The following paper was originally published in the
Proceedings of the Conference on Domain-Specific Languages
Santa Barbara, California, October 1997

BDL: A Language to Control the Behavior of Concurrent Objects

Frédéric Bertrand and Michel Augeraud

Université de La Rochelle, France

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

BDL: A Language to Control the Behavior of Concurrent Objects

Frédéric BERTRAND Michel AUGERAUD

Laboratoire d'Informatique et d'Imagerie Industrielle
Université de La Rochelle
Avenue Marillac, 17042 La Rochelle, Cedex 01, France

fbertran@mail.univ-lr.fr maugerau@mail.univ-lr.fr

Abstract

Combining concurrency and object orientation is still difficult. In an approach where methods are concurrency units, one of the main difficulties is the control of the behavior of objects.

Our proposal is BDL a language allowing to express and to achieve this control. We propose a model where each object includes a so called "execution controller" programmed with BDL. This introduces a conceptually clean separation between processing (method execution) and control. The controller ensures the respect of scheduling constraints between the executions of methods. Similarly the behavior of aggregate objects can be controlled. This language has a convenient formal base. Thus, using the expression of control, behavioral properties of an object, or even of a group of interesting objects can be verified. Our approach allows, for example, deadlock detection or verification of safety properties while the compiled object controller keeps a reasonable size.

A compiler has been implemented allowing to automatically generate the controller code from a BDL program. This compilation is achieved by producing an Esterel (reactive programming language) code from a BDL program, the Esterel compiler giving the executable code. Inter-method concurrency is implemented using lightweight processes.

1 Introduction

Through a set of mechanisms (inheritance, aggregation, prototyping) the object-oriented approach is well suited to describe complex systems. The designers of such systems have been aware of naturally concurrent applications existing (such as booking systems) which cannot be easily described with a sequential programming language [26]. So the notion of concurrent object-oriented programming has appeared naturally.

However concurrent programming is still problematic [20, p. 56]. The main problem concerns the control of concurrency, also called synchronization. This control aims at preventing the object from being in an incoherent state as, for example, when two methods concurrently executed modify attributes. A schedule of method executions is necessary to ensure that such cases do not occur.

In many concurrent object-oriented languages, the execution of methods is controlled by guards checked at runtime. This approach presents the disadvantage of mix control code and processing code. We have proposed [2, 6] an architecture for objects dissociating processings offered by the object (achieved by methods) and the control of these processings. This model gathers these conditions in a dedicated structure ensuring, for each request, that the execution is possibly related to the state of execution of the other methods. This construction improves the maintainability of the object by centralizing the execution conditions of each method in only one entity (called *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 could be used to express the control of the behavior of simple objects or group of objects. We also describe the BDL implementation based on a reactive language that allows us to verify properties on BDL programs.

In section 2, we explain the need of control for a concurrent object and we present the execution controller achieving this control. The BDL language, which allows to program this controller, is described in section 3. The semantics and the implementation of BDL are detailed in section 4. In section 5, we show how the verification

could be achieved on an object or on a group of objects. Related work is described in section 6. Finally, we conclude in section 7 by emphasizing the outlooks offered by this new approach.

2 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.

2.1 The Need of Control for a Concurrent Object

Inside a concurrent object the method execution may depend on whether other methods are or 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 same time. Their executions must be mutually exclusive. This is a constraint of scheduling.

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 only read mode (`write` method is not accessible). In this case, there are two sorts of constraints: a sequentiality constraint between the three first methods and a repetition constraint between the former and the later since the object must be permanently able to process execution requests.

To express these constraints, we have defined a set of operators representing the BDL language (see 3.1). BDL expressions use method names and operators. For example, the BDL expression specifying that instances have to execute repeatedly (indicated by `*` operator) the sequence (indicated by `;` operator) of methods `open`, `read` and `close` in this order is:

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

This specification may be refined by allowing multiple executions of `read`:

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

Now if we consider the file in read-write mode, we can specify an exclusive execution (operator "`|`") between `read` and `write`:

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

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

2.2 The Concurrent Object Model

Concurrent object-oriented languages may be classified according to three criterias [22]:

- *the object model* defining the relationship between the structures of execution (*threads*) and the object paradigm;
- *the intra-object concurrency* concerning the management of threads (number of threads, ways of creating and switching threads);
- *the interaction between objects* describing mechanisms that allow to specify the sending and the receiving of messages by objects.

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

1. defining the object as a self contained entity possessing its own executive structures;
2. designing an application as a set of communicating distributed objects;
3. improving the capacity of reaction of the object; this is achieved in two ways. Raising the degree of concurrency and therefore allowing a request to be satisfied as soon as the state of the object enables it. Moreover we give to the object a capacity to control method executions (possibility of interrupting or cancelling the execution).
4. possibility to verify the object behavior.

When describing the behavior, we imply the set of the methods of the object. Controlling the object behavior consists, in determining according to the state of the object, whether the execution of a method is authorized and, in this case, to launch this execution. If the execution is not possible, the request may be stored or rejected following a determined policy.

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 [1], Eiffel// [10] and Rosette [27]. An active object decides, according to the object state, the time when a method execution can be run. The object has a part 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 rise the capacity of service, an inter-method concurrency model has been adopted. So every method execution takes place in a thread of execution.

Finally the need of 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 verifications are 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 still the respect of 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 centralized control allows a better reuse by a clear separation between control and processing.

2.3 The Execution Controller

The controller, depicted by 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 code and thread for the execution.

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. At this time, the model of execution controller is restricted: the controller does not use object attributes to manage the executions; the activation conditions of a method that use attributes are checked in method code. This restriction is necessary to use verification tools such as FcTools which works using finite transition systems.

In fact this restriction avoids introducing numerical data in the controller. Because for checking transition systems tools use finite structure thus 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 an immediate processing, the controller stores this request. This request is still stored until the controller allows the method execution.

3 Programming Controllers With BDL

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

3.1 The operators of the BDL Language

In BDL, there are only two types: identifiers of methods and operators of scheduling. 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 [11]:

- an unary operator of *repetition*, quoted "`*`", indicates that the control specified by the term works indefinitely;
- a binary operator of *sequentiality*, quoted "`;`", indicates that the left term must be executed before the right one;
- two binary operators of *parallelism* indicates that the two terms (left and right) can be executed at same time. In that case, we consider two types of parallelism:
  - a parallelism so called *weak*, quoted "`|||`", expressing a *possibility*: the concurrent structure may be ended when a term has achieved its execution and the other has not started yet;
  - a parallelism so called *strong*, quoted "`||`", expressing a *necessity*: the concurrent structure is ended only when both terms have achieved their execution.
- a binary operator of *mutual exclusion*, quoted "`|`", indicates that executions of both terms are mutually exclusive;
- a binary operator of *priority*, quoted "#", 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 operators of *preemption* indicating that the execution of the left term can be stopped by the beginning of the execution of the right term. We consider both the following preemption types:
  - a preemption so called *weak*, quoted "`^`": the execution of the preemptive structure (right term) can take place only during the execution of the preempted structure (left term);

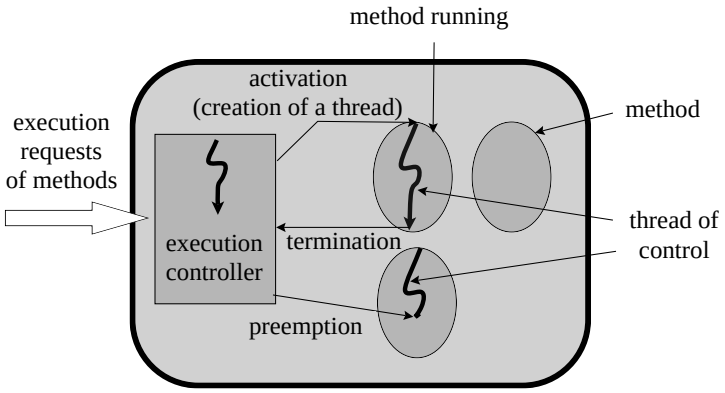

Figure 1: Architecture of a concurrent object with a execution controller

– a preemption so called *strong*, quoted "/": 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 more complex is certainly the priority operator. We will illustrate its semantics by considering the `File` object (cf. 2.1) again. The previous specification was:

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

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

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

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

BDL operators enables 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)*
```

3.2 Examples of BDL programs

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

In a first example we illustrate the need of 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 a 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 the movement along a 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 "/") 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 [21] with four functions: load a tape, play a tape, stop playing and eject a tape. The video (`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 the two objects 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 method identifiers of one of the two object produces the code of control of the other. This sort of aggregation (called *aggregation with hiding*) is conformed to the principle defined by Hartmann *et al.* [18]. This principle states that the behavior of an aggregate restricted to the methods of a component object must give the behavior of this component.

In the next section, we describe the choices that have been done for the implementation of BDL.

4 Semantics of BDL

4.1 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 quoted that, in our model, these events occur at distinct instants and are separately received (one by one) by the controller. This distinction allows 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.

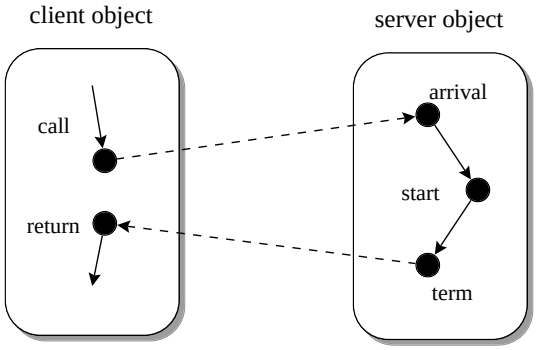

Figure 2: The events in the lifespan of a method invocation

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.

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

About the called object, the distinction between *arrival* and *start* events is important because it stands for the part played by the controller upon the execution of a method. The distinction between *start* and *term* events introduces the notion of duration for an execution. This model of execution is also present in [19] under the name SOS (Service Object Synchronisation).

4.2 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 achieve this work. For example, the BDL program

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

could be represented by the automaton of figure 3.

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

- *efficiency*: an automaton described in a programming language produces a fast executable code. This efficiency is important because this code is often executed and the object must quickly respond to a execution request;

- *proofs*: the automata are mathematical structures on which many verification tools have been developed.

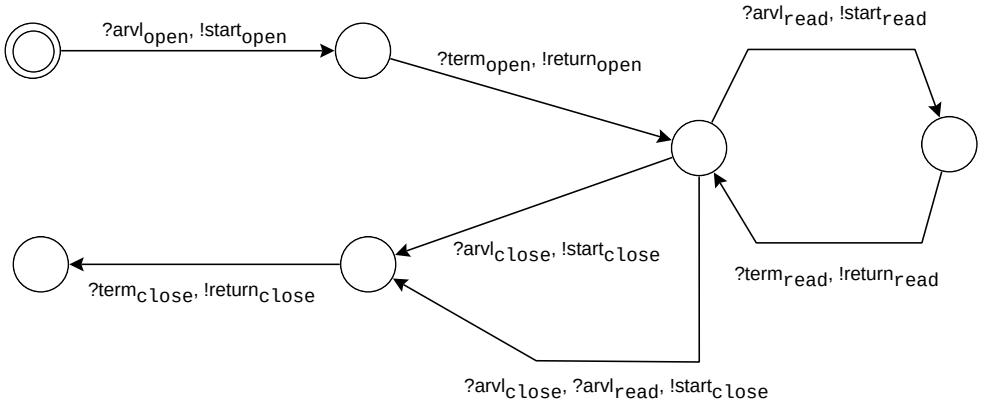

Figure 3: The (simplified) automaton of the BDL program `(open ; read* # close)`

4.3 From BDL Program to an Automaton

To produce an automaton from a BDL program is possible but a great part of this work is already done (and well done because it concerns critical systems) by a family of languages named *reactive languages*.

Reactive languages have been designed to program reactive systems. A reactive system [16] 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 develop a family of languages so called reactive among which it may be quoted Esterel [3], Statecharts [17] and Electre [11]. Due to the critical aspect of the systems implemented, these languages own a mathematical semantics allowing formal verification of properties on the behavior of these systems. The 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 describing the automaton.

In a previous section (see 4.1), 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.

4.4 The Esterel Language

We choose to use the synchronous reactive language Esterel [3] 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 with asynchronous 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 [4]. 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 unique 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 from 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.

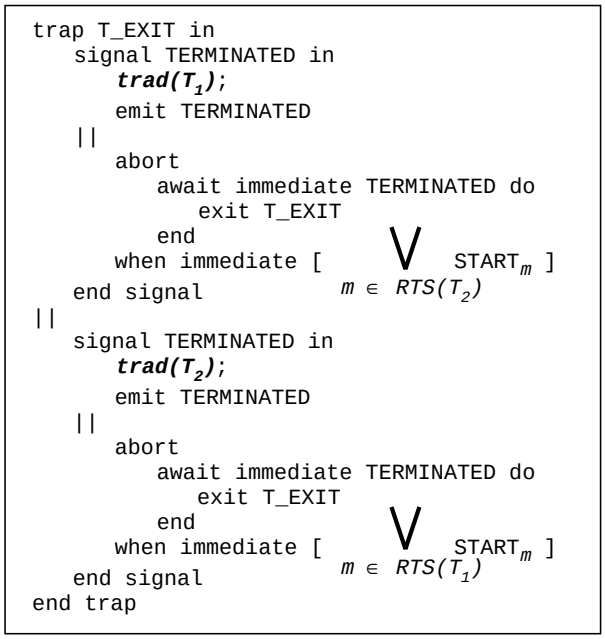

Figure 4: Semantics of BDL expression "T_1 ||| T_2" expressed with Esterel operators

The synchrony hypothesis (reaction takes no time) coupled with signal assumptions allows a rigorous processing of multiform time. Time can be handled as an ordinary signal ("clock signals"); any signal defines a particular clock.

4.5 Defining BDL Semantics from Esterel Operators

Some BDL operators have a direct equivalent Esterel operator as "*", ";", "||" operators. But the definition of "|||", "|", "#", "/", "^" operators needs a sequence of Esterel statements. The definition of the whole operators is too long for this paper, the reader will refer to [5] for the complete semantics.

We present here, as an example, the translation of the weak parallelism operator ("|||") depicted by figure 4. 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 operators).

So, in the Esterel code, at 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 Esterel preemption operator (abort ... when). We will explain the meaning of *RTS* (*Ready To Start*) set. The *RTS* set is an attribute calculated on each term T giving the name of the methods ready to be executed that are located on the left part of BDL sequential expression. For example

$$RTS(((A\ ;\ C)\ |\ (B\ ;\ D))) = \{A, B\}$$

Disjunction between the different START_*m* events corresponds to different possibilities of executions of *m* methods.

4.6 Esterel Architecture of an Execution Controller

An overview of Esterel architecture for an execution controller managing the execution of two methods (A and B) is represented on figure 5.

Each method managed by the controller needs three modules:

- a METHOD_STATE module has in charge two functions. The first one indicates the execution state at every instant by emitting (or not) the ACTIVE event. The second one to emit a CHANGE event towards BUFFER module at every method termination to request them to emit their pending requests again. Therefore a pending request can be executed only when an execution terminates.

- a BUFFER module that stores pending requests then emits them again (REQUEST emission) when receiving a CHANGE event. When a method is allowed to run, the module is informed by the reception of a START event.

- a REQUEST_HANDLER module receiving requests (ARRIVAL event corresponding to the *arrival* event on figure 2). The request is then transmitted (REQUEST) to BEHAVIOR module. If the request cannot be served then it is sent to the BUFFER module to be stored.

In the execution controller, on figure 5, there is only one BEHAVIOR module representing the decision structure. The BEHAVIOR module implements the BDL program using Esterel language. We have implemented a compiler carrying out the compilation of BDL program into Esterel code and also building the main module Esterel representing the execution controller.

The Esterel compiler allows to get a finite state automaton represented by an intermediate code (oc) that is translated in C by a postprocessor. Of course the trouble is the blow up of the size of automaton when the number of methods to control becomes important. Nevertheless, for a small number of methods, the size of the object

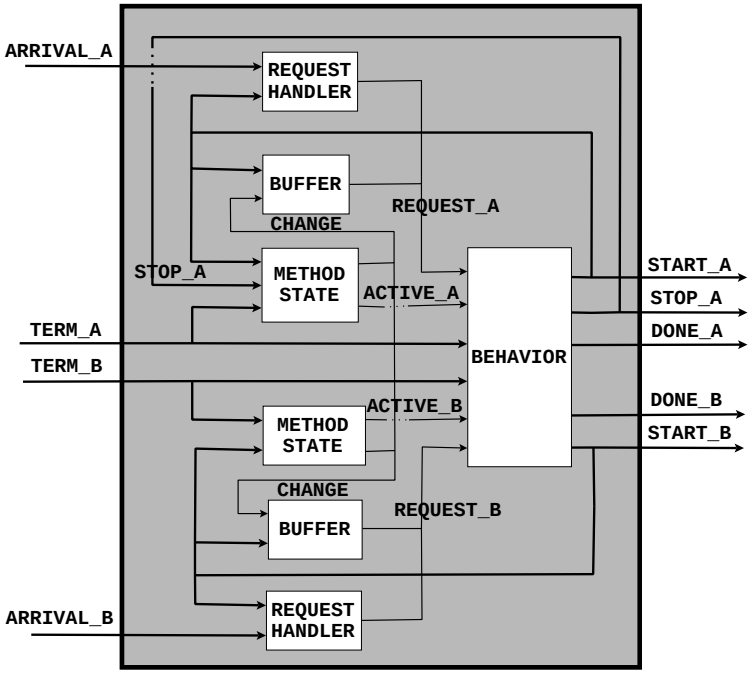

Figure 5: Esterel architecture of an execution controller managing two methods

code is not too big: for six methods managed in mutual exclusion, we have an object code of 5 Kbytes[1]. There is another alternative to reduce this size. One may use another mode of Esterel compiler that uses boolean circuits instead of automata but the verification tools working for this new data structure are under development.

4.7 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 [13] built on the Mach operating system [25]. The creation of a reactive object implicates the creation of a process holding at least a lightweight process ensuring 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++. 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.

[1]Object code obtained on a SparcStation 5 with `cc` of SunOS 4.1.4.

The implementation of reactive objects is divided into three steps. The first step consists in building the controller from a BDL program. The second one consists in the controller implementation which is obtained by linking the code of the controller to the code of the storage structure. 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 6 depicts these differents steps.

In the current version, every object has its own reactive script that is a program representing an automaton. That is trouble for the 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 reactive script for every object class. The reactive script then acts as a server 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.

5 Verification of The Control Behavior

Verification is an important step in the lifespan of an object. Object-oriented programming is concerned with reusing: an object designed for an application may be reused in other applications. To avoid an error in the

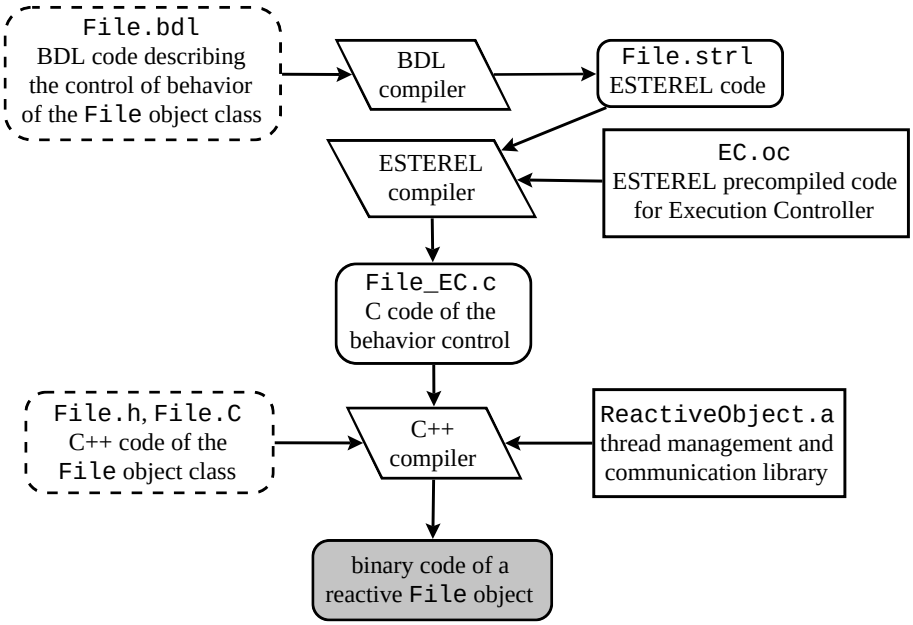

Figure 6: Development steps of a reactive `File` object

design of an object being propagated to several applications, it is important to make sure of *correct working* of this object.

In respect to correct working, 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. So we must ensure that a case (sequences of execution requests) leading to a "blocking" of the object does not exist. These blocking cases correspond to a problem frequently encountered within concurrent programming: the deadlock.

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

$$(met_1 \;||| \; met_2)*$$

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 the 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

$$(met_1 \;||| \; met_2)* =3D$$
$$(met_1 \;||| \; met_2) \;;\; (met_1 \;||| \; met_2)*$$

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[2] determinate 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 determinate 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 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 of verification by checking correctness at the object level. In assembling the simulation modules of several objects

[2]This constraint excludes obviously the mechanisms such as function pointers.

it is possible to check the correctness of the behavior of a group of objects.

Compiling a simulation module, we get an automaton on which we can verify properties with the Fc2Tools [7] verification tool developed in INRIA[3].

Fc2Tools is designed as the set of programs using the same automaton format (fc2). The internal representation of automata is carried out using binary decision diagrams (BDD) and so a more efficient representation in memory is allowed. So the limits on the size of automata to be verified are extended. If a deadlock is detected, then an event sequence leading to this deadlock is generated by the tool.

A small example is that of a micro-wave oven. This object is 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)*
```

The behavior control of the aggregate can be defined as:

```
( (open ; close) #[4] (on ; off) )*
```

But we can refine this control by precising the user to open the door while the generator is working. As this is a dangerous situation, the emission of micro-waves must be stopped:

```
          ((open ; close)
# (on ^ (open ; close) ; off))*
```

From the automaton generated by this specification, it is possible to verify the case where the generator works and the gate is open never occurs.

6 Related Work

Many works have been led to the area of the control of concurrency in the concurrent object-oriented languages. Nevertheless as far as we know few works have used a reactive model to describe the execution control in an object.

Concerning the control of concurrency in the concurrent object-oriented languages, we can distinguish four approaches:

- the mechanisms based on the *synchronization counters* developed by ROBERT and VERJUS [24] in which the execution state of methods is determinated 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 [15] or Dragoon [14]. If the approach has a great power of expression, using it is still difficult owing to its complexity.

- the *path expressions* [9] use a notation derived from regular expressions to specify synchronization constraints. We find these expressions in Procol [28]. These expressions have influenced the designers of Electre too and thus indirectly the BDL operators.

- in Eiffel// [10] there exists a special method ensuring the request processing. This method presents many similitudes with the execution controller. However our approach owns a management of the intra-object concurrency and is supported by a formal model.

- in the *enabled-sets* which has been developped with the Rosette language [27], the control is defined by defining *states of control*. For these states, a specific set of methods allowed to be executed is defined.

Concerning the works using reactive language and object, there are two main ways:

- the Objectcharts [12] which use the Statecharts. The Statecharts represent a reactive language based on a visual formalism. If this formalism is pleasant to use, the designers are not very clear about the executability of the specifications and on its implementation.

- BOUSSINOT's works [8] that consist of defining a prototype of an object-oriented language based on a synchronous execution model. In this approach, a reactive object holds attributes and methods. These last ones are "reactive agents". Opposite to our approach, these methods are reactive code and the aimed purpose is to structure a reactive program with an object-oriented approach.

7 Conclusion And Future Works

We propose BDL, a language to control the behavior of concurrent objects. This language allows to program a special entity in the object: the execution controller. Adding an execution control allows a concurrent object to make it permanently receptive to its environment. This feature of reactive objects is important for the reliability

[3] Institut National de Recherche en Informatique et Automatique – France.

[4] The priority operator "#" expressed here a notion of possibility, the working of the generator being not mandatory before open it again.

of an application because it guarantees an answer to the object requesting a service. The nature of the answer depends upon the behavioral logic of the receiver object and its state. The reactive object allows to preempt executions that allows its adaptation to the modification of its environment.

Using reactive objects is twofold: it allows intra-object concurrency management and fast adaptation to external stimuli. A reactive object offers the possibility of execution concurrently several methods with respect to a clearly expressed semantics.

This model extends the object reusability in two ways. 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 replacing must be of course made respecting the type of its inputs and outputs.

Using an automaton as target code enables us to dispose of a mathematical model upon a certain number of properties can be verified. Proofs enable us to control that the initial specification have been correctly translated.

BDL cannot express conditions of activation related to objet attributes. This restriction is a constraint imposed by verification tools. We are now working towards this way be using Toupie [23] a constraint language working on finite domains instead of Esterel.

8 Acknowledgments

The first author is supported by a grant 94/RPC-B-94-85 from Conseil Régional Poitou-Charentes.

The authors are grateful to G. LOUIS for their comments and suggestions.

References

[1] P. America. POOL-T: A parallel object-oriented language. In B.D. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 199–220. MIT Press, Cambridge, Massachusetts, 1987.

[2] M. Augeraud. A reactive part to specify dynamic objects behavior. Indo-French workshop on object-oriented systems, November 1992.

[3] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992. Postscript version available[5].

[4] G. Berry, S. Ramesh, and R. K. Shyamasundar. Communicating Reactive Processes. In *Proceedings of the 20th ACM Conference on Principles of Programming Languages*, pages 85–98, Charleston, South Carolina (USA), January 1993. Postscript version available[6].

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

[6] 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*, pages 539–544, Hong Kong, May 1994. Chinese University of Hong Kong. Postscript version available[7].

[7] A. Bouali, A. Ressouche, R. de Simone, and V. Roy. *The* FCTOOLS *Reference Manual*. INRIA / ENSMP-CMA, Sophia-Antipolis, FRANCE, 1994. Postscript version available[8].

[8] F. Boussinot, G. Doumenc, and J.B. Stephani. Reactive Objects. Technical Report RR-2664, INRIA, 2004, Route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex, FRANCE, October 1995. Postscript version available[9].

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

[10] D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

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

[12] D. Coleman, F. Hayes, and S. Bear. Introducing Objectcharts or how to use Statecharts in object-oriented design. *IEEE Transactions on Software Engineering*, 18(1):9–18, 1992.

[13] E. C. Cooper and R. P. Draves. C THREADS. Department of Computer Science, Carnegie Mellon University, September 1990.

[5] ftp://ftp-sop.inria.fr/meije/esterel/papers/BerryGonthierSCP.ps.gz

[6] ftp://ftp-sop.inria.fr/meije/esterel/papers/popl.ps.gz

[7] http://www.univ-lr.fr/Labo/L3I/L3I/equipe/fbertran/papers.html

[8] ftp://ftp-sop.inria.fr/meije/verif/primer.ps.gz

[9] ftp://zenon.inria.fr/pub/rapports/RR-2664.ps

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

[15] 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*, 24(4), april 1989.

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

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

[18] T. Hartmann, R. Jungclaus, and G. Saake. Aggregation in a Behavior Oriented Object Model. In O. L. Madsen, editor, *Proceedings of ECOOP'92*, volume 615 of *Lecture Notes in Computer Science*, pages 57–77. Springer Verlag, 1992.

[19] C. Mac Hale. *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance*. PhD thesis, University of Dublin, Department of Computer Science, Trinity College, Dublin 2, Ireland, October 1994. Postscript version available[10].

[20] B. Meyer. Systematic concurrent object-oriented programming. *Communications of the ACM*, 36(9):56–80, 1993.

[21] A. M. D. Moreira. *Rigorous Object-Oriented Analysis*. PhD thesis, University of Stirling, Department of Computing Science and Mathematics, Stirling FK9 4LA, August 1994. Postscript version available[11].

[22] M. Papathomas. *Language design rationale and semantic framework for concurrent object-oriented programming*. PhD thesis, Université de Genève, January 1992. Postscript version available[12].

[23] A. Rauzy. *Toupie v. 0.26 User's Manual*. LaBRI (URA 1304), 351, Cours de la Libération, 33405 Talence Cedex, FRANCE, October 1994. Postscript version available[13].

[24] P. Robert and J.-P. Verjus. Toward autonomous descriptions of synchronization modules. In B. Gilchrist, editor, *Information Processing 77*, pages 981–986. North Holland Publishing Company, 1977.

[25] E. Sheinbrood. *The design of the* MACH *Operating System*. Prentice Hall, February 1993.

[26] M. Tokoro. The society of objects. In *Addendum to the Proceedings of OOPSLA'93*, pages 3–11, Washington, D. C., October 1993. ACM. Invited address.

[27] C. Tomlinson and V. Singh. Inheritance and synchronization with enabled-sets. In *Proceedings of OOPSLA'89*, pages 103–112, New Orleans, Louisiana, October 1989.

[28] J. van den Bos and C. Laffra. PROCOL: A concurrent object-oriented language with protocols delegation and constraints. *Acta Informatica*, 28(6):511–538, 1991.

[10] ftp://ftp.dsg.cs.tcd.ie/pub/doc/dsg-86.ps.gz

[11] ftp://ftp.cs.stir.ac.uk/pub/tr/cs/1994/TR132.ps.Z

[12] ftp://cui.unige.ch/OO-articles/papathomasThesis.ps.Z

[13] ftp://ftp.u-bordeaux.fr/pub/Local/Info/Publications/Rapports-internes/RR-58693.ps.Z