Alien call - a synchronization mechanism for distributed processes

Bolesław J. Ciesielski
Institute of Informatics
University of Warsaw

4 May 1989

Abstract

A new communication mechanism for parallel processes is proposed. The alien call is designed primarily for high level languages featuring distributed processes. It is based on the rendezvous schema, but introduces some significant enhancements while omitting constructs most difficult to implement. The alien call is particularly suitable for object oriented languages because it uses a procedure call/message sending as the main tool for information passing. The proposed mechanism integrates in a simple way both synchronous and asynchronous communication patterns, so separate mechanisms are not needed (e.g. messages and rendezvous). Examples presented in the paper demonstrate the use of the alien call in solutions of several typical synchronization problems. The alien call is strong enough to be the only communication mechanism available in a programming language. On the other hand the alien call may be efficiently implemented using asynchronous messages as a primary hardware or system level mechanism. A prototype implementation was done on the LAN of PC-compatible computers using LOGLAN as an underlying language.

Key words: concurrency, distributed systems, synchronization, communication mechanism, programming languages

1 Introduction

The alien procedure call is an interprocess communication and synchronization mechanism integrating both synchronous and asynchronous elements.

It was designed and implemented originally for the programming language LOGLAN developed in the Institute of Informatics, University of Warsaw. LOGLAN belongs to the family of object oriented programming languages (SIMULA, SMALLTALK, ...). The main constructs for dealing with objects, i.e. classes and the notion of inheritance are borrowed from SIMULA-67, in LOGLAN however they are substantially extended. From our point of view the most interesting is LOGLAN's treatment of concurrency.

The parallelism in LOGLAN has an object oriented nature. Processes are classes, so they define a type - a set of process objects of that type. Process types are static and fixed in a given program while process objects may be dynamically generated and are deallocated when no longer useful. Moreover processes like all classes may inherit from other classes and may themselves be inherited from by other processes. This provides good facilities for non-system concurrent programming, especially for large distributed applications.

2 Processes in LOGLAN

The following example illustrates the use of processes in LOGLAN.

Listing 1: Simple processes without synchronization

```
program example1;

  unit powers:process(n : integer);
  (* declaration of a process type *)
    var
      x : integer;
    begin
      x := 1;  (*constructor *)
      return;  (* thread *)
      do              (* infinite loop *)
        x := x * n;
        writeln(x)
      od
    end powers;
var
  p1, p2, p3 : powers;
begin (* the main program - also a process *)
  p1 := new powers(5);     resume(p1);
  p2 := new powers(7);     resume(p2);
  p3 := new powers(11);    resume(p3);
  stop
end example1
```

The program in the Example 1 consists of the declaration of the process type powers and the main program body. A process is a class, so it may have parameters, local declarations and a body. Parameters of processes (as well as of procedures and classes) may have input, output or inout passing modes as in ADA. Processes are created dynamically with the help of the object generator **new**. However the process generation does not complete after the creation of the process object. The parent process executes the body of the newly created process until the first return statement is encountered (it is constructor phase). At that moment the process generation is completed. Output parameters are transmitted back to the parent process which continues its execution. The new child process remains suspended. It may be resumed (put into the execution) by another process using the **resume** statement. The **stop** statement unconditionally suspends the process executing it. In the above example the main program process creates three processes of the type powers, resumes them and stops itself. Three processes run in parallel continuously. They neither terminate nor communicate with each other.

3 Communication mechanism

Distributed processes do not share memory, so they cannot communicate through shared variables or semaphores. Instead an original communication and synchronization mechanism appropriate for distributed systems, the *alien procedure call*, is proposed for LOGLAN. It is based on the rendezvous but introduces some new concepts.

Regular (i.e. non-process) objects in LOGLAN may be accessed remotely by means of a dot notation:
X.V := ... X.V ... or
call X.P(...)
where X is an object of some class A, V is a local variable and P is a local procedure of A. For a process object only the second type of remote access (a procedure call) is allowed. It is a direct consequence of the distribution of processes. Remote access to a local variable of another process is not permitted, although it may be defined using the proposed mechanism. A process must just provide procedures for fetching and storing of the value of the variable. Those procedures must be made accessible to other processes by the alien call mechanism.

Syntax. Now we give the precise (yet informal) description of the semantics of the alien call.
An alien call is either:

- a procedure (or function) call performed by a remote access to a process object, or
- a call of a procedure which is a formal parameter of a process, or
- a call of a procedure which is a formal parameter of an alien-called procedure (this is a recursive definition).

The latter two cases come from the fact that a procedure which is passed as a parameter must be executed in the static environment of a process in which it is declared (similarly to calling by a remote access). However that form of an alien call is not used very often.

Every procedure declared directly inside a process or inherited from a superclass may potentially be alien-called by another process, but a given process object may selectively allow or inhibit alien calls to its own procedures. For this purpose the notion of an enable mask is introduced.

Enable mask is a private process attribute whose domain is the power set of all procedure names from the given process that may be potentially alien-called. At any moment the enable mask contains the subset of all callable procedures.

The receiving process allows alien calls only to the procedures contained in its enable mask. It is important to realize that the enable mask is associated with every process object (not a process type) and may be changed separately for each of them during the execution of a program. In fact the enable mask is a part of a process state.

A procedure is enabled in a process (an active object) if it belongs to the enable mask of that process. A procedure is disabled if it does not belong to that mask.

Immediately after generation of a process object its enable mask is empty (i.e. all procedures are disabled). A process may change its own enable mask using the constructs described below. The statement:

enable p1, ..., pn

enables the procedures with identifiers p1, ..., pn in the current process and the statement:

disable p1, ..., pn

disables the procedures with identifiers p1, ..., pn.

The alien procedure call has similar syntax to Remote Procedure Call (RPC) but its semantics is different. Both the calling process and the process in which the procedure is declared (i.e. the called process) are involved in an alien call. This way the alien call may be used as a synchronization mechanism.

Semantic. An alien procedure call is initiated by a calling process usu-

opisać alien call- jako pro- tokół

ally by means of a procedure call statement remotely accessing a procedure declared in another (called) process:

call X.P(e, v);

where X is a process object, P is its local procedure and e and v represent input and output parameters. A formal procedure call may also initiate an alien call.

The alien-called procedure is executed by the called process (as in rendezvous but not as in RPC). The calling process passes input parameters and waits for the completion of the execution of the procedure. Then the calling process reads back the output parameters and resumes the execution at the next statement. Thus the calling process is suspended almost all the time during an alien call.

The called process may not always be able to execute the procedure. The execution of the procedure will not begin before certain conditions are satisfied. First, the called process must not be suspended in any way. The only exception is that it may be waiting for an alien call (see the accept statement described below). Second, the given procedure must be enabled in the called process. Hence the calling process remains suspended and nothing happens until both of these conditions are satisfied. At any given moment there may be many processes waiting for an ability to alien call of the same procedure in the same process.

When the above two conditions are met one of the waiting processes is chosen and its request is served. The called process is interrupted and forced to execute the alien-called procedure in its own static and dynamic environment but with parameters passed earlier by the calling process.

Upon entry to the alien-called procedure all procedures become disabled in the called process (the enable mask is cleared). Therefore the procedure cannot be immediately interrupted by another alien call. However it may easily allow interrupts if it changes the enable mask by itself.

Upon exit from the alien called procedure the enable mask of the called process is *restored* to that from before the call (regardless of how it has been changed during the execution of the procedure). The called process is resumed at the point of interruption and continues to do whatever it was doing before. The calling process reads back the output parameters and resumes its execution after the call statement.

As can be seen from the above description, if we consider each of the processes separately, an alien call appears as a normal procedure call to the calling process and as an interrupt to the called process.

Besides the above base mechanism there are some additional language constructs associated with the alien call. A special form of the return statement (which ends the execution of a procedure): **return enable** p1, ..., pn

disable q1, ..., qn; allows to enable the procedures p1, ..., pn and disable the procedures q1,...,qn after the enable mask is restored on exit from the alien-called procedure. That form of return is legal only in the alien-called procedures and gives them the possibility to permanently change (not only for the period of the execution of a procedure) the enable mask of the called process.

Listing 2: Implementation of semaphores using the alien procedure call

```
unit semaphore: process(compNo, n:integer);
(* compNo is the number of the processor on which the active
   object of semaphore will be located. n is the semaphore counter.
   Its initial value is passed as a
   parameter by the creating process (must be >= 0). *)

  unit wait:procedure;
  (* enabled only if n > 0 *)
  begin
    n := n - 1;              (* decrement the semaphore counter *)
    if n = 0                 (* if we reached 0 *)
    then                     (* then disable further alien calls *)
      return disable wait    (* of procedure wait *)
    fi
  end wait;

  unit signal:procedure;
  (* always enabled *)
  begin
    n := n + 1;              (* increment the  counter *)
    return enable wait       (* surely,  n > 0, so enable *)
                             (* alien calls of wait *)
  end signal;

begin
  return;                    (* return to the creator *)

  (* Enable procedure signal  because
     this  operation  is  always allowed. *)
  enable signal;

  (* Enable procedure wait if the initial counter's
     value is greater than 0. Otherwise it may only
     be enabled by an alien call to signal. *)
```

```
  if  n > 0  then enable P fi;

  (* Now we wait indefinitely for the alien calls.
     For the moment a busy-waiting loop suffices. *)
  do
  od
end semaphore;


...
(* example use of the semaphore in main process  *)
var
  s : semaphore;
begin
  s := new semaphore(no, 2);
  resume(s);
  ...
  call s.wait;
  ...
  call s.signal;
  ...
end
```

In the above example a Dijkstra's semaphore is implemented using the alien procedure call. In a distributed system there are no shared variables so semaphores must be processes themselves. Once a process is created, a reference to it may be passed as a parameter to another process so it may be accessed by a number of processes. The process semaphore defined above has two alien callable procedures: wait and signal which correspond to standard operations on semaphores. The enable mask is manipulated in such a way that procedure signal is always enabled while procedure wait is enabled only if the semaphore counter is positive. Explicit queuing of the processes waiting for a semaphore to be open is not needed. It is done by the underlying alien call mechanism. Note that after the required initialization process semaphore has nothing to do except waiting for alien calls. It may do something else or loop as above. That follows from the fact that the alien callable procedures synchronize themselves modifying the enable mask without the need of a process body as a separate supervisor. This is not always the case, as we will see in the next example. Another interesting property of this solution is that an arbitrary fixed number of semaphores can be implemented by a single process. One must only repeat declarations of procedures wait and signal and the parameter n changing their names (e.g. wait1, wait2, ...). This is a direct consequence of the fact that the body of the process semaphore

albo
niepotrzebna
uwaga
albo
pogłebić

only waits for alien calls and is not involved in the synchronization.

A special construct is introduced to allow the called process avoid the busy waiting for an alien call of its procedure. The statement:
accept p1, ..., pn
adds the procedures p1, ..., pn to the current mask, and waits for an alien call of one of the currently enabled procedures (possibly other than p1, ..., pn). When such call occurs it is treated in a normal way but after the procedure return the enable mask is restored to that from before the accept statement (perhaps modified by return enable/disable). For example, suppose that procedures P and Q are enabled in a given process. If this process executes the statement accept Q, R the procedure R will become enabled (in addition to P and Q) and the process will suspend awaiting alien calls of P, Q, or R. Suppose that procedure Q is called and terminates with return enable R disable Q. Now the procedures P and R are enabled in the process.

Note that the accept statement alone (i.e. without any enable/disable statements or options) provides a sufficient communication mechanism. In this case the called process may execute the alien-called procedure only during the accept statement (because otherwise all procedures are disabled). It means that the enable mask may be forgotten altogether and the alien call may be used as a totally synchronous pure rendezvous (what it really is without the enable mask and interrupts). Other constructs are introduced to make partially asynchronous communication patterns possible: the called process does not have to wait for a communication.

The following example illustrates the use of the alien call as a purely synchronous mechanism:

Listing 3: Semaphores implemented by the alien call used as the pure rendezvous

```
unit semaphoreA:process(n:integer);
(* n is the semaphore counter. Its initial value is
   passed as a parameter by the creating process
   (must be >= 0). *)

  unit wait:procedure;
  (* enabled only if n > 0 *)
  begin
    n := n-1;              (* decrement counter *)
  end wait;

  unit signal:procedure;
  (* always enabled *)
  begin
```

```
    n := n+1;                (* increment  counter *)
  end signal;

begin
  return;

  (* We loop waiting for an alien call.
     An  accept  statement is  used to set
     proper enable mask and avoid busy waiting *)
  do
    if n > 0                 (* if the counter is positive *)
    then                     (* both wait and signal are allowed *)
      accept wait, signal
    else                     (* otherwise only signal is allowed *)
      accept signal
    fi;
  od
end semaphoreA;
```

In this example the alien procedures operate on the local variable only and do not modify the enable mask. It is the process body that controls the state of the enable mask. Since accept statement is used, there is no need for explicit manipulations on the enable mask. In fact, the above program can be understood without considering the enable mask at all.

4 The alien procedure call vs. other communication mechanisms

As we mentioned earlier, the alien procedure call is based on the rendezvous, a synchronous communication mechanism best known from its ADA implementation . Let us consider the simpler case for a moment. The programming language BNR Pascal contains the purest form of the rendezvous: only call and accept statements. The alien call is a strict extension to the pure rendezvous. It preserves all the features of the rendezvous and adds something new: an ability to define partially asynchronous communication patterns by means of the enable mask and alien calls acting as interrupts. This makes it more convenient for the programmer and often reduces the required number of processes (as we can see from the next example). On the other hand, the whole mechanism is more complex to describe and to implement. It must be used very carefully because of its low level constructs and non-structural nature. We believe, however, that in many applications the greater flexibil-

ity in the communication mechanism results in simpler, more readable and efficient programs. Rendezvous in ADA is even more flexible but also more complex and difficult to implement . It is also an extension to the basic rendezvous schema but goes into different direction. While remaining synchronous it introduces timeouts and guarded calls and entries. Both these extensions greatly increase the strength of the mechanism, particularly in real-time systems. We will show later how some of those constructs can be simulated by means of the alien call.

4.1 Using the alien call to define the asynchronous message passing

Asynchronous messages are often the basic communication mechanism offered by the hardware or the system level software (e.g. the data packets in RSX-11). Because of their flexibility they are often incorporated into high level programming languages (e.g. PLITS , CHILL). We will show how to implement asynchronous messages using the alien procedure call. To make the problem more concrete let us assume that we are implementing a print spooler in a multiuser system. User processes may send to it requests for printing files. They should not wait for the request to be serviced neither they are interested in the fact that the file is already printed. It is important that the user processes do not wait unless the print spooler queue is full. This is essentially the classic producer-consumer problem in the case of multiple producers but a single consumer.

Listing 4: Print spooler process

```
unit queue:class(type element; size:integer);
(* The auxiliary class implementing queues with
   a limited capacity. The class is parameterized by
   the element type and the maximum queue size *)

  unit insert:procedure(e:element); ...
    (* insert element into the queue *)
  unit delete:function:element; ...
    (* remove the first element from the queue *)
  unit empty:function:boolean; ...
    (* check if the queue is empty *)
  unit full:function:boolean; ...
    (* check if the queue is full
    (i.e. maximum size is reached) *)

end queue;
```

```
...
unit spooler:process;
(* The print spooler process. Requests for printing files
   are sent by alien-calling procedure print *)

var
  Q:queue,        (* queue of files to be printed *)
  f:filename,

 unit print:procedure(f:filename);
 (* Accept a request for printing file f. This procedure may
    only be called if the queue Q is not full  (forced  by
    the manipulations on the enable mask) *)
 begin
   call Q.insert(f);     (* queue the request *)
   if Q.full             (* if the queue is full now *)
   then                  (* disable further alien calls of print *)
     return disable print
   fi;                   (* otherwise restore the enable mask *)
 end print;

begin (* the body of the print spooler process *)
  (* create the file queue *)
  Q := new queue(filename, 50);
  (* end initialization *)
  return;

  (* main loop *)
  do
    (* get the first file from the queue *)
    disable print;       (* enter the critical section *)
    if Q.empty           (* if the file queue is empty *)
    then
      accept print       (* wait for a request *)
    fi;
    (* here we are sure that the queue is not empty *)
    f := Q.delete;       (* remove the first file from the queue *)
    enable print;        (* now the queue is not full, so we may *)
                         (* accept requests *)

    (* at last copy the file f to the printer *)
    ...
  od
```

```
end spooler;
```

The possibility of defining asynchronous communication with the alien call comes from the fact that the called process need not wait for the call. An alien procedure call is accepted and serviced regardless of what the process is doing at the given moment. The only requirement is that the called procedure must be enabled. In the example above procedure print is enabled almost all the time (except for the short critical section), so the request is accepted immediately. Thus we can maintain the message queue (files to be printed) directly within the receiving (spooler) process. This is not possible with the pure rendezvous. One must define an auxiliary process acting as a buffer for the messages. In the case of an unlimited capacity queue we can remove the statement if Q.full then ... from the procedure print. Now we are simulating exactly asynchronous messages as defined in PLITS. The statement: call spool.print(f) corresponds to: send f to spool ,

while the sequence: disable print; (* enter the critical section *) if Q.empty (* if the file queue is empty *) then accept print (* wait for a request *) fi; (* here we are sure that the queue is not empty *) f := Q.delete; (* remove the first file from the queue *) enable print; (* now the queue is not full, so we may *) (* accept requests *) replaces the statement: receive f. By using different procedures for different message types we can even define the PLITS mechanism of selective (with respect to the type) message receiving. However the selection of the message sender requires that each sending process uses distinct procedure for the message passing. This method is not as safe and elegant as in PLITS because it does not exclude the possibility of sending a message with a wrong sender identification (by calling a wrong procedure).

The dual problem of a single producer and multiple consumers can be solved similarly. The queue must be declared in the producer process which inserts products into the queue (with the similar critical section). The processes of consumers request the products from the producer by calling a procedure which removes the products from the queue. This procedure is enabled only if the queue is not empty. Note that in this case both the synchronous rendezvous and the asynchronous messages require an additional buffer process. Buffering may be done directly in the producer process only if we are able to interrupt it and force it to do some action (as in the case of the alien procedure call). In the most complex case of multiple producers and multiple consumers there is really a need of a separate buffer process. The simplest solution is the same as with the rendezvous.

4.2 Avoiding the starvation using the alien call

The last example shows the solution of another classic problem: readers and writers. There is a global resource to which writers can write and from which readers can read. Several readers may read simultaneously but when a writer is writing, nobody can access the resource. The straightforward solution based on simple mutual exclusion suffers from the starvation problem. If there are always readers ready to read, no writer can get access to the resource . We will present a solution which avoids the starvation by forcing certain discipline in the access to the resource. A special supervisor process is introduced to synchronize reading and writing.

Listing 5: Readers and writers without starvation

```
unit supervisor:process;
(* The supervisor process (with a single instance) servicing
requests for starting a reading or writing. *)
var
  waiting\_readers:integer,  (* number of waiting readers *)
  writing:boolean,          (* TRUE if a writer is writing *)

  (* the following variables concern the group of readers serviced
     in a single supervisor cycle *)
  cr:integer,               (* total number of readers *)
  br:integer,               (* number of readers which started *)
                            (* reading *)
  er:integer;               (* number of readers which finished *)
                            (* reading *)

  unit stamp:procedure;
  (* This procedure is called by a reader directly
before
  requesting the read access to the resource. It is used
to
  register and count the waiting readers. *)
  begin
     waiting\_readers := waiting\_readers+1;
  end stamp;

  unit startread:procedure;
  (* Called by a reader before reading. *)
  begin
    br := br+1;          (* next reader started the reading *)
    writing := false;    (* no writer is writing *)
  end startread;
```

```
  unit endread:procedure;
  (* Called by a reader after reading. *)
  begin
    er := er+1;          (* a reader finishes the reading *)
  end endread;

  unit startwrite:procedure;
  (* Called by a writer before the writing *)
  begin
    writing := true;     (* a writer is writing *)
  end startwrite;

  unit endwrite:procedure;
  (* Called by a writer after the writing *)
  begin
    (* No   actions   are   needed,   this   procedure   is
for
       synchronization purposes only. *)
  end endwrite;

begin
  return;
  (* allow the readers to register itself *)
  enable stamp;
  do
    (* start a new service cycle *)
    br := 0;  er := 0;

    (* We allow the first reader or writer to access the resource.
      The alien call mechanism ensures the strong fairness. *)
    accept startread, startwrite;
    if writing           (* if a writer was first *)
    then                 (* then we wait until it finishes the *)
      accept endwrite;   (* writing and finish the service cycle *)
    else
      (* If the reader was first we remember how many  readers
         are already waiting to access the resource. *)
      disable stamp;
      cr := waiting\_readers+1;  (* the number of waiting readers*)
      waiting\_readers := 0;      (* additional readers will be *)
                                (* serviced in the next readers *)
                                (* group *)
```

```
      enable stamp;

      (* We allow all currently waiting readers to start reading.
      At the same time we allow them to end reading. Readers which
      become waiting in the meantime are not serviced *)
      while br < cr
      do
        accept startread, endread;
      od;
      (* We allow all readers to end their reading *)
      while er < cr
      do
        accept endread
      od;
      (* and finish the service cycle *)
    fi;
  od
end supervisor;
...
unit reader:process(sv:supervisor);
(* A template for the reader process. Sv is the pointer to
the
supervisor process. *)
begin
  return;
  do
    call sv.stamp;     (* register a waiting reader *)
    call sv.startread; (* request a reading access *)
    ...
    (* reading *)
    ...
    call sv.endread;   (* reading is finished *)
    ...
  od
end reader;

unit writer:process(sv:supervisor);
(* A template for the writer process. Sv is the supervisor *)
begin
  return;
  do
    call sv.startwrite;  (* request a writing access *)
    ...
```

```
    (* writing *)
    ...
    call sv.endwrite;    (* writing is finished *)
   ...
  od
end writer;
...
begin (* the main program *)
 (* first we create and resume the supervisor process *)
 c := new supervisor;
 resume(c);

 (* next we create a number of readers and writers. All of
them
 receive the supervisor process as an argument. *)
 for i := 1 to num\_readers
 do
   resume(new reader(c));
 od;
 for i := 1 to num\_writers
 do
   resume(new writer(c));
 od;
 (* now we may end the main program *)
end
```

The idea of the above solution is that during the reading cycle only those readers are allowed to start reading which were waiting at the beginning of the cycle. All others must wait for the next reading cycle and a writer may be serviced before them. In this way the readers cannot starve the writers. As can easily be seen the solution would be simpler and more elegant if the alien call mechanism made available to a process the number of processes waiting for an alien call of the given procedure. In that case the procedure stamp and registration of waiting readers could be omitted. Instead a system count of waiting readers should be consulted. A simple extension to the alien call is described below. A solution using only the pure synchronous rendezvous would be more complex because the supervisor should consist of two separate processes: one of them for registering waiting readers and the second for actually controlling reading and writing.

5 Extensions to the base alien call mechanism

Two additional constructs are available when using the alien procedure call. They are described here because they are not implemented in the prototype implementation, although their implementation should not rise much problems. The standard function pending accepts a procedure identifier as a parameter. The expression
pending(P)
where P is an alien callable procedure has the value equal to the number of processes waiting for the service of alien calls of the procedure P. Besides other uses this operation makes easier to avoid the starvation. In the above example the statement:
cr := pending(startread)
replaces the sequence:
disable stamp;
cr := waiting_readers+1;
waiting_readers := 0;
enable stamp;
and procedure stamp may be deleted from the supervisor process.

The second extension introduces truly asynchronous messages. The statement
send X.P(e)
where X is a process, P its procedure and e represent input parameters acts like call X.P(e) except that the calling process does not wait for the completion of the alien called procedure. In this case the procedure P may have input parameters only. In fact the send statement is the asynchronous message sending. The message may be received either synchronously by means of the accept statement or asynchronously by enabling the given procedure and accepting an interrupt.

6 Conclusion

The proposed communication mechanism has both synchronous and asynchronous features which are integrated in a simple and consistent manner. The alien procedure call is flexible and convenient for the high level language programmer. Moreover it can be efficiently implemented in a distributed environment. Although it lacks some features of the more complex mechanisms (such as Ada's rendezvous) it is strong enough to be the only communication and synchronization mechanism available in a programming language.

References

[1] A. Kreczmar, A. Salwicki and M. Warpechowski, LOGLAN'88 - Report on the Programming Language, LNCS vol. 414, Springer 1990.

[2] US Department of Defense, Ada Programming Language, Military Standard, ANSI/MIL-STD-1815A, Ada Joint Program Office, 1983

[3] N.D. Gammage, R.F. Kamel, L.M. Casey, 'Remote Rendezvous', Software - Practice and Experience, vol. 17, no. 10 (October 1987), pp. 741-755.

[4] R.E. Filman, D.P. Friedman, Coordinated Computing: Tools and Techniques for Distributed Software, McGraw-Hill, 1984

[5] DEC, RSX-11M/M-PLUS Executive Reference Manual, Digital Equipment Corporation, Maynard, M.A., 1979

[6] J.A. Feldman, 'High Level Programming for Distributed Computing', CACM, vol. 22, no. 6 (June 1979), pp. 353-368

[7] Recommendation Z.200 (CCITT High Level Language CHILL), CCITT AP VII - No. 21-E, UIT, Geneva 1984

[8] M. Ben-Ari, Principles of Concurrent Programming, Prentice-Hall International (UK) Ltd., 1983