An Object Oriented Method for Implementing Layered Protocols
published in FORTE'93, boston.

A. Divin and C. Petitpierre[a]

[a]Ecole Polytechnique Fédérale de Lausanne
Laboratoire de Téléinformatique
IN - Ecublens
1015 Lausanne, Switzerland
Tel +41 21 693 47 97 FAX +41 21 693 66 00
E-mail [divin,petitp]@ltisun.epfl.ch

This paper presents a novel extension to the C++ programming language, allowing powerful expression of concurrency and synchronization in an object-oriented environment. The principle advantages of this extension over existing alternatives are its powerful yet simple expression, its ease of use, and the strict parsimony of additions made to the C++ syntax. The concurrent and object-oriented capabilities of our approach make it particularly useful for facilitating the construction of complex software systems, such as communication protocol stacks. The paper shows that the structures obtained with our proposal are simpler and more efficient than those obtained with SDL-like primitives. The semantics of the novel features are defined with respect to the Calculus of Communicating Systems (CCS).

1. Introduction

The complexity of communications systems has spurred research aimed at providing adequate tools for their development, and prompted the international organizations that define communications standards to propose three main specification languages: SDL, Estelle and LOTOS. Several tools and compilers have been developped to transform the specifications written in these languages into application code, but neither the languages nor the tools are adequate to realize products and the industrial developments of communications systems are still made, completely or partially, with general purpose languages such as CHILL, Ada, C or C++. These "industrial" environments are not quite satisfactory either for integrating the multiple layers of a communication protocol. One reason for this is that communications protocols are often presented in the academic and industrial literature without reference to integration problems.

This paper presents a new approach to concurrency, based on the concept of objects. Although this approach does not directly provide a formal specification language, the paper will show that Object Oriented Design (OOD) together with our extension is a very efficient specification *environment*. Our language can be described with the Calculus of Communicating Systems theory (CCS) [6] for validation purposes and implemented thanks to an environment based on C++, publicly available.

Our proposal is simpler than alternative proposals such as ABCL, POOL-T, Concurrent Smalltalk, COOL, Presto and ES-Kit C++ as described in [15], [20], [21]. It respects the philosophy behind C++. Thus it contains all OO features and can be acquired very quickly by C++ programmers. The paper shows how the approach can be used to build protocol stacks that are very well structured, efficient, reusable and easy to maintain.

In order to avoid any confusion, we would like to stress that our approach aims at the programming of monoprocessor systems. It is very useful to build the interfaces of distributed systems, but it has no high level primitives for such environments.

Section 2 presents our extension to C++, section 3 presents some issues raised by the use of channels such as those defined in SDL, section 4 presents a general view of a layer structure that solves these issues, and section 5 gives a semantic of the new statements, related to the CCS theory.

2. Description of the Extension to C++

The object oriented programming language C++ [9] defines data and functions within definition modules called classes. Objects are instantiations of classes. They can read or modify the data of the other objects mainly by calling the methods (functions) of those objects.

Method calls are often compared to message exchanges between objects, because the human perception of the communications between objects is very close to the message passing paradigm found in multitasking environments. Unlike other proposals, ours sticks to this view. The only addition we made to C++ is the possibility for the objects to run concurrently. Although the objects communicate with the other by synchronous method calls only, all the synchronization problems that can be solved by any other multi-tasking language, can be solved with our proposal.

We define two kinds of objects:

Soft objects:

> These are the usual C++ objects. Once an object has been instantiated and its constructor [1] has been executed, its methods can be called at any time.

Hard objects:

> The hard objects introduce concurrency. The constructor of a hard object runs in parallel with its instantiator. The instantiator can thus continue its execution before the constructor terminates. The

[1] A constructor is a function that is called automatically just after the instantiation of an object and before any use of the object. It is usually used to initialize internal data. (It has the same name as the class).

> hard objects are protected in such a way that their constructors and their methods cannot be executed simultaneously. Only one method can be executed at a time and only when the constructor is terminated or suspended. Thus hard objects provide the same protection as a Hoare monitor [10].

An object is of type hard if it inherits *task*, else it is soft. The property *task* can be inherited recursively. In that case the object thread executes successively all constructors, in the same order as for standard C++ object. Of course if one constructor loops or suspends its execution, the executions of the descendant constructors are delayed.

A thread of execution (coroutine) is automatically attached to the hard object if a constructor is defined. This thread supports the constructor. It is released once the constructor terminates (if it does).

The synchronization primitives are defined in the following sections.

2.1. Call And Accept Statements

At the lexical level, calling a soft or a hard object is absolutely identical. The compiler automatically inserts the needed synchronizations. An object can call its own methods recursively without being blocked. The bodies of the methods are defined as usual.

The syntax used to express a method call is:

object->method(parameters);
or
object.method(parameters);

The caller of a hard object method is suspended until the called method has been enabled or the constructor of the called hard object is terminated. In order to enable a method for one call, an object uses the following instruction (usually located within the constructor) :

accept method;

If no constructor is defined within the hard object or at the end of it, all methods are automatically enabled. Of course they are still executed one at a time. As soon as the synchronization takes place, the method is disabled, and its code is executed. In order to accept further calls the *accept* statement must be reexecuted. The code that executes the *accept* statement is suspended until the method has been completely executed.

2.1.1. Example

Figure 1 and the program example below show how a constructor suspends itself while enabling a method, and how a message can be passed from one object to the other. A detailed explanation follows the program.

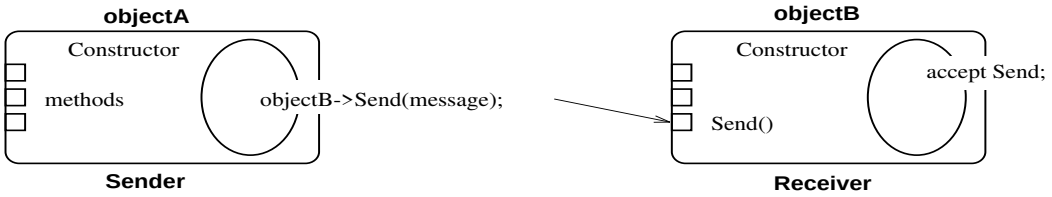

Figure 1. Communicating Objects

```
// in objectA constructor

1   B *objectB;
      .
      .
      .
2   objectB = new B;
      .
      .
      .
3   objectB->send(message);
      .
      .
      .
```

```
1   class B: public task {          // objectB's class
2     MsgType InputMessage;

3   public:
4     void send(MsgType m) {  // definition of the method
5       InputMessage = m;
      }

6     B() {        // constructor, start of objectB's execution
7       for(;;) {
⟹        accept send;
9         process(InputMessage);
        }
      }
    };
```

(line 8: $\Longrightarrow$ represents a transfer of data). *objectA* first instantiates the hard object *objectB* of class B (line 2) and thereafter calls a method of *objectB* (line 3). The execution of *objectA* is suspended until *objectB* accepts its call. The constructor of *objectB* is executed concurrently with *objectA* down to the *accept* statement (line 8), where it is suspended. When both *objectA* and *objectB* have arrived at line 3, respectively line 8, they make a **rendezvous**. Method *send* is executed during this rendezvous. Once it has been executed, the executions of *objectA* and of the constructor of *objectB* are resumed. *objectB* processes the message (line 9) and reenables method *send* (line 8) to get the next message.

Notice that we could also code the processing of the message within the method and let the constructor terminate, without coding a loop. In that case the method should not be reenabled each time it is executed, because all methods are implicitly enabled once the constructor is terminated. The difference between both cases is that with the solution choosen here, the sender is not blocked for the duration of the processing of the message.

If several objects are waiting for the same method in the same object, they are stored in a queue and one is reactivated each time the method is reenabled. They are reactivated in the arrival order.

The parameters of the method call are evaluated only when the synchronization takes place. If a parameter is defined as a function call, that function is executed during the rendez-vous, which can be very useful to send last relevant data.

In C++ language, the instantiator of a soft object cannot call the methods of the object before the constructor is finished, because a call to the constructor is automatically inserted by the compiler when a new object is created. Notice that this is also true in the case of a hard object, but for another reason. The constructor of a hard object is started in parallel with the instantiator, which can call a method at once. Nevertheless, the method is not accepted by the new object before its constructor is terminated (or it has enabled the method).

2.2. Select Statement

The following statement enables a choice of methods. Notice the symbol || representing the "or" operator in C++ used to separate the synchronization events in the *select* statement.

```
select {
  accept method1;
||
  accept method2;
||
  accept method3;
}
```

After this statement, the caller is suspended and any one of the three methods can be called. Nevertheless, as soon as one method has been selected, all methods are automatically disabled. The execution of the code containing the *select* is resumed after the selected method has been completely executed. The statement has to be reexecuted if the methods must be reenabled.

Some code can be placed directly after the *accept* statement for example to handle the results of the execution of the method and to determine which methods must be enabled next. This code can contain calls to other objects. A more complete *select* statement is given below.

```
select {
  accept send;
    ...instructions ...
||
  objectA->deliver(x,y,z);
    ...instructions ...
||
  waituntil(date1);
    ...instructions ...
||
```

```
    accept reset;
      ...instructions ...
}
```

This *select* statement is composed of four **alternatives**. When the hard object arrives at the *select* statement, its execution is suspended until either one of these **events** happens:

- method *send* or method *reset* is accepted
- or *objectA* accepts method *delivery*
- or time reaches *date1*.

Only one of the events is selected and fired by the kernel. Which one is selected depends on the behaviour of the surrounding objects. The execution of the hard object is resumed only when the event is completed (i.e. the called or the accepted method has returned or the timeout has been fired) through the instructions of the alternative triggered by the selected event.

As mentionned above, after the constructor of a hard object is terminated, all its methods are automatically accepted, of course one at a time. This corresponds to an infinite loop over a *select* statement accepting all methods.

2.3. Guards

The **guards** are boolean expressions that indicate if the events they preceed are actually included in the current execution of the *select* . The guards are evaluated when the *select* statement is entered. They have no influence on the *select* if they change afterwards. Their syntax is given below:

```
select {
  when (guard1) accept methodZ;
||
  when (guard2) object->methodY(a,b,c);
||
  when (guard3) waituntil(date);
}
```

Actually this statement is close to the *select* statement found in Ada [18]. The constructor corresponds to the task body, and the code of the entry point, to the method. However in our realization, there can be as many acceptances, as many calls and as many delays as needed.

2.4. Waituntil Statement

The *waituntil(date)* statement suspends the caller until time reaches the instant given in the parameter. The delay can be relative or absolute, as is proposed in the new version of Ada. The function *now()* can be used to get the current time. Thus, the statement

```
waituntil(now() + 1000)
```

suspends the caller for 1000 time units. That statement corresponds to a delay relative to the execution time of the *waituntil* statement, whereas the basic function, without call to *now*, points to an absolute time.

If *waituntil* is used as an alternative in a *select* statement, it will schedule a time-out. The timeout will fire if none of the other synchronization alternatives is executed before the expiry of the timer. If one of the other alternatives succeeds, the *select* exits and the *waituntil* is automatically disabled. It is thus not possible to forget to disable a timer.

3. Inadequacies of SDL Programming Style

SDL [1], Estelle [3] and CHILL [2] share the same concept of unlimited channels to pass messages from one process to another and to synchronize each other [22]. These channels determine the programming style, which will be called SDL style in this paper. They solve some problems but unfortunately introduce new ones, as is explained in the following.

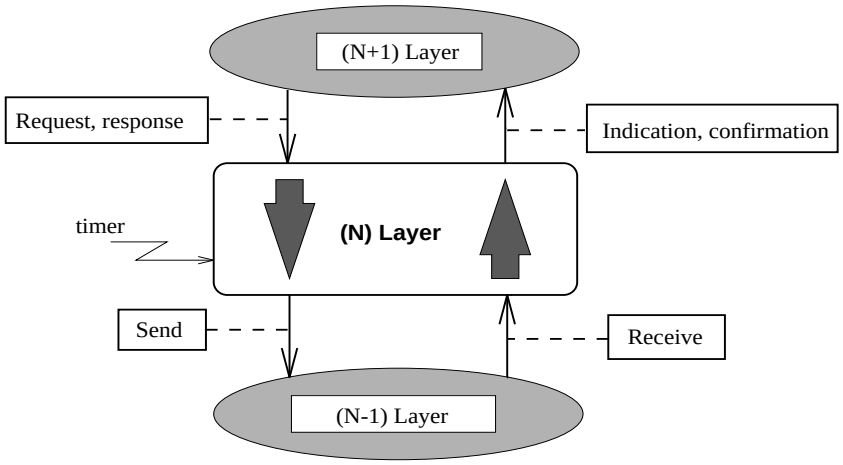

Figure 2. Structure of a layer

Figure 2 represents three protocol layers building a fragment of a protocol stack. We want to program the (N)layer in such a way that it would be easy to stack several similar layers on top of each other, in order to produce, for example, the implementation of the OSI model. The (N)layer shown on the figure, as any layer, receives data from the (N+1)layer, adds a new header and sends the data to the remote (N)layer via the (N-1)layer. It also receives data messages from the (N-1)layer and forwards them to the (N+1)layer. In the case of reliable data transfer, the reception of data is acknowledged by the remote (N)layer. If the data is not acknowledged within a predetermined time, the (N)layer has to retransmit the same data once more, because it may have been corrupted and lost during their transfer. A timer must be introduced to handle the latter operation.

Figure 3 shows a fragment of a protocol, describing a program that executes some of the operations defined above within the layer. It is taken out of the CCITT Q764, used in the SS7 (signaling system number 7). It is described using SDL graphical notation.

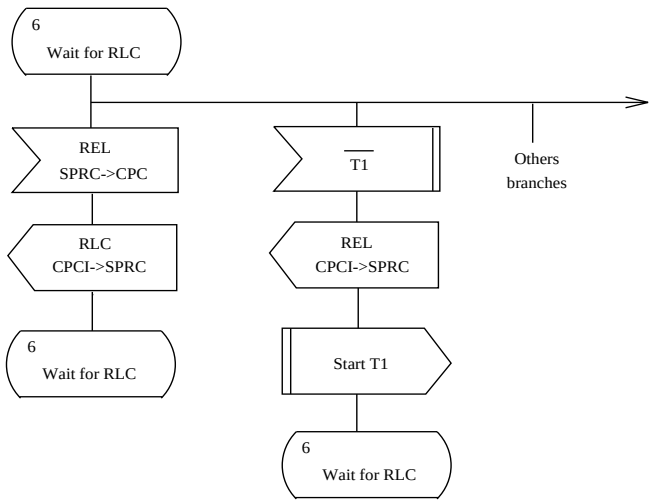

Figure 3. SDL Diagram of a protocol fragment

From a state identification "Wait for RLC", there are branches depending on first selected events.

The left branch of that figure represents:

- the reception of a message from the upper layer,
- the emission of a response to the upper layer,
- the next state.

while the right branch represents:

- the expiration of a timeout,
- the emission of a message to the upper layer,
- the activation of a timer,
- the next state.

Clearly in the state "Wait for RLC", this program must be ready to react to several events triggered by the environment. That situation is actually very different from sequential programming. Sequential programs can decide on the spot which way they will

continue on, whereas in concurrent programs, like protocols, the decision depends upon external conditions.

As already mentioned, SDL uses unlimited channels (fifos) to link the different layers. As we will see, these channels are used to avoid deadlocks. On figure 3, in the right branch, after timer T1 has been triggered, the program must send the message REL to the (N+1)layer. Assume that the function that sends the message suspends the program until the (N+1)layer does read the message. We must also assume that the (N+1)layer may also be waiting to deliver a message to (N)layer, at the same instant. That would produce a deadlock, as each layer is waiting indefinitely for the other one to take the offered message. However, if the processes are linked by channels, the messages to be sent can be deposited in the channels and each sender can proceed to the point where it can await the next event. The use of channels thus avoids this type of deadlock.

Unfortunately altough we have eliminated one problem, we have created many new ones, which are listed below.

- *Channel flushing*
  When the layer receives a reset request, it is difficult to empty the channel and it is not easy to implement a reset command in the channel, because the channel does not know what to do with the dropped messages: deallocate them, give them back ?

- *Update Acknowledgments*
  The acknowledgments usually run through the same channels as the messages. They are sometimes even written in the messages themselves. The sender should be able to send the acknowledgements in front of the messages already stored in the channel. In the case of piggyback acknolwledgments, the sender should be able to write numbers directly into the older message in the channel. This is particularly important for high speed protocols.

- *reordering of messages*
  When a selective reject has been received and several messages are already in the channel, it is impossible to reorder them !

- *Interlayer Flow Control*
  If a program sends a file to a printer, the data can be sent faster than the printer consumes it. The data speed is usually controlled by some specific layer. The upper layers do not know how to adapt their speed to that of the lower layers. If they do not, it is impossible to prevent a task from sending too many messages into a channel, which could lead to deadlocks by buffer shortage. A back pressure protocol could be used, but it needs extra signals, which must be transmitted by extra channels ...

- *Memory Management*
  The minimal memory size cannot be calculated if there is no limit on the size of the channels (Notice that if there is a limit, that limit could be 0 as well, which is close to the solution we propose).

- *Message Duplication*
  If the (N-1)layer is temporarily blocked for some reason, the (N)layer timer may induce the (N)layer to produce and store many copies of the same message in the channel. When the (N-1)layer is readied, all messages will be transmitted, which does not make sense.

- *Validation*
  Systems with unlimited channels are more difficult to validate, because the channels must be modeled by infinite state machines.

Of course there are already many protocols in use today in which these problems are solved, but not in a structured manner. We will present a clear methodology that solves them more easily and more reliably.

4. Using the New Features to Implement a Protocol Layer

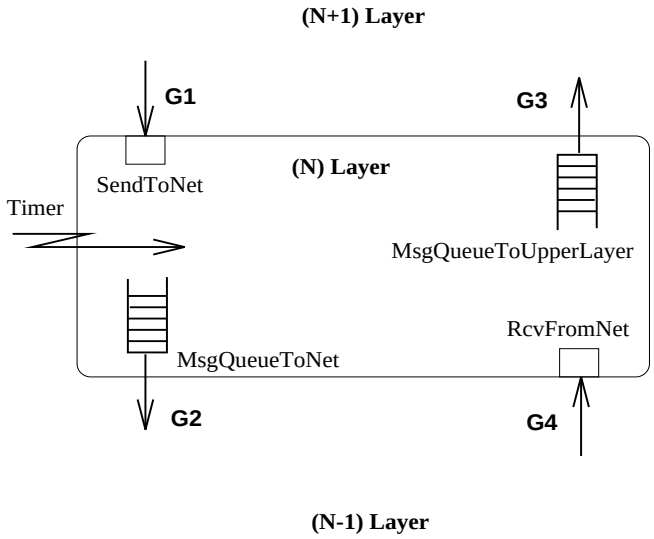

Figure 4. Diagram of a layer

Here is how we propose to build a layer without any channels between the layers. In order to memorize the messages temporarily, we use two local message queues inside the layer. The two receptions G1/G4 (see figure 4) from the adjacent layers and the two emissions G2/G3 towards the adjacent layers are made through the method calls located in the layer receiving the messages.

The program of the (N)layer (figure 5) is made of an infinite loop over a *select* statement that synchronizes the (N)layer with its neighbors on the four points G1 to G4. When the queue *MsgQueueToNet* is not full (line 3), the (N+1)layer is able to call the method

```
1  for(;;)                                                    // infinite loop
2   select {
3      when ( !MsgQueueToNet.full() )                         // event G1
4       accept SendToNet;
5         ...instructions...
6    ||
7      accept RcvFromNet;                                     // event G4
8         ...instructions...
9    ||
10     when ( !MsgQueueToNet.empty() )                        // event G2
11      LowerLayer->SendToNet(MsgQueueToNet.NextMessage());
12        ...instructions...
13   ||
14     when ( !MsgQueueToUpperLayer.empty() )                 // event G3
15      UpperLayer->RcvFromNet(MsgQueueToUpperLayer.first());
16        ...instructions...
17   ||
18     when ( Timer.Active() )                                // timeout
19      waituntil(Timer.TimeValue());
20        ...instructions...
21  }
```

Figure 5. Coding of a layer

SendToNet defined in the (N)layer (line 4) to store the message into the queue (line 5), satisfying point G1. Messages from the (N-1)layer (point G4) are always accepted and stored in the message queue *MsgToUpperLayer* by *RcvFromNet* method of the (N)layer (line 7,8). Notice that the overflow of this message queue can be avoided by the remote sliding window protocol.

Similarly, a method is defined in the (N-1)layer, *LowerLayer->SendToNet*, on point G2 and another one in the (N+1)layer, *UpperLayer->RcvFromNet*, on point G3 to transmit the data from the (N)layer to the adjacent (N-1) and (N+1) layers. The parameter of the method *SendToNet* called in line 11, *MsgQueueToNet.NextMessage()*, retrieves the oldest message out of the fifo and puts the latest informations in the message: last received number, latest status of the layer (RR or RNR in HDLC for example). Remember that the parameters of a method are evaluated only when the kernel has chosen the case of the *select* statement that will be executed. As *MsgQueueToNet.NextMessage()* is defined within the layer, it has access to all needed data and at the right time to build an up-to-date message.

Actually, what has changed from the SDL style that makes channels unnecessary now ? And why are there no deadlock risks ? Well, the most important thing we did is to put all events within the *select* statement. Unlike in SDL, the emission of a message does not block the whole layer, because the receptions and the transmissions are offered simultaneously through a single select statement. Let us examine the situation described in section 3. If two layers decide to transmit a message to each other at the same time, both have an offer and a reception in parallel in a *select* statement, and the run time kernel can decide which rendezvous is executed first. The *select* statement is thus another way of avoiding the type of deadlocks we discussed in section 3.
What about the other problems ?

- *Channel flushing*
  If the layer receives a reset command, it can easily deallocate the messages that were stored in the internal message queue and take adequate actions. Notice that if there are guards on the methods used to transfer the data to the adjacent layer, the reset command can be passed to the adjacent object by calling another unguarded method specially provided for the commands [7].

- *Update Acknowledgments*
  As we saw, the acknowledgment number can be written in the message at the latest time, i.e. as it leaves the (N)layer, when the (N-1)layer is ready to handle it. The message will thus carry up-to-date information.

- *reordering of messages*
  If a selective reject is received, it is easy to reorder the messages in the internal message queue, before going back to the select statement, reenabling sending messages alternatives.

- *Interlayer Flow Control*
  The interlayer flow control is made with a guard which takes the form of a simple boolean expression. As already mentionned, if the message offered by the (N+1)layer is not accepted immmediatly, the (N+1)layer is not blocked (*select* statement).

- *Memory Management*
  The exact memory requirement can be computed by adding the sizes of all messages that can be stored in the various layers. If there is enough memory for each layer to store the maximal number of messages each one has foreseen, the system will never be blocked by buffer shortage.

- *Message Duplication*
  Each time the timer fires, it is now easy to examine if a copy of the previous message is still in the internal queue. A new copy of the message is generated only if the previous one has left the queue, thus avoiding unecessary duplication of messages.

- *Validation*
  The rendezvous is easily modeled, as we will show in the next section.

5. Semantic of the Synchronizations

The rendezvous has first been formalized in the Communicating Sequential Processes (CSP) [12], which is at the origin of Occam [19], Ada and CCS [6]. In his paper on CSP [12], Hoare proposed a *select* statement, which he named alternative command, but, as is mentioned in the conclusion of the paper, he did not foresee the necessity of having a call in the *select* statement. Both Ada and Occam kept the same choice, which in the case of Occam facilitated the implementation of distributed rendezvous in the Transputers. Unfortunately it is impossible to interconnect several tasks together in a symmetric way if it is not possible to have calls and acceptances in the same *select* statement [11].

CCS as well as the new version of CSP [13] have symmetric selects. They gave birth to LOTOS [4]. The modeling provided by CCS is interesting, because it gives the possibility of validating protocols by analyzing the behaviour of interacting agents: detection of deadlocks, livelocks and unexpected events, final state reachability, etc... . Some verification and testing tools already exist.

Any application written with our extension is easily modeled with CCS. The *method call* and the *method acceptance* can be considered as CCS *complementary actions*. The *waituntil* statement corresponds to the CCS *silent* action performing an internal action of an *agent*. Actually our *select* statement implements exactly the CCS choice operator. The two place buffer, depicted below, can be coded with CCS as well as with the features presented in this paper.

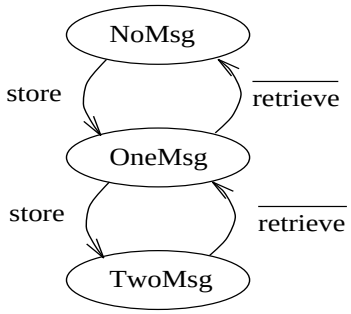

Figure 6. Buffer Management FSM

Figure 6 illustrates the finite state machine (FSM) of this two place buffer. The action "$store$" represents a message reception, while "$\overline{retrieve}$" represents the output of the stored message. In this simple example, we assume that all messages are identical. Thus their content need not be memorized.

The C++ code below implements the same example. The states are represented by labels.

```
class BufferManagementFSM: public task {

public:
  void store() { }                          // synchronization call method
  void retrieve() { }                       // synchronization call method

  BufferManagementFSM() {          // constructor, contains the FSM
    NoMsg:
      accept store;
      goto OneMsg;
    OneMsg:
      select {
          accept store;
            goto TwoMsg;
        ||
          neighbor->retrieve();
            goto NoMsg;
      }
    TwoMsg:
      neighbor->retrieve();
      goto OneMsg;
    }
  }
};
```

The program counter, at a given point, is represented by a label and thus by a state. If we follow this idea, any program is actually an FSM. The example above can be described by the following CCS equations, which model the C++ program in the same time:

$$NoMsg \stackrel{def}{=} store.OneMsg \quad (1)$$

$$OneMsg \stackrel{def}{=} store.TwoMsg + \overline{neighbor_retrieve}.NoMsg \quad (2)$$

$$TwoMsg \stackrel{def}{=} \overline{neighbor_retrieve}.OneMsg \quad (3)$$

6. Conclusions

We have presented the integration of a synchronization feature into C++, that nicely merges the object orientation and the multitasking approach. This approach remains as close as possible to the C++ philosophy and uses only a few new keywords.

The usefulness of the rendez-vous combined with the symmetrical selection has been demonstrated by a similar extension to the programming languages C and Modula2, which has previously been implemented in our laboratory [7]. These languages, Mex or C-ex, were very successfully used to develop several protocols such as X.25, TCP/IP and ISDN. These programs run today in industrial environments. C-ex was also used for creating

sophisticated queueing simulators [16, 17]. The validity of our approach has also been demonstrated in Newsqueak [14], a multitasking language very close to the above languages, but based on an interpreter that is very well adapted to the programming of window systems. This is interesting because the windowing systems face similar problems to those of communication protocols. The merger of these two worlds (concurrency and objects) is thus very promising.

The approach that we have presented here is complementary to the example of the sliding window protocol implementation given in Tannenbaum's book [5]. In this book, the procedure implementing the sliding window protocol just calls "sleep" to wait for the next message, leaving up to the programmer the solution for the synchronization with the outside world. Our approach is also closer to a real implementation than Buhr's communications systems design method using Ada [8]. In the latter reference, Buhr admits that his solution must be reviewed carefully before it can be taken as the basis of an implementation, whereas our layer structure is ready for compilation.

Our solution, although it has been inspired by the work realized in the area of telecommunications, is also very well adapted to the development of many other real time systems such as man-machine interfaces, process control, operating systems, data bases, etc ...

We are currently looking at the definition of a *private task*, the multiple and public or private inheritance of the task property and at the definition of an *and* operator in the *select* statement.

A preliminary version of an environment containing a preprocessor and a run time library is freely available on the anonymous ftp server: ftp@ltisun.epfl.ch. (Please state your e-mail address as password).

7. Acknowledgments

The authors thank G. Caal who contributed many interesting ideas to the design and the coding of the preprocessor of the C++ extended language. They also thank G. Conti and K. Warkentyne for many fruitful discussions. All members of the "Laboratoire de Téléinformatique" have participated in this development for the last few years. Nevertheless we would like to mention particularly T. Branca, B. Brunner, M. O 'Dowd, G. Semaan and M. Riese. We also thank the anonymous reviewers for their helpful comments.

REFERENCES

1. CCITT. *SDL, Specification and Description Language*, CCITT Recommandations Z.100-Z.104, 1984.
2. CCITT. *CHILL*, CCITT Recommandations Z.200.
3. ISO DP9074. *Information processing systems - Open Systems Interconnection - ESTELLE - A formal description technique based on extended state transition model*, ISO, 1985.

4. ISO IS8807. *Information processing systems - Open Systems Interconnection - LOTOS - A formal description technique based on the temporal ordering of observational behaviour*, ISO, 1988.
5. A. S. Tanenbaum, *Computer Networks*, Prentice-Hall International, 2 edition, 1989.
6. R. Milner, *Communication and concurrency*, International series on computer science. Prentice Hall, 1989.
7. G. Conti, *"Méthodologie d'implantation des protocoles de communication"*, Doctoral thesis, Departement of Computer Science, Ecole Polytechnique Fédérale de Lausanne, CH-1015 Lausanne, Switzerland, May 1990.
8. R.J.A. Buhr, *System Design with ADA*, Prentice Hall, 1984.
9. B. Stroustrup and M.A. Ellis, *The Annotated C++ reference manual.* Addison-Wesley Publishing Company, 1991.
10. C.A.R. Hoare, *Monitors : An Operating System Structuring Concept*, Communications of the ACM. Oct 1974, pages 549-557.
11. M. Riese and G. Conti, *Drawbacks of ADA's synchronization mechanism and a simple solution*, 3. international workshop on real-time ADA issues, 1989.
12. C.A.R. Hoare, *Communicating Sequential Processes*, Communication of the ACM, 21(8):666-667, August 1978.
13. C.A.R. Hoare, *Communicating Sequential Processes*, International series on computer science, Prentice Hall, 1985.
14. R. Pike, *Newsqueak: A langage for communicating with Mice*, AT&T Bell Laboratories, Computing science technical report No. 143, 1989.
15. B. Wyatt, K. Kavi, S. Hufnagel, *Parallelism in Object-Oriented Languages: A survey*, IEEE software, november 1992.
16. G. Semaan and K. Warkentyne, *Simulating Networks with C-EX*, International Journal on Computer Simulation, 1993.
17. M. Odowd, *MEX, a programming language that supports discrete event simulation*, EUROMICRO'89, Cologne, sept 1989.
18. US DoD, *Reference Manual for the Ada programming language*, ANSI, February 1983.
19. A. Burns, *Programming in OCCAM2*, Addison Wesley, 1988.
20. Yonezawa and Tokoro, editors, *Object-Oriented Concurrent Programming*, The MIT Press Series in Computer Systems, 1987.
21. K. Smith and A. Chatterjee, *A C++ Environment for Distributed Application Execution*, Tech. Report ACT-ESP-275-90, Microelectronics Computer Technology Corp., Austin, Tex., 1990.
22. Hogrefe D., *Estelle, LOTOS und SDL - Standard Spezifikationssprachen für verteilte Systeme*, Springer Verlag, 1989.