Chapter

7

Connections between the Functional and Behavioral Views

In Chap. 2 we discussed the functional view and the language of Activity-charts, which is used to specify it. As explained there, each activity in the chart may contain a control activity whose role is to supervise the behavior of its sibling activities. The internal descriptions of control activities are given by Statecharts, the language for the behavioral view discussed in Chap. 4.

In this chapter, we provide the link between the two languages, by describing the mechanisms that a statechart may use to control those parts of the activity-chart for which it is responsible. We discuss the actions used by the statechart to control activities and the events and conditions used by it to sense their status.

In addition, we show how the behavior of a basic activity (i.e., one that is not further decomposed into other activities) can be specified by a mini-spec, using the textual language described in Chap. 5.

Behavioral aspects of the communication between the activities are described in Chap. 8.

7.1 Dynamics in the Functional Decomposition

The activities participating in the functional decomposition are not necessarily always active. They may be constantly active when the functional components represent blocks of electronic design, as happens in chip-level modeling. However, in most kinds of systems many of the activities have limited periods in which they are active.

Here are some examples. Procedures and functions in software programs start when they are "called" by another part of the code, and upon completion they stop and return to the calling statement. In multitasking or multiprocessing systems, tasks (or processes) are invoked, do their job, and then are "killed" or "kill" themselves. Tasks with lower priorities are interrupted and delayed when a mission of higher priority arrives, and they are resumed when the more urgent mission completes. Interactive user interface is specified by "callback functions" of limited execution time, performed as a reaction to keyboard and mouse events. In object-based decomposition, objects are dynamically created and deleted, and operations related to an object are activated only when needed.

Let us examine the dynamic and timing issues related to the activities in our EWS example. Most of the details are obtained from the textual description of the example in Chap. 1, and others reflect decisions made later on in the text.

SET_UP	Activated by an explicit request of the operator when the system is waiting for a command. It terminates on its own.
COMPARE	Starts when the operator invokes an EXECUTE command, and stops when the event OUT_OF_RANGE occurs or when the operator stops it with the RESET command.
PROCESS_SIGNAL	Active only when the system is in the usual execution mode and the COMPARE activity is active and is consuming its output for comparison.
DISPLAY_FAULT	Starts when the processed signal has become out of range and is stopped either by the operator or after a predefined time period.
PRINT_FAULT	Activated if the predefined time period has passed and has caused DISPLAY_FAULT to stop. It terminates on its own.

Obviously, merely listing the activities and their connections, as is done in the functional view, is not sufficient. We have to specify the dynamics of controlling these activities, including the starting and stopping of the subactivities of a nonbasic activity. In the following sections we shall see how these aspects are covered in our models using the control activities and their describing statecharts. But nonbasic activities are not all we have. To complete the specification, we have to add something to describe the behavior of the basic activities, those that have no subactivities, not even a control activity. In Sec. 2.4.2 we examined the different types of basic activities—reactive event-driven, procedure-like, and data-driven—and mentioned that their behavioral description is provided in the Data Dictionary. In the last section of this chapter we show in detail how this is done.

Behavior related to the communication between activities, which deals not only with reading the inputs and sending the outputs but with synchronization aspects, is discussed in Chap. 8.

We should emphasize that the order in which the functional and behavioral views and their connections are developed depends on the

nature of the system and on the specification methodology. One can start by carrying out a functional decomposition in activity-charts, and then add the timing and other dynamic information in statecharts to capture behavior. In contrast, it is possible to start by using statecharts to describe the system's modes of operation, a collection of use-cases (scenarios), or both and then construct an activity-chart from the activities performed in these modes or scenarios.

7.2 Dynamics of Activities

To capture the dynamic behavior of nonbasic activities, that is, to manage and control their subactivities, our models employ control activities that are associated with statecharts. In this section and the next, we discuss how this controlling is carried out.

7.2.1 Statecharts in the functional view

In general, when a nonbasic activity that contains a control activity starts its execution, the statechart associated with that control activity becomes active, that is, the system enters the top-level state of this statechart, and it starts reacting to external and internal happenings, as described earlier.

Associating a statechart with the control activity is accomplished by using the @ symbol. Figure 7.1 shows a control activity named `CNTRL_ACT`, which is associated with the statechart `CNTRL_SC`. (The special dashed lines in this and similar figures are not part of our graphical languages; they are used to denote associations between boxes and charts.)

Very often the name of the control activity itself is omitted. (See Fig. 7.2 below), in which case it is referred to by the name of its associated statechart.

An activity with a reactive behavior pattern can be described by a statechart even though it is not further decomposed so that it has no

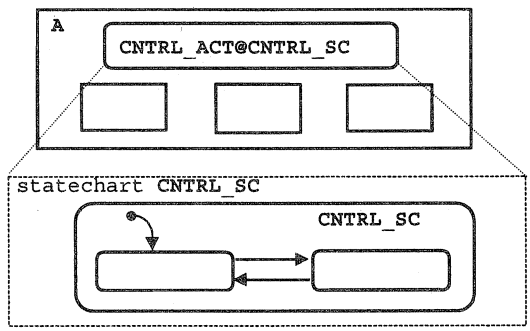

Figure 7.1 Associating a statechart with a control activity.

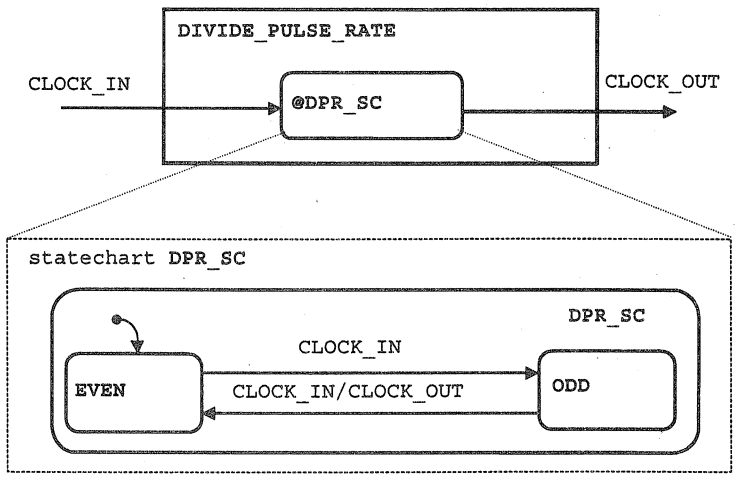

Figure 7.2 A statechart describing a simple activity.

subactivities to control; that is, its only subactivity is the control activity. See Fig. 7.2. The activity in this example doubles the rate of a clock pulse.

In some cases, the control behavior of an activity can be captured by static reactions alone, without the need for states and transitions. In such cases, the controlling statechart will consist of a single top-level state with the static reactions given in its Data Dictionary entry. If this behavior does not involve the control of sibling activities, a mini-spec can be used instead; see Sec. 7.4.

Finally, we should mention that while the controlling statechart may consume and produce external (control and data) information, its interface does not appear in the statechart itself. Rather, it shows up in the activity-chart as the interface of the control activity with which it is associated. This issue is also discussed in Chap. 8.

7.2.2 Termination type of an activity

In the discussion of dynamics in earlier chapters we saw several examples of activities that stop by themselves, from "within," and some that are stopped only from the "outside." We thus distinguish between activities that have *self-termination* and those that have *controlled termination.* (Of course, some can have both; we consider such cases to be self-terminating.)

If a self-terminating activity has a control activity, the corresponding statechart must contain a *termination connector,* also called a *T-connector.* This connector can appear anywhere in the statechart, and it is considered a final state; in particular, it has no exits. Upon entering this connector, the statechart "stops," its parent activity—call it `A`—becomes deactivated, and the event `stopped(A)` (abbreviated by `sp(A)`) occurs.

In the EWS example, the activities SET_UP and PRINT_FAULT are self-terminating. Figure 7.3 shows the controlling statechart of SET_UP, and it contains a termination connector. In contrast, COMPARE and PROCESS_SIGNAL are periodic activities with controlled termination. Also, DISPLAY_FAULT, which produces an alarm sound and displays a message on the screen, continues to do so until it is stopped (as we shall see later) by the controlling statechart of EWS_ACTIVITIES.

While reactive activities, whether data-driven or event-driven, can have either controlled termination or self-termination, procedure-like activities are always self-terminating. When invoked, a procedure-like activity performs a sequence of actions and stops. It is always a basic activity and lasts for one step of execution only.

The distinction between the two termination types can be made for both basic and nonbasic activities, and it is recorded in the Data Dictionary entry of the activity. An important point is that when a nonbasic activity stops, either because its statechart moves to a termination connector or it is stopped from the outside (e.g., by an explicit stop action, as we shall see), all its subactivities stop immediately, too.

7.2.3 Perpetual activities

We mentioned activities whose components are "always active." Because this kind of behavior pattern is very common in the specification of hardware systems, we refer to it as *hardware activation style.* This is a case in which an activity does not need to have a control activity. For nonbasic activities that do not have a control activity, we provide special default behavior: all the subactivities start when the parent activity starts, and they all stop when it stops. (The latter is always true, even in the presence of a control activity.)

In the EWS example, we may decompose DISPLAY_FAULT into two subactivities with no control activity, as shown in Fig. 7.4. When

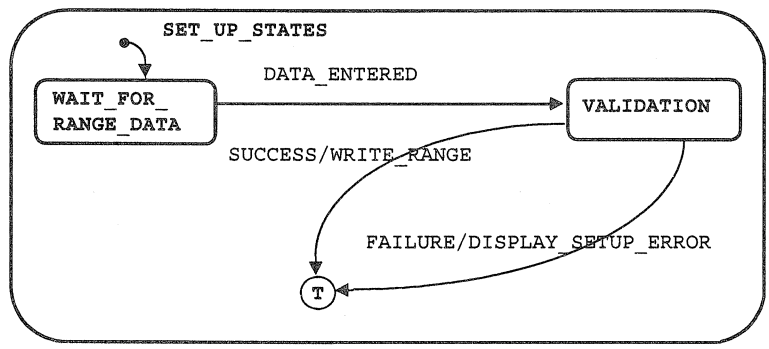

Figure 7.3 Termination connector in SET_UP's statechart.

DISPLAY_FAULT is activated, both DISPLAY_FAULT_MESSAGE and PRODUCE_ALARM_SOUND start simultaneously. They stop when DISPLAY_FAULT is stopped.

7.3 Controlling the Activities

We now show how the controlling statecharts affect and sense the status of their sibling activities.

7.3.1 Starting and stopping activities

One of the main mechanisms that statecharts use to control activities is the ability to activate (start) and deactivate (stop) them explicitly. This is usually carried out via the actions start(A) and stop(A), which are abbreviated as st!(A) and sp!(A), respectively.

To exemplify these actions, let us return to the dynamic and timing issues related to our EWS example, as described in Sec. 7.1. Here is how these decisions can be specified in the controlling statechart. Consider the statechart of Fig. 4.6 and compare it with Fig. 7.5. The event PRINT_OUT_OF_RANGE, which is generated on the transition from GENERATING_ALARM to WAITING_FOR_COMMAND, is replaced by the action st!(PRINT_FAULT).

This takes care of the activation of PRINT_FAULT. For all other activities, we can link their activation with the entrance to a state. For example, SET_UP is started by carrying out the action st!(SET_UP) upon entering the state SETTING_UP. A good way to achieve this effect is to attach a static reaction: ns/st!(SET_UP) to the SETTING_UP state. (Recall that ns abbreviates the entering event.) Similarly, the activities COMPARE, PROCESS_SIGNAL, and DISPLAY_FAULT are started upon entering the states COMPARING, OPERATING, and GENERATING_ALARM, respectively. The existence of the static reactions attached to these states is marked by a > symbol affixed to the state name, as shown in Fig. 7.6. Notice that entering these states is triggered by the events that were stated above to start the corresponding activities. For example, the SET_UP command causes entrance to the SETTING_UP state and therefore causes activation of the SET_UP activity.

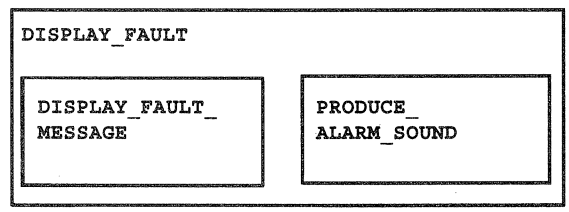

Figure 7.4 A nonbasic activity with no control activity.

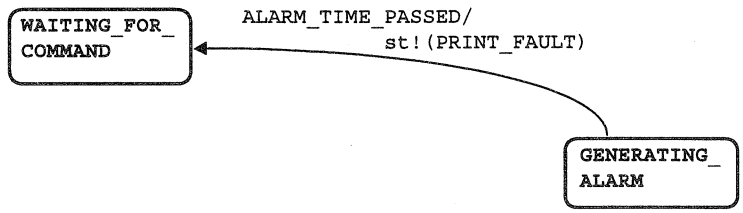

Figure 7.5 An action that starts an activity.

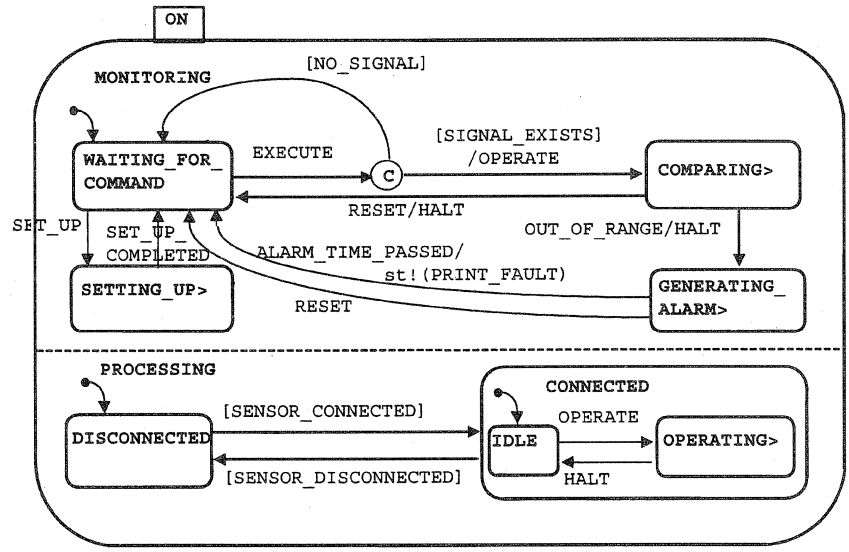

Figure 7.6 States marked as having entering and exiting reactions.

The COMPARE activity is stopped when the COMPARING state is exited by the reaction xs/sp!(COMPARE) that appears in the Data Dictionary entry of this state. (Recall that xs abbreviates the exiting event.) Note that the events OUT_OF_RANGE and RESET cause the system to exit the COMPARING state, and therefore they stop the COMPARE activity.

It is noteworthy that the action st!(A) has no effect if A is already active, and sp!(A) has no effect if it is not. Thus, for example, no message will be printed if st!(PRINT_FAULT) was executed a second time before the last activation of PRINT_FAULT terminated.

Starting an activity that has a controlling statechart with the start action will cause the statechart to begin "running" in parallel to all other statecharts that might be active in the model at that time. Similarly, stopping an activity by the stop action causes the controlling statechart of the stopped activity to abort and to remain dormant until its next activation, when it will restart in its top-level state. Moreover, if the stopped activity is nonbasic, all its subactivities stop, too.

Recall that the control activity can control only its sibling activities. Therefore, all actions that appear in its statechart may refer to the sibling activities only.

7.3.2 Sensing the status of activities

The statechart that describes a control activity is not limited to causing activities to start and stop. It can also sense whether such happenings have indeed taken place. Specifically, the control activity can sense the events `started(A)` and `stopped(A)`, and the condition `active(A)`, abbreviated as `st(A)`, `sp(A)`, and `ac(A)`, respectively. The event `st(A)` occurs when the activity `A` starts, `sp(A)` occurs when `A` terminates either by self-termination or by an external `stop` action, and the condition `ac(A)` is true for the duration of the period in which `A` is active.

In the EWS example, we can be more specific about the actual event that triggers the exit from `SETTING_UP`. It will be `sp(SET_UP)`, rather than `SET_UP_COMPLETED`. (See Fig. 7.7.)

Just as the control activity can control only its sibling activities, so can it sense only these siblings. Therefore, the events and conditions in the describing statechart are allowed to refer only to the sibling activities.

7.3.3 Activities throughout and within states

Often, we wish an activity `A` to start when a certain state `S` is entered and to stop when `S` is exited. This can be specified by associating the action `st!(A)` with the entering event `ns` as a static reaction in the Data Dictionary entry for `S` and the action `sp!(A)` with the exiting event `xs` therein. To cater for such cases in a more compact way, we may specify in the Data Dictionary entry for `S` that `A` is *active throughout* `S`. For example, the `COMPARE` activity can be specified as being `active throughout` the `COMPARING` state and `PROCESS_SIGNAL` as being `active throughout` the `OPERATING` state. See Fig. 7.8.

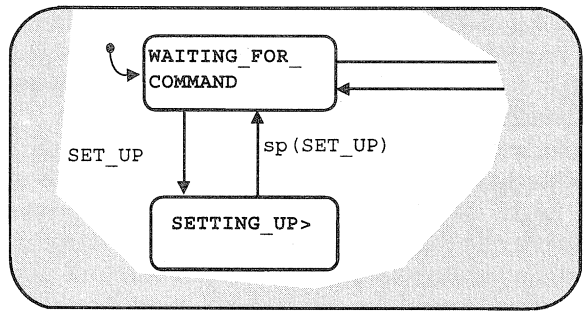

Figure 7.7 An event signifying termination of an activity.

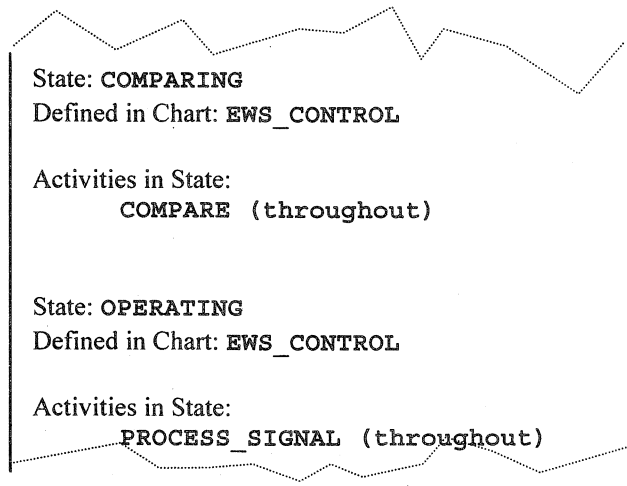

Figure 7.8 Activities active throughout states in the Data Dictionary.

The `throughout` correspondence between a state and an activity is natural for activities with controlled-termination because exiting the state will stop the activity. However, a self-terminating activity `A` may also be specified as being active throughout a state `S`. In such a case, there is usually an exit transition from `S` triggered by the event `sp(A)`; this implies that if and when `A` stops of its own accord `S` will be exited via this exit. If `A` stops and there is no such exit transition, the specification is misleading, but it is not a language error. If `S` is exited before `A` terminates on its own, `A` will stop as a result, just as if `A` had been of the controlled-termination type.

The following example illustrates this case. Assume that the self-terminating activity `SET_UP` is specified as `active throughout` the `SETTING_UP` state. Figure 7.9 shows an exit transition from this state, labeled `sp!(SET_UP)`. We have also added another exit transition, triggered by the `RESET` command, that enables the operator to abort the `SET_UP` activity during its execution.

Another similar association is *active within,* which represents a looser connection between an activity and a state. Again, we use the Data Dictionary to assert that activity `A` is `active within` state `S`. This is mainly done as a temporary specification to indicate that the activity is activated sometime during the time the system is in `S` but that we cannot be more concrete at present. One of the technical ramifications of this association is that when `S` is exited `A` stops (unless, of course, it had stopped earlier for some other reason). However, in contrast to the `throughout` connection, `A` does not necessarily start when `S` is entered.

For example, in Fig. 7.6, the activity `PROCESS_SIGNAL` can be defined as `active within` the `CONNECTED` state before that state is further decomposed into `IDLE` and `OPERATING`. The reason is that

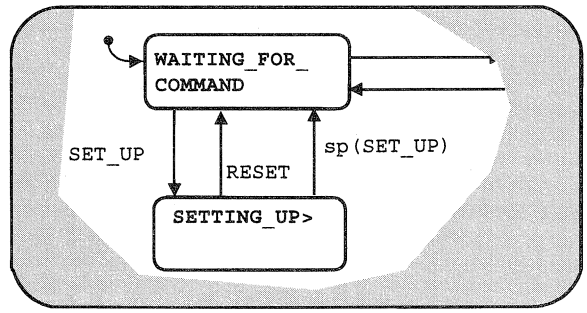

Figure 7.9 Self-terminated activity active throughout a state.

even before the decomposition we do know that it is meaningless to perform this activity when the sensor is disconnected. Later on, we can be more concrete and define the activity `PROCESS_SIGNAL` as being `active throughout OPERATING`.

7.3.4 Suspending and resuming activities

In addition to being able to start and stop activities, control activities can cause an activity to "freeze," or *suspend,* its activation, and to later *resume* from where it stopped. The relevant actions are `suspend(A)` and `resume(A)` (abbreviated as `sd!(A)` and `rs!(A)`, respectively). Associated with these actions is the condition `hanging(A)` (abbreviated as `hg(A)`), which is true as long as `A` is suspended without being resumed or stopped. It should be emphasized that an activity is considered active even when suspended. Thus whenever `hg(A)` is true, so is `ac(A)`.

Suspension may be used, for example, when we want to interrupt the progress of an activity in favor of another activity with a higher priority. Figure 7.10 shows a simple activity and its controlling statechart. The event `E` causes `A` to be suspended, while the preferred activity `B` is carried out to completion, at which time `A` is resumed.

If `A` is suspended, its descendant subactivities, including all descendant control activities, become suspended, too. This means that the corresponding statecharts stop in their tracks: they remain in the state configurations they were in at the instant of suspension. Upon resuming, all such control activities continue in a normal fashion from those configurations. While suspended, the statechart does not react to events. In fact, all events that occur during the suspension period are "lost" and have no effect on the suspended statechart.

Resuming a suspended activity sounds very much like entering a state via a history entrance (see Sec. 4.6.2). However, these notions should not be naively interchanged. To illustrate the difference, compare the state-

chart of Fig. 7.10 with Fig. 7.11. In the absence of any additional static reactions, the exit from the AC_A state in Fig. 7.11 does not cause activity A to either stop or suspend. Now assume that we remove the starting action from the default entrance in that figure and the activity A is defined to be active throughout state AC_A. In this case, the event E will cause A to stop, and returning to AC_A will cause it to start at the beginning, which is not the case in Fig. 7.10. Thus one must remember that reentering a state via a history entrance is considered an entrance nonetheless, and actions that are to be performed on entry (such as starting activities that are defined throughout) are indeed carried out.

7.4 Specifying Behavior of Basic Activities

When carrying out functional decomposition, the lower-level building blocks of the description are the basic activities, those that

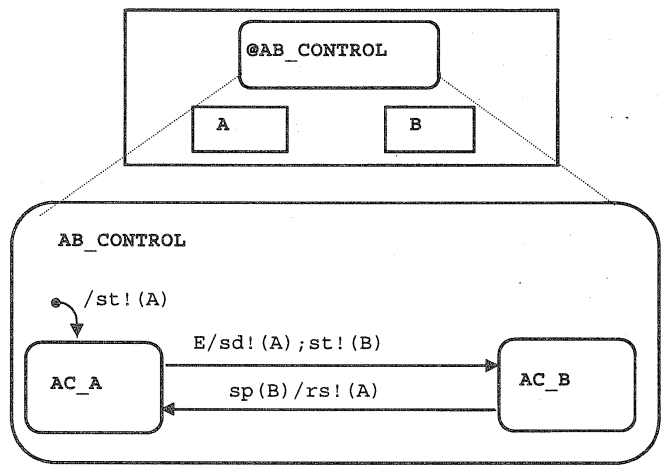

Figure 7.10 Suspending and resuming activities.

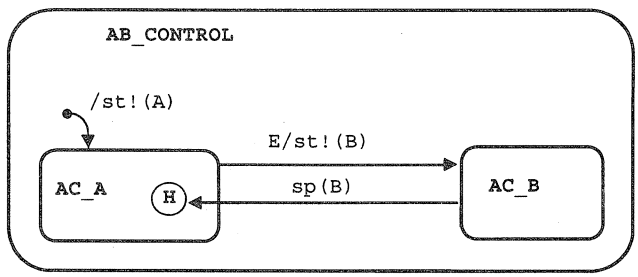

Figure 7.11 History entrance vs. resume activity.

require no further breakup. We may use the textual language of Chap. 5 to associate formal executable descriptions with basic activities without using the Statecharts language. These descriptions, called *mini-specs* and *combinational assignments*, are written in the Data Dictionary. (They were mentioned also in Sec. 2.4.2.) Basic activities that have additional textual descriptions in the Data Dictionary are marked by a > appended to their names, like states with static reactions.

7.4.1 Reactive mini-specs

In some cases the behavior of a basic activity can be described by a collection of reactions consisting of triggers and their implied actions. In these cases, a *reactive mini-spec* can be used.

The syntax of a reactive mini-spec is similar to that of a static reaction in a state; that is, it is a list of reactions of the form `trigger/action`, separated by a double semicolon (`;;`). The meaning is obvious: as long as the activity is active, an action is performed whenever the corresponding trigger occurs.

It is also possible to associate actions to be carried out when the activity starts by using the event `started` (abbreviated by `st`) as the trigger in the mini-spec. This event occurs one step after the action `st!(A)` is performed, like the event `started(A)`. Notice that the name of the activity does not appear in this event because the reaction is associated with the activity itself.

Figure 7.12 describes the `PROCESS_SIGNAL` activity of the EWS as a reactive mini-spec. This activity reads the sensor's output, `SIGNAL`, every `SAMPLE_INTERVAL` and transfers the read value to a processing user function `COMPUTE()`, which is unspecified in the model. The function's output, `SAMPLE`, is later checked for being inside the required range. The sampling cycle is implemented by an internal event `TICK`, which is first scheduled when the activity is started, and is then scheduled for the subsequent cycle.

A reactive mini-spec can be attached to both self-terminating or controlled-terminating activities. To stop a self-terminating activity, the `stop` action (abbreviated `sp!`), which also has no activity name, is used. The stopped activity becomes inactive in the next step. In the current step, it continues to apply other reactions if there are other enabled triggers, and it even completes the sequence of actions that follows the `stop` action (although it is probably bad practice to write action expressions with actions that follow a `stop` action).

To illustrate usage of the `stop` action in the EWS example, let us assume (over and above the original requirements) that the `PROCESS_SIGNAL` stops when it finds that the sensor signal is zero. This can be specified as in Fig. 7.13.

```
Activity: PROCESS_SIGNAL
Defined in Chart: EWS_ACTIVITIES
Termination Type: Reactive Controlled
Mini-spec: st/TICK;;
TICK/ $SIGNAL_VALUE:=SIGNAL;
      SAMPLE:=COMPUTE($SIGNAL_VALUE);
      sc!(TICK,SAMPLE_INTERVAL)
```

Figure 7.12 A reactive mini-spec in the Data Dictionary.

```
Activity: PROCESS_SIGNAL
Defined in Chart: EWS_ACTIVITIES
Termination Type: Reactive Controlled
Mini-spec: st/TICK;;
TICK/ $SIGNAL_VALUE:=SIGNAL;
      if ($SIGNAL_VALUE # 0) then
        SAMPLE:=COMPUTE($SIGNAL_VALUE);
        sc!(TICK,SAMPLE_INTERVAL)
      else
        sp!
      end if
```

Figure 7.13 The `stop` action in a mini-spec.

It is important to remember that states and activities cannot be referred to in the mini-spec. All the activities and states of the model are beyond the scope of an individual mini-spec.

7.4.2 Procedure-like mini-specs

Very often, an activity can be described as a sequence of actions, possibly with conditional branching and iterations. Such activities are called *procedure-like* and are actually similar to actions: they are described by a mini-spec that has an action syntax and are active for a single step only. Obviously, such activities are always self-terminating.

For example, let us return to the SET_UP activity, whose activity chart and controlling statechart are shown again in Fig. 7.14. The VALIDATE_RANGE subactivity is active throughout the VALIDATION state. It can be described by a very simple procedure-like mini-spec, as shown in Fig. 7.15. Notice the > marks in the statechart and activity-chart, which indicate that the VALIDATION state and the VALIDATE_RANGE activity have an additional behavioral description in the Data Dictionary.

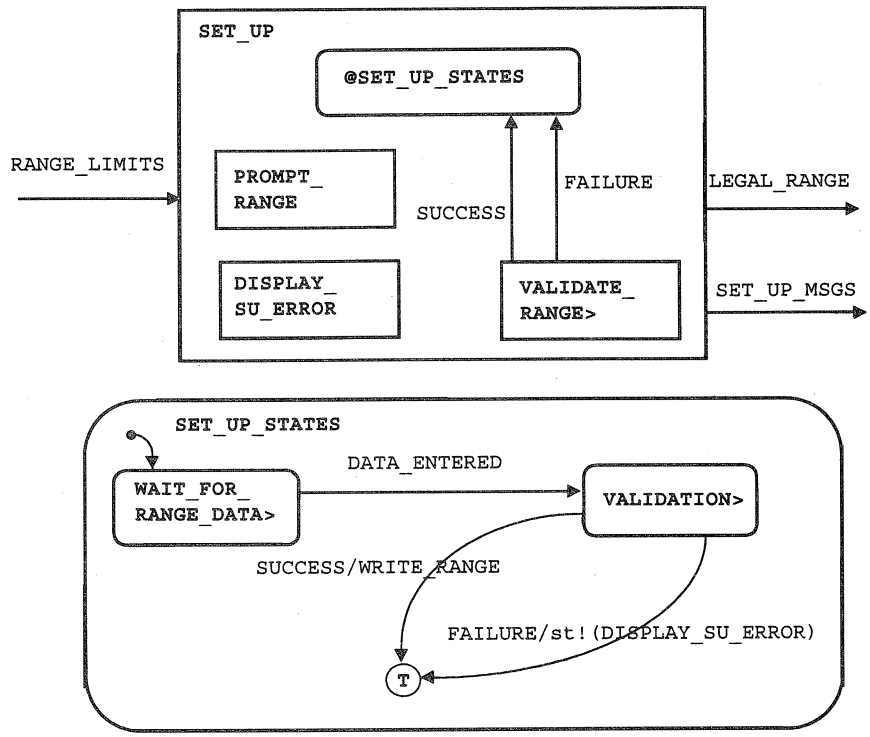

Figure 7.14 SET_UP activity and controlling statechart.

Activity: **VALIDATE_RANGE**
Defined in Chart: **SET_UP**
Termination Type: **Procedure-Like**
Mini-spec:

```
if LOW_LIMIT < HIGH_LIMIT then
  SUCCESS
else
  FAILURE
end if
```

Figure 7.15 A procedure-like mini-spec in the Data Dictionary.

As in the case of reactive mini-specs, procedure-like mini-specs are not allowed to refer to states and activities.

7.4.3 Combinational assignments

Another typical behavior for an activity is that of a *data-driven* pattern. The activity is continuously ready to perform some calculations

whenever the input changes its values. In principle, this pattern can be described by a reactive mini-spec in which the required calculations are performed when the activity starts and also whenever an event `changed(X)` occurs for any relevant data-item or condition `X`. We provide an alternative, more convenient way to describe data-driven activities: *combinational assignments.* These are associated with the activity via the Data Dictionary, just as mini-specs are.

The general syntax of a combinational assignment is:

```
X := Y1 when C1 else
     Y2 when C2 else
     . . .
     Yn
```

when `X` is a variable condition or data-item, `Y1` to `Yn` are expressions, and `C1` to `Cn` are condition expressions.

For example, let us define a subactivity `COMPUTE_IN_RANGE` of the `COMPARE` activity, with the following combinational assignment:

```
IN_RANGE:= false when (SAMPLE < LEGAL_RANGE.LOW_LIMIT)
     else false when (SAMPLE > LEGAL_RANGE.HIGH_LIMIT)
     else true
```

This combinational assignment was designed to illustrate the syntax of the construct, but there is actually a simpler way to obtain the same effect, using only a simple expression with no `when` clause, as shown in Fig. 7.16.

Whenever `SAMPLE` changes its value, the combinational assignment recomputes the value of `IN_RANGE`. (The `OUT_OF_RANGE` event will be generated by another action when `IN_RANGE` becomes false.)

The left-hand side of the assignment is called a *combinational element.* It can be of a numeric type, a string, or a condition. It can be an array component (with a constant index) or a record field. It can also be a bit-array slice but, again, only with constant range indices.

The combinational assignments are performed at the end of an execution step. If, during the step, a value of an element appearing on the right-hand side of some combinational assignment is changed, the assignment is carried out using the new values. If

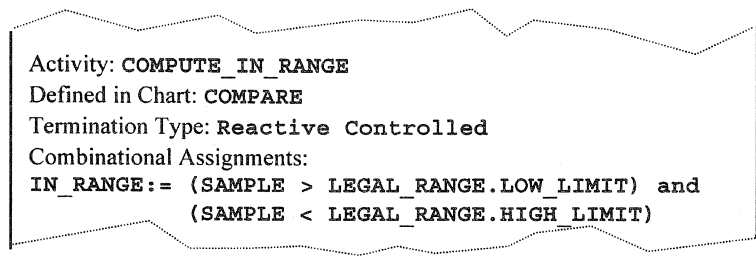

Figure 7.16 Combinational assignments in the Data Dictionary.

doing so changes the value of an element in some assigned expression, an additional computation phase is called for. Of course, this chain of computations can be infinite, resulting in an unstable design.