Chapter

4 The Behavioral View

Statecharts

This chapter describes the language of Statecharts (Harel 1987b),[1] which is used to describe the control activities in activity-charts. As explained in Chap. 2, these activities constitute the behavioral view of a model.

In this chapter and the two that follow, we concentrate on the pure features of Statecharts and their semantics, leaving those parts that pertain to the connection with Activity-charts to Chap. 7. Thus we do not concern ourselves here with the way activities are controlled by statecharts or with how statecharts are affected by activities, but only with the internal features of the statecharts themselves.

This chapter describes how states are organized into an *and/or hierarchy* and how they may represent levels of behavior and concurrency. We also show how transitions are used (with the various connectors) to describe changes in the states. In Chap. 5 we describe the textual expression language used in Statecharts to specify triggers and actions, and how it supports timing considerations. Chapter 6 describes the dynamic semantics of Statecharts. Throughout, the reader will observe that Statecharts constitute a powerful extension of conventional state-transition diagrams.

4.1 Behavioral Description of a System

A behavioral description of a system specifies dynamic aspects of the entire system or of a particular function, including control and timing.

[1]Parts of our description here follow Harel (1987b), although our version reflects some modifications and enhancements that were incorporated to make it better fit the STATEMATE modeling approach.

It specifies the states and modes that the system might reside in and the transitions between them. It also describes what causes activities to start and stop, and the way the system reacts to various events. The functional and behavioral views complete each other, as explained in later chapters.

A natural technique for describing the dynamic of a system is to use a *finite-state machine.* The described system or function is always in one of a finite set of *states.* When an event occurs, the system reacts by performing *actions,* such as generating a signal, changing a variable value, and/or taking a *transition* to another state. The events causing the reaction are called *triggers.* For example, a simple mechanism that controls a light bulb may be in one of two states, OFF and ON. The event BUTTON_PRESSED might trigger the transitions from one of these states to the other. On moving from OFF to ON, the mechanism sends a signal TURN_ON to the light bulb, and similarly, the bulb is turned off on the other transition (see Fig. 4.1).

Let us analyze the behavior of the EWS in terms of states or modes. From the informal description of the EWS presented in Chap. 1 we can identify several main states of the system:

WAITING_FOR_COMMAND	The system is idle, waiting for an operator command o start executing or to set up the range values.
SETTING_UP:	The range values are being set by the operator.
COMPARING:	The signal processing is being performed, and the processed signal is being checked.
GENERATING_ALARM:	The system is generating the alarm to indicate that the value of the processed signal is out of range and is awaiting the operator's reset.

These states are exclusive, that is, when the system is accepting new range limits, it is not performing signal processing or value comparisons. Similarly, the comparisons are not carried out when the alarm is generated. Regarding the transitions between states, when in WAITING_FOR_COMMAND, the EXECUTE command from the operator causes the system to move to the COMPARING state, and the SET_UP command causes a transition to SETTING_UP. This description implies that the system moves to the GENERATING_ALARM state in the event that

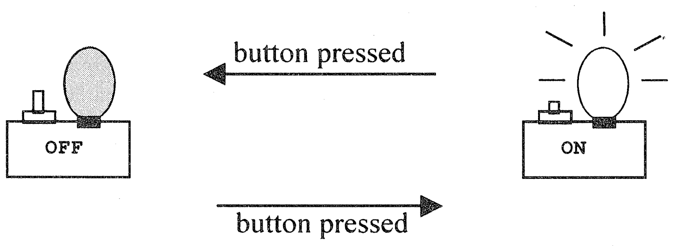


Figure 4.1 A finite-state machine that controls a light bulb.

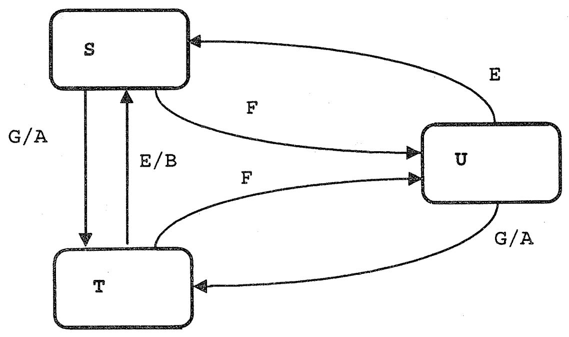


Figure 4.2 A simple state-transition diagram.

the tested signal is out of range. More details about the transitions between these states are given in the following sections.

In some of the states, certain functions from the functional description are performed (on the assumption that we are carrying out a function-based decomposition; see Chap. 2). For example, the SET_UP activity is performed in the SETTING_UP mode. In general, the functional and behavioral views are combined to yield the entire conceptual description of the system under description. This subject is discussed in Chap. 7.

Finite-state machines have an appealing visual representation in the form of *state-transition diagrams*. These are directed graphs in which nodes denote states and arrows denote transitions. The transitions are labeled with the triggering events and caused actions, using the following general syntax for a reaction: *trigger/action*. Figure 4.2 shows a simple three-state diagram that describes a system.

If, for example, the system is in state S and event F occurs, the system is transformed into state U. If, in the same state, G occurs, the system performs the action A and ends up in state T.

In our approach we use the Statecharts language to describe the behavioral view. This language is similar to state-transition diagrams, but includes many enhancements, such as hierarchy, orthogonality, expressions, and connectors. As in Activity-charts, the elements appearing in the charts have associated entities in the Data Dictionary. In the following sections and in the two subsequent chapters, we describe the details of the Statecharts language. The way in which statecharts relate to activity-charts is dealt with in Chaps. 7 and 8.

4.2 Basic Features of Statecharts

As in conventional state-transition diagrams, statecharts are constructed basically from states and transitions. The states in a statechart are depicted as rectilinear boxes with rounded corners. The names of the states appear inside their boxes and obey the name syntax given in App. A.1. The transitions are drawn as splined arrows, with the triggers serving as labels.

The main states of the EWS and the transitions and their triggers are shown in Fig. 4.3.

The triggers of the transitions in Fig. 4.3 are all events, which are regarded as instantaneous occurrences. There are two kinds:

- *External* events coming from external sources (such as the commands coming from the operator via the control panel: SET_UP, EXECUTE, and RESET).
- *Internal* events coming from internal sources (such as OUT_OF_RANGE, which is output from the COMPARE activity; ALARM_TIME_PASSED, which is the output of some invisible clock; and SET_UP_COMPLETED, which signifies that the SET_UP activity has terminated).

We shall see later that the event ALARM_TIME_PASSED can be defined to be more specific about the alarm duration, using the timing facilities provided by our languages. Note that we do not show the source of the triggering events in the statechart itself. We shall return to this issue in Chap. 8.

The trigger of a transition may be an expression that combines some events. It may also include a condition, enclosed in square brackets, or it may consist of the condition only. Thus if a transition is labeled E[C], the condition C is tested at the instant the event E occurs, guarding the transition from being taken if it is not true at that time. If the transition is labeled [C], the condition C is tested at each instant of time when the system is in the transition's source state, and the transition is taken if it is true.

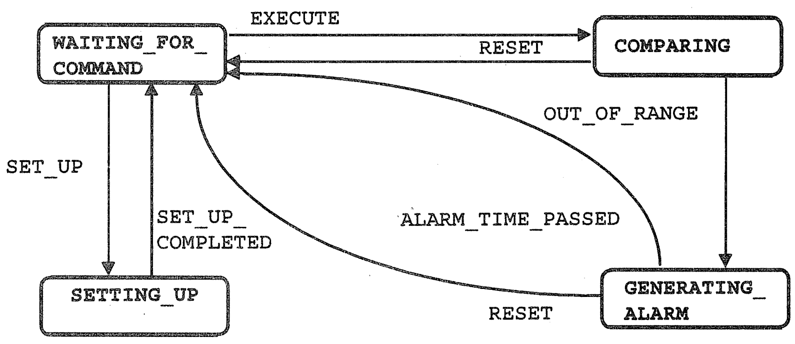


Figure 4.3 States, transitions, and event triggers.

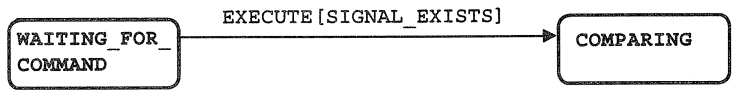


Figure 4.4 A trigger with a condition.

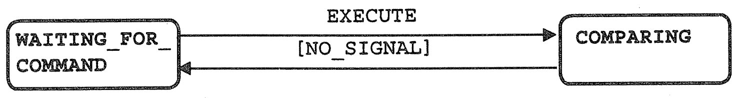


Figure 4.5 A condition as a trigger.

In the EWS example, we may want to prevent the transition between WAITING_FOR_COMMAND and COMPARING from being taken unless the sensor is connected to the system and there is a signal coming from the sensor. We could do this as in Fig. 4.4, by enriching the statechart of Fig. 4.3, with an appropriate condition.

In fact, a similar effect could be achieved differently by using a condition to trigger a transition, rather than as a guard on a triggering event. In Fig. 4.5, we take the transition to COMPARING when the EXECUTE command is issued, but once in COMPARING, we continuously monitor the condition NO_SIGNAL, returning to WAITING_FOR_COMMAND the instant we detect that it is true. In this way, if the sensor is not connected and there is no signal coming from the sensor when we enter COMPARING, we will immediately return to WAITING_FOR_COMMAND. However, if there is a signal, we will stay in COMPARING until the sensor is disconnected and the signal ceases.

A transition can be labeled not only with the trigger that causes it to be taken, but also, optionally, with an action, separated from the trigger by a slash as follows: *trigger/action*. If and when the transition is taken, the specified action is carried out instantaneously. Some actions simply generate an event, but they may also cause other effects. We shall see that actions can modify values of conditions and data-items; they can start and stop activities, and more. Several actions can be performed when a transition is taken. The actions are written after the slash in a sequence, separated by a semicolon (e.g., E/A;B;C).

A simple action incorporated into the EWS example is shown in Fig. 4.6. Here, we have decided that when the ALARM_TIME_PASSED event occurs in the GENERATING_ALARM state, two things happen simultaneously:

- The system returns to the WAITING_FOR_COMMAND state.
- The event PRINT_OUT_OF_RANGE, which is really an internal command to print a fault report on a printing device, is generated.

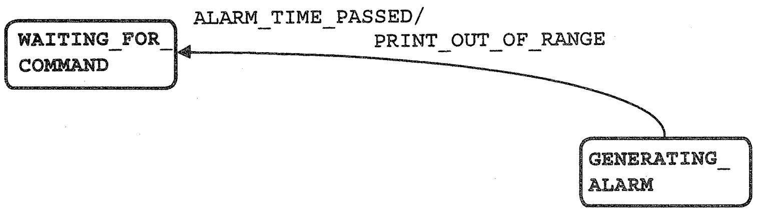


Figure 4.6 A simple action.

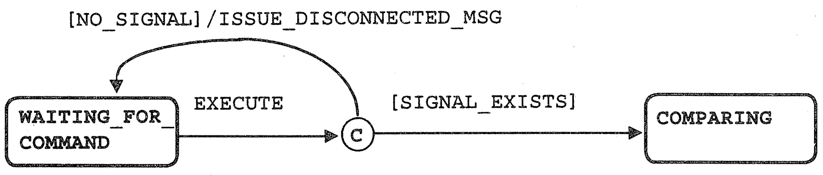


Figure 4.7 A condition connector.

Another way of using conditions to guard transitions is to employ the *condition connector.* An arrow enters the connector, labeled with the triggering event, and the connector may have several exit arrows, each labeled with a condition enclosed in square brackets and optionally also with an action. In general, any number of exit arrows from a condition connector is allowed.

Figure 4.7 shows how the EXECUTE event causes a transition from WAITING_FOR_COMMAND, with the two mutually exclusive conditions NO_SIGNAL and SIGNAL_EXISTS, that determine whether the system enters COMPARING or returns to WAITING_FOR_COMMAND. In the latter case, we have also specified that the event ISSUE_DISCONNECTED_MSG will be generated, causing an error message to appear.

Although the mechanisms of states and transitions labeled by triggers and actions allow rich and complex behavioral descriptions, they are not always enough. Later we will discuss the ability to specify reactions that do not involve transitions between states and to associate them with a specific state. These reactions, as well as the information about the activities that are active in a state, are attached to the state through the Data Dictionary. Like the other elements appearing in statecharts, such as triggers and actions, each state also has an associated entry in the Data Dictionary. Transitions, however, do not have Data Dictionary entries, mainly because they are not identifiable by name.

4.3 The Hierarchy of States

As it turns out, highly complex behavior cannot be easily described by simple, "flat" state-transition diagrams. The reason is rooted in the unmanageable multitude of states, which may result in an unstructured and chaotic state-transition diagram. To be useful, the state machine approach must be modular, hierarchical, and well structured. In this section we show how states can be beneficially *clustered* into a hierarchy.

Recall Fig. 4.2. Since event F takes the system to state U from either state S or state T, we may cluster the latter into a new state, call it V, and replace the two F transitions with one, as in Fig. 4.8.

The semantics of the new state V is as follows: to be in V is to be, exclusively, in either of its *substates,* S or T. This is the classical exclusive-or applied to states. V is called an *or-state,* and it is the *parent* of the two sibling states S and T. The F transition now emanates from on V, meaning that whenever F occurs in V, the system makes a transition to U. But because being in V is just being in S or T, the new F arrow precisely abbreviates the two old ones.

Applying this feature to our example, we may cluster the states COMPARING and GENERATING_ALARM into a new state (which does not need to have a name), simply because of the common exit transition triggered by the operator command RESET. (See Fig. 4.9.)

We can also achieve results similar to those shown in Figs. 4.8 and 4.9 not by clustering, which is a bottom-up operation, but by *refinement,* which is top-down (as in the functional decomposition that is presented in Chap. 2). For example, we could have started the EWS behavioral description with the two-state decomposition of Fig. 4.10, in which one top-level state, EWS_STATES, is decomposed into two

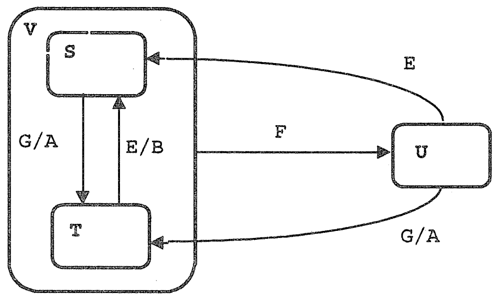


Figure 4.8 Clustering of states.

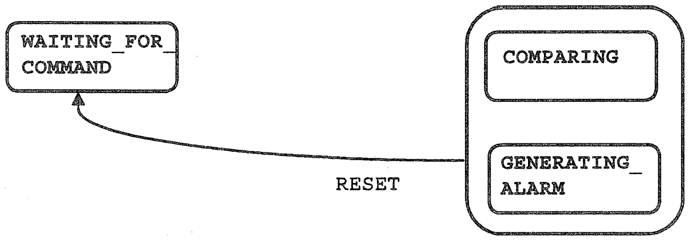


Figure 4.9 Clustering of EWS states.

substates, OFF and ON. These states are connected by two transitions, labeled POWER_ON and POWER_OFF.

We specify that the initial state of the system is OFF by using a *default transition,* specified by a small arrow emanating from a small solid circle. We can then zoom in to the ON state, and show the next-level state decomposition of the EWS. This results in the multilevel statechart of Fig. 4.11. The EWS states from Fig. 4.3 appear here as substates of the state ON. The default transition to WAITING_FOR_COMMAND indicates that this state is the default entrance of the ON state. This means that when there is a transition that leads to the borderline of the parent state, without indicating which of the substates is to be entered, like the one triggered by POWER_ON, the system enters the default substate.

The main advantage of using default transitions is in cases where there is more than one entrance to the parent state. Note that the top level of each parent state can have only one default entrance. A default transition usually leads to a substate in the first level of the state decomposition, but it can be made to directly enter a state on a lower level, as shown in Fig. 4.12.

Some terms and conventions that we use for the hierarchy of statecharts are similar to those used for activity-charts. A state that has no substates, such as WAITING_FOR_COMMAND, is referred to as a *basic state.* The state EWS_STATES is an *ancestor* of its *descendants,* which consist of all other states in Fig. 4.11. As in activity-charts, we say that a transition exits from its *source state* and enters its *target state.*

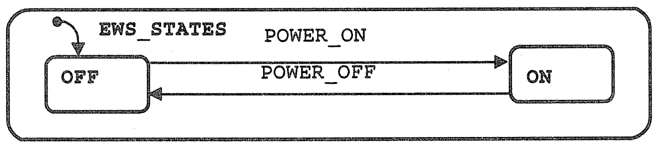


Figure 4.10 Top-level state decomposition of EWS.

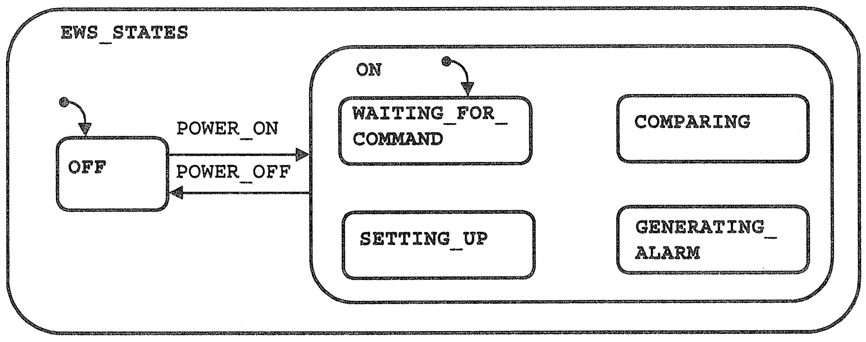


Figure 4.11 A multilevel statechart.

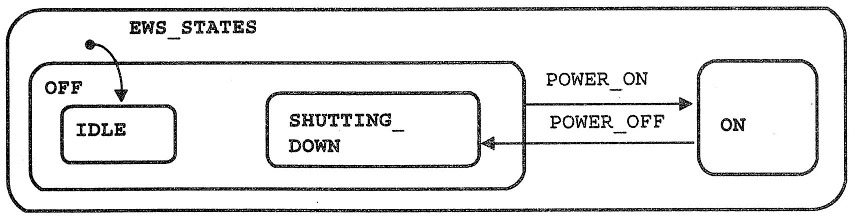


Figure 4.12 Default entrance to a lower-level state.

4.4 Orthogonality

4.4.1 And-states and event broadcasting

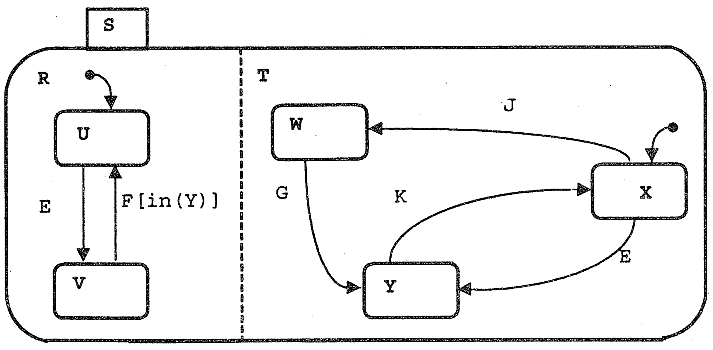
One of the main problems with descriptions of behavior is rooted in the acute growth in the number of states as the system is extended. Consider a statechart with 1000 states that describes certain control aspects of a flight-control system. Suppose that the behavior is now enriched by making its details depend to a large extent on whether the aircraft is in autopilot mode. With the features we have so far we might have to double the number of states, obtaining two versions of each of the old states—one with autopilot and one without—altogether 2000 states. As more additions are made, the number of states grows exponentially.

An additional problem arises when we want to describe independent or almost independent parts of the behavior (e.g., the behavior of several different subsystems) in a single statechart.

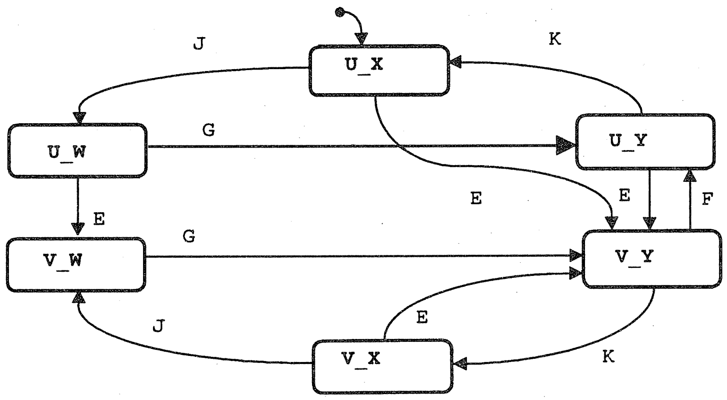
Statecharts handle these cases by allowing the *and-decomposition* of a state. This means that a state S is described as consisting of two or more *orthogonal components,* and to be in state S entails being in all of those components simultaneously. S is then called an *and-state.* The notation used is a dashed line that partitions the state into its components. The name of the and-state is attached to the state frame. The orthogonal components are named like regular states.

Figure 4.13*a* shows a state S consisting of the two components R and T, and being in S is being in both. However, because each component is an or-state, the first consisting of U and V and the second consisting of W, X, and Y, it follows that to be in S is to be in one of U or V as well as one of W, X, or Y. Such a tuple of states, each from a different orthogonal component, is called a *state configuration.* We say that S is the *parent* of its components R and T, or that R and T are the *substates* of S, as in the case of or-decomposition. The components R and T are no different from any other states; they may have their own substates, default entrances, internal transitions, and so on.

Entering S from the outside is tantamount to entering the configuration (U, X) by the default arrows. If E occurs in (U, X), the system transfers simultaneously to (V, Y), a transition that is really a form of synchronized concurrence—a single event triggering two simultaneous



(a)



(b)

Figure 4.13 Orthogonality using and-decomposition. (*a*) An and-state. (*b*) A nonorthogonal equivalent.

happenings. If K now occurs, the new configuration is (V, X), yielding a form of independence—a transition is taken in the T component, independently of what might be happening in the R component. Notice the in(Y) condition appearing in R. It signifies that the F transition from V to U is taken only if the system is in (V, Y). Thus one component is allowed to sense which state the other is in.

Figure 4.13*b* is the conventional "and-free" equivalent of Fig. 4.13*a*, and while it is not much larger than Fig. 4.13*a*, it illustrates the blow-up in the number of states: if Fig. 4.13*a* had 100 states in each component, giving a total of 200 bottom-level (basic) states, Fig. 4.13*b* would have had to contain all 10,000 combinations explicitly!

Returning to our EWS example, consider Fig. 4.14. Here we have added an orthogonal component to the ON state named PROCESSING. Its role is to describe the processing aspects of the raw signal read from the external sensor.

The conditions SENSOR_CONNECTED and SENSOR_DISCONNECTED in the PROCESSING component indicate the status of the connection with the sensor. They are set by the operator and are thus external. The OPERATE and HALT events, on the other hand, being generated by the MONITORING component, are internal. They are generated by actions when the system enters and exits the COMPARING state, respectively, and serve to indicate to the processing unit whether the system has completed the comparing of the processed signal.

Notice how these events are sensed immediately by the orthogonal component. Moreover, events generated by actions in one component are sensed by all other orthogonal components. For example, if there were more than one sensor, each with a corresponding signal processing unit, we could have modeled each of them by its own component, and the OPERATE and HALT events would have then been broadcast automatically to each one of them.

4.4.2 Conditions and events related to states

It is interesting to note that some of the events and conditions that label transitions in the EWS example now depend on and refer to states in the orthogonal component. Thus we may replace the conditions NO_SIGNAL and SIGNAL_EXISTS in Fig. 4.14 with in(CONNECTED) and in(DISCONNECTED), respectively. In fact, we may refer not only to the status of being or not being in a state as a condition but to the moment of entrance or exit as an event. The syntax is entered(S) and exited(S), with en and ex abbreviating the verbs. We may thus replace the OPERATE and HALT events in the CONNECTED state by en(COMPARING) and ex(COMPARING), respectively, and

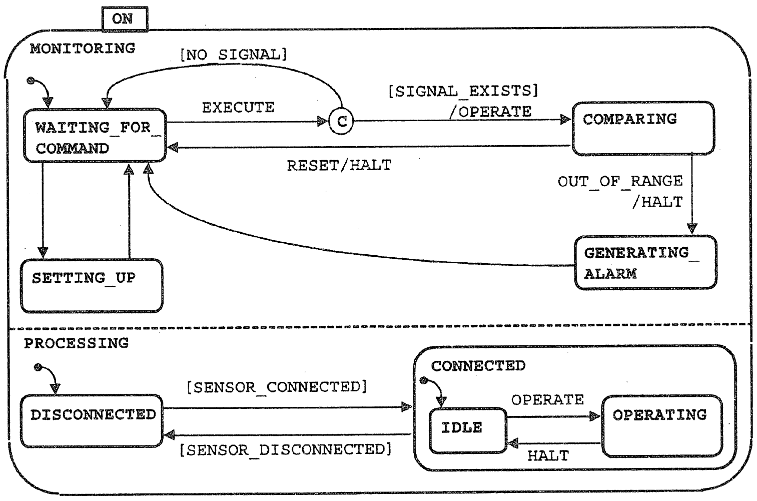


Figure 4.14 An and-state in the EWS.

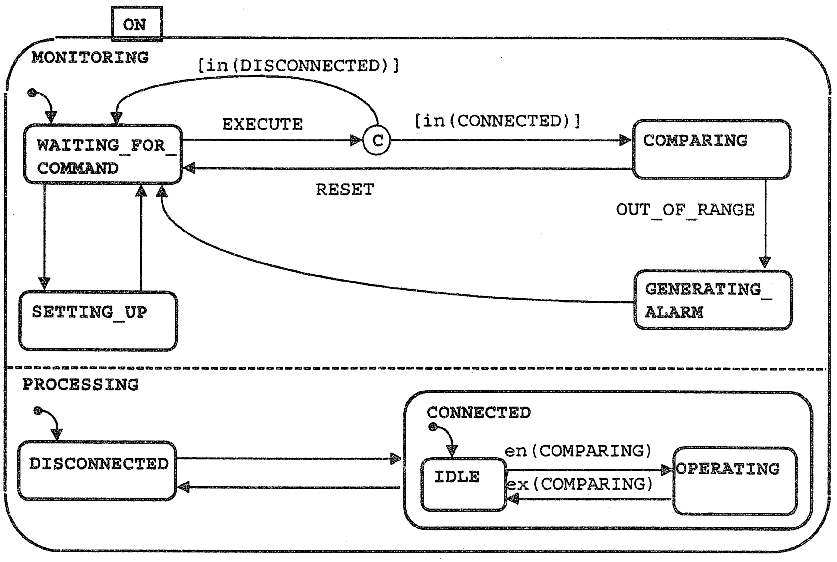


Figure 4.15 Conditions and events related to states.

the two events need no longer be explicitly generated by actions along transitions. The resulting statechart is shown in Fig. 4.15.

Note that the meaning of the and/or decomposition of states implies the following:

- If the system is in a state S, then not only is in(S) true but in(T) holds for each ancestor T of S.
- Entering a state S will trigger the event en(S), as well as en(T) for every ancestor T of S in which the system did not reside when S was entered.
- Exiting a state S will trigger the event ex(S), as well as ex(T) for each ancestor T of S in which the system does not reside after the transition.

Even in cases where states are exited and entered by looping transitions, such as the transition from or to WAITING_FOR_COMMAND shown in Fig. 4.7, the corresponding events ex(S) and en(S) occur. Section 5.4 contains further explanation about when these events occur.

The condition in(S) and the events en(S) and ex(S) may not be applied to an and-component such as PROCESSING. Instead, they should be applied to the parent and-state (in this case, the state ON).

4.4.3 Multi-level state decomposition

Orthogonal break-up into components is not restricted to a single level. For example, we might have further refined the OPERATING state of the

EWS, within CONNECTED, into two components: one deals with the clock rate of the signal sampling and the other with the computation mode. This is shown in Fig. 4.16.

Note that "high-level" transitions continue to apply, regardless of whether a state has orthogonal components. Thus the HALT event, for example, takes the system out of whatever state configuration within OPERATING it is in and causes entry into IDLE.

An important point is that there are no scoping restrictions within a single statechart, so any state can be referred to anywhere in the statechart, even if the state referred to appears in some level lower down.

As with activities, two states may have the same name if they have different parent states, in which case their names are distinguished by using *path names,* that is, by attaching their ancestors' names separated by periods. Thus had we chosen to rename CONNECTED and DISCONNECTED simply by ON and OFF, we would have to write PROCESSING.ON and PROCESSING.OFF whenever they had to be distinguished from the ON and OFF that reside within the top-level state. This convention is not limited to a single level; a sequence of several state names can be given, separated by periods, such as S1.S2.S3. Notice that no particular relationship is implied between states that have the same name.

4.5 Connectors and Compound Transitions

As in activity-charts, we allow several kinds of connectors in statecharts. They are used to help economize in arrows to clarify the specification.

4.5.1 Condition and switch connectors

As mentioned earlier, statecharts may employ *condition connectors,* also called *C-connectors.* Figure 4.7 showed an example. In general, the conditions along the branches emanating from the C-connector

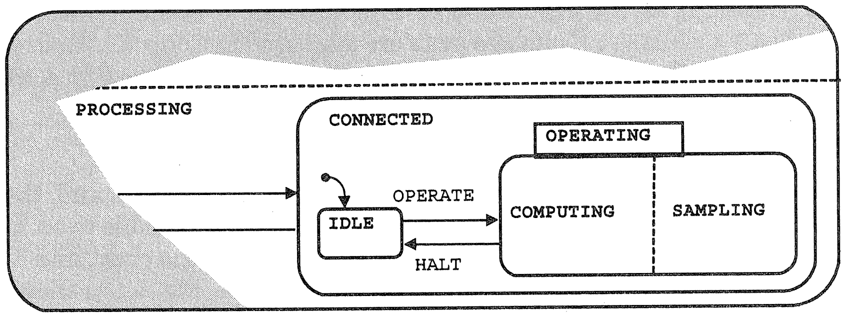


Figure 4.16 And-decomposition on any level.

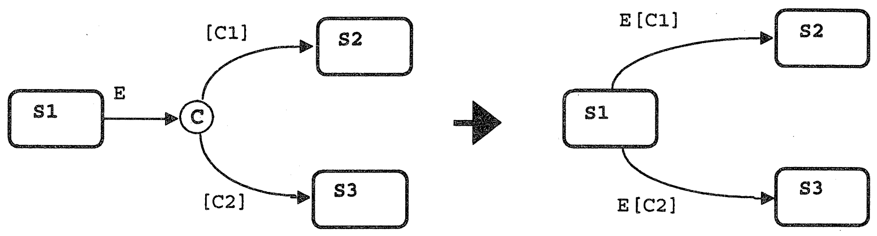


Figure 4.17 A condition connector and compound transitions.

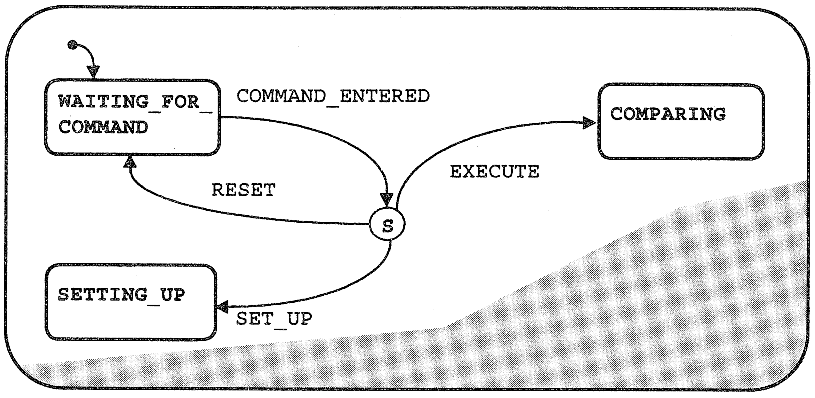


Figure 4.18 A switch connector.

must be exclusive, but there can be more than two such branches. When the conditions are not exclusive, a situation of nondeterminism ensues, which is discussed in more detail in Sec. 6.3.

Figure 4.17 shows a simple case of using the C-connector and the equivalent *logical transitions.* Each logical transition is represented by a *compound transition* consisting of two simple transitions. The transition labeled E is part of both.

Another connector, similar to the C-connector, is the *switch connector,* also called the *S-connector,* which is usually used with events rather than conditions. In our EWS example, we may define a named event, COMMAND_ENTERED, as the disjunction of three command events: EXECUTE or SET_UP or RESET. (Named events are discussed in Chap. 5.) We may then deal with the command-driven transitions of Fig. 4.3 as in Fig. 4.18.

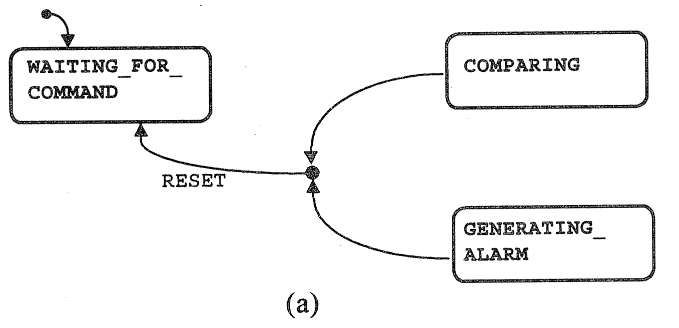
4.5.2 Junction connectors

Transition arrows can be joined using *junction connectors,* and the labels along them can be split as desired. This makes it possible to economize both in the number of lengthy arrows present in the chart and in the number of identical portions of labels. For example, Fig. 4.19*a* shows how to use a junction connector if the same event (RESET, in this case) causes exit from two states, but we do not want to cluster the two states into one.

Figure 4.19*b* shows a more subtle case, in which two events lead out of a state into two separate states, but there is a common action that is to be carried out along both. As this last example shows, the order in which events and actions appear along the parts of the compound transitions formed by using junction connectors is unimportant. However, all the triggers appearing along the parts of a compound transition must occur at exactly the same time for the transition to be taken. If and when that happens, all the actions appearing along the transition are carried out. As an example, the two parts of Fig. 4.20 are actually equivalent.

Multiple entrances and exits may be attached to a junction, and the semantics prescribes creation of logical compound transitions from all possible combinations of paths. The same is true of C-connectors and S-connectors.

The different connectors are meant to visually emphasize the distinction between different kinds of behavior: a C-connector indicates branching by conditions, an S-connector branches by events, and junction connectors are used for the remaining cases.



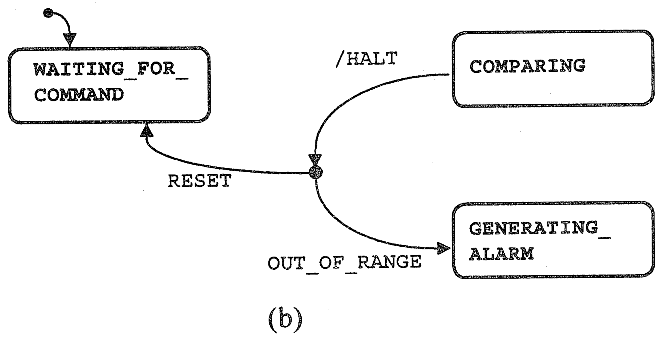


Figure 4.19 Junction connectors.

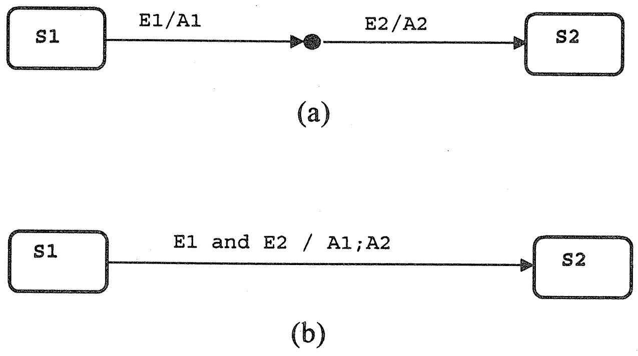


Figure 4.20 Two equivalent transition constructs.

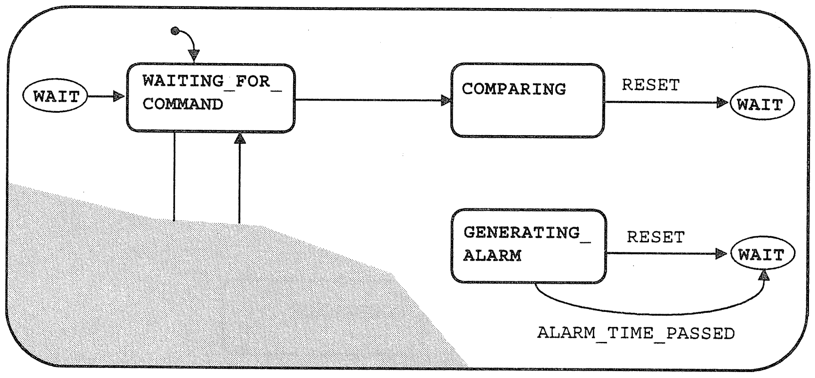


Figure 4.21 Diagram connectors.

4.5.3 Diagram connectors

As in activity-charts, statecharts also allow diagram connectors. These are simply a means for eliminating lengthy arrows from the chart in favor of marking two points in the chart and indicating that the arrow flows from one point to the other. See Fig. 4.21.

Any legal name may be used to label the diagram connectors (see App. A.1), as can any integer. Each occurrence must have only entering arrows or only exiting ones. Triggers and actions are concatenated along all possible combinations of paths that constitute compound transitions, as with other connectors.

4.6 More about Transitions

4.6.1 Transitions to and from and-states

Recall that being in an and-state is being in a configuration of states—one from each component. As a consequence, the Statecharts language allows splitting and merging arrows to denote entries to and exits from state configurations.

Figure 4.22 shows an alternative way of describing the transition from OFF to ON in our EWS example. Instead of having a default entrance in each component (as in Fig. 4.14), we have a *fork construct* that depicts the entrance to the default configuration directly. We may view a fork as another kind of compound transition, with the splitting point of the two branches as a special *joint connector*.

Such a transition is taken if and when all of its triggers occur, and when taken, all of its actions are performed. Thus Fig. 4.23, for example (while possibly misleading), shows a case in which the transition is not taken unless all of E, E1, and E2 occur simultaneously. When it is taken, both actions A1 and A2 are performed.

A dual kind of arrow can be used to exit a state configuration. Figure 4.24 shows a case in which the system will enter S5 if it was in the configuration (S2, S4) and E occurred. This is a *merge construct*.

If one portion of the transition is missing, the meaning is quite different. Figure 4.25 illustrates this case, in which the and-state is exited, and S5 is entered when E occurs and the system is in S2. The

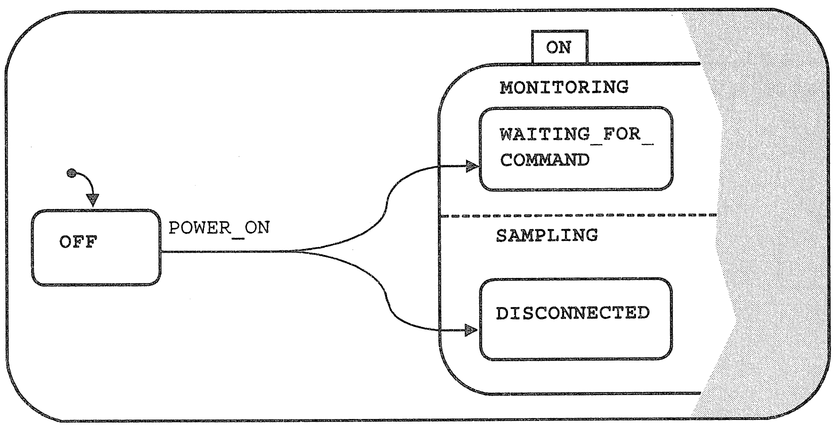


Figure 4.22 A joint connector in a fork construct.

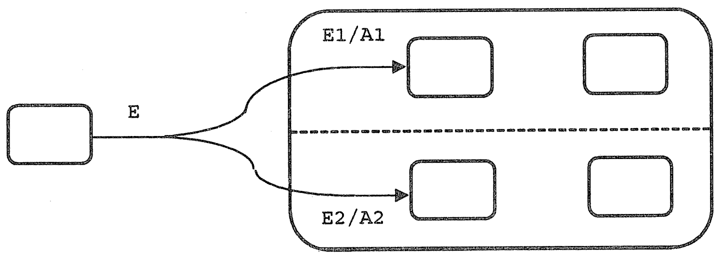


Figure 4.23 Triggers and actions on a fork construct.

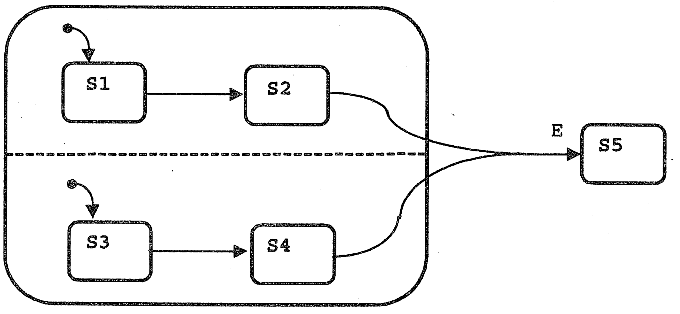


Figure 4.24 A joint connector in a merge construct.

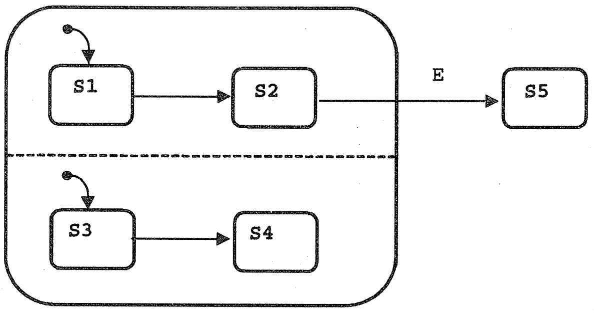


Figure 4.25 A transition from an and-state.

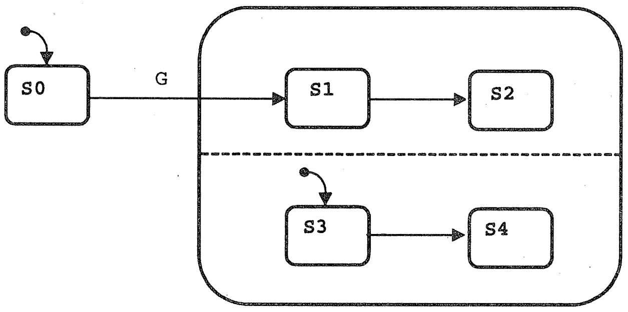


Figure 4.26 A transition into an and-state.

transition is performed independently of which of the substates in the other component the system is in (S3 or S4).

Figure 4.26 shows a transition from S0 that causes entrance to the configuration (S1, S3). The entrance to S1 is by the arrow itself, overriding any default that might exist, and the entrance to S3 is by the default transition.

4.6.2 History entrances

An interesting way to enter a group of states is by the system's history in that group. The simplest kind of this "enter-by-history" feature is to enter the state most recently visited within the group. This is depicted by the special *history connector,* also called an *H-connector.*

Returning once again to our EWS example, consider Fig. 4.27. Here we have decided that once the sensor is connected when we are in state DISCONNECTED, we make a transition to state CONNECTED and enter the inner state that was visited most recently, which will be either IDLE or OPERATING. The arrow leads to an H-connector; thus the mode the EWS reenters is the mode it left when the sensor was disconnected. Notice that the H-connector also has a regular outgoing transition leading to IDLE. This signifies that IDLE is the state to be entered if there is no history (e.g., when the CONNECTED state is entered for the first time).

The history connector specified in Fig. 4.27 indicates an entrance by history on the first level only. If state OPERATING, for example, had substates SLOW and FAST, the history entrance would not extend down to these. In other words, it would not "remember" which of these two substates the system last resided in, and the entrance would be to the one specified as default. To extend a history entrance down to all levels, the H-connector can appear with an asterisk attached, indicating an entrance to the most recently visited state (or configuration) on the lowest level. This is a *deep history connector,* and it is illustrated in Fig. 4.28. If the system was last in OPERATING.FAST, that would be the state entered, despite the fact that SLOW is the internal default.

Once we have history entrances, we must provide the ability to "forget" the history at will. In our example, we may wish to specify that when the HALT event is generated the slate will be cleaned, and the next entrance to OPERATING will be to the default state SLOW, regardless of past behavior. We have special actions for this purpose, which can be used along the appropriate transitions:

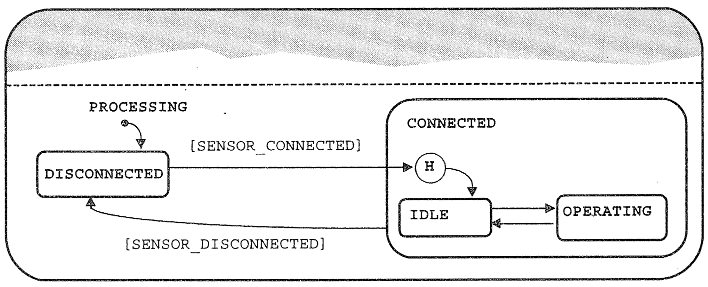


Figure 4.27 A history connector.

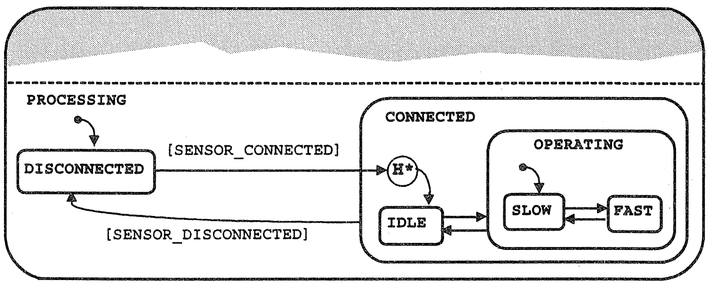


Figure 4.28 A deep history connector.

history_clear(S) and deep_clear(S), abbreviated hc!(S) and dc!(S), respectively. The former causes the system to forget the history information of state S. That is, the next time a history connector or a deep-history connector drawn in state S is entered, the system will behave as if S was entered for the first time. The latter causes the system to forget the history information of all of the descendants of S, to any depth of nesting.