Chapter 5

State Machines and Statecharts

5.1 State machines

The two methods seen in detail so far, Larch and Z, deal with the definition of a collection of abstract operations. Larch shows these operations in terms of their mutual effects, and Z shows them as schemas describing the relation between the state before and the state after the operation. This latter view can be formulated in terminology and with a formal basis known as a state machine specification.

In the state machine approach, the underlying formalism is of a finite state automaton, or of a state transition graph. We shall use the state transition graph terminology, although those familiar with automata theory should find the notation familiar. State machines express requirements about the flow of control of a system, including the conditions under which an operation is possible, and the possible states that can occur during execution.

A system viewed as a state machine is described as a collection of states, which are the nodes of a graph, and the possible transitions from state to state, which are connections among nodes. In diagrammatic representations of the graph, the connections are written as arrows. The arrows are often labeled, i.e., the transitions have names. These labels correspond to possible actions or events of the system or of the
environment in which it operates. One or more states are marked as *initial* states. The computations denoted by such a graph are the paths in the graph that begin from an initial state. These are also known as *traces* of the graph. In some versions, the traces are the sequences of states on such paths, while an event-based approach would define the traces as the sequences of labels on transitions in paths from an initial state. The state of a system can be thought of as a snapshot of the values of all variables, program counters, and other system elements such as channels.

For example, a state might be described as a 5-tuple of values, representing values for variables $a$, $b$, $c$, $d$, $e$. Then a possible transition could connect the state $(0, 1, 1, 0, 0)$ with the state $(0, 1, 1, 1, 0)$, representing a change in $d$ from 0 to 1. A nondiagrammatic representation of such a transition could be:

$$(0, 1, 1, 0, 0) \rightarrow (0, 1, 1, 0)$$

This seems to imply that the collection of transitions can be viewed as a function from states and actions to states, but this view is unnecessarily restricted: it is often reasonable to allow nondeterminism. That is, a given state and action can both appear in more than one transition, each connecting the state and action to a different result state. Graphically, more than one arrow from a state can have the same label.

Formally, a state machine graph is defined as a four tuple $(S, A, I, T)$, consisting of a collection of possible states $S$, a collection of possible labels $A$, a collection of initial states $I$ (with $I \subseteq S$), and a transition relation $T$ over $S \times A \times S$ showing all possible transitions. The triple $(s_1, l, s_2)$ is in $T$ exactly when there is an arrow from the node labelled by $s_1$ to the one labelled $s_2$ with a label $l$ on the arrow.

An example of a specification with such a graph may be seen in Figure 5.1. When the state has only one component, the value is written in the corresponding node. This example might be an abstraction of a more complex system, where we only wish to specify the relations among the values specified, which transitions are possible in which states, and what those transitions do to the value. In this case the modes of operation of a simple watch could be showing the time, showing the date, or a state where the battery is not working. The tuple representation then would have
Figure 5.1: A state machine graph

- $S = \{ \text{time, date, batoff} \}$
- $A = \{ c, \text{rem, insert} \}$
- $I = \{ \text{time} \}$
- $T = \{ \text{(time, c, date), (date, c, time), (time, rem, batoff), (date, rem, batoff), (batoff, insert, time)} \}$

One state trace of this system could be $<\text{time, date, time, batoff, time, date}>$. In this example, the labels on the transitions correspond to external events, known as triggers, that lead to the corresponding transition. Thus $\text{rem}$ and $\text{insert}$ correspond to removing or inserting a battery from a watch, while $c$ corresponds to pushing a button. Note that if the button corresponding to $c$ is pressed while the system is in the state $\text{batoff}$, no transition will occur. In many specifications, the transitions represent internal changes of the system, unconnected to physical triggers or interactions to external events.

For a complex state, the values of the variables can be listed in the nodes, as seen in Figure 5.2. Here $\text{light}$ is a Boolean variable, so only two states are needed to represent its possible values when all other components have a value. Still, the number of states in a graph with an added state component (independent of the rest of the state) is the number of values the added component may have times the number of states in the original state graph. Thus as components are added to the state, the potential size of the graph grows exponentially, since each component presumably has at least two possible values. When the values of the added component are not independent of the rest of the state, the combined number of states is smaller.
One problem with such a detailed description of every possible transition is that the number of states and/or transitions can quickly become very large and unmanageable, as noted above, exponential in the number of state components. This difficulty is known as the state explosion problem. Even a trivial counter variable $x$, initialized to zero and occasionally incremented by one, gives rise to an infinite number of states and transitions (with separate states for $x = 0, x = 1, ...$).

For complex software systems it is therefore clear that the number of states and the number of elements in the transition relation are too large to list explicitly, even when they are finite. Nevertheless, in modelling hardware designs or communication protocols, where the values of some variables, channels, or messages can be ignored, such transition graphs are commonly used. In these cases, there are a small finite number of states, and the graphic notation can be helpful in guaranteeing that all situations have been considered.

Note that in such an approach the distinction between a specification and an implementation is especially blurred, since it is hard to imagine a lower level implementation. In fact, a program to implement an explicit state machine graph is more abstract than the specification.
5.2 Textual state machines

When an explicit state graph is impractical, a notation that expresses the graph more concisely is needed. One class of specification methods expresses the transitions textually, with conditions for their ‘activation’. The changes in the state are then described as a relation between the state before the transition, and the state afterwards, much as in Z. Often the same primed versions of variables seen in Z are used to denote the state after the transition. Such a transition \( t \) and its accompanying condition express that in an explicit state machine graph representation an arrow labeled \( t \) appears between any state \( A \) satisfying the condition and any state \( B \) with values obtained by activating the transition \( t \) on the state \( A \). A system and its requirements can be described as a collection of textual transitions, along with a predicate satisfied by precisely the initial states. This form is entirely equivalent to the explicit state machine graph obtained by listing all states that satisfy the initial predicate, and then applying all transitions until no new states or transitions are added.

Such textual notations are still known as state machine methods. Z itself can be seen as a state machine notation for sequential programs, even though the schema notation is more general, and other views of systems can be expressed in Z. If a state machine view is taken, it would be necessary to make the preconditions for activations of operations more explicit, so that a system indeed becomes a collection of possible transitions along with their conditions for activation. Although the state machine view of Z was not emphasized in its initial presentations, it has been widely used in this way for sequential collections of operations.

For the trivial example seen earlier, the Z representation for the state machine would have an operation schema for each transition label. In the abbreviated notation this would be

\[
\begin{align*}
c &= [st, st'](st = time \lor st = date) \land (st = time \rightarrow st' = date) \land (st = date \rightarrow st' = time)] \\
rem &= [st, st'](st = time \lor st = date) \land st' = batoff] \\
insert &= [st, st']st = batoff \land st' = time] \\
new &= [st']st' = time]
\end{align*}
\]
As this example demonstrates, a simple one-to-one translation from an explicit state graph to a collection of Z schemas gathers the arrows that correspond to a single label into one schema. The precondition is the union of the source states with that label in the transition graph, and the predicate involves an implication for each arrow. For an arrow from a state $x$ to a state $y$, the predicate

$$s = x \Rightarrow s' = y$$

can be automatically derived. When, as in the $rem$ schema, the result state is not dependent on the source, the implication is unnecessary. In most cases it is possible to express the preconditions as general predicates in a closed form, and the changes can be expressed as state transformations.

Generally, textual state machine methods put special emphasis on control aspects and describe communication among modules in asynchronous or synchronous distributed environments. Later in this chapter, the level of atomicity of operations and possible interference among operations will be considered for textual state machine specifications. This approach has been used to describe subtle interactions among communication subsystems, including fault tolerance and real time. These aspects will be treated in later chapters.

### 5.3 Statecharts

A statechart is a graphic notation for state machines that alleviates the state explosion problem by using hierarchy and visual techniques to reduce the amount of clutter while maintaining the same information as a full state machine representation. In this approach, a graph with nodes and arrows is used to represent states and transitions, but the nodes are not always primitive objects, and can contain subgraphs. A hierarchical structure is encouraged, where a top level view hides most of the detail, while a zoom facility allows examining and developing more detailed views of the system requirements. The visual notation encourages a global picture of the system structure and internal components, while the zoom facilities allow hiding extraneous details. A simulation facility is used to check that the specification conforms to
the intuitive expectations. A commercial specification tool based on these ideas is called Statemate, and in the continuation we point out differences between the original statecharts and this system.

Some of the causes of ‘clutter’ in state graphs are statements that often arise in specifications. Consider:

- “Whenever the red button is pressed, an emergency situation is begun.” In a pure state graph, every regular state would have to have a transition to a special state that corresponds to the regular state except that the ‘emergency’ indicator is true, thus doubling the number of states.

- “There is no connection between turning on the lights and changing the gear.” This means that the state is composed of the cross product of independent components. If there are three values for a variable lights (perhaps $off$, $dim$, and $bright$) and six for a variable gear, the cross product will have eighteen possible values.

To express such requirements concisely, three basic tools are used:

- States are grouped into superstates when some of the transitions entering or leaving them are uniform. A superstate can be composed of an entire statechart of lower level states. An arrow (transition) leaving the border of a superstate means that the transition can be taken from any of the states within that superstate.

- Orthogonal parts of the state are viewed as parallel components, and the cross product of the components is not represented explicitly.

- An abstract state is considered, with the precise values of some of the system variables viewed as irrelevant. Values for such variables and the effects of some of the transitions on those values might not be specified within the statechart framework.

In Statemate, the control aspects are in statecharts, but there are also separate provisions for describing values of variables or registers. The parallel activities of the system are graphically represented by dividing the state using dotted lines. The communication among the parallel
components is modeled by using joint transitions. As a side-effect of activating one transition, it is possible to activate a second one, and this can also serve as a way to communicate among otherwise independent components.

There are also additional views of the system, outside of the statechart notation, that describe the ongoing activities, as they are affected by particular events, and the physical division of the system into components. A transition can activate or stop external (named but unspecified) ongoing activities, or can activate additional transitions in the statechart. This is also a method with a well developed notation for real-time properties.

In greater detail, a superstate without parallel components is composed of a lower level statechart, i.e., states and transitions among them. (Of course, some of the lower level states may themselves be superstates.) To be in such a superstate $S$ is to be in one of the states from which it is composed. The default state within a superstate is denoted by a dot with a short arrow to the default. An arrow to the boundary of the superstate is then the same as an arrow directly to the designated default. Arrows from outside the superstate directly to states within the superstate are still allowed.

A transition from the superstate is written as an arrow from its boundary, and means that every state within the superstate has a transition with that label and condition for activation. Arrows directly from an interior state out of the superstate are also allowed.

The two statecharts seen in Figure 5.3 are thus equivalent.

When a transition is to or from some specific inner state of a superstate, but the superstate is unopened at the level being considered (i.e., is being viewed as a black box), the head or tail, respectively, of the arrow is connected to a short perpendicular line.

Orthogonal components of a superstate are written with dotted lines separating the components. Although known as parallel states or components, they do not have to correspond to separate implementation processes, and are simply a way to avoid taking the cross product among independent parts of the state. Each parallel component of a superstate has a default. Being in a superstate means that the system is in all of its parallel components, in a particular state of each. Thus it is equivalent to being in a state made by taking the cross product of
5.3. STATECHARTS

Figure 5.3: A simple superstate example

the components. Entering such a superstate means entering all of its components, and leaving it means leaving them all. An arrow to the boundary enters the default of each component.

As in simpler superstates, there can also be arrows directly to internal states. These may be hyperarrows that have multiple heads, up to one to a state in each component. If an arrow is directly to an internal state in a proper subset of the components, the default states of the components not in that subset are entered when the transition is followed. Hyperarrows with multiple tails are also possible. If there is an arrow to outside the superstate from a state in one component, the transition is possible from that state and from every state of the other components that satisfies the condition for the transition to occur.

To keep the full power of allowing transitions from every state of the explicit cross product state graph to any possible other state, but without forming the cross product, the nature of a transition is more complex when there are parallel components. For example, it must be possible to express that a transition between states within one component is only possible when another component is in an appropriate state. The general form of a transition in a statechart is

\[ \sigma[P]/S \]

where \( \sigma \) is the name of the transition, \( P \) is an applicability condition, and \( S \) is an action associated with the transition.
Figure 5.4: Joint transitions in parallel components

The actions associated with a transition typically can be sending messages, assigning new values to variables, initiating or stopping externally defined activities, or activating another transition. Most actions are associated with aspects of a system’s requirements not treated within the statechart itself, and will be briefly described later. Coordination among the transitions in parallel components is mainly handled through the condition $P$, and through the use of joint transitions. Consider the superstate seen in Figure 5.4. If the system is in states A and C, then the transition $go$ will lead to the states B and D, i.e., both arrows labeled by $go$ are followed. This is known as a joint transition. If the system is in states B and C when transition $go$ is taken, the system will remain in B, but the other component will move to D. If control is in B and D already, no transition will occur, even if an external signal that otherwise would enable the transition takes place. The transition $c$ can occur if the third component is in E and the first is in B, because of the condition next to the transition. Taking this transition only affects the third component.

The ways in which the cross product’s transitions are represented in a superstate with parallel components can be seen in Figure 5.5, where the representations as a statechart and as an explicit transition graph are equivalent. The correspondences between the two representations show common modes of expression in statecharts. For example, the transition $g$ in the left statechart is from $E$ to $G$. Thus in the explicit state transition graph, $g$ connects the two pairs that represent
the change from $E$ to $G$ but keep the first component ($B$ or $C$) unchanged. To express that $f$ is (only) from state $(C, G)$ to state $(B, G)$ in the transition graph, the $[inG]$ predicate is used next to $f$ in the statechart. The transition $e$ in the state transition graph changes both components when done from state $(B, F)$, and so corresponds to a joint transition in the statechart when done from states $B$ and $F$.

The joint actions can be viewed as synchronization points among the parallel components, and the $in$ predicates imply that one process can examine the state of another one. These features of statecharts seem natural, and are fairly easy to implement, for a shared-memory implementation of the processes. They are less appropriate when the processes will be implemented in a distributed message-passing environment. In that case, determining the state of another process would involve sending a query message, and waiting for a response. Even then it is not clear whether the information received is current. For these reasons, implementations of statechart specifications can be difficult for some distributed models of computation and languages. Separately from the above considerations, statecharts are convenient for showing clearly the possible transitions from one component of the state. However, from a statechart representation it can be difficult to determine the overall effect of an action, because arrows with that action are scattered among parallel components.

As noted above, one of the possible actions that can be associated with a transition in a statechart is to activate another transition. This
provides another way to coordinate among parallel components. Consider one component with a transition from \( A \) to \( B \) labelled \( e/f \) and a parallel component with a transition from \( C \) to \( D \) labelled \( f \). According to the original statechart semantics, if the superstate is in the states \( A \) and \( C \), and \( e \) occurs, it immediately (in the same transition step) activates \( f \) also, so that the superstate will then be in the states \( B \) and \( D \) as if a joint transition had occurred. The resulting transition is said to be composed of two *mini-steps*. However, if an \( f \) transition occurs, only the transition from \( C \) to \( D \) is taken, with the other component unchanged. Of course, if the superstate was already in state \( D \) when \( e \) occurred, and there is no \( f \) transition from \( D \), only the \( A \) to \( B \) transition will occur.

In the Statemate system, a variant semantics is used: a transition of the form \( e/f \) activates \( f \) only in the following state, so there are no mini-states ‘within’ a transition. Note that it is possible to construct ‘chain reactions’ \( a/b, b/c, \) etc. and even cycles of activations, as in \( a/b \) and \( b/a \). Under the original statechart semantics, these could lead to a transition that never terminates, depending on how such a transition is interpreted. Such questions have posed difficulties for those trying to precisely define the semantics of this feature, and motivated the view seen in Statemate. A restriction makes such mini-steps within a transition reasonable: each component is only allowed to change states once within a chain reaction.

The statechart notation, like many of the others treated in this book, handles nondeterminism naturally. There is no reason to assume that the conditions for transitions from a state are disjoint, or involve distinct triggers. The possible traces are still clear, and the nondeterminism can either represent partial knowledge about the environment, or real choice among a collection of possible actions. The Statemate system is less tolerant of nondeterminism, evidently because it makes the behavior of simulations more difficult to reproduce.

As already noted, the statechart formalism is compact partially because it does not attempt to describe all aspects of a system. An *activity* is sustained over time, and is not described within the statechart notation. Actions associated with transitions that affect an activity \( A \) include \( \text{start}(A) \), \( \text{stop}(A) \), \( \text{suspend}(A) \), and \( \text{resume}(A) \). Thus a transition could have the form \( a/\text{start(beeping)} \) where the activity of *beeping*
is defined outside the statechart formalism. Alternatively, it is possible to link an activity to a state, so that it is started when the state is entered, and stopped when it is left. For example, a state called beep.on could be reasonably associated with the beeping activity.

Other actions can update variables not expressed explicitly within the statechart, with assignment statements. A transition of this form is \( a/ x := 2y + 1 \).

### 5.4 Examples of statecharts

Some typical statecharts can be seen in the following figures, taken directly from the Statemate tool. These can be understood almost without textual explanations, illustrating the accessibility of the notation.

Turning to the first statechart, we can see that it describes some of the states of a system, and the possible transitions. In particular, it describes some required stages in failure and recovery modes of a physical system, as related to the usual operation modes. The ok superstate is the default (seen by the small circle and arrow at the top of the chart) and when entered, will go to the default internal state, recursively, until normal is reached. Note that by using superstates, we can easily express that whenever a particular event occurs in any one of a collection of states, some particular transition should occur. Thus we can see that in the ok component, whenever a correct event occurs (that may be connected to pressing a button), then in both normal and alternate modes a transition to fix mode will occur, while if the system is already in test mode, that event will have no effect. The ok superstate can be left either by a power-failure event, or by a reconfiguration event, each leading to different states. These states also presumably have internal structure, but are not 'open' in the view given here, because that level of detail is not needed for the overall understanding of the aspect of the system being considered. Note that when the warm-start state is left in the DONE transition, then the default state of ok will again be entered.

The second statechart shown relates the rate and time components in parallel. The DONE transition is joint to the two parallel compo-
nants. In the *regular* superstate the transitions between day and night can occur only when the time variable $t$ has appropriate values. This variable is updated in the *time* component as a side effect of the *TICK* operation, and/or within the states that are not given in detail.

In the third statechart the *ok* sequential superstate seen previously is now ‘refined’ into a parallel superstate composed of the previous version of *ok* in parallel with another independent component. The new component specifies when a register is displayable or not (depending on whether the parallel component is in the superstate *regdisp*).

The final statechart shows how security levels can be changed and how they enforce doing some unspecified action *SPECIAL* only when the security level is in state *level2*.

It should be noted that there are other aspects of the Statemate system intended to describe the physical partition of the implementation. For example, if the system is to be implemented on two processors, there could be a division to a front-end and a back-end processor, each having part of the functionality. The tasks assigned to each component and the types of messages passed between these are also easy to list and to connect to the statechart level, but are not part of the statechart.
5.4. EXAMPLES OF STATECHARTS
5.4. EXAMPLES OF STATECHARTS

Security

Security MODE

TOTAL_KW_CUM

Displayable

Hidden

Not in(REGDISP)

Toolbox

Normal

Alternate

Test

Mode

Normal

Alternate

Test
SECURITY

LEVEL0

RESET
or
tm(en(SECURE),5)

KEY1

KEY2

LEVEL1

LEVEL2

SPECIAL[in(LEVEL2)]

SECURE
5.5 Methodology and tools

Statecharts are today most widely used through the commercial Statemate system due to the company Ad Cad and its subsidiary I-Logix. The system is very convenient to use graphically, and has an excellent color simulation facility that allows considering the effects of various scenarios of events, in order to provide early feedback. There are also features that allow listing real-time requirements conveniently. However, except for the simulation, the tool is weak on automatic analysis, and on proving a connection between the Statemate specification and implemented code.

Variations of statecharts have been introduced, and used for modelling applications in avionics and other distributed environments. Statecharts are strongest in modeling control, especially concurrency and real-time, with overlapping operations. They are less effective for expressing the data values needed. The Statemate system introduces some additional notation to allow recording formulas and other data manipulation aspects, as well as separating the statecharts into a collection of ongoing activities, each with its own statechart to describe control. It might be advisable to use the method in conjunction with Larch or Z (similarly to the combination seen later in LOTOS).

The level of a statechart specification is generally fairly close to the implementation, and thus can suffer from overspecification relative to the user requirements.

5.6 Atomicity and Interference

Specifications using state machines deal both with the design of complete systems and with modules within larger systems. This is the first context in which concurrent activities are explicitly taken into consideration. In this context, the interface between the unit being specified and the environment external to that unit is more complex than previously. For Larch and Z it is sufficient to consider operations as being atomic, in the sense that they transform the state before the operation to the state afterwards as an atomic step. Here it is necessary to distinguish between internal steps that can either be taken or can be delayed
arbitrarily, and operations initiated by external stimuli from the environment that must be treated by the module under consideration. The relations among concurrent components both within and without must also be specified.

In a closed system, the entire system is specified, with only very simple (generally input/output) relations with the environment. In such a system all operations are assumed known and are specified within the system. In the earlier chapters we have implicitly assumed that a closed system is being described.

Modules with a significant ongoing interaction with the environment are known as open systems. In an open system some operations are not part of the module under consideration and, for example, operations from the environment can change the state of the system while the module is ‘in the middle’ of the implementation of an operation. To specify such a module, possible interferences from the environment much be described, and only if such interference is restricted is the module expected to satisfy specified requirements.

More broadly, whenever parallel components can interact, either within or between modules of the system or environment, levels of atomicity and possible interferences among operations must be considered. This is most obvious when several operations can change the same statespace. Consider a state composed of the variable $x$ that has the value 0. What are the possible results if the assignment $x := x + 1$ is executed in parallel with the assignment $x := x + 2$? The answer depends on what is considered atomic. If the level of atomicity is an entire assignment statement, then there are two possible orders of execution. One has a sequence of states $(x = 0, x = 2, x = 3)$ and the other has $(x = 0, x = 1, x = 3)$. For each of them, the result after executing both statements will be $x = 3$. In this situation, there are four possible states that could occur.

However, if each statement is actually composed of smaller units that involve assembly commands such as loading a value to a local register, incrementing the register, and storing the value of the local register back into the memory location associated with $x$, other results are possible. In this situation, each assignment has three internal steps, and thus has four stages. The state now should relate to $x$, the local registers, and the stage at which each assignment is found. Thus there
are now sixteen possible states (if the local register for each assignment is distinct). Moreover, for some of the resultant execution sequences, either \( x = 1 \) or \( x = 2 \) could be the result after both operations are completed, in addition to the possibility of obtaining \( x = 3 \) (the reader is invited to determine how this could occur). Of course, such interferences among operation implementations influence what can be specified, as well as lower level programming of multiple processes in shared memory.

In specifications involving parallelism, it is often necessary to relate to the level of atomicity of the system. Other crucial questions are whether some part of the system (processes, or other units) could be denied service indefinitely (known as starvation), whether the system as a whole could enter a state where no actions can be taken, even though the specification is still unsatisfied (deadlock), or whether some specified crucial operations are guaranteed not to overlap (mutual exclusion).

As one way of treating such situations, an operation can be seen as a nonatomic collection of possible lower level transitions, each of which can occur at some point during the implementation of the operation. The individual transitions are atomic and define crucial moments during the implementation. For example, when an element is being inserted into a data structure, there is some moment when the new element is considered to be “in” the structure (and a concurrent query on its membership may begin to give a positive response). The details of the implementation are still not of interest in such a specification. For example, the element to be inserted could be copied bit-by-bit, and the internal representation of the data structure could be revised as part of the insert operation, but there still must be such a crucial moment when concurrent operations react as if the element is inserted, possibly even before the insert operation is completed. The considerations on when operations take effect relative to concurrent operations resemble those seen in concurrent database systems, and will be explained in more depth later.

A textual state machine specification with this approach is syntactically similar to state machines where operations are atomic, but will give a different interpretation to conjuncts true in the description of possible transitions. A specification is composed of
State functions that can be declared, e.g., using a Z schema

- Initial conditions of the state
- Invariant properties true for each possible state

In addition to usual predicates, there are two common ways to specify the invariants by describing what is possible in the system (and thus to characterize safety properties). The first has the form

\[ \text{A leaves unchanged } f \text{ when } Q \]

In the expression above,
- A is a collection of actions (atomic transitions)
- f is a state function
- Q is a predicate

This means that any action of A that is done from a state that satisfies Q leads to a state in which the value of f is the same as in the state before the action. Thus, if stack is a state function, and TOP is a collection of lower level atomic transitions that together represent the implementation of the top operation on a stack, then

\[ \text{TOP leaves unchanged stack when true} \]

The other form resembles more closely the standard description of actions as symbolic transformations seen in the view of Z as a textual state machine. The only difference is that the transitions are seen as describing crucial moments within the implementation of a nonatomic operation, rather than as part of the single transition describing an atomic operation. The \text{INSERT} (nonatomic) operation with a parameter x can be described as

\[ \text{INSERT: } \text{set}' = \text{set} \cup \{ x \} \]

More interesting specifications have several transformations, that are only partially ordered, and indicate which stages occur during the implementation of the operation. A typical case could involve copying
an input parameter to an internal variable, copying an internal variable
to a stable data structure, and copying a result to an output parameter.
As part of the specification, it is then possible, for example, to require
that the environment will not change an input parameter until the
implementation of the operation first copies it to an internal variable.
After the transition copying the input, the environment is free to modify
the input.

Some useful predicates for expressing such relations are in, at, and
after. These are true for a module representing the implementation of
an operation if the next step in the system can be one of the transi-
tions of the module (for in), if the next transition can be the first of the
module (for at), or if the most recent step of the system was one of the
module, but all of the transitions of the operation have already occurred
(for after). (Note that in the above definitions, at(M) ⇒ in(M). Now
we may write, for example,
\[
\text{ENV leaves unchanged input parameter when in(IN SERT)}
\]

\section*{5.7 Bibliographic remarks}

State machines and transition graphs originate in automata theory
and the connection with formal languages. As noted earlier, Z can
be used to describe textual state machines. Among the other speci-
fication methods in this category are the Input/Output automata of Lynch
and Tuttle [6, 7, 5], and the version of a state machine advocated by
Lamport[3, 4]. The nonatomic specification described at the end of the
chapter are based on Lamport’s approach. The use of textual state ma-
chines to model and specify fault tolerance is explained and surveyed
in a paper by Schneider[8].

Statecharts were first introduced by David Harel[1]. The Statem-
ate system is described in [2]. Statecharts are being incorporated into
the UML (Universal Modeling Language) being developed by Rational
software as a proposed standard for the specification of reusable objects
developed using object-oriented design.
Bibliography


