Chapter

8 Communication between Activities

Specifying the communication between activities consists of the what and the when, just as in other parts of the specification. The what is described by the flow-lines in the activity-charts (see Chap. 2) and relevant parts of the Data Dictionary (see Chap. 3). The "when" is to be specified by the behavioral parts of the model, that is, the statecharts and mini-specs. This dynamic aspect of communication is the subject of the present chapter. We discuss the parts of our languages that control the communication between activities and discuss how they are related to the flow-lines in the functional view. We also describe the queue mechanism in some detail.

8.1 Communication and Synchronization Issues

Functional components in systems communicate among themselves to pass along information and help synchronize their processing. A number of attributes characterize the various communication mechanisms, and different mechanisms are convenient for different application domains. The communication can be *instantaneous,* meaning that it is lost if not consumed immediately, or *persistent,* meaning that it stays around until it gets consumed (which can be achieved by queuing, for example). The communication can be *synchronous* (e.g., the sender waits for an acknowledgment or reply from the listener) or *asynchronous* (i.e., there is no waiting on the part of the sender). The communication can be *directly addressed* (i.e., the target is specified) or sent by *broadcasting.* And there are other issues. A flexible modeling and implementation environment will make it possible to use many variants of these communication patterns.

In our models, every element has a *scope* in which it is recognized. The scoping depends on the element's definition chart, and is described in Chap. 13. The central point here is that every change in the value of an element is broadcast to all activities and statecharts in the element's scope, and is thus "seen" by them all. These changes include the occurrence of an event, the assignment of a value to a data-item or condition, a change in the status of an activity, and entering or exiting a state.

In addition to events, which are instantaneous and last for one step only, all other elements keep their values until some explicit action causes a change. Therefore, for all communicated elements other than events, the receiver need not necessarily be active when the sender assigns them a value. Moreover, in all cases other than queues, the same information can be consumed an unlimited number of times.

The following sections discuss the elements related to the flow of information in our languages and illustrate how they can be used to model various communication patterns.

8.2 Controlling the Flow of Information

The statecharts and mini-specs are responsible, among other things, for controlling the flow of information between activities, and they are complemented by certain elements in the textual language. The description of the information flow given in the activity-charts completes the picture, and it should be consistent with the control specification.

8.2.1 Elements related to flow of information

Consider Fig. 8.1, in which `X` is specified to flow between activities `A` and `B`. When does `X` flow, and what triggers the flow?

Assume first that `X` is an event, and that the behavior of `A` is described by the statechart that constitutes `A`'s control activity. This statechart may contain an action that generates `X` along a transition or in a static reaction, and at that instant the controlling statechart of `B` (or of any of `B`'s descendants) can sense `X` and modify its behavior accordingly. See Fig. 8.2. Similarly, `A` and `B` can be described by mini-specs, which, respectively, contain an action that generates `X` and some reaction triggered by it. Many other alternatives are also possible.

If `X` is a condition or a data-item, it is considered to be continuous in time. This means that the value of `X` may change at any point in time

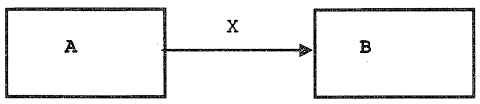


Figure 8.1 An information element flowing between activities.

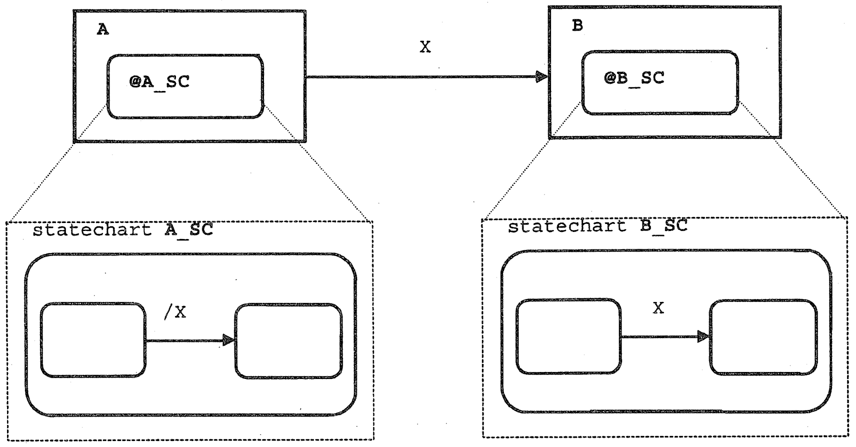


Figure 8.2 Producing and consuming an event.

as long as A is active, and B can sense and use this value at all times (even when A is no longer active). The actions and events that were described in the preceding chapters enable us to affect the values of the conditions and data-items and to sense when changes in such values occur. More specifically, if X is a condition, the source activity A (i.e., its controlling statechart or mini-spec and those of its descendants) can change X's value by the actions `tr!(X)` and `fs!(X)`. The change itself (via the events `ch(X)`, `tr(X)`, or `fs(X)`), and the current truth value of X, can be sensed anywhere in B. If X is a data-item or condition, it can be assigned values in A by actions such as `X:=E` for an expression E. In B, we can sense the event `written(X)` (abbreviated `wr(X)`), which may be viewed as occurring at the instant the assignment takes place. The value of X can also be used in any controlling statechart, mini-spec, or combinational assignment inside B.

If we are not interested in assigning a specific value to X, just in stating that some value has been assigned, A may execute the action `write_data(X)` (abbreviated `wr!(X)`), and B may sense the event `wr(X)`. Thus, informally, the action `wr!(X)` means assign a value to X but without specifying any specific value, and the event `wr(X)` means that X has been assigned a value. In a dual fashion, the target activity B of the data-item or condition X may perform the action `read_data(X)` (abbreviated `rd!(X)`), signifying that it has read the value of X, without using it in any particular computation. At the same time, the source activity A can sense the corresponding event `read(X)` (abbreviated `rd(X)`).

Note the following rules, which hold when the actions `wr!` and `rd!` are applied to structures such as records and arrays and their components. The general idea is that when dealing with structures all of whose components exist in every occurrence of the structure, the special actions and events that involve the structure as a whole apply to all components, but the converse is not true. If R is a record, then the action wr!(R) and an assignment to R trigger the event wr(R.X), for each component R.X of R, and the action rd!(R) triggers the event rd(R.X). If A is an array, the action wr!(A) triggers the event wr(A(I)) for each component of A, and the action rd!(A) triggers the event rd(A(I)). An assignment to the entire array (e.g., A:=B), or to an array slice (e.g., A(1..3):=T), triggers the event wr(A(I)) for each index I in the assigned range but not vice versa; that is, an assignment to A(I) does not cause the event wr(A).

For unions, in which the components have an exclusive nature, actions on a component imply events related to the containing union data-item but not vice versa. Thus if U.F is a component of the union data-item U, then the action wr!(U.F) triggers the event wr(U), as does an assignment to U.F. The action rd!(U.F) triggers the event rd(U).

The written and read events are relevant to the *queue* data-item, too. This is discussed in Sec. 8.4.

8.2.2 Interface between "execution" components

The actions and events described earlier provide a way to monitor the behavior of the flow of information. An important issue related to the information elements that appear in controlling statecharts, mini-specs, and combinational assignments pertains to their origins and destinations. In particular, the statecharts themselves do not explicitly deal with the flow of information. The inputs and outputs of a statechart are presented in the activity-chart as flowing to/from the control activity associated with the statechart in question.

For example, refer to Fig. 4.3, the simplest version of the statechart describing the EWS_CONTROL. The operator commands EXECUTE, SET_UP, and RESET are input events to this statechart and are shown as flowing from an external activity into the control activity (as components of COMMANDS) in Fig. 2.5. Similarly, the event OUT_OF_RANGE, which is also used in this statechart, is an input that comes from the COMPARE activity.

Not all the elements used in the statecharts come from external sources. We have seen that orthogonal components may communicate via internal information elements. The events OPERATE and HALT, shown in Fig. 4.14, are generated by an orthogonal component and, as such, they do not appear in the external interface of the control activity in Fig. 2.5 at all.

In general, each element that appears in a behavioral description unit (i.e., a statechart, mini-spec, or combinational assignment) may be either used by or affected by this description unit. Some elements, such as the events HALT and OPERATE and the event TICK in the mini-spec of Sec. 7.4.1, are both used and affected by the same statechart or mini-spec and are thus considered internal to it.

If X appears in a trigger (along a statechart transition or in a reaction in a state or mini-spec), then we say that it is *used by* the statechart or activity. The same applies if X appears in a conditional expression in the *if* or *when* parts of an action. Data-items are also said to be used by a statechart or an activity if they appear on the right-hand side of assignment actions or combinational assignments.

Consider, for example, Fig. 8.3. The event E and the condition C are used by the statechart because they appear in the transition's trigger. If C is a compound condition (say, it is defined as C1 or I=J), then its components (in this case, C1, I, and J) are also used by the statechart. The data-items X and Z in Fig. 8.3 are also used because the former is tested and the latter participates in an assignment.

Similarly, if X is an event generated by an action (along a transition or in a static reaction or a mini-spec) in the statechart or in an activity, then it is *affected by* this behavioral unit. The same applies if X is a data-item or a condition that is assigned a value in an action (e.g., Y and K in Fig. 8.3), or in a combinational assignment (e.g., IN_RANGE of Sec. 7.4.3).

In a complete specification, we expect all elements that are used by a statechart or an activity (in its mini-spec or combinational assignment), but are not affected by it, to be inputs to the corresponding control activity or the activity itself, respectively. Similarly, elements that are affected by the statechart or the activity, but are not used internally, are expected to be outputs of the control activity or the activity.

We should remark that actions related to activities (e.g., st!(A) and sp!(A)), although they can be viewed as signals that flow out of the control activity, have no corresponding flow-lines in the activity-chart. The same goes for the events st(A) and sp(A), and the conditions ac(A) and hg(A), which can be viewed as signals that flow from A into the control activity.

8.3 Examples of Communication Control

We have seen several patterns by which activities communicate. For example, the data-item LEGAL_RANGE was assigned a value by the SET_UP activity, and this value was used later by the COMPARE function. In this scheme of shared data, the exact timing of the production and consumption of the values is not significant. On the other hand, we have

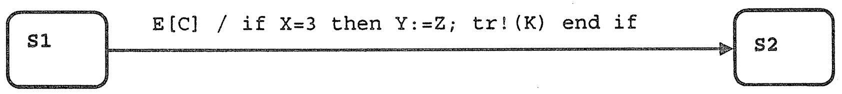


Figure 8.3 Elements used and affected by a statechart.

seen several cases in which events were used to detect an occurrence in which timing was important and an immediate response was required (e.g., the OUT_OF_RANGE notification and the RESET command).

We shall now see examples in which the communication involves synchronization aspects as well as data transfer.

8.3.1 Communication between periodic activities

In distributed computation models, the functionality is often divided among a number of periodic activities. Each of these has some mission to carry out, and upon completion it transfers control to some other activity. One activity might prepare data for processing and then notify the consuming activity when the data is ready. Figure 8.4 shows such a case from the EWS example, where we specify the activities PROCESS_SIGNAL and COMPARE. The checking that takes place in the latter is synchronized to the periodic rate at which signals are produced in the former. CHECK is a procedure-like activity that computes the IN_RANGE value for the current SAMPLE and then terminates.

In this example, like other similar ones, some assumptions are made about the processing time of the activities participating in the cycle.

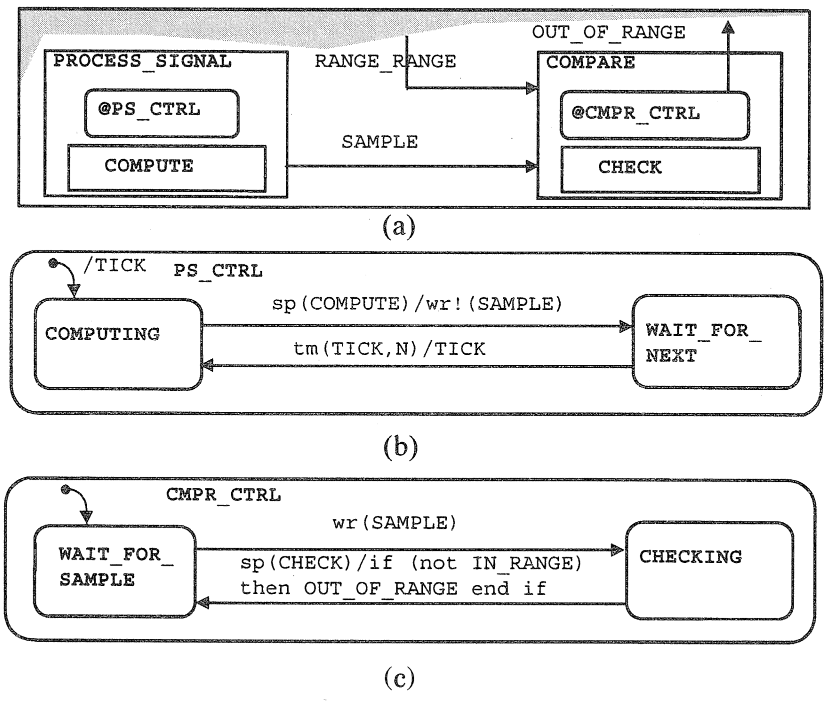


Figure 8.4 Communication between periodic activities. (*a*) The communicating activities. (*b*) The statechart of PROCESS_SIGNAL. (*c*) The statechart of COMPARE.

For instance, it is assumed that in every cycle the CHECK activity succeeds in completing its execution before the next SAMPLE is ready for processing; otherwise, some data may be lost.

In Fig. 8.4 we have shown only the top-level behavior; there is no explicit value assignment to SAMPLE, and no details about how it is used in the CHECK activity. The timing of the data transfer and how it influences the activity scheduling are expressed with the abstract write_data action and the written event. Actually, the read_data action and read event can be used in a dual manner to synchronize an activity execution with the time the data is consumed, so that a cycle of preparing new data can start.

8.3.2 Message passing

It is sometimes convenient to base the communication between activities on message passing. A good way to deal with this involves queues, which are described in the next section. However, in many cases, the mechanisms already discussed are sufficient. Dataless messages can be represented by events, while messages with data can be modeled by record data-items, whose departure from the source (or arrival at the target) can be sensed by the receiver using the written event.

As an example, assume we have a simple client/server setup, where the server waits in an idle state for a message that denotes a request for some service. The server is able to deal with three different kinds of messages, each with special data. This can be achieved using a union data structure whose components are the various message records, as follows.

First, we define a data-type MESSAGE as a record with two fields:

Field Name:	TYPE	Field Type:	Integer min=1 max=3
Field Name:	DATA	Field Type:	MSG_DATA

The first field, TYPE, holds the message type, one of three possible values, while the second holds the accompanying data. The user-defined type MSG_DATA is a union consisting of three fields, each corresponding to one of the message types:

Field Name:	D1	Field Type:	POSITION
Field Name:	D2	Field Type:	BITS
Field Name:	D3	Field Type:	KEY

Each message transfers some data represented by a different user-defined type. The client prepares and sends the message MSG (whose data-type is MESSAGE) by carrying out the following actions:

```
MSG.TYPE:=1; MSG.DATA.D1:=NEW_POSITION; wr!(MSG)
```

In Fig. 8.5 we see how the server may respond to the arrival of the message. Each of the three services activated in response to the respective message (i.e., service request) consumes its appropriate data.

In this example we did not discuss whether the server is guaranteed to be ready to respond when the request is sent, or how the client knows whether the request was fulfilled. Our language does *not* contain any built-in mechanism for identifying message senders so that replies can be automatically addressed. However, when this is required (e.g., for synchronization or confirmation purposes in a multiple-client environment), it can be implemented using explicit identification. Later, when multiple instances of generic charts are discussed, we shall see that an instance number can be used for this purpose.

8.4 Activities Communicating through Queues

Queuing facilities for messages are virtually indispensable in modeling multiprocessing environments, especially multiple client/server systems. We would like to be able to address situations in which an unlimited number of messages is sent to the same address, while the receiver is not always in a position to accept them. We also want to arrange things so that no message is consumed before one that was sent earlier. Moreover, we want it be possible for concurrently active components to write messages to the same address at the same moment and for concurrently active components to read different

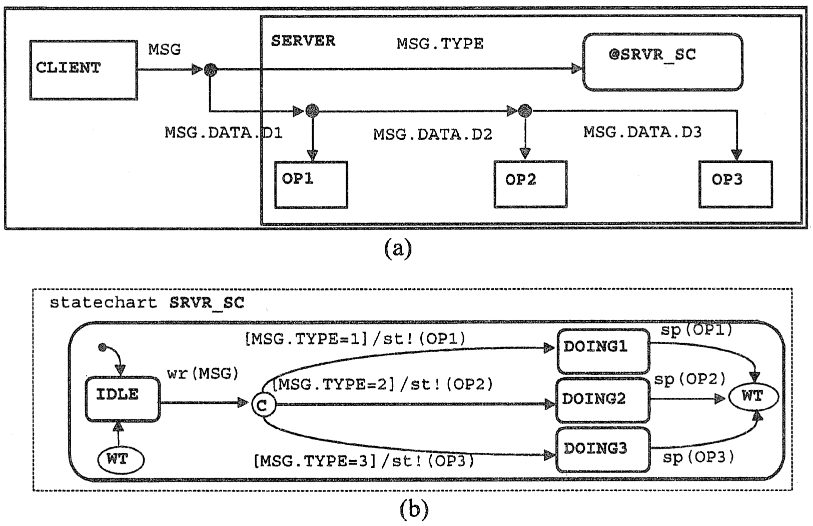


Figure 8.5 Server responding to three service requests.

messages from the same source, even at that very same moment. In our language set, we use *message queues* for this, simply called *queues* for short.

8.4.1 Queues and their operation

A queue is an ordered, unlimited collection of data-items, all of the same data-type. The queue is usually shared among several activities, which can employ special actions to add elements to the queue and read and remove elements from it. Our queues are of unrestricted length, which is in contrast to those used in some real-time kernels, which are defined with a maximal number of components.

A queue is itself a structured data-item, just like an array, and when defined in the Data Dictionary the data-type of its components must be specified. This data-type can be any basic predefined type (i.e., integer, real, etc.), or a user-defined type. There are no limitations on combining queues with other constructs, e.g., arrays, records/unions, or other queues. This means that we can define an array of queues, a record with a queue as a field thereof, or even a queue of queues. The usage of such compound constructs will be presented further shortly. A queue of records or unions, for example, is achieved by an intermediate definition of a user-defined type.

We supply several actions to manipulate a queue. The exact timing of these actions during the execution of a step is a delicate issue, which is discussed in Sec. 8.4.2.

The actions `q_put(Q,D)` (abbreviated `put!(Q,D)`) and `q_urgent_put(Q,D)` (abbreviated `uput!(Q,D)`) add the value of the expression `D` (a data-item or condition) to the queue `Q`. The former action adds an element to the tail of the queue, while the latter adds it to the head of the queue, allowing messages with higher priority to precede all others. Both these operations cause the event `wr(Q)` to occur. The type of the expression `D` must be compatible with the data type of the elements of the queue, as in assignment actions.

The action that is dual to these two is `q_get(Q,D,S)`, abbreviated `get!(Q,D,S)`. It extracts the element residing at the head of the queue `Q` and places it in `D`, removing it from the queue in the process. The data type of `D` must be compatible with the data type of the elements in the `Q`. The third operand, the *status condition* `S`, is optional. It is set to `true` if the queue contained elements when the action was carried out, and to `false` if the operation failed to find data to extract.

The action `q_peek(Q,D,S)` (abbreviated `peek!(Q,D,S)`) is similar to `get!`, but it is not destructive; it copies the element at the head of the queue into `D` without removing it from the queue.

The actions `get!` and `peek!` may succeed or fail, the latter being the case if the queue is empty. If successful, `D` and `S` are assigned values, and the events `rd(Q)` and `wr(D)` occur. The event `wr(S)` always occurs, and if the values of D and S are changed from their previous values in the process, then ch(D) and ch(S) occur, too.

In addition to these actions, a queue can be totally cleared by the action q_flush(Q), abbreviated fl!(Q). It is also possible to examine the queue length by the operator q_length(Q), which returns the length of the queue prior to the step. More about this issue in the next section.

8.4.2 The semantics of queues

A queue is inherently sequential because the order in which the messages are put in the queue determines the order in which they are consumed (with the exception of the order-overriding action uput!). A problem arises when operations on the same queue occur in parallel components during the same step. Because there is an element of nondeterminism in the order of the operations, which depends on the tool implementing the execution of the model, the end result might not be fully determined. We now describe a carefully defined semantics, whose goal is to reduce this nondeterminism.

All get actions are performed when they are encountered. Actually, a get action immediately removes the element read from the head of the queue. However, the assignment to D in get!(Q,D,S) is performed only at the end of the step, unless the assigned variable is a context variable (i.e., $D instead of D, see Sec. 5.2.2). Several get actions in the same step read the elements from the queue sequentially, and each reads a different element one after the other in a nondeterministic order. Because get fails when the queue is empty, some of the get actions succeed and some fail. Using a context variable for the status condition (i.e., $S instead of S) makes it possible to check in the current step whether the operation succeeded.

In contrast to get actions, a put does not immediately affect the contents of the queue. All put actions are accumulated and are performed at the end of the step. This scheme reduces the chances of racing (see Sec. 6.3.2) because it prevents the interleaving of get and put actions in the same step. The order in which the put actions of the same step are performed at the end of the step is also nondeterministic, and it depends on the tool implementing the execution.

The clearing action flush also takes effect at the end of the step. When issued in the same step with some put actions on the same queue, flush will be the last to be carried out, and it will result in an empty queue. Of course, this situation is considered a racing condition.

Although the actual number of elements in the queue might change during a step, the returned value of the q_length operator is not updated continuously. Rather, it returns a unique value per step retrieved before all other queue operations of that step. The following example of its use is inappropriate, and when started on a nonempty queue, it will result in an infinite loop:

```
while q_length(Q)>0 loop
    get!(Q,$MSG,$S);
    if $S then
    . . . . .
    end if;
end loop
```

The following loop is more suitable for processing all messages in the queue:

```
for $I in 1 to q_length(Q) loop
    get!(Q,$MSG,$S);
    if $S then
    . . . . .
    else
     break
    end if;
end loop
```

The status condition `$S` is checked during the loop, because there may be several consumers reading from the queue in the same step.

Figure 8.6 illustrates the order in which operations on a queue are performed during a step.

8.4.3 Queues in an activity-chart

Queues can be associated with data-stores just as data-items of other types can be. To associate a queue with a data-store, both must have the same name. Figure 8.7 illustrates the combined use of data-stores and queues, and here, too, if the incident flow-lines are unlabeled, the queue `Q` is considered an output of the source activity, `PRODUCER`, and an input to the target activity, `CONSUMER`.

Note that `P_MSG` is not an output of `PRODUCER` and is therefore not written on the emerging flow-line. It is best to view the `put!(Q,P_MSG)` action as the assignment `queue-head:=P_MSG`. In terms of Sec. 8.2.2,

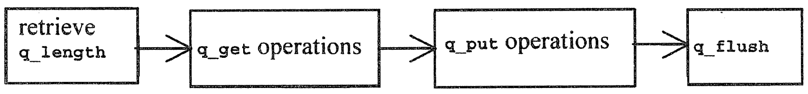


Figure 8.6 Operations on a queue during a step.

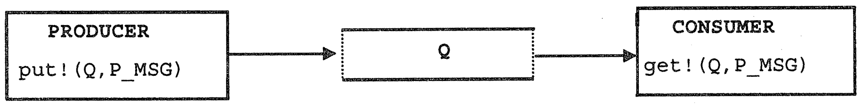


Figure 8.7 A queue associated with a data-store.

P_MSG is actually *used by* the put operation, and should thus flow into the PRODUCER activity, or, alternatively, it should be assigned internally. Moreover, P_MSG is not necessarily a variable data-item; it may be a compound expression or a constant that cannot even move along a flow-line. Dually, C_MSG is viewed as being *affected* by the CONSUMER activity, where actually it can be viewed as being assigned by C_MSG:=queue-head. Thus it is expected to be an output of CONSUMER, or used internally.

Sometimes a queue that transfers messages between activities is marked just as a label on a flow-line between the sender and the receiver. When messages flow among activities in both directions, two oppositely flowing lines can be used.

8.4.4 Example of activities communicating through queues

The special characteristics of queues make them suitable for modeling architectures consisting of several clients and servers. Before sending a new request, a client does not need to check whether its previous requests (and those of other clients) have already been granted and a server is available because all requests are kept in the queue until they are granted. However, the exclusive nature of the get operation guarantees that only one server will handle an individual request, although multiple servers may be available when the request arrives.

Let us now assume we have a multiple-EWS system, consisting of several EWS units of the kind described so far and connected to several printers. Any of the printers may serve any one of the units. See Fig. 8.8, which shows an activity-chart with four EWS units (the clients) connected via a queue PRINTING_Q to two printers (the servers). The queue, in addition to its appearance in the data-store, is defined in the Data Dictionary as a data-item whose type is queue of PRINT_REQST.

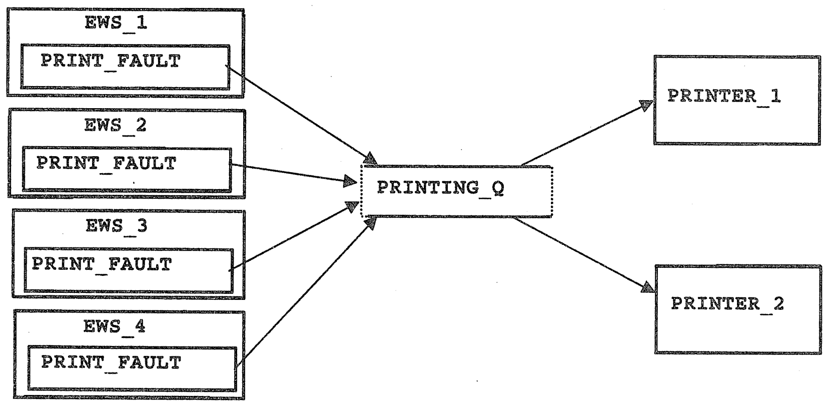
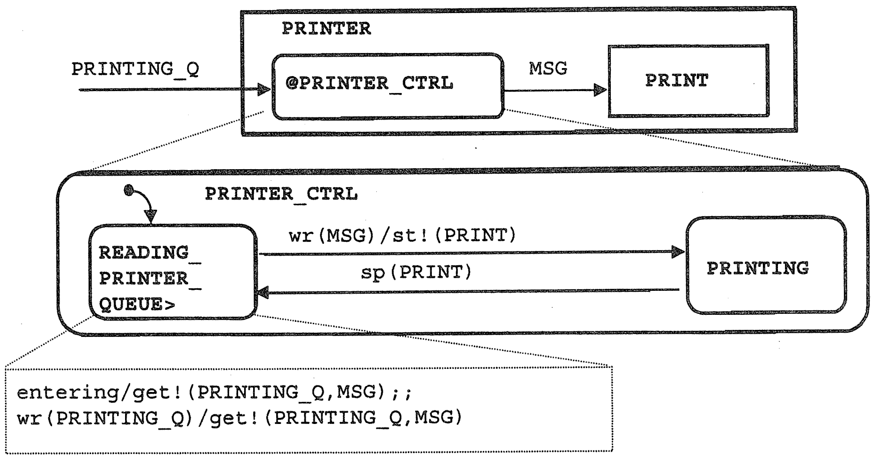


Figure 8.8 Multiple clients served by multiple servers via a queue.

```
CONSTRUCT_FAULT_MESSAGE;
put!(PRINTING_Q,FAULT_MSG)
```

(a)



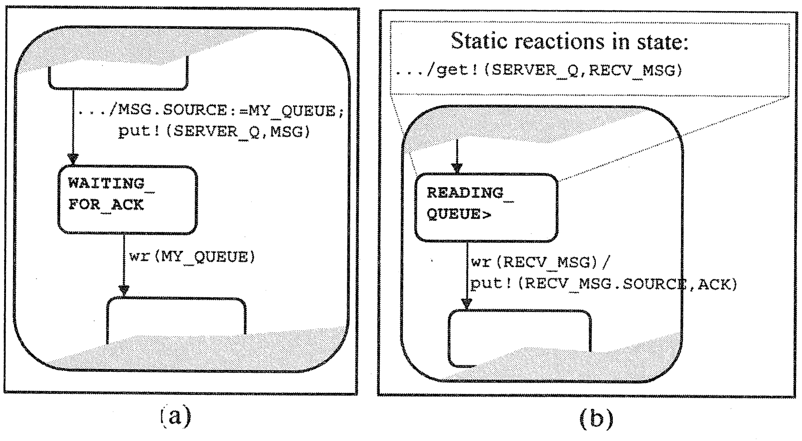
(b)

Figure 8.9 Writing and reading messages from a queue. (*a*) Mini-spec of PRINT_FAULT activity. (*b*) Description of the PRINTER.

Each of the EWS units contains a PRINT_FAULT activity that converts the OUT_OF_RANGE_DATA into a printing request (FAULT_MSG of type PRINT_REQST) and sends it to the queue PRINTING_Q. A printer, when ready, reads the next request from the queue, if there is one, and performs the actual printing. See Fig. 8.9 for the mini-spec of the PRINT_FAULT activity and the internals of each PRINTER.

8.4.5 An address of a queue

The preceding example is of loosely coupled (asynchronous) communication. Because the sender does not wait for a reply, the receiver does not need to know the identity of its clients. When the server does not have any prior knowledge of its clients and tightly coupled (synchronous) communication is required; that is, the sender waits for a response, and the address for reply should be contained in the original request. This can be supported by referring directly to the queue data-item that actually holds the address to the queue. This implies that if Q1 and Q2 are both defined as queues of the same component type, then Q1:=Q2 is a legal action, after which Q1 will point to the same data that Q2 points to. Any put and get operation using either Q1 or Q2 will affect the common queue. We should point out that two queue data-items are considered equal only if they point to the same real queue; e.g., Q1=Q2 is true after the assignment Q1:=Q2. Otherwise, even if all of their contents are the same, the two are not equal.



.../MSG.SOURCE:=MY_QUEUE;
put!(SERVER_Q,MSG)
WAITING_
FOR_ACK
wr(MY_QUEUE)
(a)
Static reactions in state:
.../get!(SERVER_Q,RECV_MSG)
READING_
QUEUE>
wr(RECV_MSG)/
put!(RECV_MSG.SOURCE,ACK)
(b)

Figure 8.10 Using a queue address for synchronous communication. Behavior of (*a*) CLIENT and (*b*) SERVER.

When synchronous communication is required, each client may have its own queue through which it receives replies. When sending a message MSG, the client includes a field, say MSG.SOURCE, to which it assigns its queue address, say, MY_QUEUE, by the action MSG.SOURCE:=MY_QUEUE. Assume that the server reads the message into RECV_MSG, and acknowledges its receipt by sending a reply using the action put!(RECV_MSG.SOURCE,ACK). The client then waits for the event wr(MY_QUEUE) that results from this put, and can then proceed with its work. See Fig. 8.10.