AN ALTERNATIVE TO THE COMMUNICATION PRIMITIVES IN ADA

Jan Stroet
Informatica/Computer Graphics
University of Nijmegen
Nijmegen, The Netherlands

1. Abstract

A critical look is taken at the ADA communication primitives by comparing them to the ITP (Input Tool Process) model, the model for process communication developed at Nijmegen. The comparison is done by means of example solutions to several problems in both models.
It is shown that by using features extracted from the ITP model, the communication facilities in ADA could be improved considerably with respect to orthogonality, clarity, flexibility and power.

2. Criticism

In a previous note of Van den Bos [3] in fact most of our criticism on the ADA communication facility is mentioned, i.e.

- Accept entries and accept statements occur in the middle of the executable code, and each entry point may be connected with a different accept body, even to accept entries with the same name; this can be rather confusing;

- The calling process is suspended until the called process has executed its accept statement. This lays a strong restriction on the parallel behaviour of the processes, especially when the called process has to execute many statements or suspends itself in its accept statement;

- Asymmetry of the identification of processes: the caller can address the destinated task, but a called task cannot select its customers;

- Communication primitives may be intermixed with all other kinds of ADA primitives.

In this paper we show how the communication facility in ADA could be improved. The improvements have been derived from the ITP model [4] for communicating processes, developed at the University of Nijmegen. The latter model is based on message exchange, with the communication protocol specified by means of a variant of path expressions.
We start by presenting a brief introduction to a subset of the original ITP model. Following that we will present an alternative notation for the communication facility in ADA by means of relevant examples from the ADA rationale [1] and reference manual [2].

3. An ITP subset

In interactive computing, and also in communication between processes, input and output play a predominant, closely coupled role. With this idea in mind the ITP model has been developed.
Exchange of information between processes is performed by message passing. Output for a sending process is input for a receiving process.

The basic concept in the ITP model is the tool. A process is a specific kind of tool. Tools can be built out of other tools. Two kinds of tools are distinguished, abstract tools and basic tools.

Basic tools are the lowest level tools. They have a receive rule. A process can communicate with another process by sending a message to a basic tool of that process. The receive rule of the basic tool specifies what type of message such a basic tool accepts. A basic tool in fact acts as a message slot. The type of the message sent must correspond with the type of the message specified in the basic tool.

The higher level tools or abstract tools have an input rule instead of a receive rule. This input rule specifies, in a regular expression, how that tool is composed out of other tools, and finally out of basic tools. Hence an abstract tool specifies what pattern of input is expected, expressed in terms of tools out of which it is built.

An abstract tool has in general the following appearance:

```
tool name = input input-pattern end

    declarations of variables
    and internal (lower level) tools

    init initialisations end

    tool body (executable statements)
end
```

Whenever a tool becomes active, which means that its specified input pattern is a candidate for occurring, the init statement (if present) is executed to perform certain initialisations. When the input pattern has occurred (the input rule is satisfied) the semantic action, called tool body, is executed.
Input patterns are specified by means of input expressions, controlled by conditionals. Sequencing (; operator), selection (+ operator), repetition ($ operator), and testing (: operator) are control structures which can be used in an input rule. The operators ; + and $ have tools as operands, the operator : has a boolean condition and a tool as operands.

It is possible to bind whole or part of an input expression to an external process or set of processes by the destination operator (->). This has the effect that (this part of) the input expression can only be satisfied by (one of) the process(es) specified.

Because tools are built out of other tools this model makes it possible to specify input patterns in a clear, hierarchical and modular way. Structured programming is almost a natural consequence of using the ITP model.

The highest level abstract tool is an independent process; this is similar to a task in ADA. The name of the process is the name of the highest level tool.
The lower level tools, that is the internal tools, are sequential objects; they behave like input-driven procedures.
Processes may send information to another process by addressing a basic tool of this process. This is done by a send statement, which looks like:

send process_name.basic_tool_name (message)

In the send statement the specification of the basic tool name and the message are optional. (In fact the process name is also optional; this however is not applied in this paper.) When the basic tool name is omitted any basic tool of the designated process which can accept the type of message sent is a possible consumer of the message. When in the send statement the message is omitted the send is called a signal.
A process issueing a send is suspended until the message is consumed by the designated process. A send statement may occur in any executable part of a tool, that is in the init rule and the tool body.

4. Comparison of ADA examples

Most of the relevant examples in the ADA rationale [1] and reference manual [2] are applications of service processes. Several of these examples are presented here.
For easy comparison both the ADA solutions (at the left hand side) and the ITP solutions are given. Because the focus of this paper is ADA's primitives, the ITP solutions were tailored after the ADA solutions. This mapping sometimes resulted in solutions that do not smoothly fit in the ITP spirit.

4.1 Protected array

As a first example we refer to the 'Protected_array' example of ADA.

```
task PROTECTED_ARRAY is
  -- INDEX and ELEM are global types
  entry READ (I : in INDEX; V : out ELEM);
  entry WRITE(I : in INDEX; E : in  ELEM);
end;

task body PROTECTED_ARRAY is
  TABLE : array(INDEX'FIRST .. INDEX'LAST) of ELEM :=
                (INDEX'FIRST .. INDEX'LAST => 0);

begin
  loop
    select
      accept READ (I : in INDEX; V  : out ELEM) do
        V := TABLE(I);
      end READ;
    or
      accept WRITE(I : in INDEX; E  : in  ELEM) do
        TABLE(I) := E;
      end WRITE;
    end select;
  end loop;
end PROTECTED_ARRAY;
```

```
tool Protected_array = input ( read + write) $ end;

     table : array(index'first .. index'last) of elem;
     i : index;  e: elem;

     tool read = receive i;
          send sender(table(i));
     end;

     tool write = receive i, e;
          table(i):=e;
     end;

     init for j in index'first .. index'last loop
               table(j) := 0;
          end loop;
     end;
end;
```

The input rule of the process 'Protected_array' specifies that it waits for an infinite number ($-operator, which is similar to the looping construct in ADA) of occurrences of the expression between parentheses. This expression 'read+write' indicates that it waits for either a 'read' or a 'write' (the 'select accept read .. or accept write ..' construct in ADA). The input rule of the process gives clear information about the kind of input this process expects.
Because the input expression is not bound to any external process it accepts input from any process which sends messages to one of its basic tools 'read' or 'write'.

When this process gets started its init rule is executed. Subsequently the input expression determines which tools are candidates for activation; in this

case the basic tools 'read' and 'write'. Now the process is suspended, waiting for a message sent to either 'read' or 'write'.

Each tool contains a (possibly empty) tool body. The body of a basic tool is executed when a message is accepted for that basic tool. The body of a higher level tool is executed when its input pattern has occurred.

In the above example, when a process issues

send Protected_array.write(1,5)

the message (1,5) is accepted by basic tool 'write'. Its body is executed, which results in making the first element (i=1) of 'table' equal to 5 (e=5). After the body is executed tool 'write' has occurred and so the expression 'read + write' is satisfied. Because the $-operator is applied to this expression, it is activated again. This means that both 'read' and 'write' are candidates for receiving the next message. Only one of the active basic tools however can actually receive the message.

When a

send Protected_array.read(10)

is issued the receive rule of the tool 'read' is satisfied. This results in executing the body of the basic tool 'read'. With the help of the sender primitive, which yields the name of the process from which the last message was accepted, the contents of 'table(10)' is returned to the caller.

The body of a basic tool is comparable to the body of an accept entry in ADA. However there are some differences:

- Each tool has only one body, so one specific basic tool performs one specific job. In ADA it is possible to have different accept bodies connected with an entry with a certain name;

- The sending process is suspended only until the sent message has been consumed. This guarantees much more concurrency than ADA does, especially when it takes a lot of time before a possible answer has been determined. The ADA approach in our case would amount to suspending a process until the body of the basic tool has been executed. After execution of the accept body ADA may return parameters. In the ITP approach the service process must send a message to the user. This has the disadvantage that when the user does not wait for the answer the service process hangs. A time out mechanism will be one of the possible solutions. However these details are beyond the scope of this paper.

In ADA with each accept entry a FIFO queue is connected. In the ITP solutions given here we assume a FIFO queue connected with each basic tool. The behaviour of such a queue is similar to the accept entry queue in ADA.
We want to remark that queues are no part of the official ITP model.

4.2 Signals and semaphores

The solutions for a task 'Signal' are as follows:

```
generic task SIGNAL is
   entry SEND;
   entry WAIT;
end SIGNAL;

task body SIGNAL is
   RECEIVED : BOOLEAN := FALSE;
begin
   loop
      select
         accept SEND;
         RECEIVED := TRUE;
      or when RECEIVED =>
         accept WAIT;
         RECEIVED := FALSE;
      end select;
   end loop;
end SIGNAL;
```

```
tool Signal =
     input (send + ¦received¦:wait)$ end;

     received : boolean;

     tool send = receive;
          received := true;
     end;

     tool wait = receive;
          received := false;
     end;

     init received := false;
     end;
end;
```

In the input rule of 'Signal' the test operator ':' is used. Tool 'wait' may occur, depending on whether the prefix test yields true or false. So the expression 'send+¦received¦:wait' is equivalent to 'send+wait' when 'received' is true and equivalent to 'send' when 'received' is false.

When process 'Signal' is started 'received' is set to false in its init rule. Consequently process 'Signal' will only accept (empty) messages (signals) to basic tool 'send'. When a message for basic tool 'send' has been accepted, the boolean 'received' is made true in the body of this basic tool. Subsequently (as a consequence of the $-operator) messages can be accepted for the basic tools 'send' and 'wait'. After one message for basic tool 'wait' has been accepted, 'received' is set to false again.
From the input rule of this process is immediately clear what this process does. The same holds for the following process 'Semaphore':

```
task SEMAPHORE is
   entry P;
   entry V;
end;

task body SEMAPHORE is
begin
   loop
      accept P;
      accept V;
   end loop;
end;
```

```
tool Semaphore = input ( P ; V )$ end;
     tool P = receive;
     end;
     tool V = receive;
     end;
end;
```

In this example the followed-by operator (';') is used. This process accepts an infinite number of times a message for basic tool 'P' followed by a message for basic tool 'V'. So a signal to basic tool 'V' will only be accepted when previously a corresponding signal to basic tool 'P' has occurred.

4.3 Line to char

The process 'Line_to_char' alternately fills a buffer with characters (by accepting a send to 'put_line') and then empties the buffer (by accepting sends to 'get_char') until all characters in the buffer have been distributed.
In fact the description of this process reflects itself immediately in the input rule specification.

```
task LINE_TO_CHAR is
  type LINE is array (1 .. 80) of CHARACTER;
  entry PUT_LINE  (L : in  LINE);
  entry GET_CHAR (C : out CHARACTER);
end;

task body LINE_TO_CHAR is
  BUFFER : LINE;
begin
  loop
    accept PUT_LINE(L : in LINE) do
      BUFFER := L;
    end PUT_LINE;
    for I in 1 .. 80 loop
      accept GET_CHAR(C : out CHARACTER) do
        C := BUFFER(I);
      end GET_CHAR;
    end loop;
  end loop;
end;
```

```
tool Line_to_char =
     input ( put_line;
             (|char_left|: get_char)$) $ end;

     buffer : array (1 .. 80) of character;
     char_left : boolean;
     index : integer;

     tool put_line = receive buffer;
          char_left := true;
          index := 1;
     end;

     tool get_char = receive;
          send sender(buffer(index));
          index := index + 1;
          char_left := index <= 80;
     end;
end;
```

In the input rule for process 'Line_to_char' an example is seen of a conditional repetition. As soon as the test 'char_left' yields false (there are no characters in the buffer) this repetition ends.

The functioning of this process is analogous to the 'Protected_array' process. Both are typical examples of service processes: they do not identify their input sources and they return information to the process which requested it, by means of the sender primitive.

This example illustrates another advantage of the ITP method. The patterns of input are specified in a distinct place, clearly separated from administrative chores performed by the executable code. The input rule gives a specification of the interactions occurring with other processes. This gives a good insight in what the process is supposed to do. In ADA this is less clear because the accept entries and so the interaction specifications may occur at any place in the executable code.

4.4 Reader writer

Several processes can read from and write to process 'Reader_writer'. Writers have priority over readers. New readers will not be permitted to read if there is a writer waiting. When a writer has finished all waiting readers will have priority over the next writer.

The ITP model does not allow procedure entries such as in ADA. Information about different processes has to be extracted from the queues for the basic tools. Therefore the ITP solution looks somewhat different from the ADA solution:

```
task READER_WRITER is
  procedure READ(V : out ELEM);
  entry WRITE(E : in ELEM);
end;

task body READER_WRITER is
  VARIABLE: ELEM;
  READERS : INTEGER := 0;
  entry START_READ;
  entry STOP_READ;

  procedure READ(V : out ELEM) is
  begin
    START_READ;
    V := VARIABLE;
    STOP_READ;
  end;

begin
  accept WRITE(E : in ELEM) do
    VARIABLE := E;
    N := START_READ'COUNT;
  end;

  loop
    select
      when WRITE'COUNT = 0 =>  -- this is safe
        accept START_READ;
        READERS := READERS + 1;
    or
        accept STOP_READ;
        READERS := READERS - 1;
    or
      when READERS = 0 =>
        accept WRITE(E : in ELEM) do
          VARIABLE := E;
        end;
        loop
          select
            accept START_READ;
            READERS := READERS + 1;
          else
            exit;
          end select;
        end loop;
    end select;
  end loop;
end READER_WRITER;
```

```
tool Reader_writer =
     input write;
           (|write'count=0 or (readers>0 and read'count>0)|:read
               + |readers = 0| : write
           )$
     end;

     variable : elem; readers : integer;

     tool read = receive;
          send sender (variable);
          if write'count = 0     --  no writers in queue
          then readers := read'count;
          else readers := readers - 1;
          end if;
     end;

     tool write = receive variable;
          readers := read'count; -- readers with priority
     end;

     init readers := 0;  end;
end;
```

The variable 'readers' indicates the number of readers that have priority over the next writer.
First a send to basic tool 'write' will be accepted. Subsequently when no external process has issued a send the basic tools 'read' or 'write', 'Reader_writer' waits for a send to any of these basic tools (both tests yield true); when there are readers which have priority over writers then 'Reader_writer' only accepts sends to basic tool 'read'; finally when sends are issued to basic tool 'write' and all readers which had priority over writing have been serviced (readers = 0) 'Reader_writer' will only accept a message to 'write'.

4.5 Control

Process 'Control' satisfies disk requests which are ordered in FIFO priority queues. The number of requests per queue is administered in the array 'pending'. The subscript for each array element serves as a priority level.
In order to service a request a send to basic tool 'signin' must be accepted first and its occurrence recorded. In a second step the tool 'perform' must be executed. The input parameter of tool 'perform' indicates the priority level to be serviced.
Process 'Control' proceeds by :
1. waiting for the first request to 'signin' if all previous requests have been serviced;

2. accepting all pending requests;
3. executing the request with the highest priority;
4. restart the mainloop of the input expression to take care of any signin's issued in the meantime.

From this description the input rule for process 'Control' is immediately derived.
In this input rule another example is seen of a conditional repetition. As soon as the test yields false (in fact case 1 (total=0) and 2 (sigin'count>0) are both handled) the repetition will end. After the completion of the repetition tool 'satisfy' is activated. In its init rule the priority level to be serviced is determined. When this is done one request will be accepted and serviced.

```
task CONTROL is
  subtype LEVEL is INTEGER range 1 .. 50;
  procedure REQUEST (L : LEVEL; D : DATA);
end;

task body CONTROL is
  entry SIGN_IN  (L : LEVEL);
  entry PERFORM (LEVEL'FIRST .. LEVEL'LAST)(D : DATA);
  PENDING : array (LEVEL'FIRST ..LEVEL'LAST) of INTEGER :=
                          (LEVEL'FIRST .. LEVEL'LAST => 0);
  TOTAL   : INTEGER := 0;

  procedure REQUEST(L : LEVEL; D : DATA) is
  begin
    SIGN_IN(L);
    PERFORM(L)(D);
  end;

begin
  loop
    if TOTAL = 0 then
      -- no request to be served:  wait if necessary
      accept SIGN_IN(L : LEVEL) do
        PENDING(L) := PENDING(L) + 1;
        TOTAL := 1;
      end SIGN_IN;
    end if;
    loop  -- accept any pending SIGN_IN call without waiting
      select
        accept SIGN_IN(L : LEVEL) do
          PENDING(L) := PENDING(L) + 1;
          TOTAL := TOTAL + 1;
        end SIGN_IN;
      else
        exit;
      end select;
    end loop;

    for I in reverse LEVEL'FIRST .. LEVEL'LAST loop
      if PENDING(I) > 0 then
        accept PERFORM(I)(D : DATA) do
          -- satisfy the request of highest level
        end;
        PENDING(I) := PENDING(I) - 1;
        TOTAL := TOTAL - 1;
        exit;  -- restart main loop in order to accept new requests
      end if;
    end loop;
  end loop;
end CONTROL;
```

```
tool Control =
     input ((|total=0 or signin'count>0|:signin) $ ; satisfy) $
     end;

     subtype level is integer range 1..50;
     total : integer; prty : level;
     pending : array(level'first .. level'last) of integer;

     tool signin = receive prty;
          pending(prty) := pending(prty) + 1;
          total := total + 1;
     end;

     tool satisfy = input perform[i] end;

          i : level;

          tool perform [j:level] = receive d;
               d : data;
               satisfy request(d);
               pending(j) := pending(j) - 1;
               total := total - 1;
          end;

          init i := level'last; -- determine which priority
                                -- level should be serviced
               while pending(i) = 0 and i >= level'first
                loop i := i - 1 end loop;
          end;
     end;

     init for j in level'first .. level'last;
          loop pending(j):=0; end loop; -- no request received
          total := 0;
     end;
end;
```

The procedure entry in connection with the accept entry enhances the power of the inter-process calling facility in ADA. Here it makes it possible to include the protocol in the process which demands it, without the user having any notice of it. In the ITP approach there are no procedure entries, so something different has to be done. Obliging the user to follow the protocol seems rather unsatisfactory. We have therefore chosen for a different approach. In the ITP model it is possible to create a family of scheduling processes, each member dedicated to one user process. A family of processes is distinguished from other processes by having (read only) parameters (enclosed between square brackets). The parameter for such a scheduling process would in this case be the name of the user process:

```
tool Sched_control[user:process] = input request end;

     l : level; d : data;

     tool request = receive l,d;
          send Control.signin(l);
          send Control.perform[l](d);
          send user;      -- acknowledgement to the user
     end;
end;
```

In fact the user now only communicates with its private scheduling process by issueing a send Sched_control[user].request(level,data).
By choosing an appropriate name for that scheduling process the user will not be aware of talking to an intermediate process, rather than directly to 'Control'.

4.6 Printer driver

The next process drives a chain printer. If the printer does not receive a printing request for 10 seconds, while the chain is going, the chain is stopped. A further print request will restart the chain and after a one second delay printing can start again:

```
task PRINTER_DRIVER is
  entry PRINT(L : LINE);
end;

task body PRINTER_DRIVER is
  CHAIN_GOING : BOOLEAN := FALSE;
  BUFFER      : LINE;
begin
  loop
    select
      accept PRINT(L : LINE) do
        BUFFER := L;
      end;
      if not CHAIN_GOING then
        -- start the chain
        delay 1.0*SECONDS;
        CHAIN_GOING := TRUE;
      end if;
      -- print the line
    or
      when CHAIN_GOING =>
        delay 10.0*SECONDS;
        -- stop the chain
        CHAIN_GOING := FALSE;
    end select;
  end loop;
end;
```

```
tool Printer_driver =
     input (printline + |chain_going|:stop_printer)$ end;

     chain_going : boolean;

     tool printline =
          input line ; |not chain_going|:start_printer end;

          buffer : array (1 .. 80) of character;

          tool line = receive buffer;
          end;

          tool start_printer = input clock[1] end;
               chain_going := true;

               init start_chain end       -- system routine
          end;

          print_out (buffer);             -- system routine
     end;

     tool stop_printer = input clock[10] end;
          stop_chain;          --  system routine
          chain_going := false;
     end;

     init chain_going := false;
     end;
end;
```

Each process has a basic tool 'clock'. This basic tool has one parameter, indicating the number of seconds after which an interrupt should occur. When 'clock' is one of the active basic tools, it occurs if within the specified number of seconds no message for another active basic tool arrives.

Initially the process only waits for the occurrence of tool 'printline', since the chain is not going ('chain_going' is false). Consequently a send from any process to 'line' will be accepted. After a line is received tool 'start_printer' is activated ('not chain_going' yields true). In the init statement of this tool the chain is started, following which basic tool 'clock[1]' will be activated. This implies a one second delay, since this is the only active basic tool at that moment. When basic tool 'clock[1]' has occurred, the input rule of 'printline' is satisfied. Consequently the body of

tool 'printline' will be executed, that is the line sent will be printed. Because of the repetition ($) in the input rule of 'Printer_driver' the expression between parentheses becomes active again. Since the printer chain is going this process now waits for either a send to basic tool 'line' from any process, or for a clock interrupt after 10 seconds (tool 'stop_printer'), following which the chain will be stopped.

Inspection of this example shows that intermixing of lines from several printing processes can happen. To avoid this it is necessary for the printer driver to accept lines from one specific process only, until all printing for that process is done. The ITP solution is given in the next example. It assumes that the last line to be printed is identified by an EOF character.

```
tool Printer_driver =
     input (printline + |chain_going|:stop_printer)$ end;

     customer : process set;
     chain_going, first_line : boolean;

     tool printline =
          input customer -> line ;
                |not chain_going|:start_printer
          end;

          buffer : array (1 .. 80) of character;

          tool line = receive buffer;
               if first_line
               then customer := {sender};
               end if;
               if buffer(1) = EOF
               then first line := true;
                    customer := Universe;
               end if;
          end;

          tool start_printer = input clock[1] end;
               chain_going := true;

               init start_chain end       -- system routine
          end;

          print_out (buffer);             -- system routine
     end;

     tool delayed_stop = input clock[10] end;
          stop_chain;          -- system routine
          chain_going := false;
     end;

     init chain_going := false;
          first_line := true; customer := Universe;
     end;
end;
```

This solution only accepts messages for basic tool 'line' from the members of process set 'customer'. At initialisation of process 'Printer_driver' this set is made equal to the Universal set, containing the names of all processes in the system. So when 'printline' gets active for the first time a send to basic tool 'line' will be accepted from any process.
When the first message to 'line' is accepted the process set 'customer' is set equal to the process which sent this message. Hence the next time 'printline' gets active, only a message to 'line' from that specific process is accepted, since it is the only member of the set 'customer'. Upon every new activation of tool 'printline' only a message from this process is accepted, until the last line is sent. At that time 'customer' is set to the Universal set again.
In the mean time sends issued by other processes which are directed to basic tool 'line' will be delayed until the current process has finished printing. Following that the next process will get control over the printer.
The queue for basic tool 'line' is not strictly a FIFO queue any more. This is caused by the fact that 'Printer_driver' at a certain moment only accepts

messages from processes which are members of the set 'customer'.

This example shows the lack of power of the ADA primitives. A process should be able to identify a customer and set up a connection with an external process. In ADA this seems only possible by letting the user explicitly serialise the printer with the help of semaphores, or by sending the entire file to be printed as a single message.

4.7 Diskhead scheduler

As a last example a disk head scheduler is shown. It handles data requests to and from a moving head disk. The requests are grouped into separate queues for each track and all requests for a particular track are serviced together. There is a basic tool for each track, so all queues are independent.
A separate process, called 'Disk_head_scheduler', controls the arm movement and the choice of the track:

```
task DISK_HEAD_SCHEDULER is
  type TRACK  is new INTEGER range 1 .. 200;
  type DATA   is ... -- other parameters of transfer
  procedure TRANSMIT(TN : TRACK; D : DATA);
end;

task body DISK_HEAD_SCHEDULER is
  type DIRECTION is (UP, DOWN);
  INVERSE: constant  array (UP .. DOWN) of DIRECTION :=
                     (UP => DOWN, DOWN => UP);
  STEP   : constant  array (UP .. DOWN) of INTEGER range -1 .. 1 :=
                     (UP => 1, DOWN => -1);
  WAITING: array (TRACK'FIRST .. TRACK'LAST) of INTEGER :=
                (TRACK'FIRST .. TRACK'LAST => 0);
  COUNT  : array (UP .. DOWN) of INTEGER := (UP .. DOWN => 0);
  MOVE   : DIRECTION := DOWN;
  ARM_POSITION : TRACK := 1;

  entry SIGN_IN(T : TRACK);
  entry FIND_TRACK(REQUESTS : out INTEGER; TRACK_NO : out TRACK);

  task TRACK_MANAGER is
    entry TRANSFER(TRACK'FIRST .. TRACK'LAST)(D : DATA);
  end;

  procedure TRANSMIT(TN : TRACK; D : DATA) is
  begin
    SIGN_IN(TN);
    TRACK_MANAGER.TRANSFER(TN)(D);
  end;

  task body TRACK_MANAGER is
    NO_OF_REQUESTS: INTEGER;
    CURRENT_TRACK : TRACK;
  begin
    loop
      FIND_TRACK(NO_OF_REQUESTS, CURRENT_TRACK);
      while NO_OF_REQUESTS > 0 loop
        accept TRANSFER(CURRENT_TRACK)(D : DATA) do
          -- do actual I/O
          NO_OF_REQUESTS := NO_OF_REQUESTS - 1;
        end TRANSFER;
      end loop;
    end loop;
  end TRACK_MANAGER;
```

```
tool Disk_head_scheduler =
    input (|count(up)+count(down) > 0|: findtrack + signin) $ end;

    type track is new  integer range 1..200;
    type direction is (up,down);
    inverse : constant array (up .. down) of direction;
    step : constant array (up .. down) of integer range -1 .. 1;
    waiting : array (track'first .. track'last) of integer;
    count : array(up .. down) of integer;
    move : direction;
    armposition : track;

    tool findtrack = receive;
         requests : integer; trackno : track;
         if count(move) = 0
         then move := inverse(move);
         else armposition := armposition + step(move);
         end if;
         while waiting (armposition) = 0
          loop armposition := armposition + step(move); end loop;
         count(move) := count(move) - waiting(armposition);
         requests := waiting (armposition);
         track_no := armposition;
         waiting(armposition) := 0;
         send sender(requests,track_no);
    end;

    tool signin = receive t;
         t : track;
         if t<armposition
         then count(down) := count(down) + 1;
         elsif t>armposition
         then count(up) := count(up) + 1;
         else count(inverse(move)) := count(inverse(move)) + 1;
         end if;
         waiting(t) := waiting(t) + 1;
    end;

    init inverse(up):=down; inverse(down):=up; count(up) := 0;
         count(down) := 0;step(up):=1; step(down):=-1;
         waiting:=0; move:=down; armpostion := 1;
    end;
end;
```

```
begin  -- DISK_HEAD_SCHEDULER
  initiate TRACK_MANAGER;
  loop
    select
      when COUNT(UP) + COUNT(DOWN) > 0 =>
        accept FIND_TRACK(REQUESTS: out INTEGER; TRACK_NO: out TRACK) do
          if COUNT(MOVE) = 0 then
            MOVE := INVERSE(MOVE);
          else
            ARM_POSITION := ARM_POSITION + STEP(MOVE);
          end if;
          while WAITING(ARM_POSITION) = 0 loop
            ARM_POSITION := ARM_POSITION + STEP(MOVE);
          end loop;
          COUNT(MOVE) := COUNT(MOVE) - WAITING(ARM_POSITION);
          REQUESTS  := WAITING(ARM_POSITION);
          TRACK_NO  := ARM_POSITION;
          WAITING(ARM_POSITION) := 0;
        end FIND_TRACK;
    or
      accept SIGN_IN(T : TRACK) do
        if T < ARM_POSITION then
          COUNT(DOWN) := COUNT(DOWN) + 1;
        elsif T > ARM_POSITION then
          COUNT(UP) := COUNT(UP) + 1;
        else
          COUNT(INVERSE(MOVE)) := COUNT(INVERSE(MOVE)) + 1;
        end if;
        WAITING(T) := WAITING(T) + 1;
      end SIGN_IN;
    end select;
  end loop;
end DISK_HEAD_SCHEDULER;
```

```
tool Track_manager =
     input (Disk_head_scheduler->adm;
            |no_of_requests>0|:transfer[current_track] $
     end;

     no_of_requests : integer;  current_track : track;

     tool adm = receive no_of requests, current_track;
          init send Disk_head_scheduler.findtrack;
          end;
     end;

     tool transfer [tn : track] = receive d;
          d : data;
          do IO;
          no_of_requests := no_of_requests - 1;
     end;
end;
```

The user, probably via a dedicated scheduling process (such as in section 4.5) would issue:

```
send Disk_head_scheduler.sign_in (t);
send Track_manager.transfer[t](record);
```

The first send causes the request to be administered and the second send causes the I/O to be performed.

A nominal investment of time in studying the ITP solution gives a good insight in what these processes are doing, and how they communicate with each other and with the user processes. The ADA solution however is much more cryptic about that.
This comparison of both solutions clearly illustrates that for more complex examples the readability and clearness of the ITP solution increases considerably over the ADA solution. This is inherent to the structured, hierarchical approach of the ITP model.

5. Conclusions

Using examples presented earlier in the ADA rationale and ADA reference manual a new approach to a communication facility has been shown. This approach is based upon input specifications, because input is one of the basic principles in communication. Exchange of information between processes only occurs by means of message passing. This has led to a somewhat lower level of communication mechanism (send/receive). The benefit of it however is that real parallel execution is possible. This is especially so when the receiving process has to execute many statements before it can return a reply to the requestor. The send and receive primitives take care of the message exchange, and synchronisation occurs implicitly. As soon as a message has been consumed both the sending and the receiving process may proceed.

Another advantage is that input specifications lead to a hierarchical, structured approach of describing the communication pattern. The communication specification is separated from the normal executable statements. It has its

own primitive operators for selection, sequencing and (conditional) repetition. This also makes it possible to discard such nasty constructs as else in ADA.

It seems a restriction of the ITP method that tools and hence basic tools have one body. The body of a basic tool can be compared with the accept body in ADA. However it is rather unnatural to have more than one body connected with a certain entry. Especially for service processes an entry corresponds with a certain task to be performed. It can only be confusing when an entry would do different things depending on the local environment of the service process.

A last remark about this new approach is that it is possible for a sending process to identify its destination, and it is possible for a process waiting for information to identify its source. This latter is certainly necessary for service processes wanting to communicate with a user process for a certain amount of time, without any other process being able to interfere with this communication.

By and large this new approach seems to lead to a simple and more powerful way of communication, which is easier to read because of its structured appearance. Especially the communication specification clearly separated from the executable code is one of the great advantages of the ITP model.

REFERENCES

1. J.D Ichbiah et al
   Rationale for the design of the ADA programming language, Sigplan Notices, June 1979.

2. J.D. Ichbiah et al
   Preliminary ADA reference manual, Sigplan Notices, June 1979.

3. J. van den Bos
   Comments on ADA process communication, Sigplan Notices, June 1980.

4. J. van den Bos, M.J. Plasmeijer, J.W.M. Stroet
   Communicating processes based on input specifications, Internal report no. 22, Informatica/Computer Graphics, University of Nijmegen, The Netherlands, April 1980.