BDL: a specialized language for per-object reactive control

F. Bertrand, Michel Augeraud

BDL: A Specialized Language for per-Object Reactive Control

Frédéric Bertrand, Michel Augeraud

Abstract— **The problem of describing the concurrent behavior of objects in object-oriented languages is adressed. The approach taken is to let methods be the behavior units whose synchronization is controlled separate from their specification.**

Our proposal is a domain-specific language, called BDL, for expressing constraints on this control and actually implementing its enforcement. We propose a model where each object includes a so-called "execution controller," programmed in BDL. This separates cleanly the concepts of what the methods do, the object processes, from the circumstances in which they are allowed to do it, the control. The object controller ensures that scheduling constraints between the object's methods are met. Aggregate objects can be controlled in terms of their components. This language has a convenient formal base. Thus, using BDL expressions, behavioral properties of objects or groups of interesting objects can be verified. Our approach allows, for example, deadlock detection or verification of safety properties, while maintaining a reasonable code size for the running controller.

A compiler from BDL has been implemented, automatically generating controller code in an Esterel program, i.e. in a reactive programming language. From this code, the Esterel compiler, in turn, generates an automaton on which verifications are done. Then this automaton is translated into a C code to be executed. This multi-stage process typifies the method for successful use of a domain-specific language. This also allows high level concurrent programming.

Keywords— **Concurrent object-oriented programming, control of behavior, verification, reactive languages.**

I. Introduction

THE OBJECT-ORIENTED APPROACH to programming has been demonstrated to be well-suited to describe complex systems, primarily because of its mechanisms for information hiding, aggregation, inheritance, and for the ability to rapidly prototype such systems. The designers of such systems have long known of the problems with supporting the programming of naturally concurrent systems (such as booking systems), that cannot be described well with sequential programming languages [1]. So the notion of concurrent object-oriented programming has arisen naturally.

The main problem that must be addressed in concurrent programming is the synchronization of object methods to prevent the object from being in an incoherent state as, for example, when two concurrent methods modify an object's attribute without mutual exclusion or proper ordering [2, p. 56]. Some form of constraint on the scheduling of method executions is necessary to ensure that such cases do not occur.

The most common approach to object-oriented synchronization is to control method execution by guards, checked at run-time. This approach mixes control and intrinsic processing code in the same code, a problem for understanding and maintenance.

Previously we have proposed [3], [4] an architecture for objects dissociating the processing activity of the object (achieved by methods) from the control of the process. This model gathers these conditions – the guards – in a dedicated structure. This construction improves the maintainability of the object by centralizing the execution conditions of each method in only one entity (called the *execution controller*). To express the execution conditions of methods and to program this controller, we have developed a language named BDL (Behavioral Description Language). This paper shows how BDL can be used to express the control of the behavior of simple objects or groups of objects. We also describe the BDL implementation based on a reactive language that allows us to verify properties of BDL programs.

In section II, we explain the need for control of concurrent objects and we present the execution controller achieving this control. The BDL language, used to program this controller, is described in section III. The semantics and the implementation of BDL are detailed in section IV. In section V, we show how verification of an object or a group of objects can be achieved. Related work is described in section VI. Finally, we conclude in section VII by emphasizing the new insights offered by this approach.

II. The Control Of Object Behavior

In this section, we explain why a concurrent object needs an execution control and we describe the object model in which the control is exercised.

A. The Need for Control of Concurrent Objects

The method execution inside a concurrent object may depend on whether other methods are or are not active. For example, if an object has two methods `read` and `write`, it is easy to understand that these methods cannot be executed at the same time. Their executions must be mutually exclusive. This is a scheduling constraint.

We will illustrate this type of constraint using a `File` object. The object owns four methods: `open`, `read`, `close` and `write`. There are different constraints on the execution of these methods. Let us suppose first that the object is in read-only mode (the `write` method is not accessible). In this case, there are two sorts of constraints: first a sequentiality constraint between the three first methods and then the `read` action must be allowed to repeatedly execute since the object must be permanently able to process `read` requests.

To express these constraints, we have defined a set of operators representing the BDL language (see III-A). BDL expressions use method names and operators. For example, the BDL expression specifying that instances have to execute repeatedly is indicated by the "`*`" operator; the sequence operator is indicated by "`;`." Execution of methods `open`, `read` and `close` of a `File` object may be ruled by the following BDL specification:

```
(open ; read ; close)*
```

A more accurate specification would allow multiple executions of `read`:

```
(open ; read* ; close)*
```

Now if we allow the file to be in read-write mode, we can use an exclusive execution operator, "`|`" to constrain `read` and `write`:

```
(open ; (write | read)* ; close)*
```

Our work aims to specify and enforce scheduling constraints between methods. Before describing our approach, we define the concurrent object model upon which the control is defined.

B. The Concurrent Object Model

Concurrent object-oriented languages may be classified according to three criteria [5]: the kind of object model used, intra-object concurrency constraints, and how interactions between objects are described.

The *object model* defines the relationship between the structures of execution (*threads*) and the object paradigm. There are three main approaches.

• *the passive approach*: concurrent execution is independent of objects. In this approach, threads and objects are orthogonal concepts. Synchronization constructs such as monitors in Java [6] or semaphores in Smalltalk-80 [7] or guards in Orca [8] must be judiciously used for synchronizing concurrent invocations of object methods. In the absence of explicit synchronization, objects are subject to the activation of concurrent requests and their consistency may be violated.

• *the active approach*: all objects are considered to be "active" entities that have control over concurrent invocations. The receipt of request messages is delayed until the object is ready to service the request. There are a variety of constructs that can be used by an object to indicate what method invocation it is willing to accept next. In Pool-T [9] this is specified by executing an explicit accept statement. In Rosette [10] an *enabled set* is used for specifying which set of messages the object is willing to accept next. In other languages such as Triveni [11] it is specified using an imperative language. The program that is written with this language represents an abstract behavior describing the interaction of the object with its environment.

• *the mixed approach*: both active and passive objects are provided. Passive objects do not synchronize concurrent requests. An example of such a language is Eiffel// [12]. The language ensures that passive objects cannot be invoked concurrently by requiring that they be used only locally within single-threaded active objects.

Intra-object concurrency concerns the level of concurrency inside the objects:

• *sequential objects* possess a single active thread of control. Objects in Pool-T are examples of sequential objects.

• *concurrent objects* do not restrict the number of internal threads. New threads could be created in different ways. Thread creation could be automatically triggered by reception of an execution request or could be controlled by the object. In the first case the thread is blocked until the activation conditions are checked. In the second case the thread is created only if the object state allows the execution.

Interaction between objects allows the specification of message sending and receiving by objects. We have chosen to follow MEYER's proposition [2, p 71] to let the calling object be master of the call policy. So a method call will be able to determine the call mode (*synchronous* or *asynchronous*) and the service mode (*immediate_service* or *normal_service*[1]). This interaction describes the degree of control that can be exercised by objects in the client and server roles. The description of this interaction goes beyond the scope of this paper because we concentrate, in this paper, on the control of intra-object concurrency, so the reader is referred to [5] for a complete description.

For each of these criteria, our choices have been selected with the following aims:

1. to define the object as a self contained entity possessing its own executive structures;
2. to design an application as a set of communicating distributed objects;
3. to improve the capacity of the object to react. This is achieved in two ways. First, by raising the degree of concurrency; therefore we allow a request to be satisfied as soon as the state of the object enables it. Secondly, we permit the object to control method executions by interrupting or cancelling their execution.
4. to allow object behavior to be formally verified.

Controlling the object behavior consists of determining, according to the state of the object, whether the execution of a method is possible and, in that case, to launch this execution. If the execution is not possible, the request may be stored or rejected following a policy determined by the client.

We have chosen an active object model. This model has been adopted by a great number of languages on concurrent object-oriented systems such as Pool-T, Eiffel// and Rosette. An active object decides, according to the object state, the time when a method execution can be run. The object plays the role of both a client (when it requires the execution of a method of another object) and a server (when it executes one of its methods on the request from another object). Furthermore, this model is well adapted to the fourth aim: the verification of the behavior control is easier to achieve if the control is carried out by the object and not an external structure, because the object is a self-contained entity.

In order to raise the capacity of service, an inter-method concurrency model has been adopted. So every method

[1]As soon as possible.

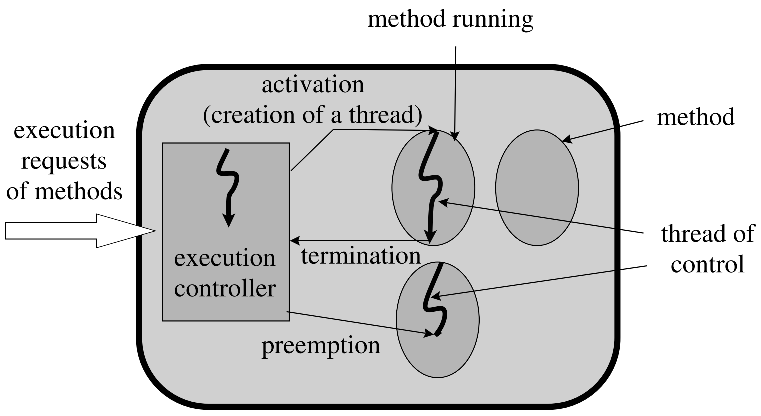

Fig. 1. Architecture of a concurrent object with an execution controller

executes in a separate thread of execution.

Finally the need for verification of some properties (safety, liveness) on the behavior control of the object is made easier using a centralized control, achieved by a dedicated structure on which the verification is carried out. These properties may be, for example, deadlock freeness, the respect of mutual exclusion during the executions of methods changing the object state, or respecting the sequentiality constraints between these executions. The previous example of `File` object has shown that scheduling constraints must be respected. We call the entity in charge of this control, the *execution controller*. Furthermore, this (per object) centralized control allows better reuse by a clear separation between control and processing.

C. The Execution Controller

The controller, depicted in figure 1, has three functionalities. The first one is related to the execution management. The controller carries out the creation of the thread for every method having to be executed and then associates the method code and the execution thread.

The second one, more complex, consists of synchronizing the execution according to the BDL program. To achieve this functionality, the controller must, permanently, be aware of the execution state of methods. Currently the model of the execution controller is restricted: the controller does not use object attributes to manage executions; the activation conditions of a method that uses attributes are checked in the method code. This restriction is necessary to use verification tools which work using finite transition systems.

In fact this restriction avoids introducing numerical data in the controller. Because for checking transition systems tools use finite structure, the use of numerical variables with infinite domains is not possible.

The third one concerns the storage of pending requests. When a request occurs, and if the object state does not allow immediate processing, the controller stores this request if the client does not request an immediate execution. This request remains stored until the controller allows the method execution. We could note that a buffer overflow may occur if an object receives multiple requests for a method which is not runnable (e.g. multiple requests for `B` and none for `A` in the BDL expression "`A ; B`"). In this case, the control mechanism may detect this overflow and sends a message indicating this problem back.

III. Programming Controllers With BDL

In the previous examples, we have used operators to specify scheduling constraints concerning method executions. These operators allow the construction of BDL expressions which specify scheduling constraints.

A. The Operators of the BDL Language

In BDL, there are only two types: method identifiers and scheduling operators. A BDL term corresponds either to a method identifier or to one (or two according to the arity of the operators) term and one operator. These operators are adapted from the asynchronous reactive language Electre [13]. As in Electre the operators are all fundamental and none can be realized by a combination of the others. Following is the BNF syntax for BDL:

T	::=	T `*`	– repetition
	|	T `;` T	– sequentiality
	|	T `|||` T	– weak parallelism
	|	T `||` T	– strong parallelism
	|	T `|` T	– mutual exclusion
	|	T `^` T	– weak preemption
	|	T `/` T	– strong preemption
	|	`(` T `)`	– parenthesis
	|	*method_identifier*	– method identifier

We explain the language by giving an intuitive semantics of the operators (the formal semantics are described in the appendix):

- a unary *repetition* operator, denoted by "`*`," indicates that the control specified by the term works indefinitely;
- a binary *sequentiality* operator, denoted by "`;`," indicates that the left term must be executed before the right one;

• two binary *parallelism* operators indicate that the two terms (left and right) can be executed at same time. In that case, we consider two types of parallelism:

– so-called *weak* parallelism, denoted by "`|||`," expressing a *possibility*: the concurrent structure may be ended when a term has achieved its execution and the other has not started yet;

– so-called *strong* parallelism, denoted by "`||`," expressing a *necessity*: the concurrent structure is ended only when both terms have achieved their execution.

• a binary *mutual exclusion* operator, denoted by "`|`," indicates that executions of both terms are mutually exclusive;

• a binary *priority* operator, denoted by "`#`," indicates that if an execution request occurs for one of the methods in the right term, then the execution requests for methods in the left term are no longer satisfied (and they will be stored). However, the execution of methods in the right term is subordinated to the termination of all the methods being executed in the left term. In this case, when no method of the left term is being executed and a request for a method of the right term occurs, then this one is immediately satisfied;

• two binary *preemption* operators indicating that the execution of the left term can be stopped by the beginning of the execution of the right term. We consider both following preemption types:

– so-called *weak* preemption, denoted by "`^`": the execution of the preemptive structure (right term) can take place only during the execution of the preempted structure (left term);

– so-called *strong* preemption, denoted by "`/`": the execution of the preemptive structure is required even if the preempted structure has terminated its execution.

• all these operators have the same precedence level, so parenthesis could be used to modify this precedence.

Among these operators, the most complex is certainly the priority operator. We will illustrate its semantics by considering the `File` object (cf. II-A) again. The previous specification was:

```
(open ; (write | read)* ; close)*
```

This specification introduces an unwanted behavior due to the semantics of the "`*`" operator which corresponds to an endless execution. Therefore `close` will never be executed. The use of preemption is inappropriate because it is too "rough." We wish to leave the reading (or writing) running until the end before closing the file by using a priority operator which allows the specification of this type of control:

```
(open ; (write | read)* # close)*
```

Now, when receiving an execution request for `close`, it is executed immediately if `read` (or `write`) is not running. Otherwise requests for the two methods are not satisfied any longer and `close` starts as soon as one of the two methods terminates.

BDL operators enable one to easily describe different policies of use. A fair policy between both modes:

```
(open ; (write | read)* # close)*
```

or then give priority to reading:

```
(open ; (write # read)* # close)*
```

or writing:

```
(open ; (read # write)* # close)*
```

In our current object architecture, each object of a class has a controller defined at class level. However, this controller can be replaced by a new controller whose interface is compatible with the previous one. We use this facility in the case of aggregate objects where the individual control of components is changed to a global control of components. The controller, ensuring this global control, is bound to the component's objects.

B. Examples of BDL Programs

We give two examples of BDL programs with an object and a group of objects.

In the first example we illustrate the need for preemption using an elevator truck. This object has five methods: `init`, `m_on`, `m_back`, `m_up`, `m_down` and `stop`. The `init` method describes the initial position of the truck and must be executed before the four others. Methods allowing the truck to move along the same axis (left/right and up/down) must have mutually exclusive executions but the movement may be simultaneous along both of them. At last, if necessary, the truck can stop any movement by invoking the `stop` method. The mutual exclusion (operator "`|`") of a movement along the same axis is expressed by:

```
m_on | m_back        and        m_up | m_down
```

Possible parallel execution (operator "`|||`") is expressed by:

```
(m_on | m_back) ||| (m_up | m_down)
```

The preemption (operator "`/`") is carried out by the `stop` method by:

```
((m_on | m_back) ||| (m_up | m_down)) / stop
```

Finally the global specification is:

```
                 init ;
((m_on | m_back) ||| (m_up | m_down))
                 / stop
```

In the second example, the control of the behavior of a group of objects (or *aggregate*) is achieved by using the behavior control for each object as a pattern. To illustrate how the behavior control of an aggregate can be defined, we use a simple video player [14] with four functions: load a tape, play a tape, stop playing and eject a tape. The video player (`VideoPlayer` object) has two components: a motor (`Motor` object) and an eject mechanism (`EjectMech` object). The `Motor` object has two methods: `play` and `stop`. The `EjectMech` object has two methods too: `load` and `eject`.

The behavior control of the `Motor` object can be expressed by the following BDL code, where the operator "`^`" means the execution of `play` method could be stopped by the execution of `stop` method:

```
(play ^ stop)*
```

and the one of the `EjectMech` by:

```
(load ; eject)*
```

When we aggregate a `Motor` instance and an `EjectMech` instance to build the `VideoPlayer` object, the behavior control of the `VideoPlayer` can be expressed by:

```
(load ; (play ^ stop)* # eject)*
```

One can notice in this code that hiding the method identifiers of one of the two objects produces the control code for the other. This sort of aggregation (called *aggregation with hiding*) conforms to the principle defined by Hartmann *et al.* [15]. This principle states that the behavior of an aggregate restricted to the methods of a component object must give the behavior of this component. Currently the composition between these different controls of each object is done by hand but we are working to define rules managing automatically this composition.

In the next section, we describe the choices that have been made in the implementation of BDL.

IV. Semantics of BDL

A. An Event-Based Semantics

The notion of event is well suited to highlight the different steps in the processing of execution requests. These requests are caught by the controller as *arrival* events. These events represent requests either from other objects or from the object itself. It must be noted that, in our model, these events occur at distinct instants and are separately received (one by one) by the controller. This distinction allows one to model distributed objects more easily. The executions triggered by the controller can be considered as reactions to these *arrival* events. The execution is triggered by the emission of a *start* event towards the runtime system. A *termination* event informs the controller of the end of execution.

When a method is called, what happens on the client object side must be also considered. The execution request is modeled by a *call* event and once the method is correctly carried out, a *return* event is sent back to the calling object.

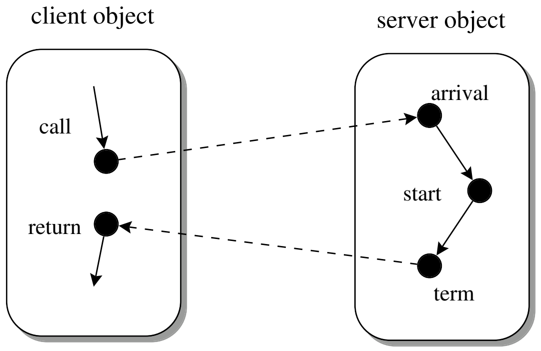

Fig. 2. The events in the lifespan of a method invocation

Figure 2 describes an event sequence happening when an object (client) requests the execution of a method from another object (server).

In the called object, the *arrival* event is distinct from the *start* event; the controller handles the *arrival* event and emits the *start* event for the actual method only if the method execution meets the scheduling constraints. The distinction between *start* and *term* events introduces the notion of duration for an execution. This model of execution is also present in [16] under the name SOS (Service Object Synchronisation).

B. Using an Automaton as Target Code

From a theoretical point of view, a BDL program ensures a correct event trace. An automaton is a well-known structure to accomplish this. For example, the BDL program

```
(open ; read* # close)
```

could be represented by the automaton of figure 3. On this automaton, the "?" symbol means an event reception, "!" means an event emission. The "+" symbol indicates an exclusive choice between several transitions.

Using an automaton as target code for BDL presents the following advantages:

- *efficiency*: an automaton described in a programming language produces fast executable code. This efficiency is important because this code is often executed and the object must quickly respond to an execution request;
- *proofs*: the automata are mathematical structures on which many verification tools have been developed.

C. From BDL Program to an Automaton

It is possible to produce an automaton from a BDL program but a great part of this work is already done (and applied to critical systems) by a family of languages named *reactive languages*.

Reactive languages have been designed to program reactive systems. A reactive system [17] is defined as reacting instantaneously to events received continually from the environment by emitting events towards it. The system does not compute or carry out a function but maintains a balance with its environment. That is: maintains a relationship between its inputs and outputs as time goes past. Most of real-time systems such as control or signal processing systems and communication protocols are reactive. Software implementation of these systems has led to a family of languages, called reactive, for example Esterel [18], Statecharts [19] and Electre [13]. Due to the critical aspect of the systems implemented, these languages have a mathematical semantics allowing formal verification of properties on the behavior of these systems. Compilers of reactive languages produce an automaton used both by verification tools to verify properties and by translators to generate a C (or Ada) code implementing the automaton.

In a previous section (see IV-A), we have shown the method execution could be represented by a sequence of events, the controller managing this sequence. Thus the controller behaves as a reactive system.

D. The Esterel Language

We choose to use the synchronous reactive language Esterel [18] for it offers high level control structures and, on the other hand, it is interfaced with different verification tools. Though its execution mode is synchronous (simultaneous perception of several events) we use it in an asyn-

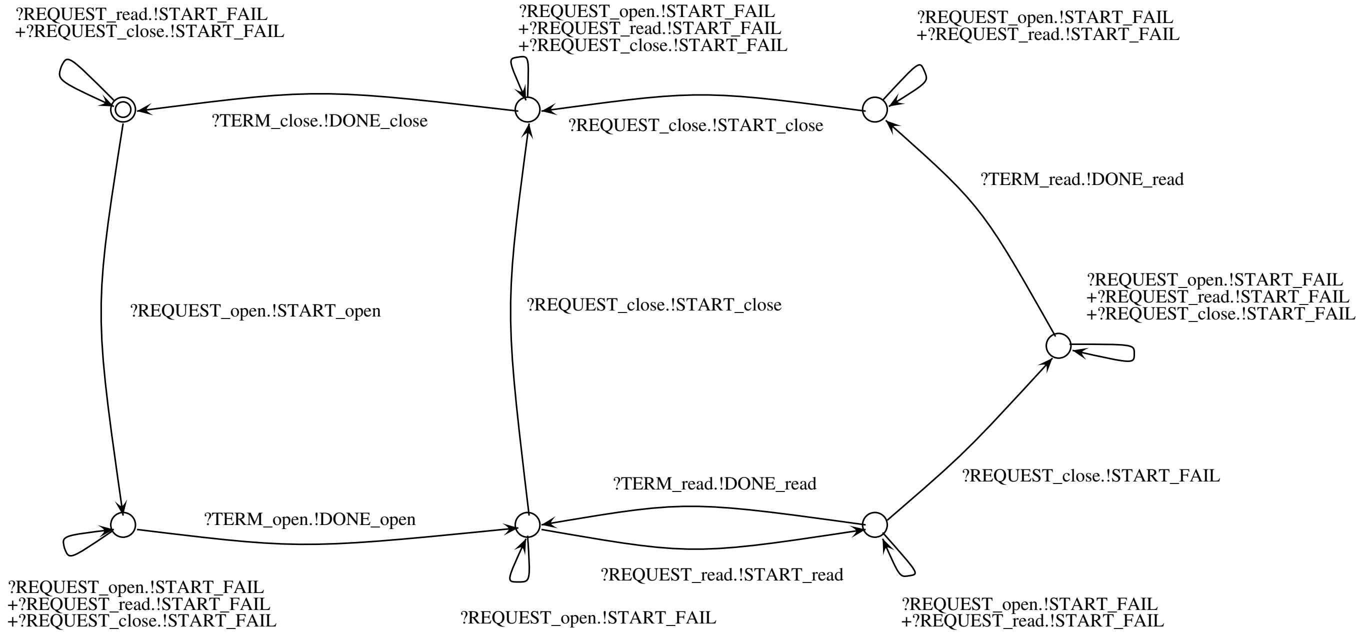
?REQUEST_read.!START_FAIL
+?REQUEST_close.!START_FAIL
?REQUEST_open.!START_FAIL
+?REQUEST_read.!START_FAIL
+?REQUEST_close.!START_FAIL
?REQUEST_open.!START_FAIL
+?REQUEST_read.!START_FAIL
?TERM_close.!DONE_close
?REQUEST_close.!START_close
?TERM_read.!DONE_read
?REQUEST_open.!START_open
?REQUEST_close.!START_close
?REQUEST_open.!START_FAIL
+?REQUEST_read.!START_FAIL
+?REQUEST_close.!START_FAIL
?TERM_read.!DONE_read
?REQUEST_close.!START_FAIL
?TERM_open.!DONE_open
?REQUEST_read.!START_read
?REQUEST_open.!START_FAIL
+?REQUEST_read.!START_FAIL
+?REQUEST_close.!START_FAIL
?REQUEST_open.!START_FAIL
?REQUEST_open.!START_FAIL
+?REQUEST_read.!START_FAIL

Fig. 3. The automaton of the BDL program `(open ; (read* # close))*`

chronous way to describe the working of the execution controller. The perception of events is restricted to only one event per instant.

Esterel is a synchronous imperative language. A quick introduction to Esterel semantics can be found in [20]. A program in Esterel consists of a collection of interacting modules. A module has an interface that defines its input and output signals and a body that is an executable statement.

There are two basic composition operators: the parallel composition operator "`||`" and the sequential composition operator "`;`." In a parallel statement, all components are activated simultaneously; the parallel statement terminates instantaneously when both components have terminated.

Interactions between modules takes place through the use of "*signals*." A signal may carry a value. Occurrences of signals that are emitted by a program's environment (input signals) are the only causes for the program to react. Input signals correspond to input events from the model of controller. An Esterel program reacts instantaneously to the receipt of input signals by emitting output signals towards its environment. Output signals correspond to the activation events from our model. A program may also emit and receive internal signals, used for inter-module communication within the program itself. Internal signals are not visible to the environment.

Two assumptions are made on signals:

- signals are broadcast within the program (ie. each module in the program receives all signals);
- signals are received instantaneously by all modules in the program.

An Esterel program does not have an associated clock. The synchrony hypothesis (reaction takes no time) coupled with signal assumptions allows a rigorous processing of multi-form time. Time can be handled as an ordinary signal ("clock signals"); any signal defines a particular clock.

E. Defining BDL Semantics from Esterel Operators

In BDL some operators have a direct equivalent Esterel operator as "`*`," "`;`," "`||`" operators, while the definition of "`|||`," "`|`," "`#`," "`/`," "`^`" operators requires a sequence of Esterel statements. For a detailed description (and discussion) of the translation of each operator, the reader is referred to [21].

Several attributes are involved in the translation of BDL operators requiring a sequence of Esterel statements. One of them, which plays a major role, is discussed in the following.

The *RTS* (*Ready To Start*) attribute is calculated on each BDL term T giving the name of the methods ready to be executed. The rules of a calculus for this attribute are the following ones:

$$\begin{array}{rcl} RTS(m) & = & \{m\} \\ RTS(T*) & = & RTS(T) \\ RTS(T_1\ op\ T_2) & = & RTS(T_1) \text{ if } op \in \{\texttt{;}\} \\ RTS(T_1\ op\ T_2) & = & RTS(T_1) \cup RTS(T_2) \\ & & \text{ if } op \in \{\texttt{|}, \texttt{||}, \texttt{|||}, \texttt{\#}, \texttt{\^{}}, \texttt{/}\} \end{array}$$

For example, RTS(`((A ; C) | (B ; D))`) = {`A`,`B`}.

For each BDL term of the form "$T_1\ op\ T_2$," where the operator has not a direct equivalent Esterel operator, corresponds an Esterel "template" including an Esterel statement set defining the semantics of the operator. We present the translation of these BDL operators and the associated template.

For the weak parallelism operator, we have the following template:

```
trad(T1 ||| T2) →

    trap T_EXIT in
        signal TERMINATED in
            trad(T1);
            emit TERMINATED
        ||
            abort
                await TERMINATED do exit T_EXIT end
            when immediate [     ⋁ (m ∈ RTS(T2))     START_m ]
        end signal
    ||
        signal TERMINATED in
            trad(T2);
            emit TERMINATED
        ||
            abort
                await TERMINATED do exit T_EXIT end
            when immediate [     ⋁ (m ∈ RTS(T1))     START_m ]
        end signal
    end trap
```

Because of the semantics of "|||" operator, when a branch terminates whereas the other branch has not yet started, an expression such as "T_1 ||| T_2" terminates (each term T_i may be composed recursively by other BDL terms). Disjunction between all the different START_m events corresponds to different possibilities of executions of m methods. So, in the Esterel code, at the end of each branch *trad(T_i)*, a TERMINATED event (local to each signal statement) is emitted. The TERMINATED event throws the T_EXIT exception only if a START event (belonging to *RTS* set) concerning a method of the other branch has not yet been emitted. This is done by the Esterel preemption operator (abort ... when).

Translation of the mutual exclusion operator is based on a selection corresponding to the following template:

```
trad(T1 | T2) →

    await
          case immediate [ (     ⋁ (m ∈ RTS(T1))     ARRIVAL_m ) ]
          do
              trad(T1)
          case immediate [ (     ⋁ (m ∈ RTS(T2))     ARRIVAL_m ) ]
          do
              trad(T2)
    end await
```

To determine which method will be executed we use a sort of "switch" statement represented by the Esterel operator "await case ... end." As in the translation of the BDL weak parallelism operator, the disjunction corresponds to the choice offered by the different methods, ready to start, included either in T_1 or in T_2.

In the translation of the priority operator, we are faced with the problem of waiting for the running methods included in T_1. This waiting is representing by the test "present ... then." This test must be present for each method identifier of T_1 and we put in parallel these different tests because, *a priori*, we do not know when the TERM event will occur:

```
trad(T1 # T2) →

    abort
         trad(T1)
    when immediate [     ⋁ (m ∈ RTS(T2))     ARRIVAL_m ]
    do
                                    present ACTIVE_m' then
             || (m' ∈ MET(T1))         await TERM_m'
                                       await DONE_m'
                                    end
    end abort;
    trad(T2)
```

The translation of BDL preemption operators is similar to the previous one. The difference is that one must emit a STOP event instead of waiting for a TERM event. For each running method, this STOP event must trigger the preemption of the method. Then, after the preemption sequence, the translation of T_2 could be executed. This semantics gives the following template:

```
trad(T1 ^ T2) →

    abort
         trad(T1)
    when immediate [     ⋁ (m ∈ RTS(T2))     ARRIVAL_m ]
    do
                                     present ACTIVE_m' then
             || (m' ∈ MET(T1))          emit STOP_m'
                                     end
        ;
        trad(T2)
    end abort
```

For the strong preemption operator, there is only one change: the translation of T_2 is always executed even if the translation of T_1 completes without abortion.

```
trad(T1 / T2) →

    abort
         trad(T1)
    when immediate [     ⋁ (m ∈ RTS(T2))     ARRIVAL_m ]
    do
                                     present ACTIVE_m' then
             || (m' ∈ MET(T1))          emit STOP_m'
                                     end
    end abort;
    trad(T2)
```

If the term T only consists of a method identifier then we use this template:

```
trad(m) →  run EXEC_METHOD [ signal ARRIVAL_m/ARRIVAL_,
                                    START_m/START_,
                                    TERM_m/TERM_,
                                    DONE_m/DONE_ ]
```

For this template, we use a feature of Esterel that allows us to reuse a module with renaming of its signals. The EXEC_METHOD module has the following form:

```
module EXEC_METHOD:

input ARRIVAL_, TERM_;
output START_, DONE_;

await immediate ARRIVAL_;
emit START_;
await TERM_;
emit DONE_

end module
```

F. *Esterel Architecture of the Translation of a BDL Program*

An overview of Esterel architecture of the translation of a small BDL program managing two methods (A / B) is represented in figure 4.

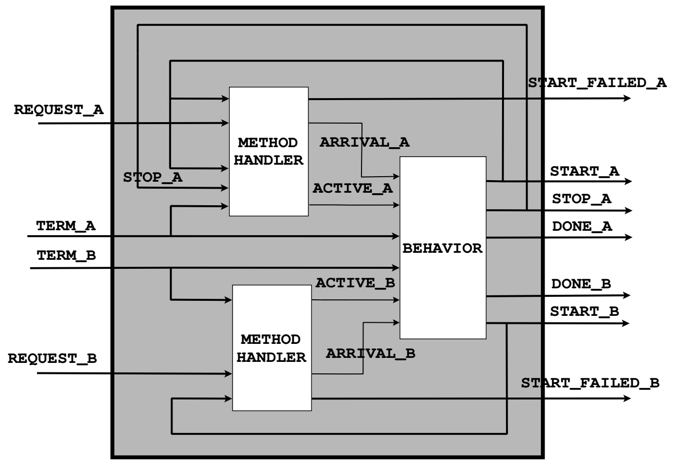

Fig. 4. Esterel architecture of the translation of a BDL program managing two methods

Each method managed by the execution controller needs a METHOD_HANDLER module receiving requests (REQUEST event corresponding to the *arrival* event on figure 2) and storing the execution state of the method (execution is indicated by the emission of a corresponding ACTIVE event). The request is then transmitted (ARRIVAL event) to the BEHAVIOR module. If the method can be executed then a START event is emitted and this emission goes on at every instant until the reception of the TERM event. Otherwise a START_FAILED event is emitted.

In the execution controller, in figure 4, there is only one BEHAVIOR module representing the decision structure. The BEHAVIOR module implements the BDL program using Esterel templates described in the previous section (IV-E). We have implemented a BDL compiler carrying out the compilation of the BDL program into Esterel code and also building the Esterel main module representing the automaton.

The Esterel compiler allows one to get a finite state automaton represented by boolean equations (BLIF[2] format) that is translated into C by a postprocessor. This format allows an implicit representation (more compact than an explicit one) of an automaton. Another advantage offered by this format is its use as input format for several academical verification tools.

G. *Implementing a Concurrent Object with an Execution Controller*

The implementation of concurrency in the reactive object model relies on the use of the *lightweight processes* library C Threads [23] built on the Mach operating system [24]. The creation of a reactive object requires the creation of a process holding at least a lightweight process. The aim of this process is to ensure the object control by receiving the execution requests and by executing the execution controller code. Each method execution gives way to the creation of a lightweight process.

The object-oriented language used is C++ [25]. When creating an object, the constructor call involves the creation of a process bound to a communication port. The object is registered to a name server (functionality offered by Mach facilitating object distribution), and the lightweight process ensuring the object control is activated. These mechanisms are gathered in a ReactiveObject class which every reactive object must inherit.

The implementation of reactive objects is divided into three steps. The first step is building the automaton from a BDL program. The second one is the building of the controller implementation which is obtained by linking the code of the automaton to the code dedicated to method activation and request memorization. The last one consists of linking the execution controller code with the code of the object and with the code ensuring thread management and communication between reactive objects. Figure 5 depicts these different steps.

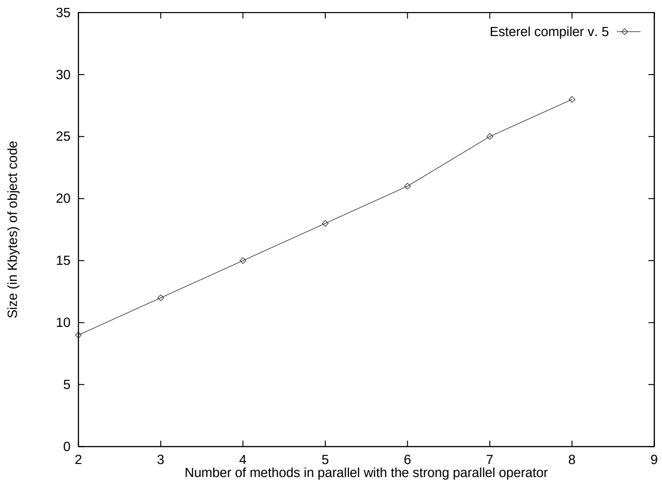

Fig. 6. Size[3] of the compiled automata

[2]Berkeley Logical Interchange Format [22].
[3]Results obtained on a Sun Sparc 5 station with gcc compiler (v. 2.7.2).

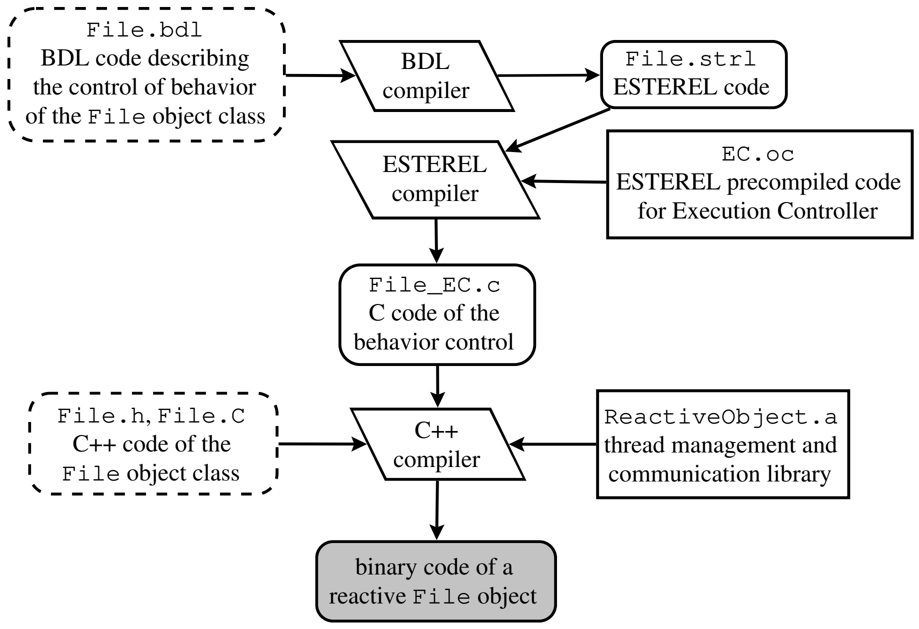
File.bdl
BDL code describing the control of behavior of the File object class
BDL compiler
File.strl
ESTEREL code
ESTEREL compiler
EC.oc
ESTEREL precompiled code for Execution Controller
File_EC.c
C code of the behavior control
File.h, File.C
C++ code of the File object class
C++ compiler
ReactiveObject.a
thread management and communication library
binary code of a reactive File object

Fig. 5. Development steps of a reactive `File` object

The size of the generated automata obviously varies according to the complexity of BDL expressions. Figure 6 shows the results obtained for a different number of methods in parallel. This figure underlines the linear growth of automata obtained with by the Esterel compiler. From the version 4, this compiler generates an implicit description of an automaton using boolean equations and so it avoids the combinatory blow-up of explicit description.

In the current version, every object has its own controller, which is a program using a variable size automaton. This is a problem for implementation of large applications where the number of reactive objects handled is significant. Nevertheless, in a new implementation, this problem will be solved by using only one controller for every object class. The reactive script then acts as a server to which the instances submit their current state and the received event. In response, the automaton sends back the new state and the events to emit.

V. Verification of The Control Behavior

Verification is an important step in the implementation of an object. Object-oriented programming is concerned with reuse: an object designed for an application may be reused in other applications. To avoid an error in the design of an object being propagated to several applications, it is important to ensure that this object is *working correctly*.

By working correctly, we mean that the object must be able to serve execution requests of methods permanently as well as respecting the control specification of its behavior. To ensure these specifications, the control of object behavior must verify two sorts of properties [26]:

- *safety properties*, which assert that the program does not do something bad.
- *liveness properties*, which assert that the program does eventually do something good.

We will illustrate the verification of safety properties with a well-known object: a micro-wave oven. This object is a compound of two objects: a `Gate` and a `MWGenerator` (micro-wave generator). The behavior control of the `Gate` object is the following:

```
(open ; close)*
```

and that of the `MWGenerator` is:

```
(on / off)*
```

where the `on` method has a continuous action (micro-wave emission) and must be stopped by preemption. The behavior control of the aggregate can be defined as:

```
((on / off) | (open ; close))*
```

In this program, when the oven is working, we must stop it before opening the door. The safety property to verify this could be expressed by: "*When the door is opened, the `MWGenerator` is not on.*" To verify this property, we will search a path in the automaton where `open` starts and, before `open` terminates, `on` could start. If this situation occurs the previous property is violated. This property can be expressed in Esterel. Figure 7 shows this code.

```
await
    case immediate START_open do
        abort
            await START_on do
                emit PROPERTY_VIOLATED
            end
        when TERM_close
    case immediate START_on do
        abort
            await START_open do
                emit PROPERTY_VIOLATED
            end
        when [TERM_on or STOP_on]
end await
```

Fig. 7. The Esterel code for the observer of the safety property

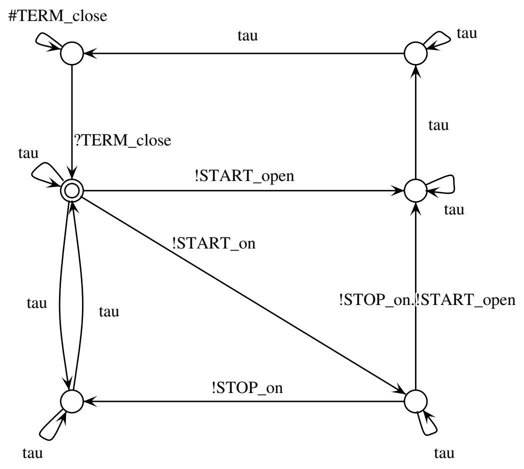

Fig. 8. The minimized automaton for the safety property verification

This sort of program is named *observer* and it catches misbehaviors and property violations for the program to verify. Observer is then put in parallel with the program under analysis. Model-checking consists simply of verifying the status of the observer events (such as PROPERTY_VIOLATED). Users can directly write observers by hand in Esterel (as in this example). However, there exists a tool called TempEst [27] offering a (linear time) temporal logic to express properties. This tool provides a translator from logic formulae into Esterel observer programs.

To verify properties on the resulting automaton (program + observer), we use the XEVE verification tool [28] developed in INRIA[4]. This tool provides two verification functions:

1. The minimization of the number of states of an automaton with respect to an equivalence called *bisimulation*. This minimization is done modulo input and output event hiding.
2. The other is the checking of the emission status of output events. The checking is made on the selected output signals, modulo fixed or hidden input events.

The analysis of the automaton with XEVE shows that the PROPERTY_VIOLATED event is never emitted; thus the safety property is verified.

Another, more illustrative, way to demonstrate this property is never violated, is to minimize the automaton in hiding the events not directly concerned by the property. The minimized automaton is depicted by figure 8. When we look at this automaton, we see that there is no path where a transition triggered by an occurrence of START_open is followed by a START_on transition without a TERM_close transition between the two (the τ-transitions stand for internal actions resulting from hiding some events). Symetrically, there is no path where a START_on transition is followed by a START_open transition without a STOP_on (because on is an endless method) between them.

[4]The French National Institute for Research in Computer Science and Control – France.

```
    await immediate START_on;
    abort
        await REQUEST_open;
        present [not START_open or not STOP_on] then
            emit PROPERTY_VIOLATED
        end
    when [TERM_on or STOP_on]
```

Fig. 9. The Esterel code for the observer of the liveness property

Concerning a liveness property we could verify: "*When the oven is working, a request to open the door will be eventually satisfied.*" This property could be verified by another observer. The Esterel code is listed in figure 9.

We make the hypothesis that the request must be immediately satisfied. The verification shows that this property could be violated when a request occurs during the time that the on method is active because the off method must be executed beforehand. This new BDL program fixes the problem:

```
((on / (off | (open ; close))))*
```

Another example of liveness properties is the deadlock detection. We must ensure that no sequence of execution requests leads to a deadlock.

For example, let us take an object having two methods met_1 and met_2, the behavior control of which is specified by the following expression

$$(met_1 \ \texttt{|||} \ met_2)\texttt{*}$$

On the other hand, let us consider the met_1 method calling met_2 during its execution.

If met_2 is the first method being executed, a deadlock may appear if, during the execution of met_2, the execution of met_1 starts. The execution of met_1 needs to execute a call of met_2. As met_2 is being executed, met_1 must wait for the end of the current execution of met_2 to know if its request is satisfied. However, when met_2 is terminated, it is no longer ready for execution

$$(met_1 \ \texttt{|||} \ met_2)\texttt{*} =$$
$$(met_1 \ \texttt{|||} \ met_2) \ \texttt{;} \ (met_1 \ \texttt{|||} \ met_2)\texttt{*}$$

So the execution request for met_1 cannot be satisfied any longer and so met_1 cannot achieve its execution. That leads to the case when neither of these methods can be executed any longer.

Because of our approach this problem can be detected formally. If we are able to statically determine the call graph of the methods of an object, it is then possible to build Esterel modules expressing this information. These modules are compiled with the module representing the execution controller to generate an automaton on which it is possible to determine deadlock states. These modules "simulate" the execution of a method and communication between methods may be introduced to allow verification.

The automaton, on which the verification is carried out, is obtained after compiling an Esterel simulation module composed of a controller module and modules representing the execution of every method managed by the controller. For this simulation module, the *start* and *term* events are considered as internal events (to the object) and so are not visible in the interface of the simulation module. This interface takes as inputs the *request* events corresponding to the methods managed by the execution controller and the *done* events of called methods. The *done* events of the methods managed by the controller and the *request* events of the called methods are outputs.

This modular architecture allows a modular approach to verification by checking correctness at the object level. In assembling the simulation modules of several objects it is possible to check the correctness of the behavior of a group of objects.

VI. Related Work

Many efforts in concurrent object-oriented languages have been related to the area of the control of concurrency. Nevertheless as far as we know few have used a reactive model to describe the execution control in an object. Process synchronization has been well studied and several synchronization patterns have been defined such as *lock protection*, *barrier synchronization*, *conditional waits*. Efforts in this area associate well-known synchronization mechanisms with object-oriented programming, keeping the benefits of object-oriented features. An implementation goal is to set concurrency control mechanisms orthogonal to other language properties. Works on this topic have been motivated essentially by the three main following issues:

1. It should be possible to design concurrent object-oriented applications by reusing components.
2. Concurrency should not interfere with other language constructs.
3. Parameterize concurrency to allow an answer to requests depending not only on the internal state of the object.

At this point we can recognize that researchers have studied synchronization problems using two different approaches to integrating objects and concurrency.

A. *Introducing Synchronization in Object-Oriented Concurrent Languages*

Concurrency in object-oriented languages has been achieved using two classes of techniques (for more details see [29]): by extending classical object-oriented languages with class libraries to allow concurrency — applicative approach — or by extending object programming concepts with concurrency abstractions — integrated approach.

A.1 Applicative Approach

The first approach has been followed to allow concurrency in Smalltalk-80 [7] where the basic synchronization primitive is represented by the `Semaphore` class. Using this approach new abstract synchronization features can be defined. B. Meyer in [2] propose to extend Eiffel with a minimal number of new mechanisms.

Taking advantage of the class facilities, libraries have been designed to support more abstract mechanisms such as in Actalk [30].

A.2 Integrated Approach

The second approach (integrated) is based on the idea that objects are code servers which will accept or delay execution of methods. As there exist similarities between object and process [2] two approaches have been followed:
• an object is an active entity with its own message queue and thread of control — *active object* model;
• in the second, activities are orthogonal to objects and synchronization is associated to method activation.

In the first class of languages the thread integrated to each object receives and services incoming requests. This class includes actor languages [30], [31], [32] and should be extended with agents [33]. Synchronization is provided by the private activity of the object. This activity may consist of an implicit program managing the input request queue such as in Actor [32] or an explicit program such as the *body* in Pool-T [9].

In the second approach, the object model has been modified to incorporate mechanisms allowing synchronization control. So in Guide [34] an object may be associated with a set of activation conditions that specify whether or not a method may be executed by a thread. But threads are execution units distinct of objects which are passive. Most of the languages in this model allow multiple threads to execute in an object at the same time. In Hybrid [35] "units of concurrency" called *domains* are defined as groups of objects in which at most one thread can be active.

B. *Techniques Used to Introduce Intra-Object Synchronization in Object-Oriented Languages*

B.1 Synchronization Abstractions

Different control abstractions of concurrency are used in concurrent object-oriented languages. We can distinguish five main mechanisms:
• the mechanisms based on the *synchronization counters* developed by Robert and Verjus [36] in which the execution state of methods is determined by updating a set of three counters *arrival*, *start*, and *term*. These counters are mainly used in guards: conditions associated to the activation of a method. We find these counters again in different languages such as Guide [34] or Dragoon [37]. If the approach has a great power of expression, using it is still difficult owing to its complexity.
• the *path expressions* [38] use a notation derived from regular expressions to specify synchronization constraints. We find these expressions in Procol [39]. These expressions have influenced the designers of Electre too and thus indirectly the BDL operators.
• a *delay queue* attached to a method which may be explicitly *closed* and *opened* (Hybrid [35])
• in Eiffel// [12] there exists a special method ensuring the request processing. This method presents many similarities with our execution controller. However our approach has a management component of the intra-object concurrency and is supported by a formal model.
• in the *enabled-sets* which have been developed with the Rosette language [10], the control is defined by using *states*

of control. For each state, a specific set of methods allowed to be executed is defined.

B.2 Integrating Synchronization by Means of Reactive Languages

In reactive languages, the state of a module only depends on the history of events that it has received. And as the controller has to manage an event flow, reactive programs are well fitted to control object behavior independently to other object properties. We earlier used facilities provided by reactive programming to extend object oriented paradigm [3], [40], especially to control concurrency in object-oriented programming using reactive modules.

Independently ROUDIER and CAROMEL propose, in [41], to set the `live` module in Eiffel// using the asynchronous reactive language Electre. Their work is close to our work but they focus more on making object-oriented the control mechanism whereas we have worked to keep the original reactive form to benefit from associated verification tools.

In the same spirit, we must cite Objectcharts [42] which use the Statecharts formalism, a graphical reactive language [19]. Even if this formalism is pleasant to use, the designers are not very clear about the executability of the specifications and on its implementation.

Other efforts have mixed reactive programming and object-oriented programming:

1. Based on a synchronous reactive model BOULANGER's "*synchronous objects*" [43] allow one to integrate reactive modules in an object-based program. A "synchronous object" is an object representation of a synchronous module. In this model channels that link reactive modules are represented by object methods. This leads to the integration of reactive programming in an object-oriented language with keeping the semantics of reactive modules and with the constraint of translating in synchronous-world operations on objects such as instantiation and destruction. This approach differs from ours in which reactive programming is used as a means to control concurrency in an object-oriented language.
2. With the aim of coping with non-determinism in concurrent object-oriented programs, BOUSSINOT [44] defines the "Reactive Object Language" where methods are reactive programs sharing the same clock. So methods are composed of instants (the code part between two stops). Opposite to our approach, methods are reactive code.

More recently, the Triveni language [45] has been introduced, integrating abstract behavior in an integrated approach where activity is explicitly expressed in an Esterel like language.

Triveni is a programming methodology for object-oriented concurrent programming. The key feature of these formalisms is a notion of abstract behavior, which is essentially the interaction of the system with its environment. In Triveni objects are instances of a class `Expr`. They are *processes* via the interface class `Controllable`. So a Triveni object is a process whose activity is described using an Esterel-based syntax. As "Esterel" modules are rewritten in Java, Triveni easily allows programs to be combined using Triveni constructs. If the broadcast of events required by Esterel is ensured using Java `Observer` and `Observable` classes, nothing is said about synchronous behavior. In Triveni provided actions are instances of an `Activity` class as are BOUSSINOT's methods. But whereas BOUSSINOT's work is to implement in an object-oriented way a reactive language, Triveni concentrates more on the methodology which is based on abstract behavior and that combines threads and events in a concurrent object-oriented language. Triveni is released as an application programming interface. It differs from our work in the choice we make to use an asynchronous reactive language to express abstract behaviors. So in principle it is much more similar to BOUSSINOT's work. Also the object structure is deeply dependent on the language Java [11, requirement 2, Introduction].

C. *Integration limits*

The first one is well known as the "inheritance anomaly" problem. A lot of studies have been done to cope with it. In [46] MATSUOKA, TAURA and YONEZAWA give a detailed analysis of this phenomenon. We have not studied inheritance problems but our work follows their requirements: "*states and transitions describing synchronization conditions are totally expressed outside of the methods code.*" The *Method sets* technique in [46] differs from ours in the language used. They use a declarative form language whereas we use a specific description language of transition systems. Reactive objects are similar to code servers which will accept or delay execution of methods according to their internal state and different parameters such as cooperation policy (MEYER's *immediate_service* or *normal_service* response mode [2, p. 71]). We do not present this last aspect in the paper but it allows the satisfaction of the third above property: "*Parameterize concurrency to allow an answer to requests depending not only on the internal state of the object.*"

VII. CONCLUSION AND FUTURE WORK

We have described BDL, a domain-specific language specialized to control the synchronization of concurrent objects. This high-level language has been specifically designed to schedule method executions in a concurrent object. BDL uses the concept of event to define the different steps in the method execution. The scheduling of the events is achieved using Esterel, another domain-specific language designed for this task.

BDL offers several benefits. First, because of the mathematical semantics of Esterel, and as BDL is compiled in Esterel programs, a certain number of properties on the object behavior can be verified. Second, BDL is a reactive language which allows one to preempt method executions. This feature enables the control of the execution process. BDL cannot express conditions of activation related to object attributes. This restriction is a constraint imposed by verification tools. We are now working in this direction by using Toupie [47], a constraint language working on finite domains instead of Esterel.

BDL is associated with an object model we called "*reactive objects*." This model offers intra-object concurrency managed by a special entity, the execution controller. This controller executes code compiled from a BDL control expression. The main rôle of this controller is to dissociate control code from processing code in an object. Another rôle of this controller is to make a concurrent object permanently receptive to its environment. This feature of reactive objects is important for the reliability of an application because it guarantees an answer to the object requesting a method execution. This model extends object reusability. The definition of an execution controller as a complete entity offers the possibility of modifying the behavior of an object in a quite simple manner by replacing the existing controller with a new one. This replacement must of course be made respecting the types of its inputs and outputs.

Using BDL and the reactive object model, end-user programmers can efficiently design concurrent objects. They can test objects with different scheduling policies and then verify various properties on the object behavior. Proofs insure that the initial specifications have been correctly translated. This domain-specific language-based method of development enhances the productivity of programmers compared to developments with low-level synchronization primitives, such as semaphores and monitors. They can focus on the specification of the object behavior and leave the implementation details to the BDL compiler.

Acknowledgments

This work and the first author have been supported by a grant from Conseil Régional Poitou-Charentes between 1993 and 1996.

The authors are grateful to D.Wile and T.Parslow for their comments and corrections.

References

[1] M. Tokoro, "The society of objects," in *Addendum to the Proceedings of OOPSLA'93*, Washington, D. C., Oct. 1993, ACM, pp. 3–11, Invited address.

[2] B. Meyer, "Systematic concurrent object-oriented programming," *Communications of the ACM*, vol. 36, no. 9, pp. 56–80, 1993.

[3] M. Augeraud, "A Reactive Part to Specify Dynamic Object Behavior," Indo-French workshop on object-oriented systems, Nov. 1992.

[4] F. Bertrand and M. Augeraud, "Control of object behavior: asynchronous reactive objects," in *Proceedings of International Conference on Data and Knowledge Systems for Manufacturing and Engineering*, Hong Kong, May 1994, Chinese University of Hong Kong, pp. 539–544, `http://www-l3i.univ-lr.fr/L3I/equipe/fbertran/pub/dksme94.ps.gz`.

[5] M. Papathomas, *Language design rationale and semantic framework for concurrent object-oriented programming*, Ph.D. thesis, Université de Genève, Jan. 1992, `ftp://cui.unige.ch/OO-articles/papathomasThesis.ps.Z`.

[6] K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, 1996.

[7] A. Goldberg and D. Robson, *Smalltalk 80 : The language and its implementation*, Addison-Wesley, 1988.

[8] H. E. Bal, *Programming Distributed Systems*, SIPR, first edition, 1990.

[9] P. America, "POOL-T: A Parallel Object-Oriented Language," in *Research Directions in Object-Oriented Programming*, B.D. Shriver and P. Wegner, Eds., pp. 199–220. MIT Press, Cambridge, Massachusetts, 1987.

[10] C. Tomlinson and V. Singh, "Inheritance and Synchronization with Enabled-Sets," in *Proceedings of OOPSLA'89*, New Orleans, Louisiana, Oct. 1989, pp. 103–112.

[11] C. Colby, L. Jategaonkar Jagadeesan, R. Jagadeesan, K. Läufer, and C. Puchol, "Design and Implementation of Triveni: a Process-algebraic API for Threads+Events," in *Proceedings of IEEE International Conference on Computer Languages*. 1998, IEEE Computer Press.

[12] D. Caromel, "Toward a method of object-oriented concurrent programming," *Communications of the ACM*, vol. 36, no. 9, pp. 90–102, 1993.

[13] F. Cassez and O. Roux, "Compilation of the ELECTRE reactive language into finite transition systems," *Theoretical Computer Science*, vol. 146, July 1995.

[14] A. M. D. Moreira, *Rigorous Object-Oriented Analysis*, Ph.D. thesis, University of Stirling, Department of Computing Science and Mathematics, Stirling FK9 4LA, Aug. 1994, `ftp://ftp.cs.stir.ac.uk/pub/tr/cs/1994/TR132.ps.Z`.

[15] T. Hartmann, R. Jungclaus, and G. Saake, "Aggregation in a Behavior Oriented Object Model," in *Proceedings of ECOOP'92*, O. L. Madsen, Ed., vol. 615 of *Lecture Notes in Computer Science*, pp. 57–77. Springer Verlag, 1992.

[16] C. Mac Hale, *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance*, Ph.D. thesis, University of Dublin, Department of Computer Science, Trinity College, Dublin 2, Ireland, Oct. 1994, `ftp://ftp.dsg.cs.tcd.ie/pub/doc/dsg-86.ps.gz`.

[17] N. Halbwachs, *Synchronous programming of reactive systems*, Kluwer Academic Pub., 1993.

[18] G. Berry and G. Gonthier, "The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation," *Science Of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992, `ftp://ftp-sop.inria.fr/meije/esterel/papers/BerryGonthierSCP.ps.gz`.

[19] D. Harel, "STATECHARTS: a visual formalism for complex systems," *Science Of Computer Programming*, vol. 8, pp. 231–274, June 1987.

[20] G. Berry, S. Ramesh, and R. K. Shyamasundar, "Communicating Reactive Processes," in *Proceedings of the 20^{th} ACM Conference on Principles of Programming Languages*, Charleston, South Carolina, USA, Jan. 1993, pp. 85–98, `ftp://ftp-sop.inria.fr/meije/esterel/papers/popl.ps.gz`.

[21] F. Bertrand, *Un modèle de contrôle réactif pour les langages à objets concurrents*, Ph.D. thesis, Université de La Rochelle, L3I, Avenue Marillac, 17042 La Rochelle Cedex, Jan. 1996.

[22] E. M. Sentovich, K. J. Singh, L. Lavagano, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Tech. Rep., UC Berkeley, May 1992.

[23] E. C. Cooper and R. P. Draves, C Threads, Department of Computer Science, Carnegie Mellon University, Sept. 1990.

[24] E. Sheinbrood, *The Design of the* Mach *Operating System*, Prentice Hall, Feb. 1993.

[25] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1991.

[26] L. Lamport, "What good is temporal logic ?," in *Information Processing 83*, Elsevier Science, Ed. IFIP, 1983, pp. 657–668, North Holland.

[27] L. J. Jagadeesan, C. Puchol, and J. E. von Olnhausen, "Safety Property Verification of Esterel Programs and Applications to Telecommunications Software," in *Proceedings of Conference on Computer-Aided Verification 1995*, Liège, (Belgium), July 1995, `http://www.cs.utexas.edu/users/cpg/TempEst/doc/95-CAV.ps.Z`.

[28] A. Bouali, "Xeve: an Esterel Verification Environment," Tech. Rep. 0214, INRIA, december 1997, `ftp://ftp-sop.inria.fr/meije/verif/RT-214.ps.gz`.

[29] J.P. Briot and R. Guerraoui, "Objets, parallélisme et répartition," *Technique et science informatiques*, vol. 15, no. 6, pp. 765–800, 1996.

[30] J.P. Briot, "Actalk: a Testbed for Classifying and Designing Actor Languages in Smalltalk-80 Environment," in *Proceedings of ECOOP'89*. 1989, pp. 109–129, Cambridge University Press.

[31] H. Lieberman, "Concurrent Object-Oriented Programming in Act 1," in *Object-Oriented Concurrent Programming*,

A. Yonezawa and M. Tokoro, Eds., 1987, Computer System, pp. 9–36.

[32] G. Agha and C. Hewitt, "Concurrent Programming using Actors," in *Object-oriented Concurrent Programming*, A. Yonezawa and M. Tokoro, Eds., pp. 37–53. Cambridge MIT Press, Cambridge, Massachusetts, 1987.

[33] J. Ferber and P. Carle, "Actors and Agents as Reflexive Concurrent Objects: a Mering IV Perspective," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 21, no. 6, 1991.

[34] D. Decouchant, S. Krakowiak, M. Meysembourg, M. Riveil, and X. Rousset de Pina, "A Synchronization Mechanism for Typed Objects in a Distributed System," *ACM SIGPLAN Notices*, vol. 24, no. 4, april 1989.

[35] O. Nierstrasz, "Active Objects in Hybrid," *ACM SIGPLAN Notices*, vol. 22, no. 12, pp. 243–253, Dec. 1987.

[36] P. Robert and J.-P. Verjus, "Toward autonomous descriptions of synchronization modules," in *Information Processing 77*, B. Gilchrist, Ed., pp. 981–986. North Holland Publishing Company, 1977.

[37] S. Crespi Reghizzi, G. Galli de Paratesi, and S. Genolini, "Definition of reusable concurrent software components," in *Proceedings of ECOOP'91*, Geneva, SWITZERLAND, July 1991, pp. 148–166, Springer Verlag.

[38] Roy H. Campbell and N. Haberman, *The Specification of Process Synchronization by Path Expressions*, pp. 89–102, Springer Verlag, Dec. 1973.

[39] J. van den Bos and C. Laffra, "PROCOL: A concurrent object-oriented language with protocols delegation and constraints," *Acta Informatica*, vol. 28, no. 6, pp. 511–538, 1991.

[40] F. Bertrand and M. Augeraud, "Asynchronous reactive objects: an attempt to control the object behavior," in *Poster in Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, Portland (USA), Oct. 1994, `http://www-l3i.univ-lr.fr/L3I/equipe/fbertran/pub/oopsla94.ps.gz`.

[41] D. Caromel and Y. Roudier, "Reactive Programming in Eiffel//," in *Object-Based Parallel and Distributed Computation OBPDC'95*, J-P. Briot, J-M. Geib, and A. Yonezawa, Eds. June 1995, vol. 1107 of *Lecture Notes in Computer Science*, pp. 125–147, Springer-Verlag, `http://www.etl.go.jp/etl/bunsan/~roudier/postscripts/OBPDC95.ps`.

[42] D. Coleman, F. Hayes, and S. Bear, "Introducing Objectcharts or how to use Statecharts in object-oriented design," *IEEE Transactions on Software Engineering*, vol. 18, no. 1, pp. 9–18, 1992.

[43] F. Boulanger, *Intégration de modules synchrones dans la programmation par objets*, Ph.D. thesis, Ecole Supérieure d'Electricité – Université de Paris-Sud, FRANCE, Dec. 1993.

[44] F. Boussinot, G. Doumenc, and J.B. Stephani, "Reactive Objects," Tech. Rep. RR-2664, INRIA, 2004, Route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex, FRANCE, Oct. 1995, `ftp://zenon.inria.fr/pub/rapports/RR-2664.ps`.

[45] C. Colby, L. Jategaonkar Jagadeesan, R. Jagadeesan, K. Läufer, and C. Puchol, "Objects and Concurrency in Triveni: A Telecommunication Case Study in Java," in *Proceedings of Conference on Object-Oriented Technologies and Systems (COOTS)*, 1998.

[46] S. Matsuoka, K. Taura, and A. Yonezawa, "Highly efficient and encapsulated re-use of synchronization code in concurrent object-oriented languages," in *Proceedings of OOPSLA'93*, 1993, pp. 109–126.

[47] A. Rauzy, *Toupie v. 0.26 User's Manual*, LaBRI (URA 1304), 351, Cours de la Libération, 33405 Talence Cedex, FRANCE, Oct. 1994, `http://www.LaBRI.U-Bordeaux.FR/~rauzy/MyPublications/techreport959-94.ps`.

APPENDIX

A. The Behavioral Semantics of BDL

The semantics of BDL operators is given by a set of rewriting rules. A rule has the form:

$$\frac{\text{Conditions}}{\text{Term} \xrightarrow[I]{O,L} \text{Term}'}$$

where I is the input event set (in our asynchronism hypothesis it is a singleton), O is the generated output event set and L is the local event set (not emitted outside). The reaction to a sequence of input events is computed by chaining elementary reactions.

There are two levels of inductive rules. The first level concerns the evolution of one method and these following rules are applied:

$$\frac{(e \neq \text{req}_m)}{m \xrightarrow[\{e\}]{\emptyset,\{\text{ready}_m\}} m} \qquad \textit{(ready)}$$

$$m \xrightarrow[\{\text{req}_m\}]{\{\text{start}_m\},\{\text{active}_m\}} \bar{m} \qquad \textit{(start)}$$

$$\bar{m} \xrightarrow[\{\text{req}_m\}]{\{\text{start\_failed}_m\},\{\text{active}_m\}} \bar{m} \qquad \textit{(active1)}$$

$$\frac{(e \neq \text{req}_m) \wedge (e \neq \text{term}_m)}{\bar{m} \xrightarrow[\{e\}]{\emptyset,\{\text{active}_m\}} \bar{m}} \qquad \textit{(active2)}$$

$$\bar{m} \xrightarrow[\{\text{term}_m\}]{\{\text{done}_m\},\{\text{active}_m\}} \emptyset \qquad \textit{(termination)}$$

$$\emptyset \xrightarrow[\{e\}]{\emptyset,\emptyset} \emptyset \qquad \textit{(nothing)}$$

The second level concerns the operators and the following rules are applied:

$$\frac{T \xrightarrow[\{\text{req}_m\}]{O,L} T'}{T* \xrightarrow[\{\text{req}_m\}]{O,L} T' ; T*} \qquad \textit{(repetition)}$$

$$\frac{T_1 \xrightarrow[\{e\}]{O,L} T_1'}{T_1 ; T_2 \xrightarrow[\{e\}]{O,L} T_1' ; T_2} \qquad \textit{(sequentiality1)}$$

$$\frac{T_1 \xrightarrow[\{e\}]{\emptyset,\emptyset} T_1' \quad T_2 \xrightarrow[\{e\}]{O',L'} T_2'}{T_1 ; T_2 \xrightarrow[\{e\}]{O',L'} T_2'} \qquad \textit{(sequentiality2)}$$

$$\frac{T_1 \xrightarrow[\{\text{req}_m\}]{\emptyset,L} T_1' \quad m \in T_2}{T_1 ; T_2 \xrightarrow[\{\text{req}_m\}]{\{\text{start\_failed}_m\},L} T_1' ; T_2} \qquad \textit{(sequentiality3)}$$

$$\frac{T_1 \xrightarrow[\{e\}]{O,L} T_1' \quad T_2 \xrightarrow[\{e\}]{O',L'} T_2'}{T_1 \,||\, T_2 \xrightarrow[\{e\}]{O \cup O', L \cup L'} T_1' \,||\, T_2'} \qquad \textit{(strong\_parallel1)}$$

$$\frac{T_1 \xrightarrow[\{e\}]{\emptyset,\emptyset} T_1' \quad T_2 \xrightarrow[\{e\}]{O',L'} T_2'}{T_1 \,||\, T_2 \xrightarrow[\{e\}]{O',L'} T_2'} \qquad \textit{(strong\_parallel2)}$$

$$\frac{T_1 \xrightarrow[\{e\}]{O,L} T_1' \quad T_2 \xrightarrow[\{e\}]{\emptyset,\emptyset} T_2'}{T_1 \,||\, T_2 \xrightarrow[\{e\}]{O,L} T_1'} \qquad \textit{(strong\_parallel3)}$$

$$\frac{T_1 \xrightarrow[\{\text{req}_m\}]{O,L} T_1' \quad T_2 \xrightarrow[\{\text{req}_m\}]{\emptyset,L'} T_2'}{T_1 \,|||\, T_2 \xrightarrow[\{\text{req}_m\}]{O,L \cup L'} (T_1' \,||\, T_2') \blacklozenge S_{T_1}} \qquad \textit{(weak\_parallel1)}$$

$$\frac{T_1 \xrightarrow[\{\text{req}_m\}]{\emptyset,L} T_1' \quad T_2 \xrightarrow[\{\text{req}_m\}]{O',L'} T_2'}{T_1 \,|||\, T_2 \xrightarrow[\{\text{req}_m\}]{O',L \cup L'} (T_1' \,||\, T_2') \blacklozenge S_{T_2}} \qquad \textit{(weak\_parallel2)}$$

$$\frac{T_1 \xrightarrow[\{\text{req}_m\}]{\emptyset,L} T_1' \quad T_2 \xrightarrow[\{\text{req}_m\}]{O,L'} T_2'}{(T_1 \,||\, T_2) \blacklozenge S_{T_1} \xrightarrow[\{\text{req}_m\}]{O,L \cup L'} T_1' \,||\, T_2'} \qquad \textit{(weak\_parallel3)}$$

$$\frac{T_1 \xrightarrow[\{\text{req}_m\}]{O,L} T_1' \quad T_2 \xrightarrow[\{\text{req}_m\}]{\emptyset,L'} T_2'}{(T_1 \,||\, T_2) \blacklozenge S_{T_2} \xrightarrow[\{\text{req}_m\}]{O,L \cup L'} T_1' \,||\, T_2'} \qquad \textit{(weak\_parallel4)}$$

$$\frac{T_1 \xrightarrow[\{e\}]{\emptyset,\emptyset} T_1' \quad T_2 \xrightarrow[\{e\}]{\emptyset,L'} T_2'}{(T_1 \,||\, T_2)\blacklozenge S_{T_1} \xrightarrow[\{e\}]{\emptyset,\emptyset} \emptyset} \qquad \textit{(weak\_parallel5)}$$

$$\frac{T_1 \xrightarrow[\{e\}]{\emptyset,L} T_1' \quad T_2 \xrightarrow[\{e\}]{\emptyset,\emptyset} T_2'}{(T_1 \,||\, T_2)\blacklozenge S_{T_1} \xrightarrow[\{e\}]{\emptyset,\emptyset} \emptyset} \qquad \textit{(weak\_parallel6)}$$

$$\frac{T_1 \xrightarrow[\{\mathrm{req}_m\}]{O,L} T_1' \quad T_2 \xrightarrow[\{\mathrm{req}_m\}]{\emptyset,L'} T_2'}{T_1 \,|\, T_2 \xrightarrow[\{\mathrm{req}_m\}]{O,L} T_1'} \qquad \textit{(mutex1)}$$

$$\frac{T_1 \xrightarrow[\{\mathrm{req}_m\}]{\emptyset,L} T_1' \quad T_2 \xrightarrow[\{\mathrm{req}_m\}]{O',L'} T_2'}{T_1 \,|\, T_2 \xrightarrow[\{\mathrm{req}_m\}]{O',L'} T_2'} \qquad \textit{(mutex2)}$$

$$\frac{T_1 \xrightarrow[\{e\}]{O,L} T_1' \quad T_2 \xrightarrow[\{e\}]{\emptyset,L'} T_2'}{T_1 \# T_2 \xrightarrow[\{e\}]{O,L} T_1' \# T_2'} \qquad \textit{(priority1)}$$

$$\frac{T_1 \xrightarrow[\{e\}]{\emptyset,\emptyset} T_1' \quad T_2 \xrightarrow[\{e\}]{\emptyset,L} T_2'}{T_1 \# T_2 \xrightarrow[\{e\}]{\emptyset,L} T_2'} \qquad \textit{(priority2)}$$

$$\frac{\begin{array}{c} T_1 \xrightarrow[\{\mathrm{req}_m\}]{\emptyset,L} T_1' \quad T_2 \xrightarrow[\{\mathrm{req}_m\}]{O',L'} T_2' \\ \exists \mathrm{active}_{m'} \in L \end{array}}{T_1 \# T_2 \xrightarrow[\{\mathrm{req}_m\}]{\{\mathrm{start\_failed}_m\},L} \underset{\mathrm{active}_{m'} \in L}{||} \bar{m}' \;; T_2} \qquad \textit{(priority3)}$$

$$\frac{\begin{array}{c} T_1 \xrightarrow[\{\mathrm{req}_m\}]{\emptyset,L} T_1' \quad T_2 \xrightarrow[\{\mathrm{req}_m\}]{O',L'} T_2' \\ \nexists \mathrm{active}_{m'} \in L \end{array}}{T_1 \# T_2 \xrightarrow[\{\mathrm{req}_m\}]{O,L'} T_2'} \qquad \textit{(priority4)}$$

$$\frac{T_1 \xrightarrow[\{e\}]{O,L} T_1' \quad T_2 \xrightarrow[\{e\}]{\emptyset,L'} T_2'}{T_1 \mathrm{pre} T_2 \xrightarrow[\{e\}]{O,L\cup L'} T_1' \mathrm{pre} T_2'} \qquad \textit{(preemption1)}$$

$$\frac{\begin{array}{c} T_1 \xrightarrow[\{\mathrm{req}_m\}]{\emptyset,L} T_1' \quad T_2 \xrightarrow[\{\mathrm{req}_m\}]{O',L'} T_2' \\ O''=O' \cup \left\{ \bigcup_{\mathrm{active}_{m'} \in L} \left\{\mathrm{stop}_{m'}\right\} \right\} \\ L''=L - \left\{ \bigcup_{\mathrm{active}_{m'} \in L} \left\{\mathrm{active}_{m'}\right\} \right\} \end{array}}{T_1 \mathrm{pre} T_2 \xrightarrow[\{\mathrm{req}_m\}]{O'',L''\cup L'} T_2'} \qquad \textit{(preemption2)}$$

$$\frac{T_1 \xrightarrow[\{e\}]{\emptyset,\emptyset} T_1' \quad T_2 \xrightarrow[\{e\}]{O',L'} T_2'}{T_1 / T_2 \xrightarrow[\{e\}]{O',L'} T_2'} \qquad \textit{(s\_preemption1)}$$

$$\frac{T_1 \xrightarrow[\{e\}]{\emptyset,\emptyset} T_1' \quad T_2 \xrightarrow[\{e\}]{\emptyset,L'} T_2'}{T_1 \hat{} \, T_2 \xrightarrow[\{e\}]{\emptyset,\emptyset} \emptyset} \qquad \textit{(w\_preemption1)}$$

B. Comments on The Rules

The rule *nothing* is obvious. Rule *sequentiality* allows the right term to evolve only if the left term is terminated. In *sequentiality3* one can notice that a request for a method belonging to the right term could not be serviced until the left term is terminated. Rule *strong_parallel* underlines the mandatory execution of each branch. The weak parallel operator is defined by a set of complex rules because the beginning of each branch must be memorized. To achieve this operation, a temporary term S_T is created (*weak_parallel1,2*); it disappears when the other branch starts (*weak_parallel3,4*). Keeping this information allows one to decide, when a branch terminates, if the other one must be executed or not (*weak_parallel5,6*). The rules are complex too for the priority operator. If a request occurs for a method included in the right term and if there are active methods in this term, so the termination of each active method is awaited (*priority3*). If there is not an active method then the execution could immediately start (*priority4*). The rules defining the semantics of the preemption operator are divided in two groups: common rules for weak and strong preemption and specific rules. When a request occurs for a method belonging to the right term then a set of *stop* events is generated to preempt the active methods of the left term (*preemption2*). The rules *s_preemption1* and *w_preemption1* determine the way the preemption term terminates.

Frédéric Bertrand was born in 1966. He received the Master in Computer Science in 1992 from the Ecole Centrale de Nantes, France and the Ph.D. degree in Computer Science in 1996 from the Université de La Rochelle, France. His Ph.D. work dealts with the development of a model of reactive control of concurrency in the object-oriented languages.
He is currently a Researcher at Industrial Imagery and Computer Science Laboratory, and Maître de Conférences in the Computer Science Department, Université de La Rochelle. His current research interests are concurrency and verification in the object-oriented languages and reactive languages.

Michel Augeraud was born in 1949. He received the Ph.D. degree in Computer Science in 1989 from the Université de Poitiers, France. He is currently a Researcher at Industrial Imagery and Computer Science Laboratory and Maître de Conférences in the Computer Science Department of the I.U.T. of La Rochelle. His current areas of interest includes concurrent and distributed object oriented programming. Since 1992 his research has been on reactive control of objects in a concurrent application. He is the author of several papers on specification of concurrency in object-oriented design.