Chapter

6

The Semantics of Statecharts

In the two preceding chapters we described the language of Statecharts and the associated textual expression language. The meaning of the various notational constructs in these languages was discussed on an intuitive level to help the reader grasp how they are used to specify behavior. This chapter defines the semantics of Statecharts more rigorously and addresses some of the delicate issues that arise in working out such a definition. A fuller discussion of the semantics can be found in Harel and Naamad (1996).

Later chapters of the book introduce additional features of our languages, and their behavioral meaning is defined in those places in a way that is consistent with the general principles of the semantics presented here.

6.1 Execution of the Model

A semantic definition of a language for specifying behavior must be sufficiently detailed to give rise to a rigorous prescription of how the model is executed, that is, how it reacts to the inputs arriving from the environment to produce the outputs. Several times we mentioned that a model is executed in steps, and in this chapter we explain what exactly that means. We first present an intuitive view and then get into a more detailed description.

6.1.1 External changes and system reactions

The input to a reactive system consists of a sequence of stimuli—events and changes in the values of data elements—that are generated by the system's environment. We call them *external changes.* The

system senses these changes and may respond by moving from state to state along a transition, by performing some actions, or both.

In general, a model can be viewed as a collection of reactions, which are `trigger/action` pairs. When external changes occur, they may cause some of these triggers to be *enabled,* which causes the corresponding actions to be performed. We have seen two kinds of reaction so far:

- A reaction related to a transition. Its trigger labels the transition, and there are three kinds of implied actions: the transfer from state to state, the actions connected with the exit from and entrance to the appropriate states, and the actions that appear on the transition itself. (Recall that when we talk about transitions, we mean the logical compound transitions; see Sec. 4.5.)
- A static reaction associated with being in a state.

At any given moment, only some of the reactions are *relevant,* depending on the current states of the system. Later we shall also see reactions that are associated with activities by mini-specs. These become relevant when their holding activities become active.

In Fig. 6.1, for example, two transition reactions are relevant in state `S1`, triggered by `E[C1]` and by `E[not C1]`, respectively. The actions that are performed if `E` occurs when the system is in state `S1` and condition `C1` is true, are: `make_false(P1)`, generate `G`, and `make_true(P2)`. (Also, of course, `S1` is exited and `S2` is entered.) Note that the exiting and entering reactions are linked with all respective exiting/entering transitions, as if they were part of their labels. Also, note that because the reaction `E/K` is associated with `S2` and the event `E` "lives" only for an instance, the event `K` is not generated. Similarly, `F/L` is active only in `S2`, and if the event `F` occurs when the system is in `S1`, it will be lost and will have no effect.

We say that the system executes a *step* when it performs all relevant reactions whose triggers are enabled. As a result of a reaction, the sys-

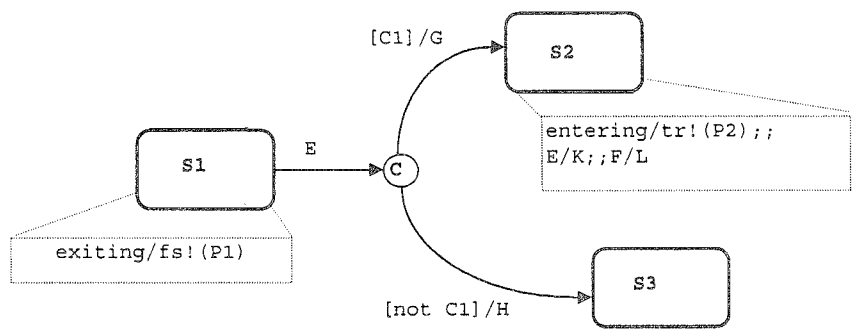


Figure 6.1 A transition reaction.

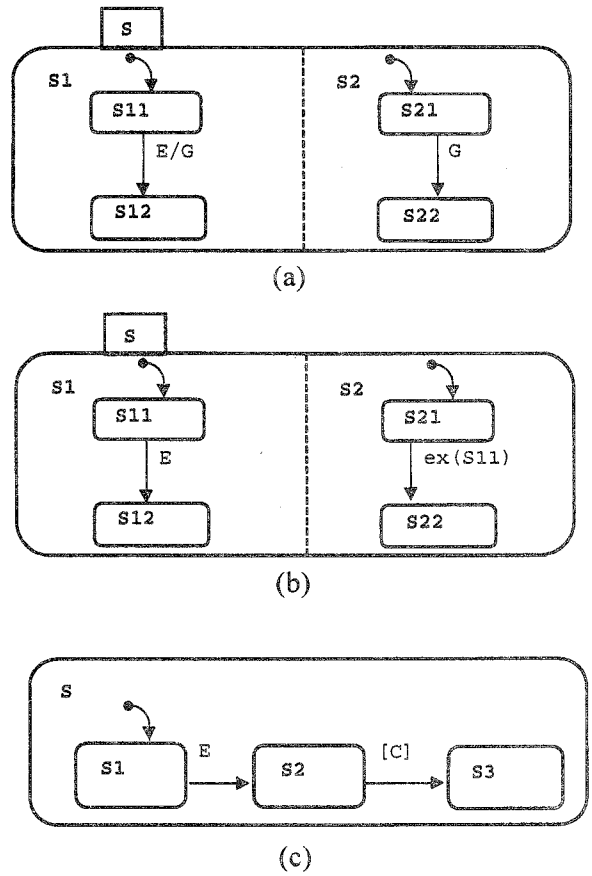


Figure 6.2 Chain reactions.

tem may change its states, generate events, and modify values of internal data elements. In addition, these can cause *derived events* to occur (e.g., `changed(D)`, if the data-item `D` changes value) and conditions to change their value (e.g., `in(S)`, if state `S` is entered). Any of these resulting changes may, in turn, cause other triggers to be enabled and, subsequently, other reactions to be executed in the next step. This has the effect of a *chain reaction,* and some of the generated events and value changes can become outputs of the system. A series of steps representing the system's responses to the sequence of external stimuli and their subsequent internal changes is called an *execution scenario,* or a *run.*

Figure 6.2 illustrates three cases of chain reactions, each consisting of two steps. All start with the system in `S1` when the external event `E` occurs. The first one, Fig. 6.2*a,* shows an event `G` generated by the reaction `E/G` in one state component `S1` and triggering another reaction

(a state transition) immediately thereafter in the orthogonal component S2. In the second case, Fig. 6.2*b*, the subsequent step in the chain takes place, triggered by the derived event ex(S11) indicating an exit from S11. The third case, Fig. 6.2*c*, is a little bit more intricate. The reaction triggered by E causes the system to move to S2, and as a result the transition labeled by [C] becomes relevant. Assuming that the condition C is true during the entire scenario, the following step will take the system to state S3. See also Figs. 4.14 and 4.15, which show chain reactions in the EWS example.

In all three parts of Fig. 6.2, the reactions are performed sequentially because each somehow entails the other. However, more than one reaction can occur simultaneously, as in Fig. 6.3. Being in S11 and in S21 when E occurs results in taking the two transitions at the same time.

Because multiple external changes can occur exactly at the same time, multiple reactions may be enabled and performed in parallel components at the same time, too, even when they depend on different triggers. Moreover, static reactions, even when in the same state, are not exclusive; that is, a number of them can be performed simultaneously. Nevertheless, there are situations when two enabled reactions *are* exclusive and cannot both be taken in the same step. One example involves two transitions exiting from the same state, a situation that is dealt with in the last section of this chapter. Another example is an enabled transition exiting a state and an enabled static reaction associated with the same state. Here, the transition has priority, and it is taken, whereas the static reaction is not.

The parallel nature of our models raises a problem regarding the order in which the actions are performed. Consider Fig. 6.4, in which the preceding example is enhanced with actions along the transitions. When E occurs, both actions are to be performed in the same step. The value of Y after carrying out the assignment Y:=X in this step depends upon whether or not the assignment of 1 to X was performed before. Our semantics resolves this dilemma by postponing

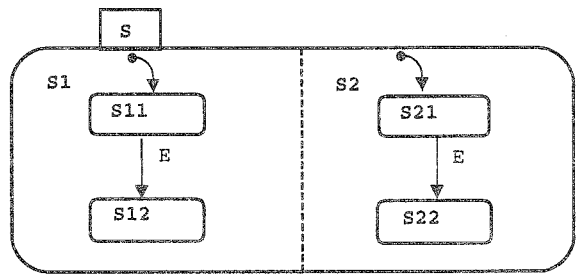


Figure 6.3 Multiple transitions taken simultaneously.

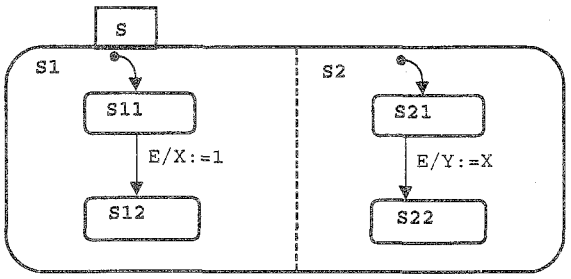


Figure 6.4 Multiple actions performed simultaneously.

the actual value updates until the end of the step, when they are carried out "at once," as we explain shortly. In this way, the evaluation of expressions that are used in actions is based on the "old" values of the variables.

It is important to realize that, by our semantics, different actions in a step are not carried out in any particular order, even when they are specified in a way that appears to prescribe such an order. For example, this includes the three kinds of actions appearing in Fig. 6.1—those associated with exiting a state, those appearing along transitions, and those associated with entering a state. The exceptional behavior of context variables, which are the ones that change their value immediately during the step (see Sec. 5.2.2), does not destroy the true concurrency among different actions performed in the same step. The scope of a context variable is the compound action it is in, and as such, it influences only the sequential evaluations carried out inside that action.

In summary, all calculations taking place in a step—both those that evaluate the triggers and determine the reactions that will be taken and those that affect the results of the actions—are based on what we call the *status* of the system prior to the step execution. The status includes the states the system is in, the values of variables at the beginning of the step, the events that were generated in the preceding step and since then, and some information about the past that we will discuss later.

Thus an execution scenario consists of a sequence of statuses, starting with the initial (default) one, separated by steps that transfer the system from one status to another, in response to external stimuli, to the actions generated in the preceding step, or both. See Fig. 6.5.

6.1.2 The details of status and step

In this section, we describe the contents of the system status and the algorithm for executing a step. Note that this description does not

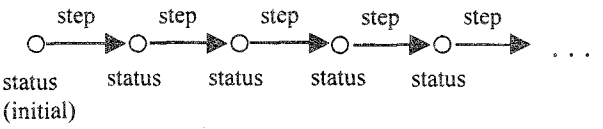


Figure 6.5 An execution scenario.

cover the behavioral aspects related to activities, although where the additional information is straightforward and does not complicate the description, we include it. This additional information will be given in Chaps. 7 and 8.

The *status* includes:

- A list of states in which the system currently resides.
- A list of activities that are currently active.
- Current values of conditions and data-items.
- A list of regular and derived events that were generated internally in the preceding step.
- A list of timeout events and their time for occurrence.
- A list of scheduled actions and their time for execution.
- Relevant information on the history of states.

The *input* to the algorithm for executing a step consists of:

- The current system status.
- A set of external changes (events and changes in the values of conditions and data-items) that occurred since the last step.
- The current time (see the discussion of time in Sec. 6.2).

The *step* execution algorithm works in three main phases:

1. First phase:
   - Calculate the events derived from the external changes and add them to the list of events (e.g., if a false condition `C` is set to be true, the event `tr(C)` is added to the list).
   - Perform the scheduled actions whose scheduled time has been exceeded, and calculate their derived events.
   - Update the occurrence time of timeout events if their triggering events have occurred.
   - Generate the timeout events whose occurrence time has been exceeded.

The first phase may modify the input status, and the new status is the one used in the following phases.

2. Second phase:
   - Evaluate the triggers of all relevant transition reactions to compute the enabled transitions that will be taken in this step (see following for how conflicts are dealt with).
   - Prepare a list of all states that will be exited and entered. This may involve the use of default entrances and history information. Note that the lists may contain nonbasic states.
   - Evaluate the triggers of all relevant static reactions to compute the ones that are enabled. Static reactions in states that are exited in this step are not included here.

The second phase ends with a list of actions to be performed in the current step. Actions specifying the exit from and entrance to states are included.

3. Third phase:
   - Update the information on the history of states.
   - Carry out all computations prescribed by the actions in the list produced in the second phase but without event generation or the value updates called for by the assignments to data-items and conditions (except for context variables, which are assigned their new values as the relevant actions are carried out).
   - Add scheduled actions from the list produced in the second phase to the list of scheduled actions.
   - Carry out all updates called for by the actions on the list produced in the second phase. This includes actually making the value assignments to data-items and conditions, and updating the list of events (i.e., removing all current events and adding the newly generated ones).
   - Update the list of current states.

The second phase can end with no enabled reactions. If this occurs, we say that the system has reached a *stationary status,* and the third phase is not performed at all. In such cases, execution will remain suspended until new external changes occur or time is advanced.

6.2 Handling Time

In reactive systems, as opposed to transformational systems, the notion of sequentiality and its relationship with time is of central importance. We now discuss this issue.

6.2.1 Sequentiality and time issues

We saw earlier that an execution scenario consists of steps triggered by external changes and the advancement of time. We also saw that

reactions triggered by such happenings may continue to generate a chain of steps caused by internal changes. This raises the following questions:

- Can external changes interleave with internal chain reactions or are the former sensed by the system only after all the internal happenings end?
- When do external changes stop being accumulated to make place for the execution of a step?

These questions deal only with the order in which things occur during execution and do not get into detailed issues involving the quantitative nature of elapsed time. They are relevant to all kinds of models. On the other hand, quantitative issues cannot be ignored when the model contains timeout events and scheduled actions, because time quantification appears within them explicitly and the current time must be used to determine whether these elements affect a particular step. When such elements are present in a model, we may also ask who causes time to progress during execution and how does this occur?

The time calculated in dealing with the explicit time expressions appearing in timeout events and scheduled actions is measured in terms of some abstract time unit common to an entire statechart. Different statecharts can have different time units, in which case the relation between them must be specified prior to model execution. When the model runs in a real environment or participates in a simulation in which concrete time units, such as seconds and minutes, are meaningful, the relationship between the model's time units and the real clock must be provided.

6.2.2 Time schemes

We now propose two time schemes and show how each of them addresses these questions. In both schemes we assume that time does not advance during the step execution itself, which can be viewed as taking zero time. The actual meaning of this assumption is that no external changes occur throughout the step, and the time information needed for any timeout events and scheduled actions in a step is computed using a common clock value.

The *synchronous time scheme* assumes that the system executes a single step every time unit. This time scheme is particularly fitting for modeling electronic digital systems, in which the execution is synchronized with clock signals and external changes can occur between any two steps. The execution proceeds in cycles, in each of which time is incremented by one time unit, all external changes that occurred since the last step are collected, and a step is executed. When different clocks are assumed for the various components of the model, time is advanced to the nearest next clock value and only the relevant components perform a step.

The *asynchronous time scheme* is more flexible regarding the advancement of time, and it allows several steps to take place within a single point in time. In general, external changes can occur at any moment between steps, and several such changes can occur simultaneously. Actually, any implementation of this scheme can choose how it deals with these possibilities. An execution pattern that fits many real systems responds to external changes when they occur by executing the sequence of all steps these changes entail, as in a chain reaction, until it reaches a stationary, stable status. Only then is the system ready to react to further external changes. Such a series of steps, initiated by external changes and proceeding until reaching a stable status, is called a *super-step*, and when adopting this execution pattern, time does not advance inside a super-step.

6.3 Nondeterministic Situations

This section discusses the nondeterministic situations that a model might run into during execution.

6.3.1 Multiple enabled transitions

Consider the simple statechart of Fig. 6.6. When the system is in `S1`, there are two relevant outgoing transitions. If `E` occurs and both `C1` and `C2` are true, the system does not know which transition to take, and *nondeterminism* occurs.

Such a situation occurs when several transitions that cannot be taken simultaneously are enabled, and no added criterion has been given for selecting only one. Tools executing the model can make an arbitrary decision in these situations or can ask the user to decide.

Now consider Fig. 6.7, which shows a portion of the main statechart of the EWS example. Assume that we are in the `COMPARING` state, which is one of the substates of `ON`. If the event `POWER_OFF` occurs at the same time as `OUT_OF_RANGE`, two conflicting transitions will be enabled. However, in this case, a nondeterministic situation will *not* occur, because the higher-level transition (i.e., the one from `ON` to `OFF`) has *priority* over the internal transition. The

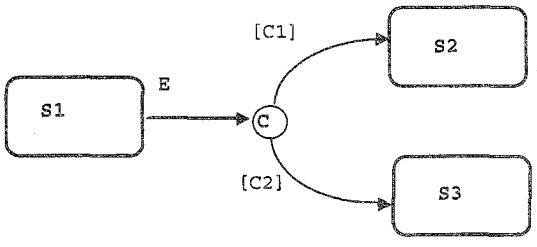


Figure 6.6 Potential nondeterminism.

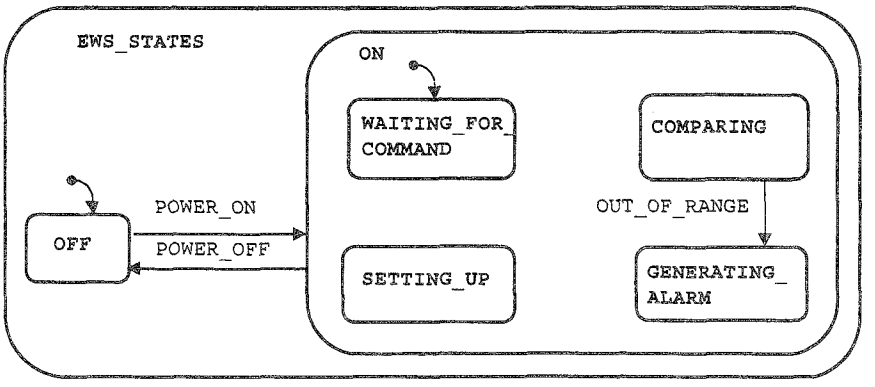


Figure 6.7 Priorities on transitions.

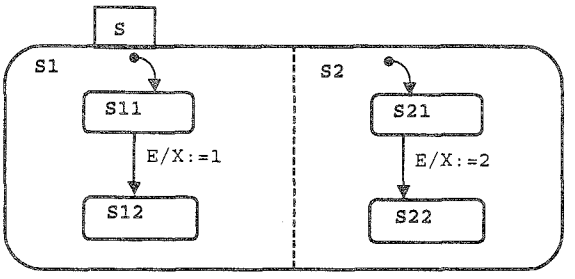


Figure 6.8 Write-write racing situation.

criterion for priority of transitions prefers the transition whose source and target have a higher common ancestor state, if possible. If the common ancestors of both transitions are identical, then nondeterminism indeed occurs.

6.3.2 Racing

We say that a *racing situation* occurs if during execution an element is modified more than once or is both modified and used at a single point in time. Situations like this usually indicate some problem in the preparation of the model.

Figure 6.4 showed a case where data-item `X` is both assigned a value and used in the same step. It is an example of what we call a *read-write racing situation.* Figure 6.8, in contrast, presents a *write-write racing situation.* According to the definition of a step presented earlier, it is clear that multiple assignments to a data-item or a condition in a single step are meaningless, because the values are updated only at the end of the step. In a write-write racing situation, the element will be assigned one of the values arbitrarily.

More information on racing situations can be found in Harel and Naamad (1996).