Communicating Process Architectures 2002
James Pascoe, Peter Welch, Roger Loader and Vaidy Sunderam (Eds.)
IOS Press, 2002

Synchronous Active Objects Introduce CSP's Primitives in Java

Claude PETITPIERRE
EPFL, Laboratoire de Téléinformatique, 1015 Lausanne, Switzerland

Abstract. This paper presents a proposal of a language, based on synchronous active objects that introduces the CSP primitives into Java. The proposal does not use channels to realise the inter-process communications, but is shown to offer the same expressive power as the channel based solutions. The paper also shows that the rendezvous defined by CSP or our synchronous objects are very well adapted to the industrial development of event driven applications, handling simultaneously GUIs and accesses to remote applications.

1 Introduction

Concurrent programs are simpler if programmed in the right way, and in this regard CSP's rendezvous is an invaluable model that allows a programmer to avoid most problems linked to Java's approach to concurrency. In this paper, we present an approach based on the concept of rendezvous, implemented directly at the level of the method calls without resorting to any kind of channels (neither logical as in CSP, nor physical as in occam).

We then discuss the compatibility of our approach with the channel-based approaches and show that it is very relevant to the development of interactive and distributed applications, which make up the bulk of today's developments.

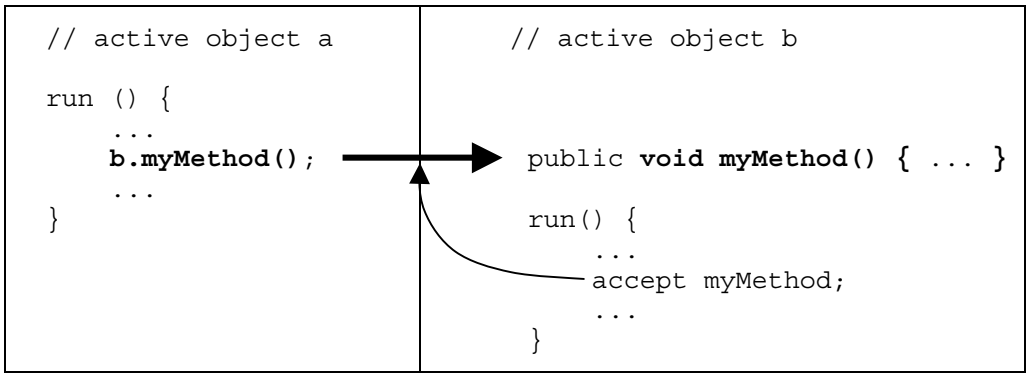
2 Synchronous Active Objects

2.1 The Concept of an Active Object

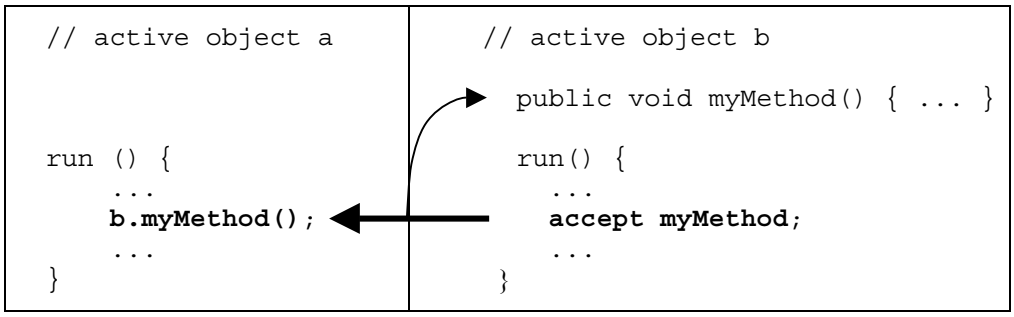
Java's threads are often considered to be orthogonal to objects. In this paper we claim that threads are too delicate to be handled at the application level. They must be tightly bound to objects, from which they should not be distinguished more than the movement of a watch is distinguished from the watch itself. This view defines a new kind of objects, named active objects, that can be contrasted with the passive (i.e., usual) objects. Passive objects are acted upon, whereas active objects may act on their own without being called. The behaviour of active objects is defined in their *run* method, which is executed on a thread attached to the object when it is instantiated. An active object is conceived as such, and we consider that a method of a passive object cannot be run on a bare thread without complicating an application's structure. We propose thus to define a way of creating active objects (by decorating their class with the keyword *active*) that does not require programmers to start threads directly. The *active* keyword tells the compiler that it must start a thread at the end of the execution of the constructor and synchronise all the calls made to its methods as described in the following.

2.2 The Concept of a Synchronous Active Object

A communication between two active objects is naturally made with method calls. However, as active objects introduce concurrency, these calls must be synchronized. This synchronization can be observed under two angles. Under the first angle, shown in the following frame, the focus is on the method call: the object on the left calls a method in object *b,* defined on the right. As *b* is an active object, this call blocks and is only executed when object *b* accepts that this specific method be called.



Under the second angle, the focus is on the communication. As the *accept* statement in object *b* is also blocking, and is executed only when a call to method *myMethod* has been executed, a communication between two synchronous active objects also expresses the rendezvous shown below. The first thread must be at the call statement and the second one at the *accept* statement for the rendezvous to occur. The method is executed during the rendezvous, while both objects are blocked.



Depending on the situation, one view or the other is preferred. When one studies the internal functioning of an object, the important fact is that a method has been executed, whereas when one analyses the global behaviour of a multi-active object application, the important aspect is the interleaving of the rendezvous.

2.3 The Select Statement

Simple synchronous calls are of course too constraining. We have thus introduced a statement that implements CSP's alternation, allowing a program to wait for as many *calls*, *accepts* and *delays* as needed. We have also introduced a guarded statement, shown in the second case of the *select* described below. Guards are optional, but they can be added to the three kinds of cases.

```
public void run () {
    for (;;) {
        select {
        case
            y = obj.method(x);
                ...
        case
            when (guard) accept myMethod;
                ...
        case
            waituntil (currentTimeMillis()+1000);
                ...
        }
    }
}
```

The first statement of each case must be a (possibly guarded) trigger, namely a call to another active object, an *accept*, or a *waituntil*. The cases that have a false guard when the program enters the *select* are discarded for that execution of the *select*. The statements placed in the body of a case are executed after the rendezvous that triggers the case has taken place. They may contain passive or active calls as well as new *select* statements. Our *select* statement is very close to Ada's. The main difference is that ours may contain any number of synchronous calls, *accepts* and *waituntil*.

2.4 n x m Communications

Our communication concept does not involve any channel, a method in an object being the sole identification used by a caller to address the receiver. Actually, it is commonly admitted in object-oriented programming that calling a method in an object is equivalent to passing a message to the object. The method thus plays the role of a channel. However, with channels, it is possible to implement an *n x m* communication pattern (Figure 1, left), whereas simple method calls only implement the *n x 1* communication pattern (Figure 1, middle). We have thus devised an extra statement to cope with that aspect.

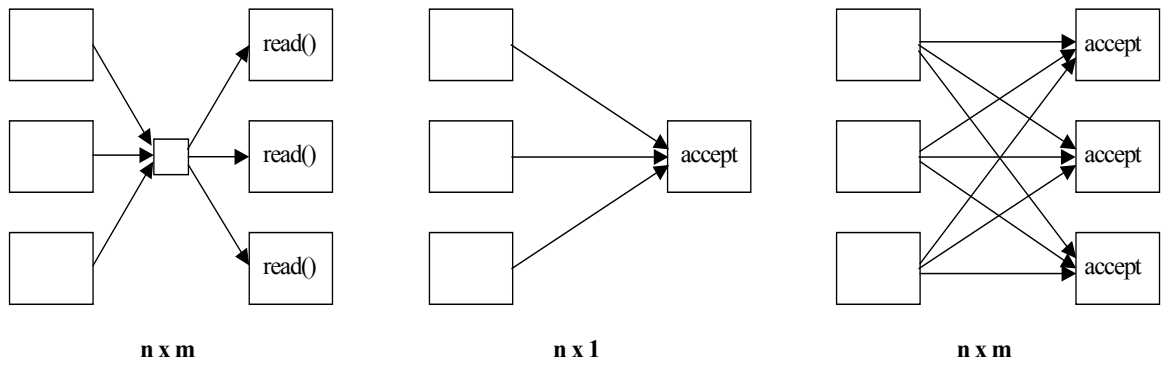


Figure 1. Channels versus synchronous calls.

The statement below implements the last pattern of Figure 1, which is equivalent to the first pattern.

```
select {
cases (i<N)
    obj[i].myMethod();
        ...  statements containing i
case
        ...
}
```

The *cases* statement represents a set of *N* cases. The code is not replicated - the kernel just returns the value of *i* that corresponds to the selected case and this value is used during the execution of the body of the case. The handling of this kind of communication is a little bit more complex than with the use of a channel, but to pass from the situation *n x 1* to the situation *n x m*, the programmer only needs to replace an object by an array of objects and to indicate how many objects are involved in the call. An advantage over channels is that the receiver automatically gets an identification of the sender (the *i* value). Note also that all receivers need not be of the same class. They just have to implement the same interface, namely the one that comprises the called method.

2.5 The CSP Primitives `SEQ` and `PAR`

The CSP primitives `SEQ` and `PAR` have not been explicitly introduced in our approach. Sequentiality appears only at the statement level, not at the object level. The sequential execution of two objects must be organized with an explicit synchronization by synchronous calls. As far as the `PAR` statement is concerned, each active object that is instantiated runs in parallel with the other objects, but the junction of the execution of two objects must be explicitly programmed.

3 Synchronous Calls, Channels and Select Statements

3.1 Buffered Channels

A buffered channel can easily be implemented with synchronous calls (as well as with zero-slot channels) and actually, we cannot imagine a better specification of a buffered channel than the following one: a buffered channel is an active object that defines a *write* method and a *read* method with the signatures and the scheduling *"specified"* below:

```
active class Channel {
    int N;
    ...  constructor delivers N (the buffer size)
    List list = new List(0);
    // signature
    void write (Object obj) { list.add(obj); }
    Object read () { return list.remove(0); }
    // scheduling
    public void run () {
        for (;;) {
            select {
            case
                when (!list.isEmpty()) accept read;
            case
                when (list.size() < N) accept write;
} } } }
```

This *"specification"* clearly identifies the data that can be passed to/from the channel, as well as the conditions under which the *read* and the *write* methods are executable. Note that the signature is clearly separated from the scheduling of the operations. In order to create an *unlimited* channel, one just has to drop the limitation – i.e. the second *when*. This channel uses a thread, but an optimiser could detect that there are no call statements in the *run* method and somehow integrate the code executed on the thread at the end of the methods.

3.2 Zero-buffer Channel

In this paragraph, we show that a zero-buffer channel (namely, a logical channel, used for identification purposes only) can also be described with synchronous calls, although it is obviously better to stick to the simple method call when developing applications.

```
active class SynchroChannel {
    Object x = null;
    void write (Object obj) {
        x = obj;
        accept read;
    }
    Object read() n{
        return x;
    }
    public void run () {
        for (;;)
            accept write;
    }
}
```

This implementation is not as clean as the previous one. Firstly, the *accept read* statement must be placed in method *write*. Secondly, the *write* cannot be called from a *select* statement, because it would be selected even if the rendezvous with the receiver is not ready. This example highlights both the flexibility and the limitations of the approach.

3.3 Characteristics of Synchronous Calls

The synchronous method calls actually integrate the *read* method in the receiver, which allows the programmer to replace the symbol *write* by significant names and to pass any set of parameters, some of which being specifiers indicating the type, the size or other characteristics of the passed data. The methods can return indications on the success of a call, which is not the case with channels; another channel is needed for that purpose.

The methods store data directly under the form expected by the receiver, in collections, databases, or whatever data structure the receiver requires. This eliminates an intermediate storage and the packing/unpacking of the data into/from transmission messages.

The synchronous calls make it possible to use object-oriented features such as overloading. Of course, these features could be integrated in the channels, but the programmer should then have a means to redefine the *read* and the *write* primitives.

The synchronous active objects can be integrated in libraries and called like the passive objects. They are very useful for creating synchronous as well as asynchronous APIs that avoid the need to poll to determine when their operations are finished (see paragraph 4.2).

These calls extend the call-channel concept defined by JCSP [1], by allowing *n* x *m* communications, as well as the placement of both ends of the communications within *select* statements. The details of the implementations are otherwise very similar.

3.4 Calls in Select Statements

Often, a potential deadlock can be avoided either by using unlimited channels or by inserting a call – or the writing of data into a channel – within a *select* statement.

In Figure 2, two clients send messages to a server, which broadcasts them back to the clients, for example in order to manage a distributed blackboard.

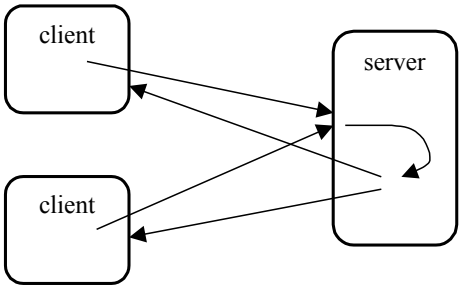


Figure 2. Broadcast server

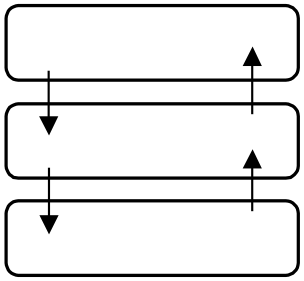


Figure 3. Protocol stack

If the client and the server are implemented with synchronous calls, zero-slot buffers or bounded buffers, and without *select* statements, it may happen that one client tries to send a message while the server is itself busy, trying to send a message to that client through a synchronous call, a zero-slot buffer or a bounded buffer that has reached its upper limit. In that case, the system enters a deadlock state. This situation can be avoided either by using unlimited buffers, in which case, the senders can never block, or by having the clients accept all incoming calls or messages in parallel with their attempts to send messages, namely having the clients accept and send messages within *select* statements. Note that if the *select* statements cannot contain calls with the accepts statements, the solution becomes much more complex.

Another example is represented in Figure 3. This figure depicts a protocol stack as defined by the OSI model, in which each layer is an active object. As it is not possible to use *select* statements that contain only *accepts* in two consecutive layers (one of them must indeed contain a caller), either a *select* statement with calls and *accepts* or unlimited channels must again be used to avoid potential deadlocks. Note that channels would establish an auxiliary protocol between the layers used to create the protocol stack and multiply the number of states that must be examined to analyse the protocol stack.

4 Communications Over a Network

Synchronous calls can be considered as the most basic object-oriented communication means within a single processor because they just add synchronization together with the calls. Accesses to remote objects however are definitely different. In the following, we present our approach to remote method calls, which allows a seamless integration of the current concepts (RMI, CORBA, SOAP) in a CSP-based environment. The main aspect we have considered is the possibility of handling remote accesses from *select* statements.

4.1 Blocking Call

Method *remoteMeth* shown below is obviously blocking, and if it is called from a *select* statement, the case that begins with this call is immediately triggered and thus the caller of *remoteMeth* is blocked while waiting for the returned data, which typically produces the kind of deadlock situations described above.

```
class Proxy { // client side
    TCPSocket socket;
    result remoteMeth (type param) {
        string = encode (param);
        send (string, socket);
        string = receive (socket);
        result = decode (string);
} }
```

4.2 Asynchronous Accesses

We have thus decomposed the call into two synchronous calls and executed the message transmission on the *run* method of the proxy. After the *post_* has been accepted, the client must execute a *ready_* before it can execute a new *post_*.

```
                        // asynchronous version
active class Proxy {            // client side
    TCPSocket socket;
    void post_remoteMeth (type param) {
        string = encode (param);
    }
    result ready_remoteMeth ( ) {
        result = decode (string);
    }
    void run () {
        for (;;) {
            select {   // there may be other cases
            case                  // in this select
                accept post_remoteMeth;
                send (string, socket);
                string = receive (socket);
                accept ready_remoteMeth;
} } } }
```

The following code shows how to use those stubs in a situation where the end-user is given the possibility to cancel the call in case s/he is getting impatient. The proxy should of course contain a cancel method, but these details go beyond the scope of this article.

```
  void run () {  // client
     ...
     proxy.post_remoteMeth();
     select {
     case
         proxy.ready_remoteMeth(x);
     case
         cancel.pressed();
     }
     ...
  }
```

We have developed a generator that creates stubs and skeletons from an interface. The resulting stubs contain the three methods, ***m***, *post_**m*** and *ready_**m***, for each method ***m*** defined in the interface. The program may thus decide at runtime which kind of call it uses. The skeletons generated only handle the normal methods, as the same message is sent in all cases. The methods of our stubs can thus access the normal CORBA servers.

4.3 Modeling a Remote Connection

One can argue that a language such as the one we propose should offer channels to implement remote communications. However, the developer must have a tight control on the way this "channel" is implemented.

In some situations, the call is just made to get some information; in which case, there is no problem to cancel it and to perform the *post* method again later if needed.

However, the application may require that if the network breaks after the message has arrived at the destination and before the result has returned to the client, the client and the server either agree that the operation has been executed or that it has not started at all. This is a complex task that must be programmed at the application level, as it must be solved for example by resorting to a two-phase commitment. The fact that the client can back off the *ready_method* in the stub proposed above allows it to get the processor control back and to secure the negative result of the operation on the disk, in order to be capable to transmit it to the server when the connection is reestablished.

Simple remote communications such as the connections between distant objects implemented by means of RMI or CORBA stubs, skeletons and TCP sockets can be modeled easily with CSP, considering the fact that two remote calls are executed sequentially, and that a new call can only be performed if the previous one has been completed. In the following, we model a connection accessed by our *post* and *ready* methods, keeping in mind that the former two methods placed in sequence can model a simple method.

$$stub = stub.post\_m\ ?\ x \rightarrow server\ !\ x \rightarrow server\ ?\ y \rightarrow stub.ready\_m\ !\ y \rightarrow stub$$

This model allows the analysis of distributed applications composed of synchronous active objects and based on standard environments (RMI, CORBA).

5 Examples of GUI Applications

The few statements defined in the previous paragraphs can be used across the whole spectrum of concurrent problems. We have written thousands of lines of code for interactive and distributed applications, without the need of any semaphore, monitor or any kind of channels. In the following, we present a GUI application and its coding with synchronous calls, then with channels and finally with listeners (see the complete sources at [7]).

5.1 Specification of the Problem

We assume we want to read a username and then a password in the same field (in a dialog box similar to the one shown in Figure 4). The entry of the username may take any amount of time, but once a username has been typed in, either:

- a password is typed in **or**
- the cancel button is pressed (and the demand is cancelled) **or**
- nothing happens for a predefined amount of time (and the demand is cancelled).

Figure 4. A dialog box.

5.2 Design with Synchronous Calls

The solution based on synchronous active objects is very close to the definition of the problem. The highlighted words in the source below can be mapped directly into the words of the problem statement. Note however that the clicks made when the program does not read the corresponding device must be ignored. This program can readily be compiled with our compiler and run; one just needs to add the imports and a main.

```
public active class Dialog {
    Jdialog dialog = new JDialog();
    Jlabel label = new JLabel("username: ");
    AtextField name = new ATextField(20);
    Abutton cancel = new AButton("Cancel");
    Container pane;
    String passwd, username;

    public void run () {
       pane = dialog.getContentPane();
       pane.add(label,BorderLayout.WEST);
       pane.add(name,BorderLayout.CENTER);
       pane.add(cancel,BorderLayout.SOUTH);
       dialog.pack();
       dialog.setVisible(true);

       username = name.read();
       label.setText("password: ");
       select {
          case
             passwd = name.read();
             System.out.println("U: " + username + " P: " + passwd);
          case
             waituntil(System.currentTimeMillis()+5000);
             System.out.println("Cancelled !");
          case
             cancel.pressed();
             System.out.println("Cancelled !");
       }
       System.exit(0);
} }
```

The first statements instantiate the GUI elements and pack them in a dialog box. As the GUI elements *AButton* and *ATextField* extend *Jbutton* and *JTextField* respectively, they can be handled and integrated in a window exactly like the original elements.

The way this program handles non-determinism is worth noticing. The *select* statement contains several calls, which allows a program to instantiate the GUI elements and then to call them like it would do with passive objects instantiated from a library (and unlike what must be done with the listeners shown in paragraph 5.4 !). This is a very important point, as it avoids the program control inversion introduced by the listeners.

5.3 The GUI Application with Channels

The following code shows how the above *select* statement would be coded with channels in a JCSP alternation. Like with synchronous calls, the events produced by clicks not taken care of overwrite the previous events.

The fact that one channel must be generated for each GUI element makes things a bit more complicated than the solution based on synchronous calls, but the two solutions are otherwise very similar.

```
AltingChannelInput eventB
    = Any2OneChannel.create (new OverWriteOldestBuffer (10));
AltingChannelInput eventT
    = Any2OneChannel.create (new OverWriteOldestBuffer (10));
CSTimer tim = new CSTimer ();

ActiveButton button = new ActiveButton (null, eventB, label);
ActiveTextField activeText
   = new ActiveTextField (null, eventT, string);

Guard[] guards = {eventB, eventT, tim};
final BUTTON = 0, TEXT = 1, TIMEOUT = 2;  // guard indices ...
                                          // optional, but nice
Alternative alt = new Alternative (guards);

tim.setAlarm (tim.read () + timeout);

switch (alt.read()) {
   case BUTTON:
      String s = (String) eventB.read ();
      System.out.println ("Button " + s + " pressed");
      break;
   case TEXT:
      String s = (String) eventT.read ();
      System.out.println ("Text field returned " + s);
      break;
   case TIMEOUT:
      System.out.println("Timeout");
      break;
}
```

5.4 The Same Application with Listeners

Listeners are often considered a very nice concept. However, the implementation of problems such as the one stated in the previous paragraphs with listeners leads to programs as bad as programs written before structured programming was recognized. The source code below presents the part of the implementation that handles the text field.

```
1 textField.addActionListener (new ActionListener () {
2    boolean readUsername = true;
3    public void actionPerformed (ActionEvent e) {
4       if (readUsername) {
5          names[0] = textField.getText ();
6          label.setText ("password: ");
7          readUsername = false;
```

```
 8           cancelButton.addActionListener (
 9             new ActionListener () {
10               public void actionPerformed (ActionEvent e) {
11                 timer.stop ();
12                 de.continuation (names);
13               }
14             });
15             timer.start (); // code executed somewhere else
16           } else {
17              names [1] = textField.getText ( );
18              timer.stop ();
19              de.continuation (names);
20           }
21         }
22     });
23 }
```

The order in which these lines are executed does not correspond at all to the lexical order, which makes their analysis, debugging and maintenance a nightmare and may even explain why the developments of event-driven applications often end up with delays and unstable results.

- Lines 1 and 2 are executed at the initialisation of the program.
- Lines 3 to 9 and 15 are executed when the text field is filled in for the first time. The Boolean *readUsername* had to be added to indicate if the method is called to provide the username or the password, although it did not appear in the specification. Line 9 actually modifies the program at runtime.
- Lines 16 to 20 are executed after the second time the text field has been filled in.
- Lines 10 to 13 are executed when the button *cancel* has been clicked.
- The timer is dependent on a listener. It had to be created (not shown above), but nevertheless still had to be started and stopped (lines 11, 15 and 18). Again, that was not necessary in the previous implementations.
- It is not possible to wrap the code presented above in an object and read the names obtained from the user by calling a method of that object (without resorting to a synchronous call or a channel !).

Moreover, the problem analysed here is quite simple. In a complex application that requires many GUI interdependent elements, it becomes almost impossible to discover how these elements are related to each other. Design patterns are supposed to help here, but they just add to the nightmare. The reader who wants to convince himself of that fact can simply read the program given at [2], which is assumed to be an example of the use of patterns. The structure of that program is indeed very difficult to understand.

5.5 Connecting Synchronous Calls to the Listeners

The following frame shows how a listener can trigger a synchronous call. Actually, this construct rectifies the program control inversion introduced by the listeners. Instead of directly calling the application code, the listener resumes the main program's execution.

The *accept* must just transmit the indication that the button has been pressed to the kernel, but not wait for the rendezvous (which has the same goal as the overwriting of data mentioned in paragraph 5.2).

```
class Sbutton extends JButton {   // listener
    public void pressed () {};
    public SButton () {
        addActionListener (new ActionListener () {
            public void actionPerformed (ActionEvent e) {
                accept pressed;  // no wait!
            }
        });
    }
}
```

The programmer must thus pay attention to the fact that this *accept* must not be re-executed before method *pressed* has actually been executed, and to disable the button for that purpose. This construct is thus a bit delicate, but it is usually made by a system engineer and stored in a library reused without any modification in all applications that need it. Moreover, this construct can be used in the same way for all GUI elements, including the mouse.

6 An Example with a Network Connection

The following example is taken from [3]. It demonstrates how synchronous objects can simplify the development of interactive applications and support UML. Figure 5 describes the steps necessary to authenticate a customer in an ATM (automatic teller machine).

This diagram does not give any hint as how to implement the corresponding program, and there is a good chance that the programmer will spend much time devising listeners, dialog box, and so on.

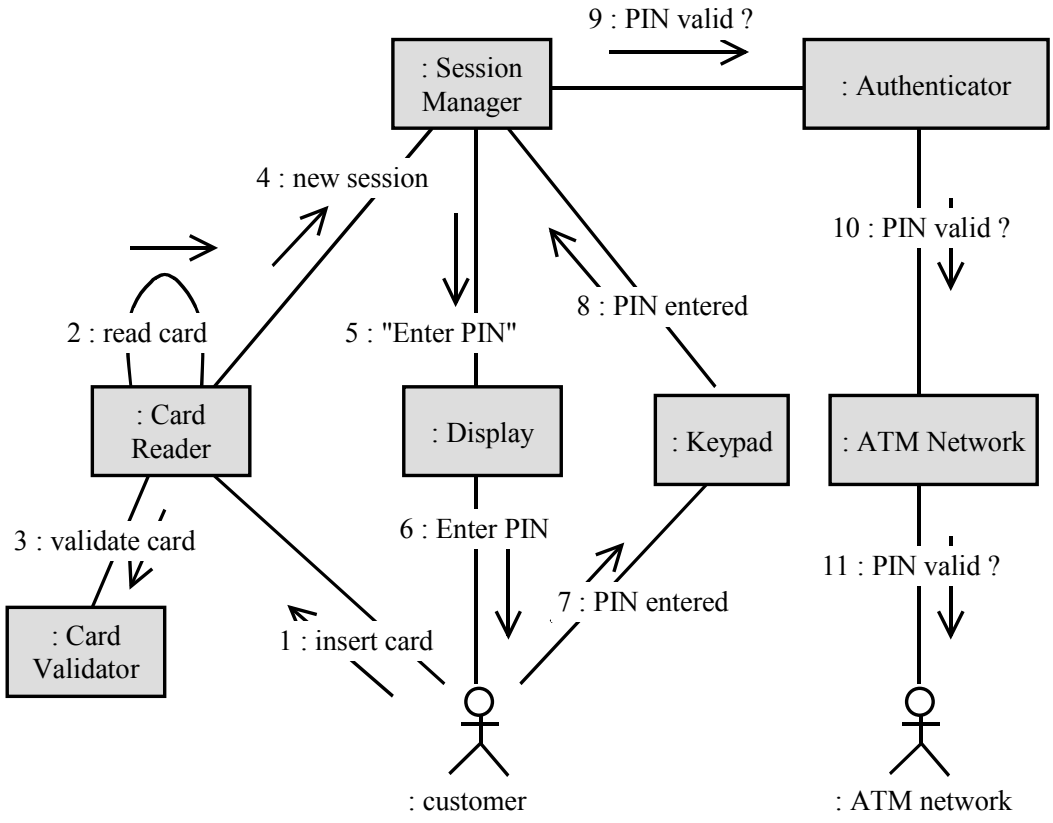


Figure 5. Collaboration diagram.

We rather propose to transform the diagram shown in Figure 5 into the one shown in Figure 6. This new diagram contains all the devices and almost all the actions that were defined in the former diagram …. We have just grouped the modules and the devices and hooked all actions to the actors that execute them (customer, session). The program is now sequential, and its code is obvious. All calls are synchronous and even do not require selections. We nevertheless added a selection between a cancel button and the waiting of the response of the authenticator to show how simple it is to handle non-determinism. The cancellation of the reception of the message does not introduce any problem here because it is a read statement that does not modify the authenticator.

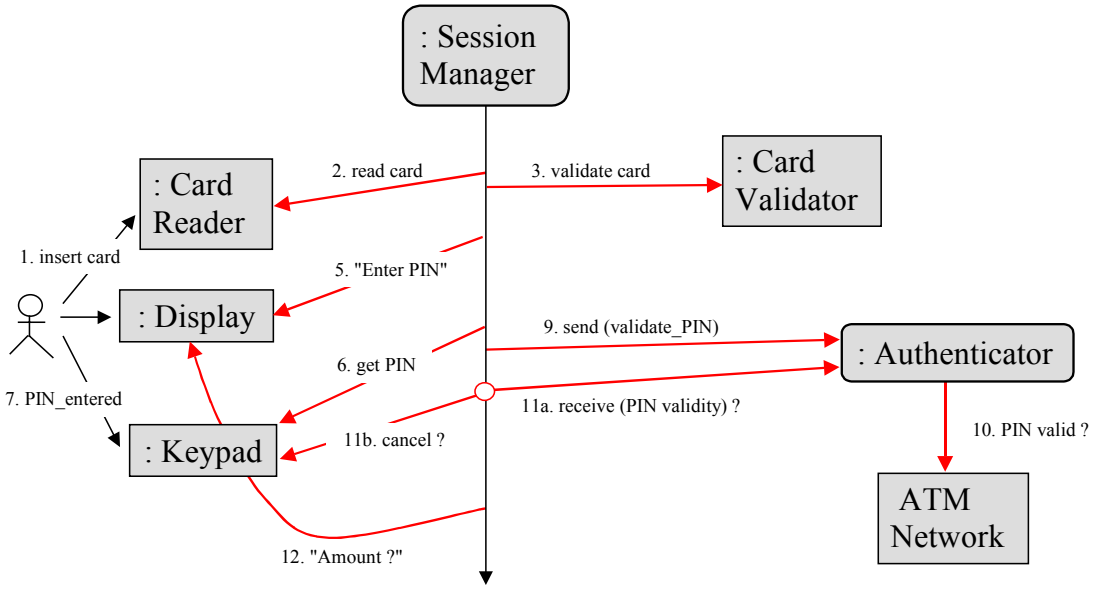


Figure 6. Diagram based on synchronous objects.

7 Conclusion

We have presented a concept of synchronous calls [4,5] that matches the synchronization aspects of CSP very well, and shown that this concept is very useful to circumvent the problems raised by the use of listeners and the program control inversion they induce.

8 Available Tools

We have developed a compiler that handles the synchronous calls and the *select* statements and produces both binary and Java source files. Actually, the statements that handle the synchronization are simple and could very well be coded by hand too. The developer is thus not tightly bound to the use of the compiler, but the program structure is of course clearer when the compiler is used.

We have also developed a state analyser that can handle a subset of Java and determine if the code contains deadlocks. This analyser also produces a CCS (Milner's calculus of communicating systems) description of the application.

We have developed several libraries and generators, and our students and ourselves have written thousands of lines of code with much success. Our developments are available on our web server [6].

Acknowledgements

I am grateful for the help that Dr. Fernando Pedone and Professor Sanjiva Prasad provided me in the writing of this paper.

References

[1] P. Welch and P.D. Austin, *The JCSP Home Page*, http://www.cs.ukc.ac.uk/projects/ofa/jcsp/, 2002.

[2] http://www.javaworld.com/javaworld/jw-10-1999/jw-10-toolbox.html

[3] P. Kruchten, *The Rational Unified Process – An Introduction*, Addison-Wesley, 2000.

[4] C. Petitpierre, *Synchronous C++, a Language for Interactive Applications*, IEEE Computer, September 1998, pp 65-72.

[5] C. Petitpierre, A. Eliëns, *Active Objects Provide Robust Event-driven Applications*, SERP'02, Las Vegas, June 2002, pp 253-259.

[6] Claude Petitpierre, *Synchronous Active Objects*, http://ltiwww.epfl.ch/sJava

[7] Claude Petitpierre, *Synchronous GUI Dialog Sources*, http://ltiwww.epfl.ch/sJava/version2/Inversion