Chapter

5 The Textual Expression Language

This chapter describes the textual expression language that appears in several places in our models. This language is used to define the triggers of transitions and the implied actions in statecharts. It is also used to describe static reactions that can be attached to a state (which are also discussed in this chapter), as well as the mini-specs of activities and combinational assignments (which are discussed in Chap. 7).

The textual language supports data manipulation using arithmetical and logical operations; it allows sensing the status of other elements, handling timing issues, and many types of actions. Its syntax and semantics are somewhat similar to procedural programming languages, although there are some important differences that relate mainly to the stepwise execution of a model, as clarified in Chap. 6.

A complete description of the textual expression language is presented in App. A.

5.1 Event, Condition, and Data-Item Expressions

In the preceding chapters, events and conditions were used in triggers, and data-items in interface definitions. We also saw some examples that used more complicated expressions, and not just element names. There were dependencies of expressions on other elements in the system (e.g., in the condition expression `in(S)`) and compound expressions that involved several elements (e.g., the event `E1 or E2`). We now describe in more detail how to construct such expressions for events, conditions, and data-items.

5.1.1 Event expressions

Most of the transition triggers shown in Chap. 4 consisted of just an event name. However, a trigger can be any event expression, as described here. Figure 4.15 showed events that occur upon entering or exiting a state, `en(S)` and `ex(S)`. Other events indicate changes in the status of other elements, such as changes in the values of conditions and data-items, and in the status of activities. These will be discussed when the manipulation of the relevant elements is presented.

Expressions for *compound events* can be constructed by using the Boolean operations `and`, `or`, and `not`. In the EWS example, the two transitions from `GENERATING_ALARM` to `WAITING_FOR_COMMAND` in Fig. 4.3 may be combined, using an event disjunction, as in Fig. 5.1. The transition labeled with the event disjunction is taken when at least one of the events occurs.

The negation of an event using the *not* operation must be approached with caution. This negation means that the specified event did *not* occur, and it makes sense only when the negated event is checked at a specific point in time, that is, when combined with other events. This is achieved by using an *and* operation or a compound transition. Thus, for example, if the event `E` has been defined as `E1 or E2 or E3 or E4 or E5` (using the Data Dictionary, as explained in the next section), then we may use either Fig. 5.2*a* or Fig. 5.2*b* instead of Fig. 5.2*c*. Recall that the junction connectors used in this figure denote the conjunction of triggers.

Note also that the combination of an event and a condition, `E[C]` (even if the event is absent, as in the trigger `[C]`), is considered an event, so that `E1[C1] and [C2] or E2`, for example, is an event expression, too.

Event expressions are evaluated according to the conventional precedence rules of logical operations, and parentheses can be used in the usual way to override the default orderings. See App. A for detailed information about precedence of operations.

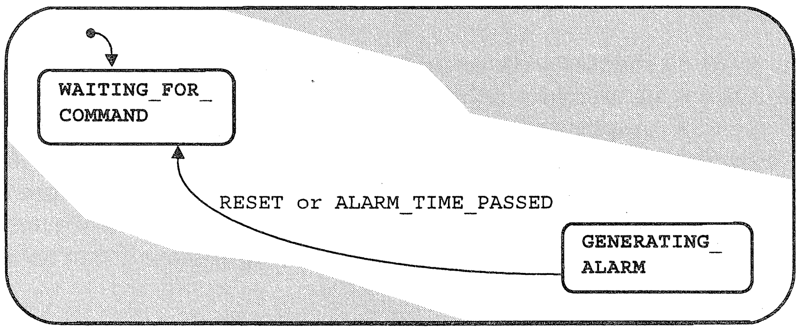


Figure 5.1 Disjunction of events.

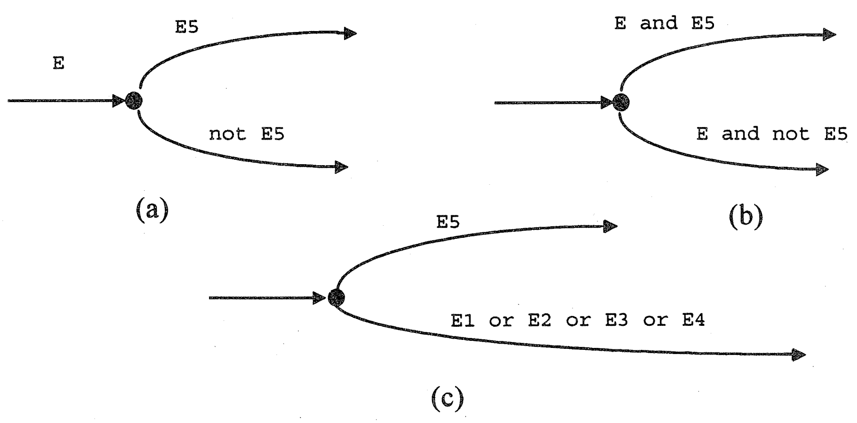


Figure 5.2 Negating an event.

All the aforementioned event expressions evaluate to yield a single event (as opposed to an event array). A component of an event array can be used whenever an event expression is allowed, and it is identified by its index, for example, `DIGIT_PRESSED(1)`.

Very often we want to detect the fact that some unspecified component of an event array has occurred. We use the operator `any` for this. For example, the event expression `any(DIGIT_PRESSED)` refers to the event array defined in Fig. 3.1, and captures the pressing of any one (or more) of the ten digit keys on the operator keyboard of the EWS. Similarly, although rarely used, the `all` operator captures the simultaneous occurrence of all events in the array.

5.1.2 Condition expressions

Some of the transition labels presented in Chap. 4 include condition expressions. In the simplest case, the condition expression is just the condition name, such as `SIGNAL_EXISTS` in Fig. 4.4, but we also saw the condition expression `in(CONNECTED)` used in Fig. 4.15. Other condition expressions are also related to the status of other kinds of elements, and they will be presented as we go along.

We often want a condition to compare data-items in one of several ways. To do so, we allow the following comparison conditions, where # depicts inequality:

```
exp1 = exp2,        exp1 # exp2,
exp1 > exp2,        exp1 < exp2,
exp1 <= exp2,       exp1 >= exp2
```

Assume that we have chosen to represent the operator command of the EWS by a string data-item `COMMAND` that has three possible values, namely, `` `execute` ``, `` `set-up` `` or `` `reset` ``. The exit from the

WAITING_FOR_COMMAND state would be triggered by the event COMMAND_ENTERED, denoting the assignment of a value to this data-item, and would be channeled to the appropriate state, depending on that value. See Fig. 5.3.

The expressions on both sides of the comparisons must be of the same type, both numeric or both strings. They can also be arrays or records and are then compared component-wise. Arrays must be of the same length and component type, and records can be compared only if they are of the same user-defined type. For strings we allow only = and #.

As in the case of events, the Boolean operations and, or, and not can be used to construct *compound conditions.* Figure 5.4 shows two alternative ways to restrict the transition from WAITING_FOR_COMMAND to COMPARING by using the conditions SET_UP_DONE and in(CONNECTED).

Because conditions can be organized in arrays or constitute fields in a record, the condition expression can have the corresponding syntax. For example, if the EWS monitors an array of sensors and SENSORS_CONNECTED is the array of conditions representing their connection status, then SENSORS_CONNECTED(I) is a legal condition expression specifying the status of the Ith component. To capture the condition of at least one of the sensors being connected or all of them, we may use the any and all operators, respectively, as in any(SENSORS_CONNECTED). These operators can also be used to refer to a slice of the array, as in all(SENSORS_CONNECTED(1..3)), which is true when the three first sensors are connected.

5.1.3 Data-item expressions

We mentioned conditions that compare data-item expressions. Data-item expressions can also be used in other places in the textual language, such

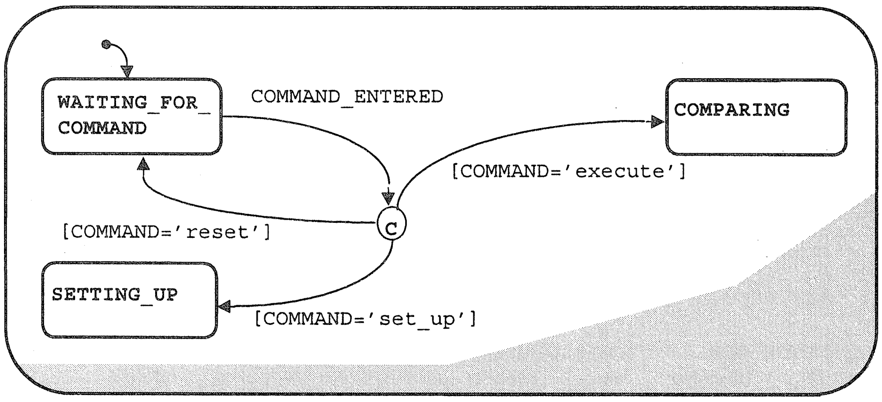


Figure 5.3 Comparison conditions.

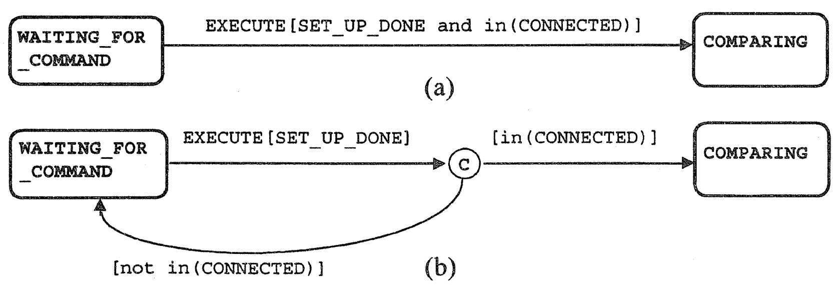


Figure 5.4 Condition expressions.

as in assignment actions, and can be of different types: numeric (integer, real, bit, and bit-array), strings, and structured.

Numeric expressions consist of constants and numeric data-items (or numeric components of structured data-items), combined by conventional arithmetic and bit-wise operations with the usual precedence rules. An example is `Y+3*R.X-A(I+J)`. There is also a set of predefined functions that can be used within numeric expressions, such as arithmetic and trigonometric functions (e.g., `abs(X)`, `sin(A)`), bit-array operations (e.g., the logical shift-left operation `lshl(B)`), and random number generators (e.g., `rand_normal(R,S)`). See App. A.

In addition, in numeric expressions we allow the use of functions that are not predefined. These are called *user functions,* and may employ data-item and condition parameters that come from the model. These functions usually denote parts of the system whose details are not currently essential. They may remain unspecified, and eventually could be taken from an existing implementation.

Numeric expressions can involve the various numeric types, and type conversion is carried out as needed. Integer and bit-array constants can use bases other than the decimal base, such as binary (e.g., `0B00101011`), octal (e.g., `0O053`), and hexadecimal (e.g., `0X2B`). Real constants can be represented in exponential format (e.g., `2.5e-3`).

As mentioned earlier, string constants are enclosed in quotes, (e.g., `` `abc` ``). There are no operations on strings, but the language offers several functions for string manipulation, such as concatenation, substring search in another string, and conversion between integer and string. See App. A.

Structured data-items (i.e., arrays, records, and unions) do not support operations, either. There is a special representation for array constants that uses commas between the components. An asterisk for repetitions is also allowed. For example, `{20*0}` is an array constant consisting of 20 zeros, while `{1,2,3,10*1,0,0}` is an array constant consisting of 15 integer components. However, the language provides no record or union constants.

Appendix A describes the full set of operations and functions that can be applied to data-items and their relative precedence.

5.1.4 Named expressions

We mentioned earlier that an element expression can be abbreviated by a simple element name. This is carried out by associating a *definition* with an element in the Data Dictionary, and here are the most common reasons for doing so:

- To shorten a lengthy expression that appears many times in transitions or in other places where the textual language is used. A short definition in the Data Dictionary prevents errors of inconsistency, enhances clarity, and economizes in writing. In the EWS example, we can define the condition `READY` to be `SET_UP_DONE and in(CONNECTED)`. This will shorten the trigger on the transition in Fig. 5.4*a*, yielding `EXECUTE[READY]`.
- To abstract away the expression, hiding details that we might not have decided upon yet or might want to change later on. In the EWS example, we can define a data-item `ALARM_DURATION` whose value will be specified later. In this example, the reason could be our desire to be able to change the duration in a flexible way. Also, the exact time is not really important in an early stage of the specification.

Such an abbreviating definition can be associated with any event, condition, or data-item. An element with no definition is called a *primitive element,* or a *variable,* and can be generated (in case of an event) or modified (in case of a condition or a data-item) in the model in the usual way. An element that has an expression definition is called a *compound element* (see Fig. 5.5*a*). The element is referred to as a compound element even when the expression is just the name of some other element. For example, the event `E` is defined to occur when event `G` occurs. An element, a data-item, or condition, is referred to as a *constant* when its definition is a literal constant expression (see Fig. 5.5*b*).

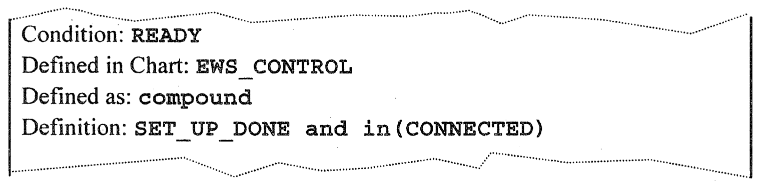
Because compound elements or constants depend for their values on their associated expressions, they cannot be affected directly by actions. For example, such an element cannot appear on the left-hand side of an assignment action, cannot label a flow-line, and cannot be a component of an information-flow.

These limitations do not apply to the special case of attaching an *alias* to a bit-array slice, which can be useful in applications such as digital chip design and communication protocol specification. In such applications an individual bit or a slice of a bit-array might carry special meaning, and it helps to be able to refer to the bit-array portion by a special name. For example, a message can be composed of a series of bits divided into groups that denote the message type, command code, data fields, etc. Assume that MSG is a message that is implemented by a bit-array of 64 bits, indexed from 0 to 63. The first three bits, MSG(0..2), denote the message type. An integer data-item MSG_TYPE will be defined to be an alias of MSG(0..2) (see Fig. 5.5*c*). Now the message sender can assign a value to MSG_TYPE, which is just like assigning a value to MSG(0..2), and the message reader can check the value of MSG_TYPE in the decoding process. A lower level of the communication protocol that handles these messages can be made to access the individual bits, with no extra conversion to another data structure.

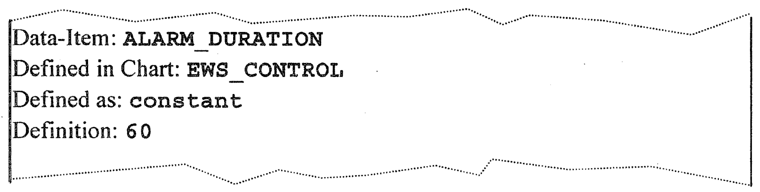
It is possible to define a data-item of type integer, bit, or bit-array to be an alias of an expression for a bit-array slice with a constant range of indices. As mentioned, an alias is treated like a variable and can be used wherever a variable is allowed, unlike compound and constant data-items.

Any occurrence of the element that has an expression definition can be viewed as if the expression were written out in full. Moreover, the expression is reevaluated whenever there is need to evaluate the element.

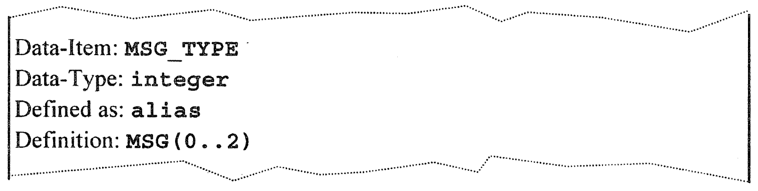
It is also possible to define named actions, as discussed in the next section.


Condition: **READY**
Defined in Chart: **EWS_CONTROL**
Defined as: **compound**
Definition: **SET_UP_DONE and in(CONNECTED)**

(a)


Data-Item: **ALARM_DURATION**
Defined in Chart: **EWS_CONTROL**
Defined as: **constant**
Definition: **60**

(b)


Data-Item: **MSG_TYPE**
Data-Type: **integer**
Defined as: **alias**
Definition: **MSG(0..2)**

(c)

Figure 5.5 Elements with definitions in the Data Dictionary: (*a*) compound, (*b*) constant, and (*c*) alias elements.

5.2 Actions

In addition to the transitions between states, other things may happen during execution of the model. They are usually specified by the *actions.* We saw some examples of actions written along transitions. In addition, actions can appear in *static reactions,* as described later in this chapter, and in *mini-specs* of activities, as described in Chap. 7.

The textual language allows various types of actions, which are described in the following sections. They can be classified as follows:

- Basic actions that manipulate elements, causing changes that can be checked and triggering other happenings in the system.
- Conditional and iterative actions, similar in structure to those in conventional programming languages.

5.2.1 Element manipulation

The most basic actions manipulate three types of elements: events, conditions, and data-items.

Event manipulation is really just sending the event. This is performed by the action that is simply the name of the event. We saw examples of actions that send events in Fig. 4.14: the events OPERATE and HALT are sent when the transitions to and from the COMPARING state are taken, respectively.

Condition manipulation is a little more flexible. Special actions can cause a condition to become true or false. In our EWS example, we may want to distinguish between success and failure of the setting-up procedure to ensure that we start comparing values in the COMPARING state only if the set-up succeeded. This may be achieved as follows. In Fig. 5.4, we added theguarding condition SET_UP_DONE to the transition from WAITING_FOR_COMMAND to COMPARING. Now, in Fig. 5.6, we add the two self-explanatory events SET_UP_SUCCEEDED and SET_UP_FAILED, which label two separate exits from the SETTING_UP state. In the case

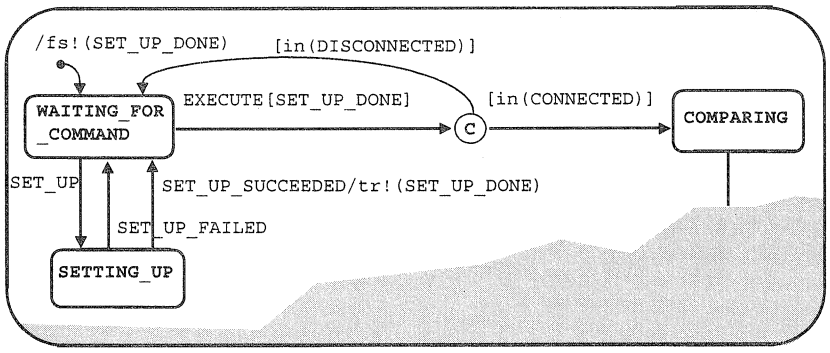


Figure 5.6 Actions on conditions.

of success—and in that case only—we carry out the action make_true (SET_UP_DONE) (abbreviated tr!(SET_UP_DONE)).

In general, the action tr!(C) has the effect of setting the truth value of condition C to true, and the corresponding action make_false(C) (abbreviated fs!(C)) sets it to false. The default entrance to WAITING_FOR_COMMAND, for example, is labeled with a make_false action that assigns a false value to SET_UP_DONE. So the system will react to the EXECUTE command only if the setting-up procedure ended successfully at least once.

Instead of the actions tr!(C) and fs!(C) we may use the assignment actions C:=true and C:=false, respectively. In general, the right-hand side of such a condition assignment can be any condition expression.

In addition to these actions, a condition C has two associated events, true(C) and false(C), which occur precisely when C changes from false to true and from true to false, respectively. We abbreviate them as tr(C) and fs(C). The condition C can be a condition expression that depends on other conditions or data-items. Interestingly, this makes it possible to replace events by conditions. In Fig. 4.10, for example, instead of the two events POWER_ON and POWER_OFF, we could have a single condition, POWER_IS_ON, and use the two events true(POWER_IS_ON) and false(POWER_IS_ON).

A subtle point concerns the precise relationship between the actions tr!(C) and fs!(C) and the events tr(C) and fs(C). For example, does tr(C) always occur when tr!(C) is executed? The answer is no. The events occur only when the truth value of C changes value, but the actions can be executed without changing the truth value if it was the desired one to start with. Thus, for example, if the setting-up procedure completed successfully twice in succession, then the first execution of the action tr!(SET_UP_DONE) will trigger the event tr(SET_UP_DONE), but the second execution will not.

Assignment actions can also be used to manipulate data-items, and as in the case of conditions, there are events and conditions associated with them. In the EWS example, we may be interested in producing an alarm only after three occurrences of OUT_OF_RANGE. This may be achieved as in Fig. 5.7.

All types of data-items can be involved in assignments. The right-hand-side expression of the assignment must be type consistent with the assigned data-item on the left-hand side. Both sides must be either numeric or string. They can also be arrays, in which case their lengths must be the same and the component types must be consistent. Assignments of an entire structured data-item (record or union) are also allowed, but both sides must be of exactly the same user-defined type.

Whenever an assignment to X takes place, the event written(X) (abbreviated wr(X)) occurs. Thus, we may replace the trigger COMMAND_ENTERED in Fig. 5.3 with the event wr(COMMAND). The exit

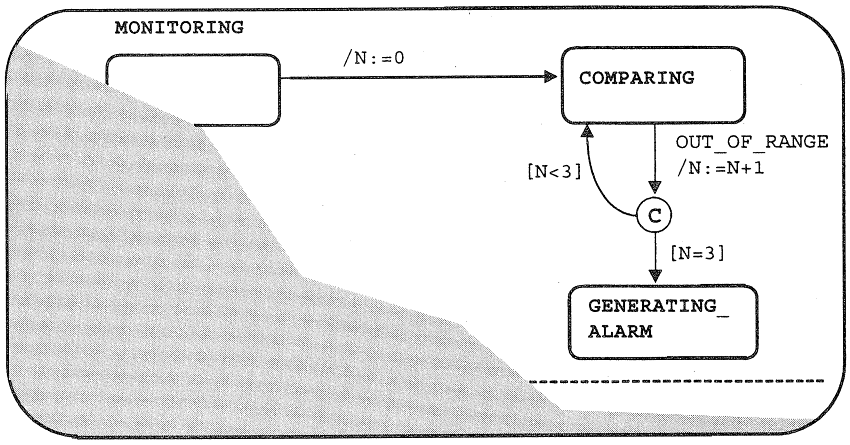


Figure 5.7 Actions and conditions on data-items.

from the WAITING_FOR_COMMAND state would be triggered by an (external) assignment to COMMAND, and would be channeled to the appropriate state, depending on its value. See Fig. 5.8.

A similar event is changed(X) (abbreviated ch(X)), which occurs when and if there was a change in the value of the data-item expression X. Thus, in our example, we cannot replace the event wr(COMMAND) by ch(COMMAND) (as a trigger of the transition from WAITING_FOR_COMMAND), because that would make it impossible to carry out two successive entries to SETTING_UP.

We may also use the actions write_data(X) and read_data(X) (abbreviated wr!(X) and rd!(X), respectively). These actions apply to all types of data-items, including ones that are structured, and even to conditions. These actions cause the occurrence of the events written(X) and read(X), respectively. They will be discussed further in Chap. 8.

Note that we do not allow actions to be carried out on named compound elements. It makes no sense to perform the action tr!(C) when C is defined as C1 or C2, or similarly to assign a value directly to X1+X2. (Of course, these changes can be achieved by operating on the components, i.e., by changing the values of C1,C2,X1, or X2.)

5.2.2 Compound actions and context variables

We already mentioned that it is possible to perform more than one action when a transition is taken. This compound sequential action is written by separating the component actions by a semicolon (e.g., A1;A2;A3).

Another kind of compound action is the *conditional action,* in which the actual action carried out depends on a condition or an event. The two cases differ in their format:

```
if C then A else B end if
when E then A else B end when
```

where A and B are actions, C is a condition expression and E is an event expression. The meaning of these is self-explanatory. In both cases the else B part is optional.

For example, in the EWS we may define an event SET_UP_COMPLETED to be the disjunction SET_UP_SUCCEEDED or SET_UP_FAILED. Figure 5.9 may then be used to specify the transitions from SETTING_UP to WAITING_FOR_COMMAND concisely, and it should be compared with Fig. 5.6.

Actions can be lengthy sequences of compound actions, and may involve complex expressions. It is thus helpful to attach a name to an

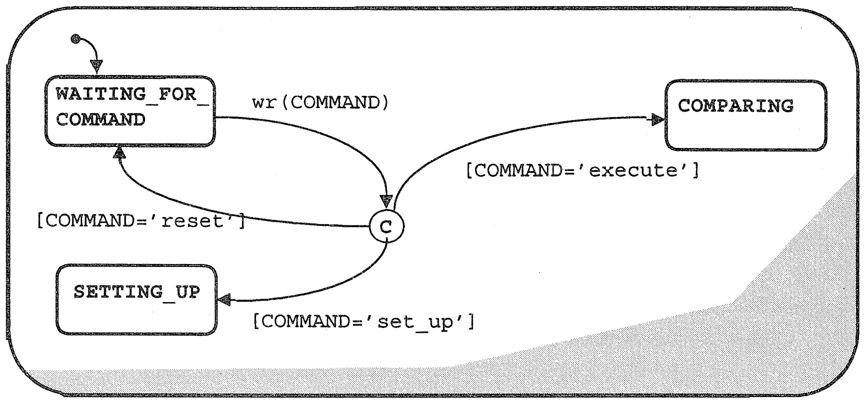


Figure 5.8 An event related to a data-item.

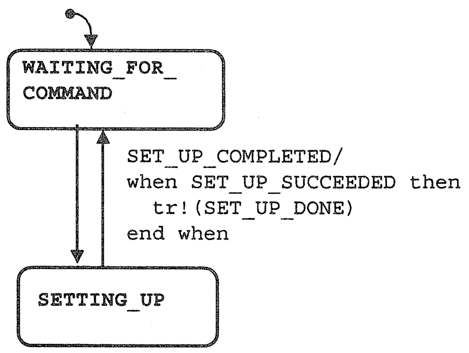


Figure 5.9 A compound action.

action, using the Data Dictionary. For example, the conditional action in Fig. 5.9 can be named `SET_SUCCESS`, shortening the transition label to `SET_UP_COMPLETED/SET_SUCCESS`.

When a sequence of actions involves assignments, the timing in which the left-hand-side variable gets its new value is significant. As explained in Chap. 6, the model is executed in steps, and the actual assignment is performed only at the end of the step using the values from the end of the preceding step. Therefore, an action like `X:=1;Y:=X` will result in `Y` becoming equal to the value that `X` had *before* the action execution, which may not necessarily be 1. Moreover, if we check the value of a variable `X` in a conditional action that follows an assignment to `X`, the value used will be the one from the preceding step. For example, in the action sequence `X:=Y; if X=Y then A1 else A2 end if`, the action `A1` is *not* necessarily carried out because `X` and `Y` might have had different values before the action.

This method of computation is sometimes inconvenient, especially when true sequentiality is required. For this purpose we provide *context variables,* identified by a prefixed `$`. In contrast to regular data-items and conditions, context variables get their values immediately, so that `$X:=1; Y:=$X` results in `Y` being equal to 1, and `$X:=Y; if $X=Y then A1 else A2 end if` causes `A1` to be performed in any case.

Context variables have limited scope. They are recognized only within the action expression in which they appear, and their value is not saved between different invocations of the same action. Thus context variables used in the definition of a named action `A` are not recognized in an action instantiating `A`, and vice versa. Also, actions that appear in labels of different transition segments connected by a connector do not share the context variables, even when they are performed in the same step. Context variables have no entry in the Data Dictionary; thus they inherit their type from the expression first assigned to them.

5.2.3 Iterative actions

We have seen how to define arrays of events, conditions, and data-items. To help manipulate these arrays we provide iterative actions. In particular, the *for loop* action makes it possible to access the individual array components in successive order. The `for loop` action has the following syntax:

```
for $I in N1 to N2 loop
     A
end loop
```

Here, `$I` is a context variable, `N1` and `N2` are integer expressions, and `A` is an action. For example, assume that there is an array of sensors monitored by the EWS. For each sensor `I`, there is a corresponding `SAMPLE(I)` whose value is checked for being in the desired range, producing an array of self-explanatory IN_RANGE conditions. This can be done as follows:

```
for $I in 1 to NUMBER_OF_SENSORS loop
    IN_RANGE($I):=(SAMPLE($I) => LEGAL_RANGE.LOW_LIMIT)
                and (SAMPLE($I) =>LEGAL_RANGE.HIGH_LIMIT)
end loop
```

The iterations can be carried out with the context variable repeatedly decremented, by using the keyword downto instead of to in the range designation.

Assume now that instead of producing the IN_RANGE values for all the sensors, it suffices to identify one sensor for which the value is out of legal range. When this happens, the OUT_OF_RANGE event should be produced. This can be done by the following for loop action:

```
for $I in 1 to NUMBER_OF_SENSORS loop
    if ((SAMPLE($I) < LEGAL_RANGE.LOW_LIMIT) or
        (SAMPLE($I) > LEGAL_RANGE.HIGH_LIMIT)) then
        OUT_OF_RANGE;
        break
    end if
end loop
```

The action break, which is performed when an out of range situation is detected, will skip the rest of the loop's iterations, and the action that follows the loop construct (if there is such an action) will be the next one to execute.

Another iterative action, the *while loop* construct, iterates until some condition becomes false. The preceding operation for the sensors can be implemented with this construct as follows:

```
$I:=1;
$ALL_IN_RANGE:=true;
while
        (($I =< NUMBER_OF_SENSORS) and $ALL_IN_RANGE) loop
    if ((SAMPLE($I) < LEGAL_RANGE.LOW_LIMIT) or
            (SAMPLE($I) > LEGAL_RANGE.HIGH_LIMIT)) then
        OUT_OF_RANGE;
        $ALL_IN_RANGE:=false
    end if;
    $I := $I+1
end loop
```

The break action can also be used in the while loop to jump out of the loop without completing the iteration.

Notice that the iteration counter in the for loop action and the iteration condition in the while loop involve context variables. The reason is that the values of these expressions must change during the execution of the action, that is, within the same step.

Iterative actions can be used wherever any other action can be written, particularly inside another iterative action. No limit is set on the level of nesting of iterations.

5.3 Time-Related Expressions

Many kinds of reactive systems have timing restrictions, and their behavioral specification must involve reference to time delays and timed-out events. Our textual language provides several constructs to deal with timing.

5.3.1 Timeout events

One way to introduce explicit timing information into a statechart is by using the *timeout event*. The general form is `timeout(E,T)` (abbreviated as `tm(E,T)`), where `E` is an event and `T` is an integer expression. This expression defines a new event, which will occur `T` time units after the latest occurrence of the event `E`. In the EWS example, we may replace the event `ALARM_TIME_PASSED` of Fig. 4.3 by the more informative and detailed event: `tm(en(GENERATING_ALARM),ALARM_DURATION)`. The new event will occur `ALARM_DURATION` time units after the state `GENERATING_ALARM` is entered. The waiting time, `ALARM_DURATION`, is measured in some abstract time unit. The way these units refer to concrete time units, such as seconds or minutes, is not part of the language and may be specified informally in the Data Dictionary. In addition, the relationship can be fixed in related tools, such as simulators, where concrete units are meaningful. In any case, the same abstract time units are used in all timing expressions throughout the entire statechart.

A subtle point related to the `timeout(E,T)` event is that the clock that "counts" the time from the occurrence of `E` is reset to zero each time `E` occurs. Thus if less than `ALARM_DURATION` time units elapsed since the system entered the `GENERATING_ALARM` state, and in the meantime that state was left and reentered, thus retriggering the event `en(GENERATING_ALARM)`, the counting of `ALARM_DURATION` will restart and the alarm will last until this new duration ends.

5.3.2 Scheduled actions

A construct that is in a way dual to the timeout event is the *scheduled action*. The general format is `schedule(G,T)` (abbreviated as `sc!(G,T)`), where `G` is an action and `T` is an integer expression. It schedules `G` to be performed `T` time units from the present instant. Referring to Fig. 4.7 of the EWS example, we can define the action that should be taken if `NO_SIGNAL` is true to be `sc!(if NO_SIGNAL then ISSUE_DISCONNECTED_MSG,3)`. This will cause the system to wait for 3 time units and then check whether there is still no signal before issuing the message.

It is interesting to compare two ways of specifying that `G` is to occur `T` time units from a present occurrence of the event `E`. If we do this by using `E/sc!(G,T)`, then indeed nothing can prevent `G` from being carried out on time. In contrast, if we use `tm(E,T)/G`, then, as mentioned earlier, a second occurrence of `E` before `T` units elapse resets the clock to zero, and `G` might take longer to occur or might never get around to doing so.

5.4 Static Reactions

5.4.1 Reactions on entering and exiting a state

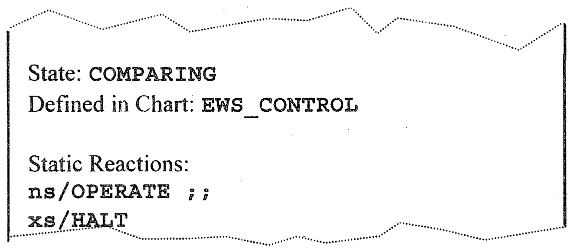
We are often interested in associating actions with the event of entering or exiting a particular state. This may be done by adding the required actions to all entering or exiting transitions. A better way, especially when there are many such transitions, is to associate corresponding reactions with the state in the Data Dictionary. These reactions are triggered by `entering` and `exiting` events (abbreviated by `ns` and `xs`).

To use the EWS as an example, refer to Fig. 4.14. The event `OPERATE` is generated on all transitions entering the `COMPARING` state, and the event `HALT` is generated on the exiting transitions thereof. We may instead omit these actions from the chart and associate two reactions with the `COMPARING` state in the Data Dictionary (separated by a double semicolon), as shown in Fig. 5.10.

Exactly when the events of entering and exiting a state occur in standard cases was explained in Sec. 4.4.1. However, there is a somewhat more subtle case—that of looping transitions. In Fig. 5.11, assume we are in state `S2`. In Fig. 5.11*a* the only entering and exiting events that occur when the transition is taken are those related to `S2`, but, in contrast, in Figs. 5.11*b* and 5.11*c* the ones related to `S` occur, too.

5.4.2 General static reactions

The reactions attached to a state in the Data Dictionary are called *static reactions*. The general static reaction construct makes it possible to define the reaction of the system to an event within a particular state, even without associating it with a transition between states.


State: **COMPARING**
Defined in Chart: **EWS_CONTROL**

Static Reactions:
```
ns/OPERATE ;;
xs/HALT
```

Figure 5.10 Reactions on entering and exiting a state.

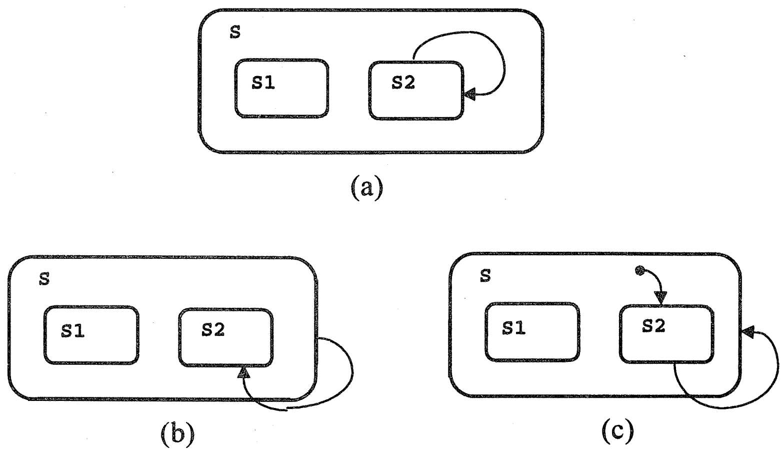


Figure 5.11 Looping transitions.

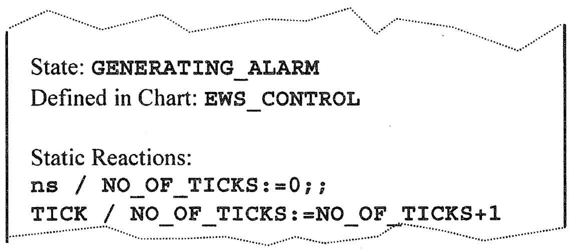
Associating the reaction `trigger/action` with state `S` in the Data Dictionary means that as long as the system is in state `S`, the action is performed whenever the trigger occurs. As in the case of a label of a transition, the trigger can be any event expression (not only `entering` and `exiting`, which are special cases), and the action can be any action expression.

In the EWS example, assume that there is no built-in clock that allows us to use the event `tm(en(GENERATING_ALARM),ALARM_DURATION)` to exit from the `GENERATING_ALARM` state. We may instead employ a "self-made" clock that, from the moment `GENERATING_ALARM` is entered, generates an event `TICK` every time unit. We can then introduce the data-item `NO_OF_TICKS`, and associate two static reactions with the `GENERATING_ALARM` state, as shown in Fig. 5.12.

We may then exit from `GENERATING_ALARM` when we have "seen," say, three ticks. This could be achieved by a transition exiting from `GENERATING_ALARM`, and labeled with the condition `[NO_OF_TICKS=3]`.

It is often tempting to replace a static reaction with a self-looping transition labeled with the reaction, so as to depict more of the specification graphically. This should be done with care. For example, we cannot naively replace the second static reaction for the `GENERATING_ALARM` state with the transition in Fig. 5.13 because each time we reenter `GENERATING_ALARM`, the first static reaction will set the data-item `NO_OF_TICKS` to zero.

Finally, let us note that it is useful to mark on the chart those states that have associated static reactions in the Data Dictionary. We use the > character for this. Thus, for instance, when we add static reactions to the `GENERATING_ALARM` state, the name will be appended with a >, to mark the existence of additional information. See Fig. 5.14.



State: **GENERATING_ALARM**
Defined in Chart: **EWS_CONTROL**

Static Reactions:
```
ns / NO_OF_TICKS:=0;;
TICK / NO_OF_TICKS:=NO_OF_TICKS+1
```

Figure 5.12 General static reactions in a state.

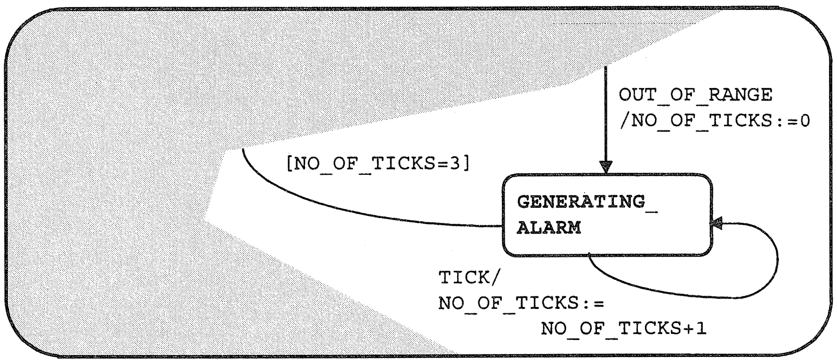


Figure 5.13 Looping transition instead of static reaction.

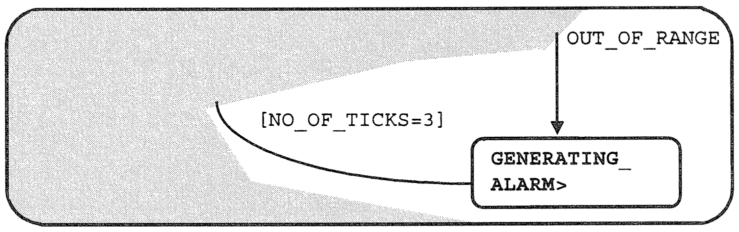


Figure 5.14 Marking a state having static reactions.