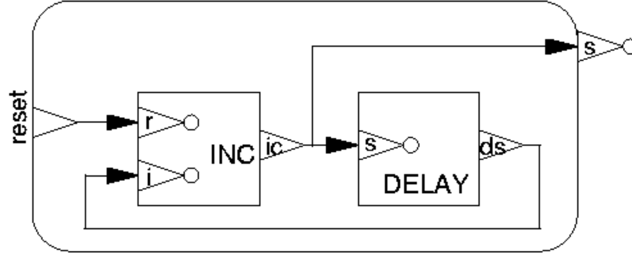

Fig. 5. Use of input-independent signals

A formal proof can be given, we give below the general idea: Condition 3 insure that there is no causality loop that would cause a signal to be a function of itself. Therefore, starting from a minimal object (according to $\dot{\prec}$) and using condition 4, one can determine the value of its input signals. Then, using condition 5, one can determine the value of its output signals, and its state for the next instant. So one can completely determine the values of all signals and the state of all objects at the next instant in a stepwise manner.

Figure 5 shows how an input-independent signal breaks a causality loop. This module is a counter, which increments its output at each instant of the clock. The signal *reset* resets the counter to 0. The module DELAY breaks the causality loop: at instant n, it stores the value of its input signal in the state of instant n+1. The module INC produces 0 as output when its input signal r is present, and n+1 where n is the value of input signal i, otherwise.

INC is an instance of class CI, with

$$CI_{/\Sigma} = e, CI_{/\sigma_0} = e, \tau = t_1, t_2, \mathcal{D}_{t_1} = \mathbf{N}, \mathcal{D}_{t_2} = \mathcal{B}, CI_{/\Gamma}(i) = t_1, CI_{/\Gamma}(r) = t_2$$

We denote by $\begin{pmatrix} r & : & (value, presence) \\ i & : & (value', presence') \end{pmatrix}$ the element f of $(\mathcal{V} \times \mathcal{B})^{S_i}$ such that $f(r) = (value, presence)$ and $f(i) = (value', presence')$

$\forall k \in \mathbf{N}, \perp$ denoting any value,

$$CI_{/\mathcal{R}}(\begin{pmatrix} r & : & (\perp, T) \\ i & : & (k, \perp) \end{pmatrix}, e) = ((ic : (0, T)), e)$$

$$CI_{/\mathcal{R}}(\begin{pmatrix} r & : & (\perp, F) \\ i & : & (k, T) \end{pmatrix}, e) = ((ic : (k+1, T)), e)$$

$$CI_{/\mathcal{R}}(\begin{pmatrix} r & : & (\perp, F) \\ i & : & (k, F) \end{pmatrix}, e) = ((ic : (k, F)), e)$$

DELAY is an instance of class DE, with

$$DE_{/\Sigma} = \{\sigma_i\}_{i \in \mathbf{N}}, DE_{/\Gamma}(s) = DE_{/\Gamma}(ds) = t_1$$

$$DE_{/\mathcal{R}}((s:(i,T)),\sigma_j) = ((ds:(j,T),\sigma_i)$$

The value of ds is completely determined by the state σ_j, and therefore ds is input-independent.

2.5. Dynamicity

There are two types of dynamicity in our model:

- dynamicity of processes,
- dynamicity of connections.

Since we view the objects as black boxes, without analyzing their semantics, we can only describe dynamicity in the model.

2.5.1. Dynamicity of objects

The destruction of a synchronous object O at instant n, is expressed by:

$O \in \mathbf{H}_{n/T}$

$O \notin \mathbf{H}_{n+1/T}$

$\forall s \in S_{I_{H_n}}, \mathbf{H}_{n/\mathcal{L}}(s) \in O_{/S_o} \Longrightarrow \mathbf{H}_{n+1/\mathcal{L}}(s)$ is not defined unless explicitly stated by a connection request.

If $\mathbf{H}_{n+1/\mathcal{L}}(s)$ is not defined, and $\mathbf{H}_{n+1/\mathcal{E}}(s) = (k,b)$ where $k \in \mathcal{V}$ and $b \in \mathcal{B}$ then $\mathbf{H}_{n+1/\mathcal{E}}(s) = (k,F)$. 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 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 Figure 2)

The instantiation of a synchronous object O of a class C at instant n, is expressed by:

$\forall t \leq n, O \notin \mathbf{H}_{t/T}$

$O \in \mathbf{H}_{n+1/T}$

$\mathbf{H}_{n+1/\mathcal{S}}(O) = \sigma_0$ (initial state of class C)

2.5.2. Dynamicity of connections

At instant n the connection of an input signal $s \in S_{I_{H_{n+1}}}$ to an output signal $s' \in S_{O_{H_{n+1}}}$ imposes that $\mathbf{H}_{n+1/\mathcal{L}}(s) = s'$.

A connection is requested at instant n, and effective at instant n+1.

3. Execution machine for clocks

For a clock, the execution machine, at instant n will do the following:

1. Process requests of creation and destruction of objects (determination of $\mathbf{H}_{n/T}$)
2. Process requests of (dis)connection (determination of $\mathbf{H}_{n/\mathcal{L}}$), check that property 3 is satisfied
3. Schedules an order of reaction for the modules that satisfies dependency relations (there exists one thanks to property 3)
4. Evaluates $\mathbf{H}_{n/\mathcal{E}}\lfloor(\mathbf{H}_{n/S_O})$ from the environment
5. Make each object $O \in \mathbf{H}_{n/T}$ react, which allows to compute $\mathbf{H}_{n/\mathcal{E}}\lfloor O_{/s_o}$ and $\mathbf{H}_{n+1/\mathcal{S}}(O)$ from $\mathbf{H}_{n/\mathcal{E}}\lfloor O_{/s_i}$. The latter can be computed thanks to $\mathbf{H}_{n/\mathcal{L}}\lfloor O_{/s_i}$ and property 4.

As part of its reaction, an object may request the instantiation or the destruction of another object, and may request the connection or disconnection of two signals. At this time, the model does not yet specify how these requests are send and processed.

4. Composite synchronous classes

In this section we describe two ways of building new classes, without (for brevity's sake) their completely formal definitions.

4.1. Composition of classes

It is possible to build new classes whose instances are composed of interconnected instances of existing classes. The formal definitions are not given here for brevity's sake, but can be deduced easily from the definitions of the input, output, state function and evaluation function of a clock.

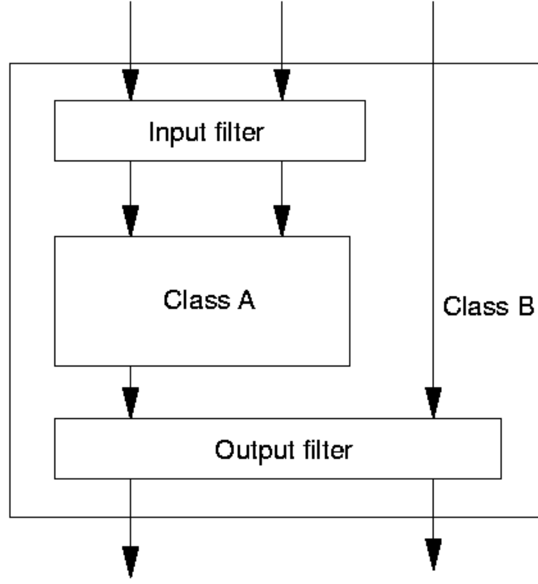

Fig. 6. Construction of a derived class by filtering

Figure 5 is an example of a composite class COUNTER built from an instance of class INC and an instance of class DELAY.

4.2. Inheritance

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

A synchronous class

$$\mathbf{B} = \langle s_i, s_o, \Gamma, \Sigma, \sigma_0, \mathcal{R} \rangle$$

inherits from a synchronous class $\mathbf{A}$ implies:

- $\mathbf{A}_{/s_i} \subseteq \mathbf{B}_{/s_i}, \mathbf{A}_{/s_o} \subseteq \mathbf{B}_{/s_o}$,
- $\mathbf{A}_{/s_i} \cap \mathbf{B}_{/s_o} = \emptyset, \mathbf{A}_{/s_o} \cap \mathbf{B}_{/s_i} = \emptyset$,
- $\forall s \in (\mathbf{A}_{/s_i} \cup \mathbf{A}_{/s_o}) \cap (\mathbf{B}_{/s_i} \cup \mathbf{B}_{/s_o}), \mathbf{B}_{/\Gamma}(s) = \mathbf{A}_{/\Gamma}(s)$.

These conditions express that the interface of class $\mathbf{B}$ includes the interface of class $\mathbf{A}$. A way to derive a class B from a class A is to filter the inputs and the outputs of an instance of A with two additional objects as shown in Figure 6. A typical example would be to derive an ABS brake from a usual brake by filtering the output of the usual break according to an additional signal that indicates if the wheel is skidding.

5. Realizations

Synchronous classes could be described in any suitable language. We have chosen the family of synchronous languages, namely ESTEREL and LUSTRE [5]. It could be extended to SIGNAL [3]. These languages are compiled into an intermediate language named OC. An OC program is basically a finite automata with outputs.

The basic components of our system are:

Occ++ which translate an OC program into a C++ class,

MDL the Module Description Language, used to describe the interface of synchronous modules and to build new classes through composition and inheritance,

Mdlc the MDL compiler, which translates an MDL description into a C++ class.

libSync the class library which contains the classes needed to support synchronous objects.

Composite classes may be used to avoid dynamicity when it is not needed. The schedule of the components will be determined at compile time by Mdlc.

The communication between synchronous objects that belong to different clocks, or between a synchronous object and a usual object is possible but nothing is imposed by the model. Like in PTOLEMY, the reason is to let as much freedom as possible to the user. For example, the user's policy could be to take into account all events, for safety reasons. In this case the communication will be done with unbounded buffers. Another policy could be to take into account only the last signal emitted. Different policies are implemented with classes whose objects are associated with the input and output signals of clocks.

A library of synchronous classes contains "useful" classes, like classes for interfacing clocks with the external world (i.e. files, keyboard, graphical interfaces).

One advantage that was felt by users is the fact that is possible to combine synchronous objects that are compiled separately, and react like one synchronous object, thus avoiding a combinatorial explosion on the number of states. There is also an implementation on the real time kernel VxWorks.

6. References

1. C. André and M.-A. Peraldi, *Effective Implementation of* ESTEREL *Programs*, Workshop on Real-Time Systems, Euromicro'93, Oulu (Finland), June 1993
2. A. Benveniste and G. Berry, *The Synchronous Approach to Reactive and Real-Time Systems*, Proceedings of the IEEE, **vol. 79**, n° 9, September 1991

3. Albert Benveniste and Paul le Guernic, *Hybrid Dynamical Systems Theory and the* SIGNAL *Language*, IEEE Transactions on Automatic Control, **vol. 35**, n° 5, May 1990
4. J.T. Buck, S. Ha, E.A. Lee and D.G. Messerschmitt, PTOLEMY*, a Framework for Simulating and Prototyping Heterogeneous Systems*, International Journal of Computer Simulation, **vol. 4**, April 1994
5. N. Halbwachs, P. Caspi, P. Raymond and D. Pilaud *The Synchronous Data Flow Programming Language* LUSTRE, Proceedings of the IEEE, **vol. 79**, n° 9, September 1991
6. N. Halbwachs, F. Lagnier and C. Ratel, *Programming and verifying Real-Time Systems by Means of the Synchronous Data-Flow Language* LUSTRE, IEEE Transactions on Software Engineering, **vol. 18**, n° 9, September 1992