4 Direct Interaction and Communication of Actors

Introduction

So far, the various actors did not interact with each other; but usually, there will be a certain amount of interaction between actors during their lifetime.

There are two ways in which actors can interact: *directly*, by one actor calling a procedure of another, and *indirectly*, by access to shared data. In this chapter, we focus on direct interaction. Chapter 5 addresses indirect interaction.

Note that we regard communication to be bidirectional, i.e., information is sent from a *sender* to a *receiver*. An actor can be either a sender or a receiver. Furthermore, an actor is allowed to undergo a transition from sender to receiver (or vice versa).

4.1 Direct Interaction and Communication of Actors in Ada

4.1.1 Entries, Accept Statements, and the Rendezvous

"The execution of an Ada program consists of the execution of one or more tasks. Each task represents a separate thread of control that proceeds independently and concurrently between the points where it interacts with other tasks." (ARM, 1995, [9(1)])

Direct communication between tasks is achieved by a technique known as the *rendezvous*.

A task passes information to another (and thereby requests that something be done with these information) by calling an *entry* of that task. Entries can be declared in a task specification similarly to the declaration of a procedure in a package specification:

```
task_definition ::=
  {task_item}
[ PRIVATE
  {task_item}]
END [task_identifier]

task_item ::= entry_declaration | representation_clause
entry_declaration ::=
  ENTRY defining_identifier [(discrete_subtype_definition)] parameter_profile;
```

Following below is a simple example of an entry declaration:

```
task T is
   entry E(Some_Parameter : Some_Type);
end T;
```

An entry resembles a procedure in many ways: it can have a formal parameter list (with `in`, `out`, and `in out` parameters) and is called like a procedure:

```
T.E(Some_Parameter => Some_Actual_Parameter);
```

Specifying the task is required since a task name cannot appear in a `use` clause (it cannot appear in a `with` clause either). For completeness, we note that a task can have a private part in which entries not visible to the external user (private entries) can be declared.

Consequently, these private entries can only be called by local tasks, i.e., tasks that are locally declared inside the task containing the private entries.

Direct communication in Ada "is based on a client/server model of interaction. One task, the server, declares a set of services that it is prepared to offer to other tasks (the clients)." (Burns, 1998, [p. 87])

To initiate communication, the caller issues an entry call on the server (callee) by identifying both the callee and the required entry. The server indicates its willingness to provide the requested procedure by executing an accept statement. Thus for the communication to occur, both tasks must have issued their respective requests. When they have done so, we say that the rendezvous (so called because both tasks "meet" at the entry at the same time) takes place. In the following, we will use the term *rendezvous* rather than *communication.*

(Barnes, 1996, [p. 396]) gives the following description of the rendezvous: "The statements to be obeyed during a rendezvous are described by corresponding accept statements in the body of the task containing of the entry. An accept statements usually takes the form

```
accept E(Some_Parameter : Some_Type) do
   -- handled sequence of statements
end E;
```

The formal parameters of the entry `E` are repeated in the same way that a procedure body repeats the formal parameters of a corresponding procedure declaration. The `end` is optionally followed by the name of the entry. A significant difference is that the body of the accept statement is just a sequence of statements plus optional exception handlers. Any local declarations must be provided by writing a local block.

The most important difference between an entry call and a procedure call is that in the case of a procedure, the task that calls the procedure also immediately executes the procedure body whereas in the case of an entry, one task calls the entry, but the corresponding accept statement is executed by the task owning the entry. Moreover, the accept statement cannot be executed until a task calls the entry and the task owning the entry reaches the accept statement. Naturally, one of these will occur first and the task concerned will then be suspended until the other reaches the corresponding statement. When this occurs, the sequence of statements of the accept statement is executed by the called task while the calling task remains suspended. This interaction is called a rendezvous. When the end of the accept statement is reached, the rendezvous is completed and both tasks then proceed independently. The parameter mechanism is exactly as for a subprogram call; note that expressions in the actual parameter list are evaluated before the call is issued."

In the following example (Burns, 1998, [p. 87]), consider a task that realizes a telephone operator who provides a directory enquiry service:

```
task type Telephone_Operator is
   entry Directory_Enquiry(Person : in Name; Addr : in Address;
                           Num : out Number);
end Telephone_Operator;

An_Op : Telephone_Operator;
```

A client may then issue an entry call:

```
-- client task
An_Op.Directory_Enquiry("Stuart Jones", "Wall Street", Stuarts_Number);
```

and the telephone operator can execute a corresponding accept statement:

```
accept Directory_Enquiry(Person : in Name; Addr : in Address;
                         Num : out Number) do
   -- look up telephone number
```

```
    -- and assign the value to Num
end Directory_Enquiry;
```

Example 4.1: Entry call and accept statement

"It is quite possible that the client and the server will not both be in position to communicate at exactly the same time. For example, the operator may be willing to accept a service request, but there may be no subscribers issuing an entry call. For the simple rendezvous case, the server is obliged to wait for a call; whilst it is waiting, it frees up any processing resource (for example, the processor) it is using; a task which is generally waiting for some event to occur, is usually termed *suspended* or *blocked*. Similarly, if a client issues a request and the server has not indicated that it is prepared to accept the request (either because it is already servicing another request or it is doing something else), then the client must wait.[1] Clients waiting for a service at a particular entry are queued."

There is a logical entry queue associated with each task entry and accepting an entry call removes exactly one call from the corresponding queue. This is called servicing the entry. A given entry may have several accept statements; the execution of any of these services this entry. Since a task, when running, has a single thread of control, it can only execute one statement at a time. Hence, if a task, `T`, executes an accept statement, `E`, for some client, `C`, other clients calling an entry of `T` generally have to wait and are queued (see Example 4.2 for an illustration). That is, `T` can only deal with at most one client at any time; in this case, `C`. As mentioned above, for the rendezvous to take place, both tasks must meet at the same entry at the same time. The entry is hence a synchronization point and the Ada rendezvous is a form of synchronous actor communication.

Now look at Figure 4.1 for a pictorial description of the rendezvous.

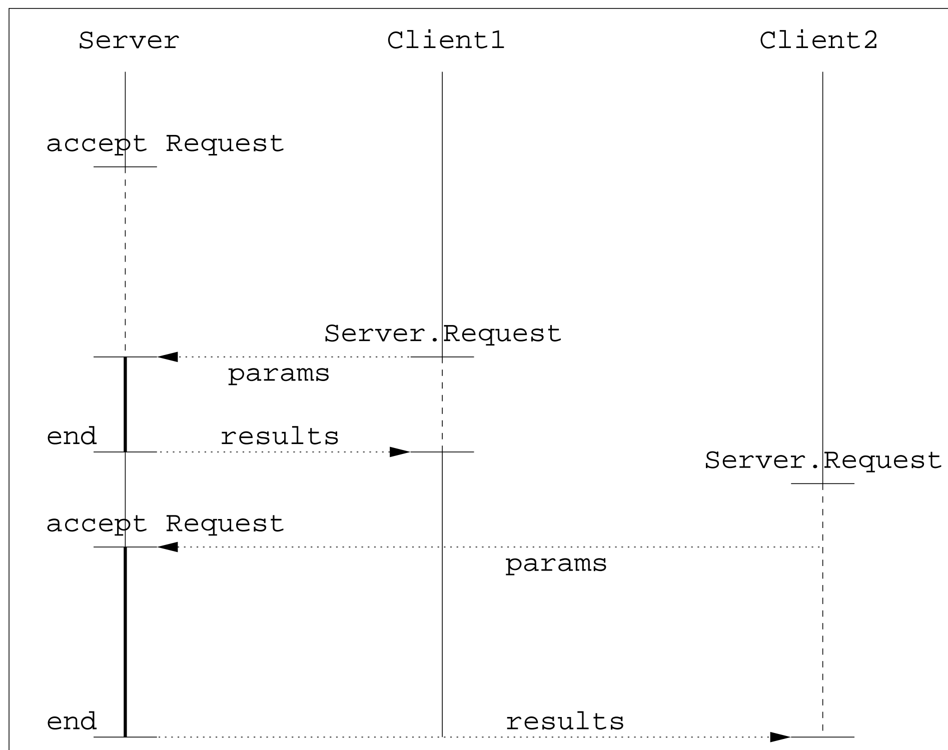

Figure 4.1: The Ada rendezvous

[1] we will, in this chapter, describe mechanisms that allow the server and the client to withdraw their offer of communication if they cannot enter the rendezvous immediately or within a specified time period

4.1.2 General Notes on the Rendezvous

The nature of the rendezvous to be a synchronization means between two tasks can cause minor problems that a programmer needs to be aware of. The first is that, although synchronization is essential for communication, this interaction blocks the caller for the duration of the rendezvous (remember, and look at Figure 4.1, if the callee is not in a position to enter communication, then the caller is blocked until the callee is ready as well). Actors, however, are meant to be active and independent and should not be bound unduly to other actors. Hence, it is desirable to make the blocking phase for both parties as short as possible, i.e., to provide as short a rendezvous as possible. The caller is blocked whilst the corresponding accept statement is executed by the callee; therefore, the key to the solution is to place only those statements in the accept statement that are really required for the communication. Actions that are not essential for the communication, but which have to be completed before a new client can be serviced (i.e., bookkeeping actions) should be performed outside the rendezvous (after the accept statement) after the client has left. In particular, so-called *potentially blocking operations* should be avoided; these include: entry calls, delay statements, protected operations, I/O operations, task creation, or yet another accept statement. Thus, in our telephone example above, logging all calls can be conveniently done after the actual call has been dealt with.

The other issue to bear in mind is exceptions. (Barnes, 1996, [p. 428]) says why: "The exception `Tasking_Error` is concerned with general communication failure. As we have seen, it is raised if a failure occurs during task activation and it is also raised in the caller of a rendezvous if the server is aborted. In addition, no matter how a task is completed, all tasks still queued on its entries receive `Tasking_Error`. Similarly, calling an entry of a task that is already completed also raises `Tasking_Error` in the caller.

If an exception is raised during a rendezvous (as a consequence of an action by the called task) and is not handled by the accept statement, then it is propagated into both tasks as the same exception on the grounds that both need to know. Of course, if the accept statement handles the exception internally, then that is the end of the matter anyway.

It might be convenient for the called task, the server, to inform the calling task, the user, of some event by the explicit raising of an exception. In such a case, it is likely that the server will not wish to take any action and so a null handler will be required. So in outline, we might write

```
begin
   accept E(...) do
      ...
      raise Error; -- tell user
      ...
   end E;
   ...
exception
   when Error =>
      null; -- server forgets
end;
```

If an exception is not handled by a task at all, then, like the main subprogram, the task is abandoned and the exception is lost; it is not propagated to the parent unit because it would be too disruptive to do so. However, we might expect the runtime environment to provide a diagnostic message. If it does not, it might be good practice for all significant tasks to have a general exception handler at the outermost level in order to guard against the loss of exceptions and the consequential silent death of the task."

It can be argued that this mechanism can be used as a form of communication between the server and the client. It is, of course, a very limited form since only the server can

"notify" the client and it is not possible to transmit parameters via an exception. However, being superimposed by the rendezvous, this form of "communication" is also synchronous. We have mentioned this for the sake of completeness only and will not further address it.

"There are a few constraints on the sequence of statements in an accept statement. They may include entry calls, subprogram calls, blocks, and further accept statements. However, an accept statement may not contain an asynchronous select statement (to be introduced later in this chapter), nor an accept statement for the same entry or one of the same family." (Barnes, 1996, [p. 399])

4.1.3 Select Statements

We have said that the Ada rendezvous is a form of synchronous actor communication. To achieve the synchronization, both tasks must meet at the same entry at the same time—the caller must have called the entry and the callee must be on the corresponding accept statement. This approach, however, is too restrictive if a client has called an entry a corresponding accept statement the server is currently not at. If there is only one client, then no progress can be made: the server waits to provide a service that can never be requested since the caller is blocked as it is waiting for the server. Consider a task that models a protected variable, i.e., a variable that provides for mutually exclusive access:

```
task Protected_Variable is
   entry Read(X : out Item);
   entry Write(X : in Item);
end Protected_Variable;

task body Protected_Variable is
   V : Item := Initial_Value;
begin
   loop
      accept Write(X : in Item) do  -- (1)
         V := X;
      end Write;
      accept Read(X : out Item) do
         X := V;
      end Read;
   end loop;
end Protected_Variable;
```

Example 4.2: A protected variable

The server task is currently blocked at the accept statement corresponding to the `Write` entry, i.e., at the position indicated by (1). If now the client calls `Read`, deadlock results. The requested service, `Read`, cannot be accepted since the server is currently not in a position to do so. On the other hand, the client cannot call `Write`—which would cause the server to "move on" to the `Read` accept statement (after servicing the `Write` request)—since it is blocked. No further progress is possible.

Generally, the simple entry-call-accept-statement interaction can only take place when the server and the client have issued their respective requests. We would like, however, the task to be able to select one of several possible rendezvouses. This can be achieved by the select statement, the definition of which reads as follows:

```
select_statement ::=
  selective_accept
  | timed_entry_call
  | conditional_entry_call
```

| asynchronous_select

The first form, selective_accept, is a controlling facility (for the rendezvous) for the server executing accept statements whereas the other three forms can only be used in conjunction with entry calls—that is, on the client side. Using the first form, we can now rewrite the body of `Protected_Variable` to meet the above mentioned requirements:

```
task body Protected_Variable is
   V : Item := Initial_Value;
begin
   loop
      select
         accept Write(X : in Item) do
            V := X;
         end Write;
      or
         accept Read(X : out Item) do
            X := V;
         end Read;
      end select;
   end loop;
end Protected_Variable;
```

Example 4.3: Select statement with multiple accept statements

"The task body consists of an endless loop containing a single select statement. A select statement starts with the reserved word `select` and finishes with `end select`; it contains two or more alternatives separated by `or`. In this example, each alternative consists of an accept statement—one for `Write` and one for `Read`.

When we encounter the select statement, various possibilities have to be considered according to whether calls of `Read` or `Write` or both or neither have been made.

- If neither `Read` nor `Write` has been called, then the task is suspended until one or the other is called and then the corresponding accept statement is obeyed.
- If calls of `Read` are queued, but none of `Write` are queued, then a call of `Read` is accepted and vice versa, with `Read` and `Write` reversed.
- If calls of both `Read` and `Write` are queued, then an arbitrary choice is made.

Thus each execution of the select statement results in one of its branches being obeyed and one call to `Read` or `Write` being dealt with. We can think of the task as corresponding to a person servicing two queues of customers waiting for two different services. If only one queue has customers, then the server deals with it; if there are no customers, then the server waits for the first irrespective of the service required; if both queues exist, the server rather capriciously serves either and makes an arbitrary choice each time." (Barnes, 1996, [p. 412])

In order to exercise even more control, each branch of the select statement can be preceded by a guarding expression `when Condition =>`, where `Condition` is of type `Boolean`. Each time such a select statement is encountered, all guards are evaluated and only accept statements whose guards evaluate to `True` are considered as select alternatives (the corresponding entries are then said to be *open*). The dynamic semantics of a guarded select statement is thus as for a select statement without guards but consisting only of those branches for which the conditions are true. Only if an entry is open, can (but need not) its corresponding accept statement be executed. Incidentally, if we omit a guard, a default guard of the form `when True =>` is assumed. Note that when all guards evaluate to `False` (are *closed*), the exception `Program_Error` is raised.

We now continue to describe the other forms of the select statement. Using the general select statement (guarded or not), enables a task to wait for any of its services to be requested. If, however, there are no such requests, the task might be unduly inactive, which might not be desirable. It would be appropriate to let the task, whilst there are no requests, do something else or even interprete this situation as a kind of emergency case. To achieve this, one or more branches of a select statement are allowed to start with a delay statement rather than an accept statement. Consider a database management system task that decides to reorganize the database if there are no requests for other operations within, say, 10 minutes:

```
select
   accept Update_Tuple(The_Tuple : Tuple_Type) do
      -- update The_Tuple
   end Update_Tuple;
or
   accept Insert_Tuple(The_Tuple : Tuple_Type) do
      -- insert The_Tuple
   end Insert_Tuple;
or
   accept Delete_Tuple(The_Tuple : Tuple_Type) do
      -- delete The_Tuple
   end Delete_Tuple;
or
   delay 10 * Minutes;
   Reorganize_Database;
end select;
...
```

Example 4.4: A select statement with a delay alternative

If no client requests an insert, update, or delete within 10 minutes, the DBMS server starts to reorganize the database. If, however, after (say) 9 minutes an update is requested, then the delay is cancelled and the corresponding accept statement is executed. "A delay alternative can be guarded and indeed, there could be several in a select statement although clearly only the shortest one with a true guard can be taken. It should be realized that if one of the accept statements is obeyed, then any delay is cancelled—we can think of a delay alternative as waiting for a rendezvous with a clock. A delay is, of course, set from the start of the select statement and reset each time the select statement is encountered. Finally, note that it is the start of the rendezvous that matters rather than its completion as far as the time out is concerned." (Barnes, 1996, [p. 417])

Continuing with our DBMS example, let us now consider the case in which the server should immediately perform some kind of action (such as flushing the system buffer to the hard disks) if no service is requested. In a first attempt, we might write:

```
...
or
   accept Delete_Tuple(The_Tuple : Tuple_Type) do
      -- delete The_Tuple
   end Delete_Tuple;
or
   delay 0.0; -- times out immediately
   Flush_System_Buffer;
end select;
```

which forces the task to take the delay branch if none of the other branches can be immediately accepted. The delay times out at once and, hence, flushing the buffers is instantly

performed. Although this approach solves the problem, it is somewhat contrived and there is a risk of race conditions interfering with normal execution flow (note that a `delay 0.0` cannot naively be "optimized away", see Chapter 6). We can, in fact, do better in Ada by using a select statement with an else part:

```
...
or
   accept Delete_Tuple(The_Tuple : Tuple_Type) do
      -- delete The_Tuple
   end Delete_Tuple;
else
   Flush_System_Buffer;
end select;
...
```

Here, the last alternative is preceded by an `else` instead of an `or` and contains simply a sequence of statements. Otherwise, it behaves essentially as the previous `delay 0.0` solution. Note that a select statement with an else part cannot cause `Program_Error` to be raised due to all guards (if any) being closed. A select statement is not allowed to contain both a delay alternative and an else part, however.

Please note that, generally, we cannot use an else part containing a delay as a substitute for a delay alternative (in the special case of the `delay 0.0`, this was possible). If we had changed the last `or` in Example 4.4 into an `else`, then the effect of the delay would be quite different. The delay then happens to be just a statement in a sequence of statements. In particular, the delay, once obeyed, cannot be cancelled. The DBMS server thus would wait for 10 minutes (regardless of whether requests for operations have arrived in the meantime) and then start the reorganization of the database!

"There are two other forms of select statement which are rather different; they concern a single entry call rather than one or more accept statements. The timed entry call allows a sequence of statements to be taken as an alternative to an entry call if it is not accepted within the specified duration. Thus

```
select
   Operator.Call("Put out fire");
or
   delay 1 * Minutes;
   Fire_Brigade.Call;
end select;
```

will call the fire brigade if the operator does not accept the call within one minute. The entry call can be to a task or protected object and it is the acceptance of the call that matters rather than its completion. Finally, there is the conditional entry call. Thus

```
select
   Operator.Call("Put out fire");
else
   Fire_Brigade.Call;
end select;
```

will call the fire brigade if the operator cannot immediately accept the call.

Timed and conditional entry calls are quite different to the general select statement. (The latter is used by the server when accepting entry calls whereas the former are used, as we saw above, by the client to issue entry calls [cmnt. by author]) They concern only a single unguarded call and so these select statements always have exactly two branches—one with the entry call and the other with the alternative sequence of statements. Timed and

conditional calls apply only to entries. They do not apply to procedures or even to entries renamed as procedures.

Timed and conditional calls are useful if a task does not want to be unduly delayed when a server task or protected object is busy. They correspond to a customer in a shop giving up and leaving the queue after waiting for a time or, in the conditional case, a highly impatient customer leaving at once if not immediately served." (Barnes, 1996, [p. 418]) They thus provide a means that allows the client to withdraw its offer of communication if it cannot enter the rendezvous within a specified time period or immediately.

Before we turn to the final form of the select statement—asynchronous transfer of control—, we introduce a further concept of Ada tasking: the requeue facility. The reason for the postponement of ATC is its nature: ATC is by far the most sophisticated and complicated part of Ada tasking. The authors, therefore, feel that it is essential that everything else related to Ada tasking be explained prior to approaching this subtle topic of Ada tasking.

4.1.4 The Requeue Statement

"A requeue_statement can be used to complete an accept_statement or entry_body, while redirecting the corresponding entry call to a new (or the same) entry queue. Such a requeue can be performed with or without allowing an intermediate cancellation of the call, due to an abort or the expiration of a delay.

requeue_statement ::= REQUEUE entry_name [WITH ABORT];

The execution of a requeue_statement proceeds by first evaluating the entry_name, including the prefix identifying the target task or protected object and the expression identifying the entry within an entry family, if any. The entry_body or accept_statement enclosing the requeue_statement is then completed, finalized, and left (see 7.6.1).

For the execution of a requeue on an entry of a target task, after leaving the enclosing callable construct, the named entry is checked to see if it is open and the requeued call is either selected immediately or queued, as for a normal entry call (see 9.5.3).

For the execution of a requeue on an entry of a target protected object, after leaving the enclosing callable construct:

- if the requeue is an internal requeue (that is, the requeue is back on an entry of the same protected object—see 9.5), the call is added to the queue of the named entry and the ongoing protected action continues (see 9.5.1);
- if the requeue is an external requeue (that is, the target protected object is not implicitly the same as the current object—see 9.5), a protected action is started on the target object and proceeds as for a normal entry call (see 9.5.3).

If the requeue_statement includes the reserved words WITH ABORT (it is a requeue-with-abort), then:

- if the original entry call has been aborted (see 9.8), then the requeue acts as an abort completion point for the call, and the call is cancelled and no requeue is performed;
- if the original entry call was timed (or conditional), then the original expiration time is the expiration time for the requeued call.

If the reserved words WITH ABORT do not appear, then the call remains protected against cancellation while queued as the result of the requeue_statement." (ARM, 1995, [9.5.4]

As an example, consider a client task that issues an entry call on a server which, in a corresponding accept statement, requeues the caller (in this case, to the same entry):

```
-- Server
```

```
Seq1;
loop
   select
      when Guard_1 =>
         accept E do
            Seq_A;
            if Condition then
               Seq_B;
               requeue E;
            end if;
            Seq_C;
         end E;
   or
      when Guard_2 =>
         accept E;
            Seq_D;
         end E;
   end select;
   Seq2;
end loop;
```

Figure 4.2 is an illustration of a possible execution. For the first iteration of the loop, let `Some_Guard` be true but `Another_Guard` shall be false. For the second iteration, let it be vice versa.

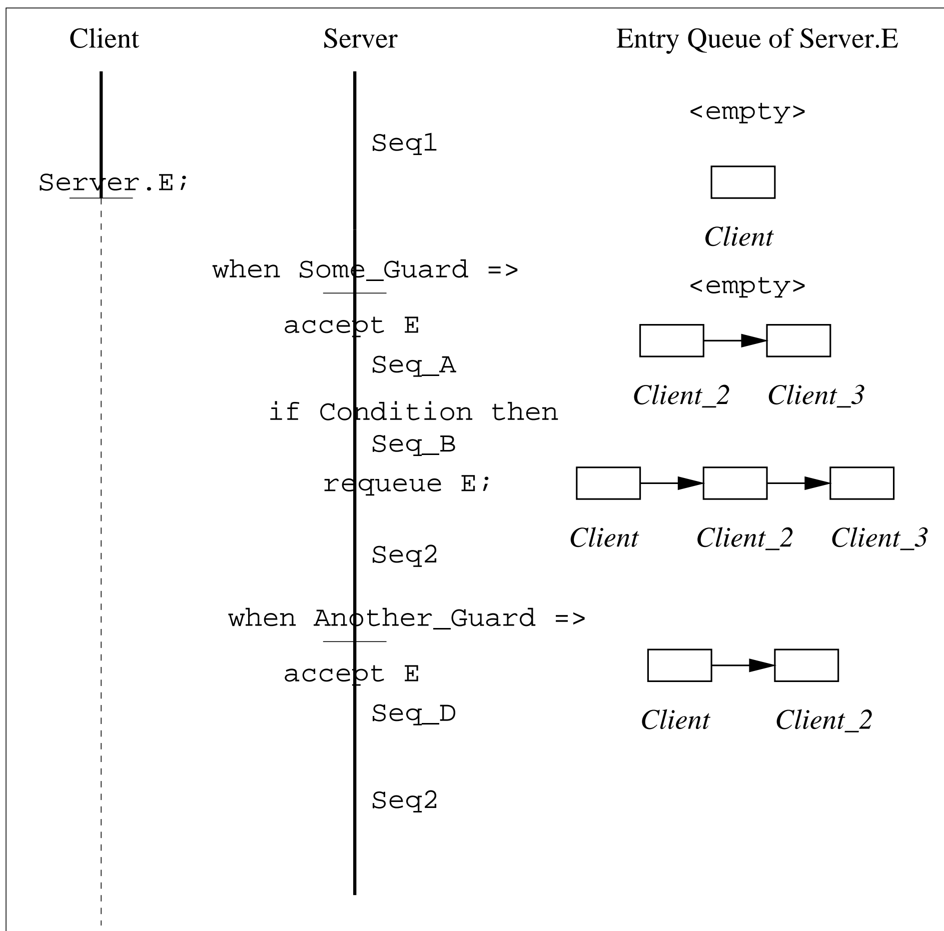

Figure 4.2: Illustration of requeue

As can be seen from Figure 4.2, the caller is *not* aware of the fact that its call is being requeued. Note that we have assumed that entry queues are processed in FIFO order; we will not further address this here since it is more comprehensively covered in Chapter 6.

"The optional `with abort` clause has two uses. When a task is on an entry queue, it will remain there until serviced unless it made a timed entry call or is aborted. Once the task has been accepted (or starts to execute the entry of a protected object), the time-out is cancelled and the effect of any abort attempt is postponed until the task comes out of the entry. There is, however, a question as to what should happen with requeue. Consider the time-out issue; clearly, two views can be taken:

1. As the first call has been accepted, the time-out should now be cancelled (that is, it cannot have an effect).
2. If the requeue puts the calling task back onto an entry queue, the time-out expiry should again be possible.

A similar argument can be made with abort; if the task is again on an entry queue, it should be abortable. The requeue statement allows both views to be programmed; the default does not allow further time-outs or aborts; the addition of the `with abort` clause enables the task to be removed from the second entry." (Burns, 1998, [p. 179])

To appreciate the benefits of requeuing, we will now study a problem that is known as the *Resource Allocation Problem*. This is a fundamental problem in all aspects of concurrent programming. Given a set of resources, we would like to offer an arbitrary number of resources (up to some maximum, of course) to external clients. Either all requested resources are given to the client or none are and the caller blocks until the resources are free.

The means of communication in Ada considered so far allow for a technique known as *avoidance synchronization*—guards (and entry barriers for protected objects) can be used to prevent communication from starting if the conditions are not appropriate. However, "this resource allocation problem is difficult to program with avoidance synchronization. In order to determine the size of the request, the communication must be accepted and the parameter read. But if, having read the parameter, the internal state of the resource controller is such that there are currently not enough resources available, the communication must be terminated and the client must try again." (Burns, 1998, [p. 168]) Terminating the rendezvous without allocating the resources clearly requires the server to inform the client that the resources could not be granted. Hence, a control parameter has to be transmitted, which is cumbersome and error-prone. Unfortunately, in a guard or entry barrier, access to the formal parameters of the accept statement or entry, respectively, is not allowed. Thus

```
-- resource controller task
select
   when Available_Resources >= Required_Resources =>
      accept Allocate_Resources(Required_Resources : Natural) do
        ...
```

is not legal Ada. After presenting a few workarounds, all of which suffer from several drawbacks, (Burns, 1998, [p. 176]) presents the final solution for the Resource Allocation Problem using the requeue facility and a protected object. We, here, outline a resource controller task that is based on Burns' solution (but which is suffering from the risk of busy waiting!):

```
task Resource_Controller is
   entry Allocate_Resources(Required_Resources : Natural);
   entry Release_Resources(Released_Resources : Natural);
private
   The_Resource : ...;
end Resource_Controller;

task body Resource_Controller is
```

```
begin
   loop
      select
         when Available_Resources > 0 =>
            accept Allocate_Resources(Required_Resources : Natural) do
               if Required_Resources <= Available_Resources then
                  -- allocate Required_Resources instances
                  -- of The_Resource
               else
                  requeue Allocate_Resources; -- try again later
               end if;
            end Allocate_Resources;
      or
         ...
```

Example 4.5: The use of requeue in the Resource Allocation Problem

We see that if there are not enough resources available, the caller is requeued to the same entry for another trial. "The key notation behind requeue is to move the task (which has been through one guard or barrier ...) to 'beyond' another guard [or the same, cmnt. by author]. For an analogy, consider a person (task) waiting to enter a room (protected object) which has one or more doors (guarded entries) giving access to the room. Once inside, the person can be ejected (requeued) from the room and once again be placed behind a (potentially closed) door." (Burns, 1998, [p. 175])

4.1.5 Asynchronous Transfer of Control (ATC)

The final form of the select statement is the asynchronous select statement which enables a task to respond *quickly* to an asynchronous event such as an expiration of a delay or a completion of an entry call. To be even more precise, an activity can be abandoned if some condition arises (such as running out of time) so that an alternative sequence can be executed instead.

```
asynchronous_select ::=
   SELECT
      triggering_alternative
   THEN ABORT
      abortable_part
   END SELECT;
triggering_alternative ::= triggering_statement [sequence_of_statements]
triggering_statement ::= entry_call_statement | delay_statement
abortable_part ::= sequence_of_statements
```

"For the execution of an asynchronous_select whose triggering_statement is an entry_call_statement, the entry_name and actual parameters are evaluated as for a simple entry call (see 9.5.3), and the entry call is issued. If the entry call is queued (or requeued-with-abort), then the abortable_part is executed. If the entry call is selected immediately, and never requeued-with-abort, then the abortable_part is never started.

For the execution of an asynchronous_select whose triggering_statement is a delay_statement, the delay_expression is evaluated and the expiration time is determined, as for a normal delay_statement. If the expiration time has not already passed, the abortable_part is executed.

If the abortable_part completes and is left prior to completion of the triggering_statement, an attempt to cancel the triggering_statement is made. If the attempt to cancel succeeds (see 9.5.3 and 9.6), the asynchronous_select is complete.

If the triggering_statement completes other than due to cancellation, the abortable_part is aborted (if started but not yet completed—see 9.8). If the triggering_statement completes normally, the optional sequence_of_statements of the triggering_alternative is executed after the abortable_part is left." (ARM, 1995, [9.7.4])

In essence, when the triggering event completes first, it aborts the abortable part, and when the abortable part completes first, it aborts the triggering event.

"*There is a restriction on the sequence of statements that can appear in the abortable part. It must not contain an accept statement.* The reason for this is to keep the implementation as simple as possible.

If the cancellation of the triggering event fails, because the protected action or rendezvous has started, or has been requeued (without abort), then the asynchronous select statement waits for the triggering event to complete before executing the optional sequence of statements following the triggering statement. ... Clearly, it is possible for the triggering event to occur even before the abortable part has started its execution. In this case, the abortable is not executed and therefore not aborted.

Consider the following example:

```
task Server is
   entry Act_Event;
end Server;

task To_Be_Interrupted;

task body Server is
begin
   ...
   accept Act_Event do
      Seq2;
   end Act_Event;
   ...
end Server;

task body To_Be_Interrupted is
begin
   ...
   select --- ATC statement
      Server.Act_Event;
      Seq3;
   then abort
      Seq1;
   end select;
   Seq4;
   ...
end To_Be_Interrupted;
```

When the above ATC statement is executed, the statements which are executed will depend on the order of events that occur:

```
if the rendezvous is available immediately then
   Server.Act_Event is issued
   Seq2 is executed
   Seq3 is executed
   Seq4 is executed
elsif no rendezvous starts before Seq1 finishes then
   Server.Act_Event is issued
   Seq1 is executed
```

```
      Server.Act_Event is cancelled
      Seq4 is executed
   elsif the rendezvous finishes before Seq1 finishes then
      Server.Act_Event is issued
      partial execution of Seq1 occurs concurrently with Seq2
      Seq1 is aborted and finalized
      Seq3 is executed
      Seq4 is executed
   else (the rendezvous finishes after Seq1 finishes)
      Server.Act_Event is issued
      Seq1 is executed concurrently with partial execution of Seq2
      Server.Act_Event cancellation is attempted
      execution of Seq2 completes
      Seq3 is executed
      Seq4 is executed
   end if;
```

Example 4.6: Illustration of ATC using pseudo code

Note that there is a race condition between Seq1 finishing and the rendezvous finishing. The situation could occur when Seq1 does finish but is nevertheless aborted." (Burns, 1998, [p. 232])

If Seq1 contains an abort-deferred operation, *Op*, then its cancellation will not occur until *Op* is completed.

The four cases of Example 4.6 are, respectively, depicted in Figures 4.3–4.6.

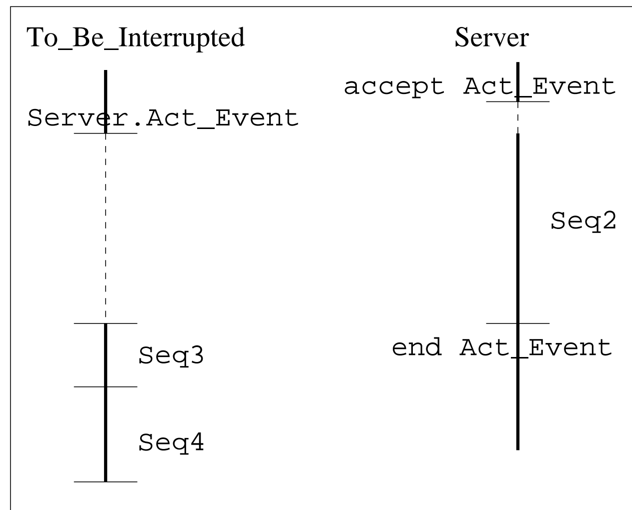

Figure 4.3: Case 1: rendezvous immediately available

In the second case (Figure 4.4), the rendezvous starts too late; there is no interested client anymore.
Note that in the third case, Seq1 is only partially executed—the dashed horizontal line in To_Be_Interrupted indicates the premature end of Seq1.
Note that in the fourth case—although Seq1 finishes prior to Seq2—, Seq2 cannot be immediately aborted since a rendezvous is an abort-deferred operation. Therefore, the trigger is not aborted (a (vain) attempt is made, however) by the abortable part but completes normally. Hence, Seq3 is executed.

Let us now consider a few examples of ATC:

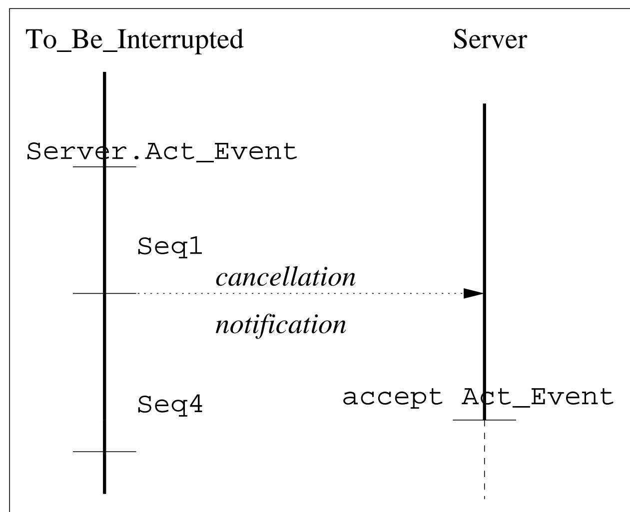

Figure 4.4: Case 2: no rendezvous starts before Seq1 finishes

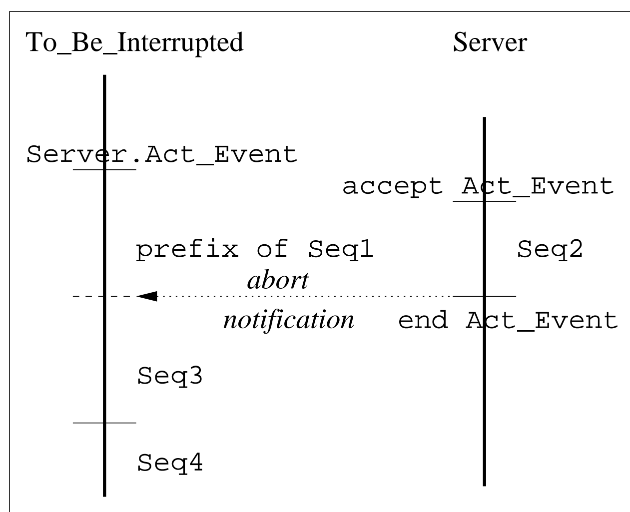

Figure 4.5: Case 3: the rendezvous finishes before Seq1 finishes

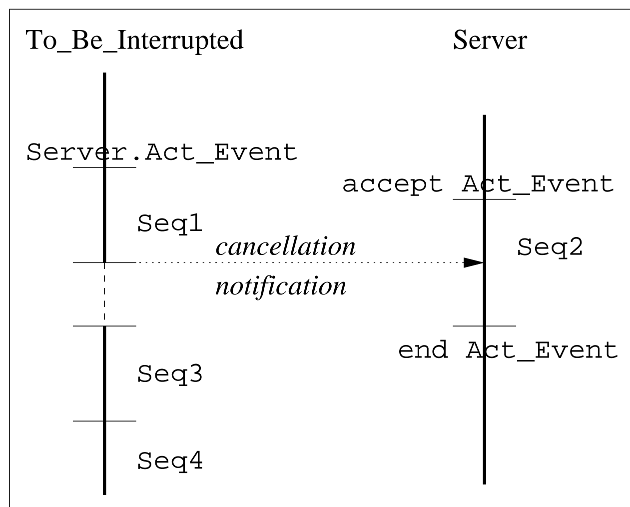

Figure 4.6: Case 4: the rendezvous finishes after Seq1 finishes

```
select
   delay 5.0; -- triggering alternative
   Put_Line("Calculation did not complete"); -- alternative
                                             -- sequence of statements
then abort
   Invert_Giant_Matrix(M); -- abortable part
end select;
```

Example 4.7: ATC using a delay as the trigger (Barnes, 1996, [p. 429])

Example 4.7 illustrates the use of a delay as the triggering alternative. Thus, if we cannot invert the giant matrix in five seconds, we give up and print a message. Incidentally, this example also illustrates how time supervision can be programmed in Ada. Time supervision will be one of the subjects considered in more detail in Chapter 6.

The following example (ARM, 1995, [9.7.4]) illustrates the main loop of some command interpreter. Upon receipt of a terminal interrupt, normal processing of commands is abandoned:

```
loop
   select
      Terminal.Wait_For_Interrupt;
      Put_Line("Interrupted");
   then abort
      -- This will be abandoned upon terminal interrupt
      Put_Line("-> ");
      Get_Line(Command, Last);
      Process_Command(Command(1 .. Last));
   end select;
end loop;
```

Example 4.8: Using an entry call as the triggering event

We proceed by giving a larger example which is based on (Burns, 1998, [p. 236]). We return to our DBMS server which, in case of an error, should undertake some appropriate recovery:

```
type Error_Type is (Transaction_Failure, Client_Not_Responding);

-- assume that we have a central error reporting task that sends
-- error information to all listeners (broadcast)

task body DBMS_Server is
   Error : Error_Type;
begin
   loop
      ...
      select
         Error_Server.Receive(Error); -- listen
         case Error is
            when Transaction_Failure =>
               Roll_Back_Transaction;
            when Client_Not_Responding =>
               Notify_The_Administrator;
         end case;
      then abort
         loop
            -- normal DBMS operation
```

```
            end loop;
         end select;
         ...
      end loop;
   end DBMS_Server;
```

Example 4.9: Error recovery

Still continuing with our DBMS server, the reader is referred to (Rat, 1995, [9]) for an example that shows how database transactions could be implemented by using ATC.

We conclude this subsection by returning to the issue of threads of control in conjunction with ATC (as promised in Chapter 3).

Using ATC, it is possible for the abortable part and the triggering event to be executed in parallel. See Example 4.6 in which, in the third and fourth branch of the pseudo if statement, `Seq1` and `Seq2` were executed in parallel.

Generally, if the triggering entry call cannot be executed immediately (because the barrier of the entry is closed), then the abortable part is executed. If, then—whilst the abortable part is being executed—, the triggering entry call can be accepted (that is, the entry barrier becomes open), it is executed concurrently with the abortable part. Note that it is the *completion* of the trigger that causes the abortable part to be aborted. If the trigger is an entry call to another task, then a rendezvous is started and the callee executes the corresponding accept statement. Hence, the task that issues the ATC statement executes the abortable part and the task that is named in the triggering statement executes the trigger. If, however, the triggering entry call is to a protected object, which is a passive entity (without a thread of control) and which can therefore not execute the entry body, a question arises as to which thread of control should be used to execute the entry body. Consider

```
   -- task T
   ...
   select
      P.E; -- let P be a protected object and E an entry of P
   then abort
      -- sequence of statements to be aborted upon
      -- completion of P.E
   end select;
   ...
```

`T`'s thread of control cannot be used to execute `P.E` since `T` executes the abortable part. Note, however, that `T`'s thread of control was used to issue `P.E`. On the other hand, as we have explained earlier in this paper, a task has exactly one thread of control associated with it. So whose thread of control is used? The answer is given by (ARM, 1995, [9.5.3(22)]): “An implementation may perform the sequence of steps of a protected action using any thread of control; it need not be that of the task that started the protected action.” Thus, for example, the thread of control of the main program waiting for dependant tasks to terminate or that of a client task engaged in a rendezvous could be used instead. But what happens if all other threads of control are “in use”? Is an implementation allowed to introduce an anonymous thread of control that executes `P.E`? This question was posed to the authors of the Ada Reference Manual, and Robert A. Duff[2] of Oak Tree Software, member of the Ada 9X Design Team at Intermetrics, Inc., was so kind in answering: “Yes, the implementation can create extra anonymous threads of control, if it likes. For example, the abortable part could be put inside a separate task.”

[2] `bobduff@inmet.com`

Although creating a new thread of control solves the problem, it is somewhat clumsy and heavy-weight. We will, however, not go further into the discussion but refer the inclined reader to (Rat, 1995, [9]) for an alternative solution.

The essence of the above discussion is that, although the abortable part and the triggering statement of an asynchronous select statement have the potential to be executed in parallel, the task issuing the ATC statement does not have two threads of control just for the purpose of achieving parallelism (note that nested ATC is possible; hence, this task would then be required to have associated with it any (positive) number of threads of control!).

4.2 Direct Interaction and Communication of Actors in CHILL

4.2.1 Asynchronous Communication in CHILL

As we have seen in Chapter 3, a call to one of a task's, `T`, simple guarded procedures, `P`, results in `P` being asynchronously executed by `T`. The caller, therefore, can immediately proceed with the next action after issuing the call. Thus, communication can be initiated by the caller by calling one of the simple guarded procedures of a task. We now look at the syntax and semantics of such a call action (see (Z200, 1996, [6.7]): (we use horizontal rules to mark the beginning and the end of the quotations; a left justified rule indicates the beginning whereas a right justified rule marks the end)

Syntax:

<call action> ::=
  <procedure call>
  | <built-in routine call>
  | <moreta component procedure call>

<moreta component procedure call> ::=
  <moreta location>.<moreta component procedure call> [<priority>]
  | <bound reference moreta location primitive value> ->.
  <moreta component procedure call> [<priority>]
  | <moreta component procedure call> [<priority>]

Semantics: A call action causes the call of either a procedure, a built-in routine, or a moreta component procedure. A moreta component procedure call `L.name(...)` causes the call of that moreta component procedure which is identified by `name` in the mode of `L`. `L` is passed as an initial location parameter to the procedure. A moreta component procedure call has always the structure "location.procedure call". This is characterised by the expression "the procedure call is applied to the location". For a moreta component procedure call, the following steps are performed:

The called procedure is applied to a task mode location `TL`: The caller performs the following steps:

1. evaluation of the actual parameters
2. send procedure identification, actual parameters, and priority to `TL`
3. continue with the next action

`TL` performs the following steps:

1. receive procedure identification and actual parameters according to priority
2. check of the precondition

3. check of the complete invariant
4. execution of the body of the procedure
5. check of the complete invariant
6. check of the complete postcondition

Static conditions: A priority can only be used in a call of a procedure applied to a task location. The procedure primitive value must not deliver a procedure defined within a process definition whose activation is not the same as the activation of the process executing the procedure call (other than the imaginary outermost process) and the lifetime of the denoted procedure must not have ended. If a call is applied to a task location, `TL`, then `TL` must not be ended.

The astute reader may recall that in Chapter 1, we mentioned that priorities cannot be associated with actors in CHILL. This seems to contradict the first static condition. However, this is not the case. Note that a priority can only be associated with "a call of a procedure applied to a task location." This does not imply that the task uses this priority to compete for the processor when it eventually executes the procedure. In fact, the priority is only used when the task decides which request is to be handled (there could be, of course, several requests waiting for the task). Interestingly, CHILL does not specify the mechanism of the selection of a request based on its priority. We **assume** that a request with a high priority is preferred to a request with a low priority. Furthermore, after the request has been handled by the callee, the priority is lost (since it is only associated with the call); it is not saved in the task and used for further actions by the task (remember that the task enters the waiting state again). Hence, it is inappropriate to regard priorities as associated with the task.

For the second static condition, think of the "procedure primitive value" as an access value that can point to a subprogram. This condition is simply a formal paraphrase for the fact that procedures defined inside a process definition can neither be called directly from outside this process using their simple names (since they are not exported), nor indirectly by first pointing to them (from outside) and then dereferencing the pointer (from outside). Hence, processes can be completely ignored when discussing this form of communication.[3]

The last static condition stands to reason.

From the semantics of the procedure call to a task mode location, it can be seen, again, that such a call is of asynchronous nature. After the caller has executed step 2, sending the procedure identification, actual parameters, and priority to `TL`, it can carry on with the next action (step 3). The caller sends, the callee receives (step 1), checks certain things (step 2 and 3), executes the request (step 4), checks again (steps 5 and 6) and is done. *Note that the callee does not send result parameters, if any, back to the caller.* This is not an ommission of the CHILL definition but a general problem inherent to asynchronous communication. The fact that the caller carries on with its next statement immediately after issuing the request makes it hard, if not impossible, for the callee to transmit result parameters. The lack of synchronization might cause a scenario in which the callee can send the results back to the caller, but in which the caller is not in a position to receive them (either because it is doing something else or because it is no longer existing). Actors communicating asynchronously are required to synchronize otherwise—by means of a third party or communication object (see next chapter). Communication always requires a minimal degree of synchronization. There is a forthcoming paper by the authors dealing with that issue.

[3] they could, of course, act as callers

4.2.2 Sending Signals between Actors in CHILL

It is debatable whether signal sending should be considered as a form of direct actor interaction and communication. The general notion of direct communication is that a client issues a request on a server, and the server, then, executes the request on behalf of the client. With signals, the "server" (the receiver) can ignore an incoming signal completely by not at all executing a receive action. Thus, a signal sent to a receiver does not necessarily have an impact on that receiver. However, since when sending a signal, it is possible for the sender to directly specify the receiver, we would like to regard signal sending as a special form of direct actor interaction and communication.

Signals can be used to transmit values between process instances. It is possible for the sender to specify the recipient and the values that are to be transmitted. For the definition of signals, a signal definition statement is used (Z200, 1996, [11.5]):

Syntax:

<signal definition statement>::=
SIGNAL <signal definition> {, <signal definition> }

<signal definition>::=
<defining occurrence> [= (<mode> {, <mode> })] [TO <process name>]

Semantics: A signal definition defines a composing and decomposing function for values to be transmitted between processes. If a signal is sent, the specified list of values is transmitted. If no process is waiting for the signal in a receive case action, the values are kept until a process receives the values.

Static properties: A defining occurrence in a signal definition defines a signal name. A signal name has the following properties: It has an optional list of modes attached, that are the modes mentioned in the signal definition. It has an optional process name attached, that is the process name specified after TO.

The following example

```
SIGNAL Notify, Carry = (INT);
```

defines two signals, `Notify` and `Carry`. Both can be used to send a signal to another process. Since no process name is given, any process can be the recipient. Furthermore, `Notify` is just a bare signal whereas `Carry` can be used to transmit an integer (`INT`) value to the recipient.

To transmit a signal requires the sender to execute a send signal action, the definition of which reads as follows (Z200, 1996, [6.18.2]):

Syntax:

<send signal action> ::=
SEND <signal name> [(<value> {, <value> })]
[TO <instance primitive value>] [<priority>]

Semantics: A send signal action evaluates, in any order, the list of values, if present, and the instance primitive value, if present. The signal specified by signal name is composed for transmission from the specified values and a priority. The priority is the one specified, if any, otherwise, 0 (lowest). If the signal name has a process name attached, only processes with that name may receive the signal; if an instance primitive value is specified, only that process may receive the signal. Otherwise, any process may receive the signal. If the signal has a non-empty set of delayed processes attached, in which one or more may receive the signal, one of these will be re-activated. If there are several such processes, one will be selected in an implementation defined way. If there are no such processes, the signal becomes pending.

If a process becomes re-activated, it is removed from all sets of delayed processes of which it was a member.
Static conditions: The number of value occurrences must be equal to the number of modes of the signal name. The `EMPTY` exception occurs if the instance primitive value delivers `NULL`. The lifetime of the process indicated by the value delivered by the instance primitive value must not have ended at the point of the execution of the send signal action. The `SENDFAIL` exception occurs if the signal name has a process name attached which is not the name of the process indicated by the value delivered by the instance primitive value.

To receive a signal, a process must issue a receive signal case action which enables the recipient to react accordingly upon reception of the signal. The syntax is (Z200, 1996, [6.19.2]):

Syntax:

<receive signal case action> ::=
RECEIVE CASE [SET <instance location> ;]
<signal receive alternative> { <signal receive alternative> }
[ELSE <action statement list>] ESAC
| RECEIVE [SET <instance location>]
(<signal name> [IN <location list>])

<location list> ::=
<location> {, <location> }
<signal receive alternative> ::=
(<signal name> [IN <defining occurrence list>]) : <action statement list>

Semantics: A receive signal case action evaluates the instance location, if present. Then the executing process: (immediately) receives a signal or, if `ELSE` is specified, enters the corresponding action statement list, otherwise becomes delayed. The executing process immediately receives a signal if one of a signal name specified in a signal receive alternative is pending and may be received by the process. If more than one signal may be received, one with the highest priority will be selected in an implementation defined way. If the executing process becomes delayed, it becomes a member of the set of delayed processes attached to each of the specified signals. If the delayed process becomes re-activated by another process executing a send signal action, it receives a signal. If the executing process receives a signal, the corresponding action statement list is entered. Prior to entering, if an instance location is specified, the instance value identifying the process that has sent the received signal is stored in it. If the signal name of the received signal has a list of modes attached, a list of value receive names is specified; the signal carries a list of values, and the value receive names denote their corresponding value in the entered action statement list.
Dynamic properties: A process executing a receive signal case action becomes *timeoutable* when it reaches the point of execution where it may become delayed. It ceases to be timeoutable when it leaves that point. All signal name occurrences must be different. The optional `IN` and the defining occurrence list in the signal receive alternative must be specified if and only if the signal name has a non-empty set of modes. The number of names in the defining occurrence list must be equal to the number of modes of the signal name.

This is what the CHILL definition says about sending and receiving signals. In order to enhance understanding, we will now give an annotated example (Z200, 1996, [D 15]). In the remainder of this subsection, we will discuss some questions that arise in connection with signals in CHILL.

```
01 definitions:
02 MODULE
03    SIGNAL
04       acquire,
05       release = (INSTANCE),
06       congested,
07       ready,
08       advance,
09       readout = (INT);
10    GRANT ALL;
11 END definitions;
12 counter_manager:
13 MODULE
14 /* To illustrate the use of signals and the receive case (buffers
15  might have been used instead), we will look at an example where an
16  allocator manages a set of resources; in this case, a set of
17  counters. The module is part of a larger system where there are
18  users that can request the services of the counter_manager. The
19  module is made to consist of two process definitions, one for the
20  allocation and one for the counters. Initiate and terminate
21  are internal signals sent from the allocator
22  to the counters. All the other signals are external, being sent
23  from or to the users. */
24
25 SEIZE /* external signals */
26    acquire, release, congested, ready, advance, readout;
27 SIGNAL initiate = (INSTANCE),
28   terminate;
29 allocator:
30 PROCESS ();
31    NEWMODE no_of_counters = INT (1:100);
32    DCL counters ARRAY(no_of_counters)
33      STRUCT (counter INSTANCE, status SET (busy,idle));
34    DO FOR each IN counters;
35       each := (: START counter(), idle :);
36    OD;
37    DO FOR EVER;
38       BEGIN
39          DCL user INSTANCE;
40          await_signals:
41          RECEIVE CASE SET user;
42          (acquire):
43             DO FOR each IN counters;
44                DO WITH each;
45                   IF status = idle
46                      THEN
47                         status := busy;
48                         SEND initiate (user) TO counter;
49                         EXIT await_signals;
50                   FI;
51                OD;
52             OD;
53             SEND congested TO user;
54          (release IN this_counter):
55             SEND terminate TO this_counter;
56             find_counter:
```

```
57              DO FOR each IN counters;
58                 DO WITH each;
59                    IF this_counter = counter
60                       THEN
61                          status := idle;
62                          EXIT find_counter;
63                    FI;
64                 OD;
65              OD find_counter;
66           ESAC await_signals;
67        END;
68        OD;
69        END allocator;
70        counter:
71        PROCESS ();
72           DO FOR EVER;
73           BEGIN
74              DCL user INSTANCE,
75                count INT:= 0;
76              RECEIVE CASE
77              (initiate IN received_user):
78                 SEND ready TO received_user;
79                 user := received_user;
80              ESAC;
81              work_loop:
82              DO FOR EVER;
83                 RECEIVE CASE
84                 (advance): count + := 1;
85                 (terminate):
86                    SEND readout(count) TO user;
87                    EXIT work_loop;
88                 ESAC;
89              OD work_loop;
90           END;
91           OD;
92        END counter;
93        START allocator();
94 END counter_manager;
```

Example 4.10: Allocating and deallocating a set of resources in CHILL

Let us consider the two processes `allocator` and `counter`: In line 27 and 28, two signals, `initiate` and `terminate`, are defined; the latter being just a bare communication facility, the former additionally carrying a location of mode `INSTANCE`. That is, `initiate` can be passed a process instance which, by the receiver, can be evaluated as we shall see in a moment. In line 39, such a location is declared, `user`. `allocator` then, in line 41, executes a receive signal case statement by first saving the sender in that instance location. The general contract is that an external user (not shown) sends the signal `acquire` to the `allocator` which keeps this external user in the instance location `user` (logging). If, then, `acquire` is received, line 42, the `initiate` signal is sent to the `counter`, line 48. `user` is passed as the actual process instance to this signal. `counter`, in line 76, executes a receive signal case action; the alternative labelled `initiate` is taken. The `user`, transmitted via the signal is saved in `received_user` before the corresponding action statement list is entered. In this action statement list, the signal `ready` is sent to the external user (to `received_user`, which, in turn, is `user`, i.e., the process instance that was transmitted from `allocator` to

counter via the initiate signal in the first place). We can imagine that this external user, then, executes a receive signal case action to make use of ready.

4.2.3 Noteworthy Points of the Signal Handling in CHILL

We would like to stress a few points about signal handling in CHILL that are not clearly expressed in the specification. Firstly, CHILL speaks of senders and receivers of signals only in terms of a process instance. Does that mean that only process instances can send and receive signals? In particular, are tasks not allowed to do so? Both process bodies and procedure bodies are allowed to contain a definition statement list and an action statement list, the latter of which is allowed to contain any CHILL action, including sending and receiving signals. Furthermore, not allowing a task to send or receive a signal is too severe a restriction since it would not, for example, enable a task to inform a client via signalling that the results of its request are valid. This, however, means that signalling cannot be used to overcome the obstacles of asynchronous communication. We thus, again, **assume** that speaking of a process rather than a thread when referring to signalling is just a lapse.

Secondly, there is the question as to what should happen if a sender issues several send actions. Clearly, the manual says that if there are no threads waiting for the signal in a receive case action, the transmitted values are kept and the signal becomes pending. Now if several signals are sent and if there is no receiver, then this would mean that several signals (and their attached values, if any) must be kept. How many signals can be kept, i.e., is there a signal overflow?

The third oddity is the nature of signals in general. Not that we would like to claim that they are useless, but CHILL would not suffer losses in its expressive power if signals were not included; buffers could be used instead. Although we have not defined them yet (Chapter 5 deals with that), buffers are an adequate substitute for signals. When defining a buffer, its length and contents are specified and then a process can send a data item to the buffer and another process can receive this data item from the buffer. Depending on the actual nature of this received data item, actions statement lists can be entered accordingly. Furthermore, buffer overflow is handled by the language which requires a sender to become delayed if the buffer is full. Likewise, a receiver becomes delayed if the buffer is empty. Unfortunately, we can only **assume** that tasks can make use of buffers, too.

4.3 Direct Interaction and Communication of Actors in Java

4.3.1 Invoking a Method of a Java Thread

Communication requires at least two actors—a caller and a callee, the former of which asking the latter to execute the called procedure. So far, Ada and CHILL stuck to this paradigm; an entry call (issued by the caller) is executed by the called task and a call to a task's guarded procedure in CHILL is also executed by the callee (being asynchronously does not matter here). If we, then, try to adopt this model to Java threads, we encounter a fundamental problem: For a given thread, T, the thread of control, at runtime, is associated solely with this thread's run() method, i.e., T.run(). Thus, although T might offer public methods in its interface, it will never execute any of these on behalf of another thread. For when another thread, S, calls one of these public methods, P(), S executes the code of T.P(). T's thread of control remains associated with T.run(); indeed, T is not even aware of the fact that P() is being called and executed (by S).[4] There is no synchronization or even a rendezvous—a call to T.P() is as much as if T were an object without its own thread of control. There is, in fact, no communication partner for S. See Figure 3.1 (in Chapter 3) for an illustration.

[4] it would be too disruptive to allow such notification

In the light of this experience, it is difficult to describe what direct actor interaction and communication should look like in Java. The authors are convinced that there is no such interaction and communication.

Even if we consider calling `T.run()` to force `T` to do useful work, we cannot get rid of this problem. If `S` calls `T.run()`, then again `S` executes `T.run()`! Even if we could force `T` to allot its thread of control to an incoming request (via calling `T.run()`), only little is achieved. Remember that there is exactly one `run()` method (originally defined in the class `Thread`) that is associated with `T`'s thread of control. As a consequence, `T` could only be offering just one true service! Furthermore, no parameters could be passed upon the call and, after all, what should be the semantics of the scenario in which `S` calls `T.run()` (thereby forcing `T` to execute the incoming request), but `T` is already executing its `run()` (as a result of starting `T`)? Should the so-far executed `run()` be abandoned? Should it be suspended and later, after the "request" has been dealt with, resumed?

We thus see that this discussion is futile and will, therefore, abandon it.

Summary and Comparison

In the light of last section's discussion, it should be clear that Java can be completely excluded from the discussion about direct actor interaction and communication.

We have, in this chapter, laid emphasis on direct interaction between running actors. Although actors are self-contained, interaction and communication is essential if actors wish to transmit data. Accordingly, Ada and CHILL provide high level means to achieve this communication.

The communication in Ada is based on a synchronized client/server model in which the client issues a request to the server and the server executes this request. While such action is in progress, both parties are said to be synchronized, meaning that the caller waits for the callee to complete the request. They are said to be engaged in a rendezvous. Various additional means, for both the client and the server, are provided to control this synchronization: timed and conditional entry calls which allow the client to withdraw its offer of communication and, for the server, the select statement with its rich set of variations (selective accept, delay alternative, else part) can be used to cater for fine control over when a request is to be accepted. We then discussed the requeue facility which allows a request, after it has been accepted, to be ejected if the internal conditions are not appropriate for handling the request. Since Ada does not allow for the formal parameters of an entry to be read prior to executing the entry, the requeue facility comes in handy to overcome this little difficulty. Generally, with the requeue and the possibility to equip branches of a select statement with guards, Ada's model of synchronization is known as avoidance synchronization. Following that, we turned to the subtle topic of ATC which allows a task to announce that it is willing to alter its flow of control upon the change of some condition. Examples illustrating these points were given.

Albeit a trifle controversial, we set accord that signal sending be regarded as a (special) form of direct interaction and communication between actors in CHILL. A sender can send a signal, possibly having data items attached, to a receiver which may receive this signal, make use of the transmitted values, and enter a corresponding action statement list. Additionally, CHILL allows actors to communicate asynchronously via calls to a task's guarded procedures. Note that the CHILL signal facility is of more general nature than the mechanism of using exceptions for "communication" in Ada.

Note that in this chapter, emphasis was laid on *direct* interaction and communication between actors. The calling protocol used in Ada and CHILL, however, is semi-anonymous. The caller always names the callee while the callee is generally unaware of the caller. At times, though, it might be necessary for the callee to know about the identity of the caller.[5]

[5] this would, to a great extend, facilitate the programming of, for example, a scheduler

In CHILL, for the case of signal sending, the SET clause in the RECEIVE CASE action can be used for that particular purpose, as demonstrated in Example 4.10. A corresponding construct for task objects is not available. Ada allows the use of the attribute `Caller` (exclusively) within an accept statement (or entry body, for protected objects) to interrogate the caller's identity:

```
with Ada.Task_Identification;
use Ada.Task_Identification;

-- body of task T
accept E do
   if E'Caller = Client_1'Identity then
      -- T'Identity yields a value of the type Task_ID (as does 'Caller)
      -- that identifies the task denoted by T
      -- Ada provides for task identification by means of the
      -- language-defined library package Ada.Task_Identification,
      -- which is declared in the optional Systems Programming
      -- Annex, Annex C
      ...
   end if;
   ...
end E;
```

The main difference between Ada and CHILL with respect to direct communication is the fact that in Ada, communication is always synchronized whereas in CHILL, communication is always asynchronous or, at least, semi-asynchronous. Neither sending a signal nor calling a task's guarded procedure delays an actor. The receiver always synchronizes with the reception of the signal—it either receives the signal immediately (if the signal is pending) or becomes delayed when it executes a receive case action for which there is no signal pending. The attempt to artificially introduce "full" synchronization by having the sender, immediately after executing the send action, execute a receive action for a signal that the receiver (of the first signal) sends as a form of acknowledgement fails because this explicit synchronization is both cumbersome and error-prone (race conditions!). Owing to the problems of stand-alone asynchronous communication, we proclaimed that communication always requires a minimal degree of synchronization.

Note that CHILL, since its communication is not based on a synchronization model, does not need to include the rich variety of communication controlling facilities found in Ada. A sender cannot, after issuing a request, withdraw its offer of communication; hence, things like conditional or timed send or call actions are not required. Interestingly, for the recipient, a few of Ada's features can be simulated—namely, a select statement with a delay alternative, a select statement with an else part, and an ATC statement using a delay statement as the triggering alternative. We conclude this chapter by considering how this can be done. The select statement with an else part is easily accomplished: the `ELSE` clause in a receive case statement realizes just that. For the other two things, we need to make use of CHILL's time supervision facilities. Without going too much into the details (Chapter 6 will address this), we note that it is possible to impose time supervisions on a running process. When such a time supervision expires, a transfer of control takes place. The following example realizes an Ada select statement with a delay alternative:

```
AFTER SECS(10) DELAY IN
   RECEIVE CASE
      ...
   ESAC
TIMEOUT
   /* sequence of statements */
```

```
END
```

Example 4.11: CHILL code simulating an Ada select statement with a delay alternative

If, after ten seconds, the receive case action has not been finished, the executing process receives a time interrupt and a transfer of control to the sequence of statements in the `TIMEOUT` part takes place. Note, however, that there is no cancellation of the time supervision (as it is with a select statement with a delay alternative in Ada) when the receive case action is eventually executed. Hence, the time interrupt can still intervene while the receive case action is executed.

The same approach can be used to simulate ATC with a delay trigger.

Communication always requires some kind synchronization and Ada provides for more fine control over synchronized communication than CHILL.