An Event-Driven Programming Paradigm Compatible with OO-Programming

C. Petitpierre
Laboratoire de téléinformatique
Ecole polytechnique fédérale de Lausanne (EPFL)

Keywords: multithreading, event driven programming, object-oriented programming, program inversion, Java, Motif, sC++.

Abstract: This paper discusses several possibilities of implementing the behavior of an object-oriented application. It assesses the use of the callback or inverse-control programming paradigm, and opposes it to a direct programming style. The discussion is illustrated with two examples of the common approach, namely a Motif program using its main event loop, and a Java program using active objects. The paper then presents the direct approach, based on a limited extension to C++ that is close to Java, but avoids some limitations of its synchronization means. Finally, the paper shows how to model a behavior described by the new approach.

1. Event-Driven Programming

The much tooted object-oriented programming paradigm has been very successful at the description of data structures, and the operations they can be applied to. Actually, the concepts this paradigm defines, such as inheritance, aggregation, relationships, etc., are very well suited to represent data dependencies and storage structures, as well as to define the interfaces of data operations in an implementation independent way. However, every application also needs an engine that decides what operations to execute on which object, and that activates the right operations at the right times. This engine is usually triggered by events produced by a human clicking on a human-computer interface, by messages coming from the network, or even by the components of the application themselves. This engine must thus manage several sources of events, and maintain the information that relates each event to a specific operation on a specific object.

If an application does not exerts any dependencies between the successive events, its implementation is straightforward: Every event activates a specific function or object method, a concept called callback, because it calls back the functions indicated by the application. However, even if the first versions of an application can be implemented in this way, it happens often that new functionality is requested afterward, and that the application cannot be extended easily without breaking this constraint.

Figure 1 shows a typical architecture that implements the callback concept. All events are delivered by the underlying operating system to an event dispatcher. At the initialization of the program, the dispatcher receives the information that relates each event to a given entry point, i.e., a function or an object method. In order to hide the idiosyncrasies of the underlying system, an intermediate level of objects, usually called wrappers, helps to structure the application [1].

These wrappers may contain a description of the behavior of the application. They know the format of the operations, as well as the sequences in which they must be called. If some event triggers different operations, possibly on different objects, as a consequence of the events previously handled, the corresponding wrapper may also store some state that keeps track of what has happened earlier.

When an event occurs, the dispatcher calls the corresponding wrapper, which calls in turn some function or object method. Of course, each

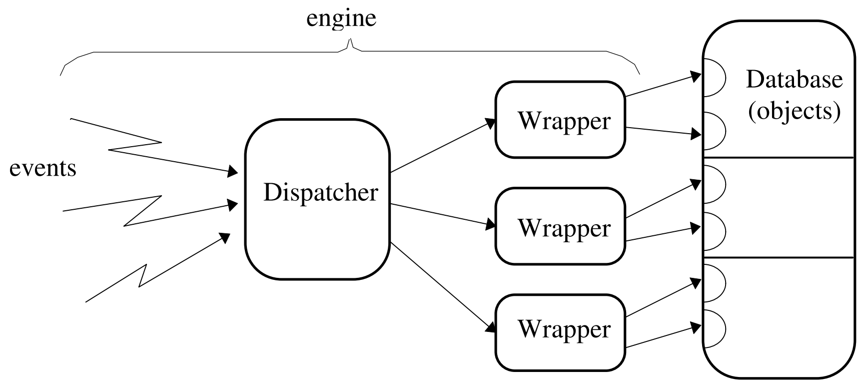

Figure 1. Architecture of an event driven application.

operation must have returned to the dispatcher before the latter can service the following event. The execution paths determine thus a tree rooted at the dispatcher, with the operations of the objects at the leaves.

This approach sticks to the OO-development paradigm: The data structure is laid down in the database, and the behavior is added as an external engine (often called *glue* !) orthogonal to the description of the operations.

Unfortunately, this structure is not adapted to the cases where the components at the leaves of the previously mentioned tree must exchange information. It is not that data cannot flow from one leaf to the other, but this architecture needs the involvement of the wrappers to trigger the execution of the leaves that have received data, which breaks the modularity of the program. It is not possible either to have some object at a leaf access a remote database, for example, by means of some well encapsulated communication protocol, as the latter is heavily dependent on the network events and on time-outs, which must also transit through the dispatcher (it might even be unprepared to handle the network events). Finally, this architecture stores the data structure and the behavior at two different places, which makes reuse more difficult.

In the first two next sections, we will use two common examples, namely a toy application embedded in the Motif framework, and another one coded in Java, to show how to embed the behavior into the objects themselves. The first example will emulate a multitasking program by control-inversion, and the second one will take advantage of the active objects offered by Java to support the behavior. The article then proposes a new approach that is more modular than the previous ones in that it avoids the crossed exchanges of information between the application and the wrappers, and thus allows a better separation between the operating system and the application. The paper shows how this approach also alleviates some constraints of the Java synchronization means, pointed out in the following sections, and how it supports behavior modeling and analysis. Finally, it shortly presents a development environment that supports the advocated methodology.

2. The Motif Callback Loop

This first example illustrates the use of Motif, the well known human-computer interface library. This application has been chosen, because it exerts a dependency between events, i.e., the handling of an event is a function of the events that have previously occurred. Its goal is to allow a person to type in a name, and thereafter any series of numbers, until s/he types in a dot at the beginning of the line. When this happens, the program repeats the sequence, requesting a new name and a new series of numbers. As the names and the numbers come from the same window, the application must always memorize if it is in a state where it reads a name or a number. It could

```
1int main (int argc, char *argv[]) {

    2       Widget  rowcol, txt, text_w;
    3       void handle_event();
    4       txt = XtVaCreateManagedWidget ("Text", xmTextWidgetClass,
    5                             rowcol, XmNvalue, NULL, NULL);
    6       XmTextSetString(txt, "Give a name: ");
    7       XtAddCallback(txt, XmNactivateCallback, handle_event, "Text");
    8       XtAppMainLoop(app);
    9     }

//   An Inverse-Control Task  (in a Callback Function)

   10     int state=0;
   11     char *name, *value;

   12     void handle_event( Widget widget, XtPointer client_data,
                     XtPointer call_data) {
   13       switch (state) {
   14       case 0:
   15         name = XmTextGetString (widget);
   16         XmTextSetString(widget, "Give a number: ");
   17         XmTextSetInsertionPosition(widget, 15);
   18         state = 1;
   19         break;
   20       case 1:
   21         value = XmTextGetString (widget);
   22         displayTxt(name, value);
   23         if (value[15]=='.') {
   24           state = 0;
   25           XmTextSetString(widget, "Give a name: ");
   26         } else {
   27           XmTextSetString(widget, "Give a number: ");
   28         }
   29       }
   30     }
```

be argued that this application would be clearer for the person at the keyboard, if it would use different fields to get the names and the numbers. However, the real purpose does not matter, and a useful task that really needs this behavior would be too complex to illustrate our point. Another remark is that this implementation is not object oriented. As a matter of fact, it can be easily completed with wrappers that relay the events to a set of objects. This point will also resurface later on.

The listing of the program, given above, shows only the important statements, explained in the following. The function *main* initializes the windows, and displays the first request to the user (lines 4 to 6). Line 7 registers the function *handle_event*, predefined on line 3, and completely defined from line 12 on. It tells the main loop that it must be called whenever the new line key is typed in the text field, which is referred to by the parameter "Text", appearing in the widget (line 4) and in the registration (line 7). Finally, the function *main* enters an infinite loop (line 8), waiting for the events that have been registered.

The code of function *handle_event* is placed in the two cases of a switch statement (line 13). Whenever an event is generated by a user typing the new line key in the window, the function *handle_event* is executed. The first time it is entered, the integer *state* (line 10) is equal to 0, and thus the case beginning at line 15 is executed. The value of *state* is changed inside the function, so that the next time it is called, the program jumps to case 1, starting at line 21. The value of *state* is set back to 0 when the user enters a dot on the window (lines 23 and 24).

The use of the switch statement provides the function *handle_event* with the characteristics of a task, with variable *state* functioning like a program counter. Note that it is mandatory to declare *state* outside the function, in the global area, to have it keep its value from one call to the other. Similarly, all values that must last from run to run, must be stored globally (line 11). This situation is improved if the function is replaced by a method within an object, which retains its attributes between the calls to its methods. However, in this case, the object must define (inherit) a standard method that is known a priori by the dispatcher, in our case the main loop, although an intermediate user-designed wrapper can alleviate this by translating the default name to the various operations requested by the object. Finally, if different events must be handled by the same function, all these events must call that function, and be sorted out within the function.

Note that it would have been possible to place the two cases of function *handle_event* in two different functions, and change the identification of the function that must be called from the main loop each time it is necessary. Applications are often build in that way, but the technique proposed here reveals more clearly the behavior of the application. It is much closer to a finite state machine, and creates implementations that are closer to the specifications. The transformation from a specification, expressed as a task that awaits events, to the construction shown here has been dubbed "program inversion" by Jackson [2] in the seventies.

Several such tasks could be prepared, and thus replace a multitasking environment. However, if data have to be exchanged between two tasks, communication means should be build from scratch. This is complex, and anyway needs a high-level concept, which actually could be derived from the ideas discussed in the following sections.

3. The Java Approach

The Java example relies on the multi-threading capability of this language. In Java, the window events are forwarded to methods defined within objects, which provides a much more comfortable storage facility for lasting data, because the attributes of the objects retain their values from call to call. Moreover, in Java, a thread can run in an object concurrently with the other activities of the application. This thread can wait for a signal sent from a method triggered by an event, and step through its code at each new event occurrence. The behavior of the object can thus be encoded within the flow control statements. Of course, the methods that send signals can also be called by the other threads.

The object instanciated from class *Input* below is inserted in a window frame (not shown in the program), which automatically forwards the events that occur in its scope to method *action* of the object. We only show here the two critical methods (lines 4 and 10 below). They are preceded by the keyword *synchronized*, which specifies that only one of these methods can run in the object at any one time. This keyword is also mandatory here, because the methods contain synchronization statements (lines 7, 15 and 21). Method *run()* (line 10), whose name is standard, is started on a new thread after the object has been instanciated. It contains an infinite loop, and thus never terminates. It suspends its execution periodically on *wait()* statements (lines 15 and 21), to await each following event. The insertion of the *wait()* statement within a *try* and *catch* statement is requested by the language. It allows the program to get a signal if the rescheduling is owing to a reason different from the event, but this statement can be ignored in this paper.

When method *action* of line 4 is triggered by the user typing the new line key, the method stores the string written in the window into attribute

```
1    public class Input extends Applet implements Runnable{
2      private TextArea area1, area2;

3      String buff;  // ! could well be overwritten !

4      synchronized public boolean action(Event aEvent, Object arg) {
5        area1.selectAll();
6        buff = new String(area1.getSelectedText());
7        notify();
8        return true;
9      }

10     synchronized public void run () {
11       String name;
12       boolean lop;
13       for (;;) {
14         area1.setText("Give a name:");
15         try { wait(); }
16         catch(InterruptedException e) {}
17         name = new String(buff);
18         lop = true;
19         while (lop) {
20           area1.setText("Give a number:");
21           try { wait(); }
22           catch(InterruptedException e) {}
23           if (buff.equals(".")) {
24             lop=false;
25           } else {  // buff is assumed to contain a number
26             area3.appendText(name+":  "+buff+"\n");
27           }
28         }
29       }
30   }
```

buff of the object (line 6), and it executes the statement *notify()* (line 7). The latter reactivates method *run()*, which can thus process the data signaled by the event. This program treats a name or a number depending on the position of its program counter. There is no need of flags any more, which makes the program clearer, not only for the engineer who writes the program, but also for the one who has to update it later on.

If the system can assure that after the *notify()* statement, method *run*, protected by the *synchronized* keyword inside the object, gets priority over any other external call (which may depend on objects priorities), the string delivered to *buff* by the event will be consumed by method *run* before the next event calls method *action* again. Otherwise, a previous unconsummated string could be overwritten and lost.

This solution makes the implementation of a behavior defined by finite state machines very easy, and one does not need to resort to the inversion technique mentioned in the first example. Inter-process communications can be implemented between active objects with the *wait-notify* pair, exactly in the same way as between the event dispatcher and the active object described above. However, beside the risk mentioned above, there are some other limitations imposed by the synchronization scheme of Java, which will be discussed in the following section.

4. Hoare Monitor

The history of concurrency has seen quite a lot of synchronization primitives, which were increasingly reliable and well adapted to concurrent applications. Many scientists have been worried by the risk of forgetting some statements, such as a *notify* call somewhere, and have thus put emphasis on constructions well integrated in some programming language. Actually, synchronization errors can lead to erratic behavior, which is very difficult to analyze and reproduce, because these errors may depend on changing timing conditions. A missing statement is quickly found, but it has been shown with the example of the overwritten message that the errors can be much more subtle than a simple command oversight.

The Hoare monitor [3], which has inspired the Java concept, proceeds of this research, and offers a mature environment. The Hoare monitor allows a process to wait for some signal from another process, and to get this signal in a tight manner, i.e., without any risk of a third party intervening between the emission and the reception of the signal. It clearly specifies that the receiving process (the one that executes *wait()*), has priority over any other one, within the object. The risk of overwriting the previous message that was feared in the Java example is thus avoided in this case.

However, in the following situations, the Hoare monitor is not the definitive solution. Let us first consider a process that must handle two sources of information, and does not know which one will provide an event next. It is not possible to wait simultaneously within two different Hoare monitors, each one being attached to a source of events. Considering another situation, let us assume that an object must get data through a Hoare monitor, but keep an eye on the possible occurrence of errors from components it has to manage, such as time-outs, broken communication link, or any device interrupt. As the monitor blocks the caller, it prevents it to handle these events before the monitor releases the caller.

It is actually possible to start extra threads to assure the required concurrency, but these threads need to be synchronized with the initial thread, and the monitor is thus only a partial solution.

Finally, if one needs to exchange signals in both directions between two monitors, the monitors must call each other, and use flags to keep track of the previous calls if they were made while the receiver was just not awaiting them. However, the use of flags is very delicate, and an error in their management can be very difficult to identify. Moreover, this approach produces cross references between the objects, as each object must call the other one, which prevents a careful encapsulation of the functions. All these characteristics imply that it is difficult to build a regular multi-layered architecture in a bottom up manner, where each successive layer uses the same scheme to communicate with its neighbors.

5. sC++ (Synchronous C++)

sC++ defines active objects that are similar to Java's, by extending C++ in a straightforward fashion that remains fully compatible with all C++ libraries. sC++ active classes just have the keyword ***active*** in front of the keyword *class* (lines 101 or 201 below), which, unlike Java's active objects, leaves all the inheritance syntax untouched. An sC++ active object is defined, created, referenced, called, deleted, and inherited with the same syntax as a passive (usual) object. Each public method of an active object is automatically protected in such a way that only one method at a time can enter the active object. An active object possesses an internal thread that runs a specific method, called the body, that has the name of the class, preceded by @ (lines 107 or 203). The syntax of its definition is thus close to the syntax of a destructor. The body is activated just after the constructor returns to the object that has instanciated the new object.

An object calls another one at line 206 below. The call is not executed as long as another method is running within the called object. Actually, the body has the same execution constraints as the methods, and thus no other method can be executed while the body is

```
101    active class Class1 {
102    public:
103      void send (MSG &msg) {
104        .....;
105      }
106    private:
107      @Class1 () {
108        for (;;) {
109          accept send;
110          actionsA();
111        }
112      }
113    } obj1;
```

```
201    active class Class2 {
202    private:
203      @Class1 () {
204        for (;;) {
205          actionsB();
              obj1.send(xxx);
207        }
208      }
209    } obj2;
```

executing. In order to give free way to calls, the body can terminate, in which case the methods are still executed in mutual exclusion of each other, or it can ***accept*** a specific method, identified by its name (line 109). If the body arrives at the *accept* statement before the caller, it is suspended until the caller executes the method, and conversely, if the caller arrives first, it is suspended until the callee accepts the method execution. There is thus a rendezvous between both objects (Fig. 2).

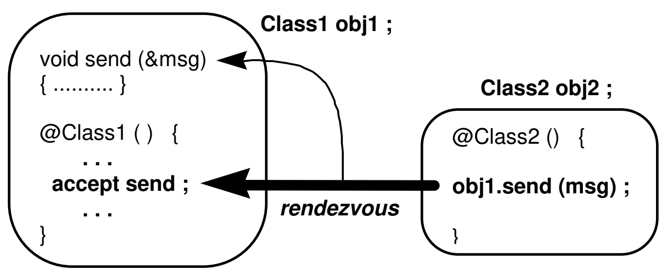

Figure 2. Inter-object communication.

Of course, it may be required that the receiver awaits several calls concurrently. This can be realized by the ***select*** statements shown below. A sequence of select statements that accept various operations can determine several possible ordering of the operations, provided that this order is compatible with the order required by the callers. This point will be discussed later on. The select on the right shows that it is possible to mix calls with accepts and delay statements in the same selection.

These *accept* and *select* statements are close to the corresponding statements provided by Ada. The left case below maps directly onto the Ada concept. However, Ada can contain only *accept* (and delay) statements, no calls, which raises the same problem as the Hoare monitor. Consider

```
select {
  accept operation1;
    .....
||
  accept operation2;
    .....
||
  accept operation3;
    .....
}
```

```
select {
  accept method;
    .....
||
  obj1->methodx();
    .....
||
  waituntil(now()+100);
    .....
}
```

two active objects that must exchange messages. Obviously, one of the object must contain a call, and the other one an acceptance. Thus, as it is impossible to put the call in a *select* statement, it is impossible to have both objects await external events in parallel with the waiting of their next exchange. The possibility to have both *accept* and *call* statements in a *select* (lines 201 to 207) with sC++ avoids this problem.

The Ada proponents have refrained from allowing calls within a *select* statement, because it was thought that Ada would have to run on multiprocessors, where it is rather difficult to realize rendezvous, but there is a need for an efficient language for programming interactive mono-processor application, which is not filled by Ada.

A big difference between a Hoare monitor and our proposition is the place where the object can be suspended. As already mentioned, it is impossible to waiting for events external to the monitor while waiting within a Hoare monitor, because the thread of the caller is blocked. In sC++, as the thread is blocked ***before*** it enters the object, it is possible to await events from several sources simultaneously. The first one that is ready accepts the method, which restarts the caller. The waiting thread can thus step through its code down to the waiting of the next events. The risks of overwriting a value is completely avoided with this approach too. In the code below, the caller is obviously not accepted before the body is ready to accept the next call (line 10), even if it takes some time to process *x* (line 11).

```
 1    active class Server {
 2      text x;
 3    public:
 4      void deposit (text xx) {
 5        x = xx;
 6      }
 7    private:
 8      @Server () {
 9        for (;;) {
10          accept deposit;
11          process(x);
12        }
13      }
14    }
```

6. Direct Programming

As previously discussed, Motif imposes an inverse-control programming style that is not very efficient. Java has improved this situation. It allows the embedding of the behavior description into the flow control statements, but it still links the events to callback methods. sC++ goes one step further by allowing the program both to instanciate, and to read the objects that support the devices, which makes it possible to develop a library that contains active objects managing buttons, text fields, menus, network sockets, etc., and offering methods that can be called by the application, concurrently if needed, to read the display, write to it, and to manage all kinds of device.

For example, a button can define a method *Pressed()* that is accepted only when the user has clicked on the button icon. A text area may define a method *ReadText()* that is accepted only when the user has pressed the new line key. In that way, these methods are blocking exactly like the usual keyboard read function. Of course, only the object that reads the text area is blocked, as the other objects may run in parallel. Moreover, it is possible to wait for several sources of events simultaneously within a select statement. Actually, we have created an interface library that encapsulate most Motif primitives.

The listings below shows an implementation of the example that has been coded with Motif and Java with this new approach. The first lines show the creation of the window elements.

```
1   RowColumn rc(gui,"Title");
2   RowColumn row(rc, 1,
       GUIOrientation::HORIZONTAL);
3   TextField area1(row);
4   TextField area2(row);
5   Text area3(rc,8,30);
6   GUIRealizer g(rc);
```

Line 1 generates the main window frame (*gui* is the active window monitor). It contains a column that contains in turn two objects: A row (line 2) of text areas (lines 3 and 4), and a third text area (line 5). Line 6 calls the library to tell it to build

```
int lop;
CharString name, buff;
void main () {
  for (;;) {
    lop=1;
    area1.ReplaceText("Give a name: ");
    area2.Clear();
    area2.ReadText(name);
    while (lop) {
      area1.ReplaceText("Give a number:");
      area2.Clear();
      area2.ReadText(buff);
      if (!strncmp((const char*)buff,".",1))
        lop=0;
      else {
        area3.AppendText(name);
        area3.AppendText((const char*)buff);
        area3.AppendText("\n");
      }
    }
  }
```

the window, and have it appear on the display.

The program itself is written like any sequential program. It uses no flags, and the synchronization means are completely hidden. Of course this code could have been stored in an active object as well.

There are two reasons to call this program direct. Firstly, the device objects are both instanciated and accessed by the application, and secondly, the context switching is specified within the flow control, and executed by the underlying kernel.

The following lines present another program that uses the same approach. This program manages several devices. The set of events it recognizes varies according to its execution state. The objects defined by *TCPSocket*, *Text* and *PushButton* (line 1-4), whose role should be obvious, do not accept their methods, when they are called by the program (lines 7, 9 and 11), as long as, respectively, the corresponding button has not been depressed, the text field has not been filled with a text terminated by a line return, or the socket has not received any data.

As already mentioned, the program can be decomposed into independent active objects and use the select statement to await several events in parallel. With this approach, every programmer can implement functions that act like the keyboard read function used in earlier programs, and then concentrate on the application, not on

```
 1    TCPSocket sd = new TCPSocket ("destination", portno);
 2    RowColumn rc = new RowColumn (gui, "Title");
 3    Text txt = new Text(rc);
 4    PushButton button = new PushButton (rc, "Click here");
 5    void main () {
 6      for (;;) {
 7        button1->Pressed();
 8        select {
 9          txt->Read(buffer);
10          handle(buffer);
11        ||
12          sd->Recv(message);
13          handle(message);
14        }
15      }
16    }
```

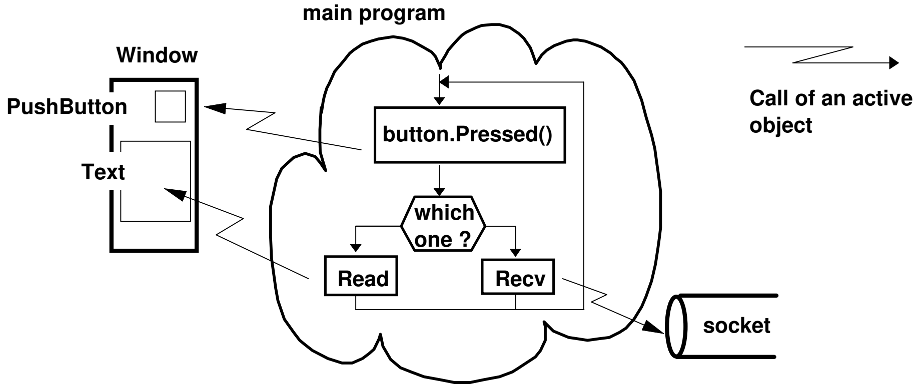

Figure 3. A direct program managing active objects.

the implementation of a communication infrastructure.

Figure 3 gives a diagram of the program. The reader may imagine how to inverse this program to implement the form used in the Motif example.

The problems mentioned previously, namely reading several sources without knowing which one will react first, awaiting a communication with an object while managing devices that may produce errors, homogeneously synchronizing several task interconnected within a mesh, and avoiding overwriting have found a solution. The Motif and Java examples have been specified and implemented from a direct view point, in which the application is the master of the devices providing the events.

The description of the object behavior, inserted in the body is orthogonal to the description of the operations, and it needs not be known from the outside. The components can thus be embedded in a communication mesh (it now really corresponds to a *glue* !), expressing the application dataflow. The global behavior determined by the kernel, automatically corresponds to the intersection of all local behaviors. Of course, it may happen that this intersection is empty, in which case the application would be in a deadlock, but unlike with the previous methods, this can be analyzed easily with analysis and modeling techniques, as explained in one of the following sections.

7. Implementing the Hoare Monitor in sC++

Although our approach also draws from the Hoare monitor, it is not possible to map it to the elements of sC++ directly. In sC++, a method can actually be put on hold while awaiting another method by means of the accept statement. The new method can accept another call in turn, and thus several calls can be nested. However, the methods can only be restarted in reverse order of their blocking. It is thus not possible to pass the control of the processor from a calling method to the waiting method, as is requested by the Hoare monitor.

We have developed an sC++ library that implements the Hoare monitor exactly, and we have evaluated ways to implement it directly into the language. However, the Hoare monitor also has its weaknesses, as has been explained above, and it seemed preferable to stick to the concept of method call. The monitor scenario can be realized by splitting the method that must be encapsulated in a monitor into two methods, one for the part defined before the signal, and another one for the part defined after the signal transmission. Both methods may then be called from outside the monitor, which avoids any unwanted blocking. The errors produced by forgetting the second call are obvious to find out, whereas if the developer discovers, after the monitor has been implemented, that external events need treatment while waiting within the monitor, s/he may have to reconsider the whole program architecture.

8. Modeling sC++ Applications

The approach proposed in this paper simultaneously provides a design methodology, as well as a good support for the analysis and the debugging of concurrent applications. Actually, it is as close to process theories as are SDL or Promela, two languages specifically developed to specify applications in such a way that they can be analyzed systematically, but sC++ is fully compatible with C++.

Let us consider again the first sC++ program of section 5 that indefinitely repeats a rendezvous between its two objects. These objects run for a while, then block to await the other object, and run again. They handle no data, but if they would, this would only make the problem a bit more difficult, not intractable. The behavior of these objects can be described with CCS (Calculus of Communicating System [4]) :

$$\mathtt{obj1 = send\ .\ actionA\ .\ obj1}$$
$$\mathtt{obj2 = actionB\ .\ \overline{send}\ .\ obj2}$$

The first line means that *obj1* executes a *send* command, the actions defined by *actionA*, and that it then repeats these actions from the beginning. The second line has a similar meaning. The *send* is coupled with the $\overline{send}$: These two commands can only occur simultaneously. The two lines above give a very precise model of the behavior of the program: The only action that can be executed by the program, just after the initialization, is *actionB*. Thereafter, no object can proceed on its own, the only possibility is both objects commonly executing the *send* command. After it is done, either *actionA* and then *actionB*, or *actionB* and then *actionA* can be fired. These sequences are all what these objects can do. They represent thus the complete program behavior.

The model of the two selections of section 5 is given below (the + sign represents a choice) :

```
    select1 = operation1.next + operation2.next + operation3.next
and select2 = method.next + methodx.next  +  τ.next
```

next represents a label that points to the continuation of the program. More complete examples can be found in [5, 6]. If during an analysis made with this calculus, one discovers a situation where no object can proceed further, i.e., and the only possible actions are communications commands that have no complements, one has discovered a potential deadlock that may arise at any time during the execution of the application. There are tools to do such analysis [4,5,6,7,8,9]. As a matter of fact, the power of these analyses are limited by the so-called state explosion (with the exception of [9]), but if they are used during the early specification steps already, these limitations are much less severe than if the analyses are done only after the application has been implemented. When used correctly, they can put a development on the right track, right away.

9. Conclusion

We have shown that the callback or inverse-control programming style has a very interesting alternative that allows a much better structuring of the specification, and closely related to the implementation. This approach lends itself to validation, in a much easier way than approaches based on control inversion. In [10], we propose an environment with debugging, specification, and analysis tools that supports the ideas presented here. This environment provides software developers with engineering technology that should help shift their job from art towards science.

As mentioned earlier, the sources of errors in concurrent applications can be very subtle. Concurrent applications are very difficult to debug. Actually the difficulties encountered during the specification of standards like the OSI or CORBA environments, are good indications that progress still needs to be done in this area. A number of experiences, within and outside our laboratory have shown that once the concept has been understood, it has a very positive impact on the way further developments are envisaged, even if they are based on other languages. We have developed a CORBA library that uses multithreading, includes the Motif GUI, an *http* server that handles *cgi* (common gateway interface) commands internally, etc., within a few man-months, and we have validated it by modeling and thorough process analysis [11].

A carefully chosen subset of sC++ could be used as a specification language, whereas the whole language could be used as an implementation or a fast prototyping language, with a sound theoretical background.

Reference

[1] D.C. Schmidt, *Applying Design Patterns and Frameworks to Develop Object-Oriented Communication Software*, in Handbook of Programming Languages, ed. P. Salus, MacMillan Computer Publishing, Vol.1, 97.

[2] M. A. Jackson, *Principles of program design*, Academic Press, 1975.

[3] C.A.R. Hoare, *Monitors: An Operating System Structuring Concept*, Communications of the ACM, 12(10), October 1974.

[4] R. Milner, *Communication and Concurrency*, Prentice Hall International, 1989.

[5] C. Petitpierre, *sC++, a Language Adapted to Interactive Applications*, IEEE Computer, around July 1998.

[6] C. Petitpierre, *sC++, Programmation pseudo-parallèle orientée objets*, PPUR, Lausanne, 1998, English version to appear.

[7] R. Cleaveland, J. Parrow, B. Steffen, *The Concurrency Workbench: A semantics-Based Tool for the Verification of Concurrent Systems*, ACM Toplas, Vol. 5, No. 1, January 1993, pp. 36-72.

[8] G. J. Holzmann, *Design and Validation of Computer Protocols*, Prentice Hall, 1991, AT&T Bell Laboratories.

[9] C.H. West, *Protocol Validation by Random State Exploration*, in Protocol Specification, Testing and Verification, VI, North-Holland, Amsterdam, 1986, pp. 233-242.

[10] http://ltiwww.epfl.ch/sCxx

[11] G. Duval, *Specification and Verification of an Object Request Broker*, ICSE'98, Kyoto, Japan.