CONTROL OF OBJECT BEHAVIOR*:  
ASYNCHRONOUS REACTIVE OBJECTS

Frédéric Bertrand       Michel Augeraud

Laboratoire d’Informatique et d’Imagerie Industrielles
Université de La Rochelle
Avenue Marillac
17042 La Rochelle Cedex 1 (FRANCE)

Abstract

In this paper, we present a model of computation combining a reactive approach and an object-oriented approach. The asynchronous reactive objects allow to describe an object by separating its actions, transformational part, from its behavior, reactive part. Through several examples, we present how, with this new object type, the behavior may be defined in a very precise way, that is, in which way the methods will be executed one in relation to each other (priority and concurrency) according to the event flow received by the object.

1 Introduction

The important development of object-oriented languages during the last few years has lead to the linkage of object and concurrency concepts to increase processing capacities. One of the major problems in the current object-oriented languages (C++ [13], EIFFEL [8], ...) is the embedding of control and of processing in methods. A separation between these two parts would allow:

- A distinct inheritance of control and of processing thus enhancing the object reusability.
- The definition of an oriented-object design method better suited to concurrent systems.

The creation of concurrent object-oriented languages (POOL [1], PROCOL [15]) has even better highlighted this ambiguity with the addition of concurrency inherent problems. Analyzing this problem brings into light two distinct problems [14]:

- The definition of an oriented-object design method better suited to concurrent systems.

- The behavior control of the object, where the definition of its different states determines the methods executable in each of its states.

Our approach is a proposal to deal with these problems. We consider an object as a conceptual method unit working according to a client/server model. Access to an object is asynchronous: two method calls from the same object cannot occur at the same moment. This property has no relation with the fact that the communication between two objects may or may not be synchronous.

Every method execution corresponds to an event pair representing the call and the end of the method execution. So, an object reacts in a continuous way to an event flow from itself, other objects, the operating system on which the objects are implemented.

The object subjected to this events flow behaves as a reactive system [11] and may be described with the help of a reactive language.

Object-oriented languages offer a great flexibility of use but that causes semantic problems. Hence program verification is difficult especially when concurrency problems have cropped up. On their side, reactive languages, with their rigorous mathematical semantics, enable the use of many proof tools. We show in this paper how the introduction of a reactive language, the ELECTRE asynchronous reactive language, may bring a greater clarity to the definition of control for the object methods.

The asynchronous reactive objects, synthesis results from reactive approach (based upon an asynchronous reactive language: ELECTRE) and object orientation, are presented in section 2, and the related works in section 3. Finally, we conclude in section 4 in evoking the viewpoints offered by this new approach.

2 Asynchronous reactive objects

Our work stands within the framework of object-oriented languages where a reactive language is used to describe the object behavior, and it continues in part from the concepts of dynamic object [3] where the usefulness of a behavioral description of an object is highlighted.

2.1 Definition of an asynchronous reactive object

An asynchronous reactive object (o.r.a.) is made up of two parts:

- A structure describing an object with its attributes and its methods, transformational part.
- A behavior, reactive part, represented by an ELECTRE program, describing the reactions of the object to the different events that it receives. This behavior is structured in two subsets:
  - A class behavior describing how an object class responds to events such as class methods calls (creation, ...).
  - An instance behavior describing the behavior of the instance object in response to events such as instance methods calls.

One can distinguish the events being received by the object according to two criteria:

- The origin that may be external or internal to the object.
- The type: method call (c_method) or information.

These different events, which may occur and affect the reactions of the objects, must be spelled out in its behavior description. The structure of an a.r.o. is summarized in Figure 1.

![Figure 1: Structure of an asynchronous reactive object (class and instance).](image)

2.2 Appropriateness of the ELECTRE language to the asynchronous reactive object model

The use of a reactive language allows us to describe the way in which the object is going to react to events (internal or external) and on what conditions. The hypothesis of asynchronism [12] is important because the method execution may have to be interrupted then resumed, and also because a method cannot receive two simultaneous calls.

From these premises, the choice of language is very simple: the one asynchronous reactive language currently available is ELECTRE. The automaton obtained, after compilation of an ELECTRE program, describes the behavior of the method execution system when it is in the presence of an environment where the activity is revealed by events. With MEC [2], a tool for analyzing and constructing automata, one can calculate state sets or transition sets that verify a number of propositions allowing to determine properties such as safety or liveness.

For a precise description of ELECTRE, the reader is referred to the formal semantics of the language [10].

---

The used notation is taken from OBJECTIVE-C, class methods are preceded by a ‘+’, instance methods by a ‘−’.
2.3 Behavior of an asynchronous reactive object

The behavior of an a. r. o. will be able to respond to the problems of:

- Priority between events and between methods.
- Inter-method concurrency.

2.3.1 Priorities management

The problem of priority of events can be solved in stating a hierarchy of efficiency for the event occurrences. From the asynchronism assumption, two event occurrences cannot occur simultaneously: that rules out any conflict of two events having the same priority. What we are interested in is to be able to specify if in a behavior phase where several events may be efficient (that is they have a direct influence on the behavior) then the occurrence of a $p$-priority event will make all the occurrences of a $q$-priority events inefficient, $q < p$.

It is possible to express this specification using ELECTRE. We note $e_i$ the events of $P_i$-priority knowing that $p_i < p_{i+1}$:

$$1/\ e_i = \ldots [[1/e_i]/e_2]/\ldots /e_n$$

If an occurrence of event $e_i$ takes place that prevents the taking into account of all event $e_j$ occurrences, $j < i$.

Concerning the execution priority of one method in relation to another, it corresponds to the fact that a method of priority level $n$ may be interrupted by the method execution request of level $n + 1$. Using the previous schema, each event is associated with a method:

$$1/\ (a_{H_1}:M_1) = \ldots [[1/a_{H_1}:M_1]/a_{H_2}:M_2]/\ldots /a_{H_n}:M_n$$

The execution priority of a method $M_i$ in comparison with a method $M_j$ where $j < i$ is explained by the fact that if $M_j$ is running and a $M_i$ occurs then $M_i$ will be preempted and $M_j$ will be executed.

The methods may therefore place themselves at different priority levels. For example, a Truck object may possess several methods (cf. Tab. 1): stop, m_forward and m_back. It is easy to understand that, for safety reasons, stop must have priority over the other methods. On another side, m_forward and m_back cannot be executed in concurrency, so, they are mutually exclusive.

Note the unnecessary preemption “↑” used to preempt the execution of m_forward and m_back. With a necessary preemption “↗”, if execution of one of the two methods has ended before any event c_stop occurs, the object would be blocked until the event occurs, that is neither useful nor desirable.

One notes also the multiple-storage qualification “♯” for events appearing in the instance behavior, so as not to lose any occurrence of these events if they could not be immediately taken into account.

<table>
<thead>
<tr>
<th>Truck Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methods</td>
</tr>
<tr>
<td>+creation { ... }</td>
</tr>
<tr>
<td>-destruction { ... }</td>
</tr>
<tr>
<td>-stop { ... }</td>
</tr>
<tr>
<td>-m_forward:(int) distance { ... }</td>
</tr>
<tr>
<td>-m_back:(int) distance { ... }</td>
</tr>
<tr>
<td>Behavior</td>
</tr>
<tr>
<td>Class</td>
</tr>
<tr>
<td>Instance</td>
</tr>
</tbody>
</table>

Table 1: The Truck class.

Another classic example from the real-time field, the “watchdog”, may illustrate this first point. A “watchdog” is an alarm mechanism that allows the temporal control over the right progress of a task. If the execution of task exceeds the allotted time, then this must be suspended and a special process will be carried out. This system is composed of two objects (cf. Tab. 2,3), the object effecting the task which must be watched (Watched object), and the watching object (Watchdog).

Note the fleeting property “@” of the event e for it will not be stored. That because if e occurs after the execution end of compute and it can be stored, then as soon as a new call to compute takes place, this cannot be executed because it will be immediately interrupted by the stored occurrence e. This represents a semantic error because this occurrence of e concerned the previous execution of compute. So, for this reason, the event e must not be storable. Note also the initial restart property “>” of compute which specifies that if it is interrupted, it will have to resume the execution at its beginning.
### Table 2: The WatchedObject class.

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>WatchedObject</td>
<td>Attributes: <code>evt e;</code></td>
</tr>
<tr>
<td></td>
<td>Methods: <code>+creation {...)</code> <code>destruction {...}</code> <code>compute: (int) data : (id) aWDg : (int) delay</code></td>
</tr>
<tr>
<td></td>
<td><code>int res;</code> <code>aWDg init: delay];</code> <code>res = ...;</code> <code>return (res);</code></td>
</tr>
<tr>
<td></td>
<td><code>-exception { return (e); }</code></td>
</tr>
</tbody>
</table>

### Table 3: The Watchdog class.

<table>
<thead>
<tr>
<th>Class</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>Watchdog</td>
<td>Attributes: <code>evt e;</code></td>
</tr>
<tr>
<td></td>
<td>Methods: <code>+creation {...}</code> <code>destruction {...}</code> <code>init: (int) delay</code></td>
</tr>
<tr>
<td></td>
<td><code>wait(delay);</code> <code>return (e);</code></td>
</tr>
<tr>
<td></td>
<td><code>exception { return (e); }</code></td>
</tr>
</tbody>
</table>

### 2.3.2 Concurrency management

According to the sort of object, methods could execute concurrently within the same object. This is illustrated by the classic example of readers-writers where several readers can read simultaneously, as opposed to writers, who can write in mutual exclusion when they access to the buffer. This is described by the following specification:\(^2\)

\[
[[1/\{\prod_{i=1}^{n} c_{\text{read}_i} : !\text{read}_i\}]\uparrow\{c_{\text{write}} : \text{write}\}*.
\]

The symbol "\(\prod\)" means that the parallel structure can completed when the execution of branches, *having started*, has finished.

If the previous behavior is adapted to a bounded buffer where three methods are available: *put, get, count_items*, it may be explained:

\[
[[1/\{\prod_{i=1}^{n} \#c_{\text{count_items}_i} : !\text{count_items}_i\}]\uparrow\{\#c_{\text{put}} : \text{put} \mid \#c_{\text{get}} : \text{get}\}*.
\]

One should note here that we are interested, at the behavior level, uniquely in the events. The tests having to be performed (to test if the buffer is not empty before removing an item, if it is full before adding item, ...) at the beginning of the method body, that poses no problem since the mutual exclusion provided by the behavior.

---

\(^2\)Some notations used in the ELECTRE programs do not belong to the syntactic kernel of the language and they are only used for writing convenience. They are equivalent to the mathematic notations. For instance, \(\{ \prod_{i=1}^{n} c_{\text{read}_i} : \text{read}_i \} = \{ c_{\text{read}_1} : \text{read}_1 \mid c_{\text{read}_2} : \text{read}_2 \mid ... \mid c_{\text{read}_n} : \text{read}_n \} \)
2.4 Functional description of an asynchronous reactive object

Having described structural description of the objects by defining their attributes and their methods, and after their description of behavior with ELECTRE, it is interesting to describe the a. r. o. in a functional way to get a comprehensive view of a system. The a. r. o. may be compared to components whose interconnection allows the building of a system. In the same way as an electronic component has a distinct control part and processing part (different pins), a. r. o. have a reactive part providing the control (which, in our graphic notation, is represented by the top part of the object) and a transformational part providing the processing (bottom part).

Using the watchdog example again, we obtain a component that consists of two cooperating subcomponents (cf. Fig. 2). The connections stand for communication links where events or data pass. The inputs $D_{\text{method}}$ (Data) represent the channel by which the parameters needed for the method execution enter, $R_{\text{method}}$ (Results) represent the channel by which the results of method execution exit. For instance, the parameters (data, aWDg, delay) of compute are transferred from the input 2 of the global component to the input 3 of the Watched object.

These notions are closely akin to plug compatibility concept described in [9] and to signatures composed of a method names set, with their arguments and their types and moreover, in our case, with the events received and emitted by the object.

![Figure 2: Functional description of a watched compute system.](image_url)

3 Related works

The notion of reactive object was introduced in [6] from a combination C++ and REACTIVE-C languages. REACTIVE-C is a reactive language build as an extension of C where functions and procedures allow management of behavior according to event reception or event emission. The reactive model used is ESTEREL’s [4] that is synchronous thus these objects are described as synchronous reactive objects.

As far as we concerned, there are several points in our approach which are different:

- The type of the reactive model used is, in our case, asynchronous because we consider, in an object-oriented language, the execution of methods takes time.
- The embedding of control and processing that seems incompatible with inheritance and composition problems hence their separation.
- The communication model used that is, in our case, object to object contrary to the REACTIVE-C model where the broadcasting mechanism is used. This has the advantage of allowing the addition of new objects to the system without modification of existing objects, but presents the major drawback of bringing us semantic problems (in particular, problems of indeterminism).

Other work has been also carried out to model the object behavior with the help of a particular type of a reactive language: the statecharts [7]. This has led to the term objectcharts [5]. While they have the same appeal of the “visual formalism” of statecharts, they present the disadvantage of dealing only with static systems of objects.
4 Conclusion

The marriage between concurrent computation and object-oriented programming is not easy because many problems (concurrency management, priority management, . . . ) are not yet resolved in a satisfactory way.

The use of a reactive language is a way of responding to these problems. The a. r. o. allows the simplification of the specification separating transformation and reaction. It is able to express locally in an object the control of its behavior and concurrency. The functional model that we present here reveals the necessary exchanges to dialog between objects for cooperation. An object appears as a subsystem that offers one interface for cooperating management and another one for application services.

As well as a greater clarity of specification, this approach offers many possibilities, as the expression of execution priorities and the possibility to express explicitly the concurrency management.

The behavior correctness of each object may be proved using proof tools on automaton. This possibility of validation is due to the use of a reactive language to describe the behavior. An object validated manner appears as a reliable software component. Important work of formalization still remains to be done, inheritance as well as object composition, and the design method must be linked to this approach.

Acknowledgments

The first author is supported by a grant from the Conseil Régional Poitou-Charentes. The authors wish to thank Olivier Roux and Georges Louis for reading and commenting on the manuscript version of this paper and Nadine Eboueya, Constance Fleuriot and Stuart McLellan for the English version of this paper.

References