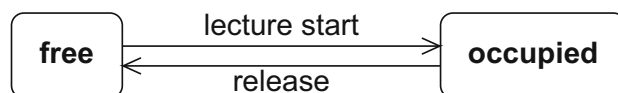


Chapter 5

The State Machine Diagram

Over the course of its life, every system, or to be more precise every object, goes through a finite number of different states. Using a *state machine diagram*, you can model the possible states for the system or object in question, how state transitions occur as a consequence of occurring events, and what behavior the system or object exhibits in each state.

As a simple example consider a lecture hall that can be in one of two states: free or occupied. When a lecture starts in the lecture hall, the state of the lecture hall changes from free to occupied. Once the respective event in the lecture hall has finished and the hall has been released again, its state reverts to free (see [Fig. 5.1](#)).



State machine diagram

Figure 5.1
State machine diagram of a lecture hall (simplified presentation)

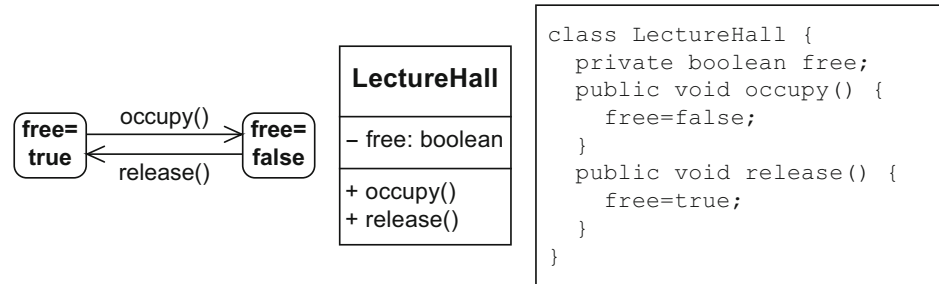
The state machine diagram is based on the work of David Harel [22] and uses concepts of finite automata. UML differentiates between two types of state machines, namely behavior state machines and protocol state machines. In this book, we present only behavior state machines, which are widespread in practice and are also referred to as state machine diagrams or state charts.

In the same way as every other diagram, a state machine diagram only models the part of a system that is necessary or relevant for the respective purpose. For example, if you want to model only the states that a lecture hall can take, either for collecting requirements or for documentation purposes, a model as shown in [Figure 5.1](#) can be sufficient. However, if you are already in a late phase of the development process, a representation that is close to code, as shown in [Figure 5.2](#), is ben-

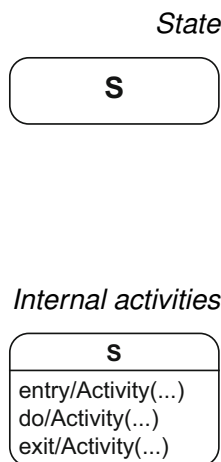
eficial. This figure shows a class LectureHall with an attribute free that can take the values true and false. Calling the operation occupy sets free to false and the lecture hall object changes to the state free=false, which corresponds to the state occupied in Figure 5.1. The events in the state machine diagram are the equivalent of calling the respective operations of the class LectureHall.

Figure 5.2

State machine diagram,
class diagram, and pseu-
docode of a lecture hall



5.1 States and State Transitions



A state machine diagram is a graph with *states* as nodes and *state transitions* as edges. In the diagram, a state is shown as a rectangle with round corners and is labeled with the name of the state. When an object is in a specific state, all internal activities specified in this state can be executed by this object. If internal activities are specified for a state, it is divided into two compartments: the upper compartment of the rectangle contains the name of the state; the lower compartment includes *internal activities*, whereby an activity can consist of multiple actions. We will present the relationship between activities and actions in detail in Chapter 7, which looks at activity diagrams.

Within a state you can model three activities that are executed at a predefined moment. When an activity is specified after the keyword *entry*, this activity must then be executed when the object enters the state; conversely, the *exit* activity is executed when the object exits the state. An activity preceded by the keyword *do* is executed while the object remains in this state, that is, as long as this state is *active*. The respective activity is always specified with a prepended forward slash that clearly identifies it as an activity.

Figure 5.3 shows an extension of the example from Figure 5.1. As long as a lecture hall remains in the state free, that is, as long as the state free is active, the activity Display as available is executed and the lecture hall is displayed in the reservation system. If the lecture hall is occupied,

it changes from the state *free* to the state *occupied*. At the moment the lecture hall enters this state, the activity *Save user reservation* is executed and the name of the person occupying the lecture hall is saved. While the lecture hall remains in the state *occupied*, the activity *Display as occupied* is executed. Once the lecture hall is no longer required, it is released and changes to the state *free*. When the lecture hall exits the state *occupied*, the activity *Delete user reservation* is executed.

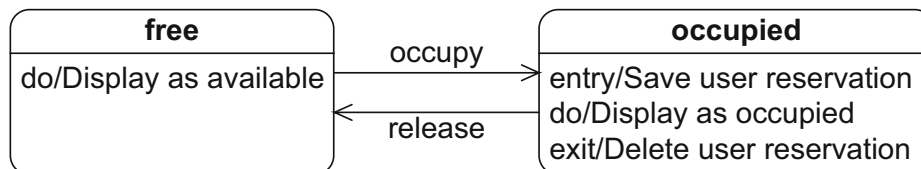


Figure 5.3
State machine diagram of
a lecture hall with internal
activities

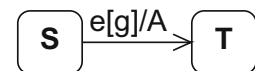
The change from one state to another is referred to as a state transition or simply *transition*. A transition is represented with a directed edge, that is, an arrow. The arrowhead reflects the direction of the transition. The origin of the transition is referred to as the *source state* and the end of the transition is referred to as the *target state*. You can specify various properties for a transition:

- The event (also called “trigger”) that triggers the state transition
- The guard (also called “guard condition” or simply “condition”) that enables the execution of the transition
- Activities (also called “effects”) executed during the change to the target state

Events are exogenous stimuli (that is, stimuli that come from outside the system/object) that can trigger a state transition. If the event specified for the transition occurs, the *guard* is checked. The guard is a boolean expression. At a specific point in time, it evaluates to either true or false. If the guard is true, all activities in the current state are terminated, any relevant exit activity is executed, and the transition takes place. During the state transition, any *activities* defined for that transition are executed. A transition—at least from a conceptual perspective—requires no time. Therefore, the system is always in a state and never in a transition. The activities specified for a transition must therefore also not require any significant time.

If the guard evaluates to false, there is no state transition and the event is lost and not consumed. Even if the guard becomes true at a later point in time, the event must occur again for the transition to take place. If no guard is modeled at a transition, the default value [true] applies. If no event is specified at a transition, the transition is triggered when the entry activity and do activities of the current state are completed.

Transition



Synonyms:

- *Transition*
- *State transition*

Event (Trigger)

Guard (Condition)

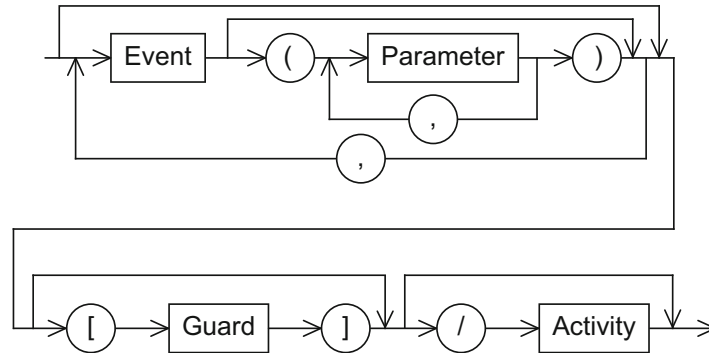
Activity (Effect)

Completion event and completion transition

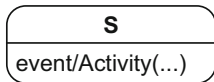
Finishing these activities creates a *completion event* that triggers the transition. This type of transition is also referred to as a *completion transition*. If an event occurs for which no behavior is specified in the current state, the event is not consumed and is lost.

Guards are always set within square brackets to differentiate them from events and activities. Activities are always prepended with a forward slash (including activities in the states). Figure 5.4 illustrates the syntax of a transition specification.

Figure 5.4
Syntax of a transition specification



Internal transition ...



You can model *internal transitions* within states. These internal transitions handle the occurrence of events within a state. You use them to model the reaction to an event when the system does not exit the state that is currently active, meaning that entry and exit activities are not executed.

... in contrast to "external" transition

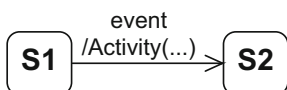


Figure 5.5 shows the two states that a student can take with reference to an exam, namely not registered and registered. As long as the student remains in the state not registered, every time a new exam date is published the student checks whether there is enough time to take the exam on this date—meaning that every time the event new date occurs the activity Check date is executed. If the event register occurs, provided the guard registration possible is true, the student switches to the state registered and the date of the exam is entered in the calendar. As long as the state registered is active, the student is studying. Any time the student encounters a problem, it is discussed with the student's colleagues. If the event withdraw occurs in the state registered, two different cases are possible. If the guard withdrawal possible is true, the activity Study for exam is interrupted and the student switches to the state not registered. When the student exits the state registered, the date is deleted from the calendar. However, if the guard withdrawal possible is false, the student remains in the state registered and must continue to study for the exam. (Believe it or not, in the home country of the authors it is possible to withdraw from an exam without consequences.)

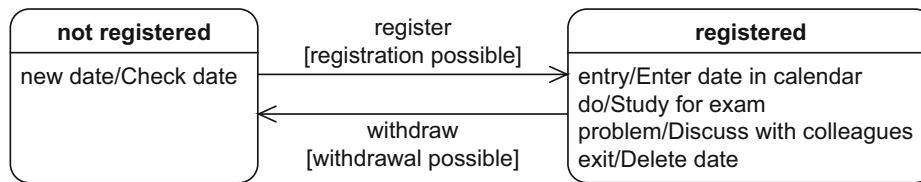


Figure 5.5
State machine diagram of
the registration status for
an exam

To further illustrate the concept of events, guards, and activities, Figure 5.6 shows abstract examples of transitions. The transition in Figure 5.6(a) has no event and no guard. Therefore the transition can take place as soon as A1 is completed. Figure 5.6(b) is similar to Figure 5.6(a) but activity A2 is executed during the transition. In Figure 5.6(c), the transition takes place as soon as event e1 occurs. If e1 occurs, the execution of the do activity A1 is immediately interrupted and the system switches to state S2. When the system exits state S1, the exit activity A2 is executed.

In Figure 5.6(d), guard g1 is checked as soon as e1 occurs. If the guard is true, A1 is terminated and there is a change of state to S2. If the guard is false, event e1 is lost and A1 is not interrupted. Figure 5.6(e) is similar to 5.6(d) but in 5.6(e), activity A2 is executed in addition during the transition.

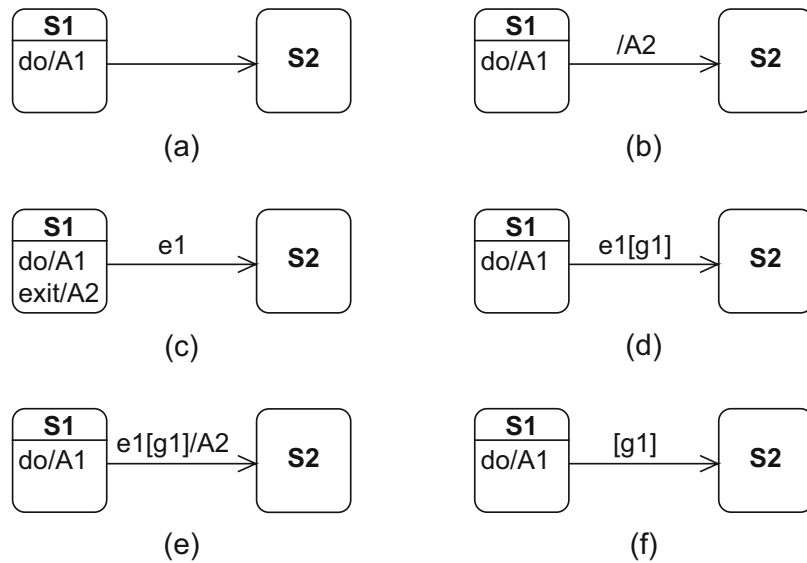
Figure 5.6(f) shows an “unclean” use of a guard. The system stays in state S1 until A1 is completed. Guard g1 is not checked until this point and the transition takes place if g1 is true. If g1 is false, the system remains in state S1 and it will never be possible to exit S1 via this transition as the completion event of the do activity was lost when it was not consumed. This type of transition specification only makes sense if, for example, there is a further transition with a complementary guard, meaning that there is no dead end (not depicted here).

5.2 Types of States

In addition to the states discussed in Section 5.1, there are further types of state that enable you to model more complex content with state machine diagrams. There is a distinction between “real” states and *pseudostates*. Pseudostates are transient, which means that the system cannot remain in a pseudostate. They are not states in the actual sense but rather control structures that enable more complex states and state transitions. You cannot annotate activities to pseudostates. These pseudostates include the initial state, the decision node, the parallelization and synchronization nodes, the history state, the entry and exit points, and the terminate node. These are described in more detail below.

*Pseudostates are
transient*

Figure 5.6
Examples of transitions

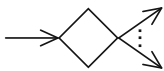


Initial state



The *initial state* is represented in the diagram as a small black circle and marks the “start” of a state machine diagram or a composite state (introduced in Section 5.5). The initial state has no incoming edges and usually one outgoing edge which leads to the first “real” state. If multiple outgoing edges are used, their guard conditions must be mutually exclusive and cover all possible cases to ensure that exactly one target state is reached. As soon as the system is in the initial state, it immediately—that is, without consuming any time—switches to the next state. Therefore, you cannot specify any events to the outgoing edge from the initial state. The only exception to this rule is the event that creates the modeled object itself—`new()` or `create()` for example. However, you can specify activities.

Decision node



The *decision node* is represented in the diagram with a diamond. You can use it to model alternative transitions. It has exactly one incoming edge and at least two outgoing edges. At the incoming edge, you model the event that triggers the transition; at the outgoing edges, you specify the guards for the alternative paths for the state transition. You can also specify activities at the incoming edge and all outgoing edges. If the event modeled at the incoming edge occurs, the system enters the transition. However, it pauses briefly at the decision node—but from a conceptual perspective without consuming any time—to evaluate the guards and thus select the outgoing edge to be used. To prevent the system getting “stuck” in the decision node, you must ensure that the guards cover all possible situations. Using `[else]` at one of the edges will allow you to do this. If the guards are not mutually exclusive, and if two or more edges are evaluated as true, one of these valid edges is selected nondeterministically. [Figure 5.7\(a\)](#) shows an example of the use of the

decision node. If event $e1$ occurs, the transition takes place. Once the system has arrived at the decision node, the guards $[b \leq 0]$ and $[b > 0]$ are evaluated and the system switches to state $S2$ or state $S3$. You can also model the same behavior without using a decision node, as shown in Figure 5.7(b). In contrast, Figure 5.7(c) and Figure 5.7(d) show different behavior. Figure 5.7(c) shows that if event $e1$ occurs, the transition starts and b is increased by the value 1. The guards are then evaluated. In Figure 5.7(d), b is only increased by the value 1 after the evaluation of the guards. Therefore, depending on the value of b , transitions to different states can occur in the two models. Figures 5.7(c) and 5.7(d) are therefore not semantically equivalent.

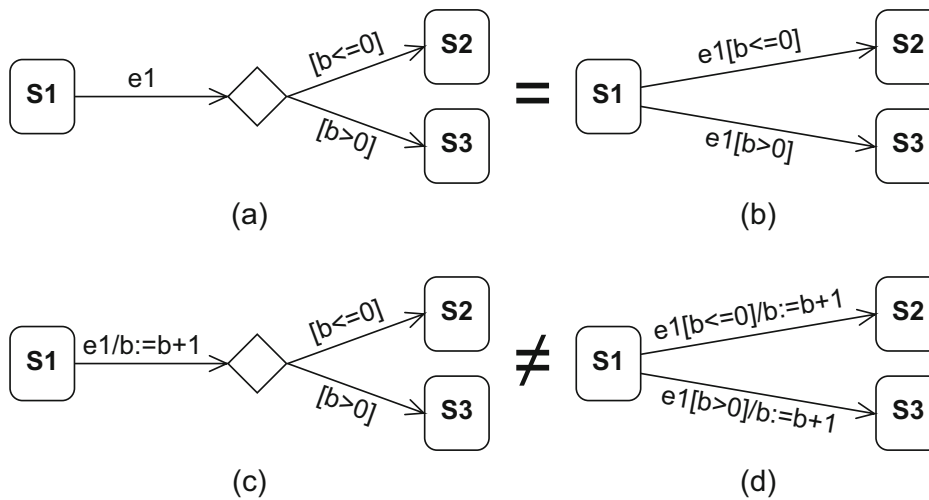


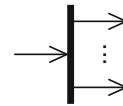
Figure 5.7
Modeling with and without
decision nodes

Figure 5.8 shows the states that a student has when participating in a specific course. If the student is in the state not graded and the event grade occurs, depending on the grade the student receives, the student switches to the state positive or the state negative. The X in the model is called terminate node, which we will introduce later on in this chapter.

The *parallelization node* is represented with a black bar. It has exactly one incoming edge and at least two outgoing edges and is used to split the flow into multiple concurrent transitions. No events or guards may be specified at the outgoing edges of a parallelization node in a state machine diagram.

The *synchronization node* is also represented with a black bar. It has at least two incoming edges and exactly one outgoing edge and is used to merge multiple concurrent flows. No events or guards may be specified at the incoming edges of a synchronization node. For more information on these two pseudostates and a description of the history state, see Section 5.5. Note that parallelization nodes must not be confused with decision nodes.

Parallelization node



Synchronization node

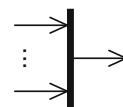
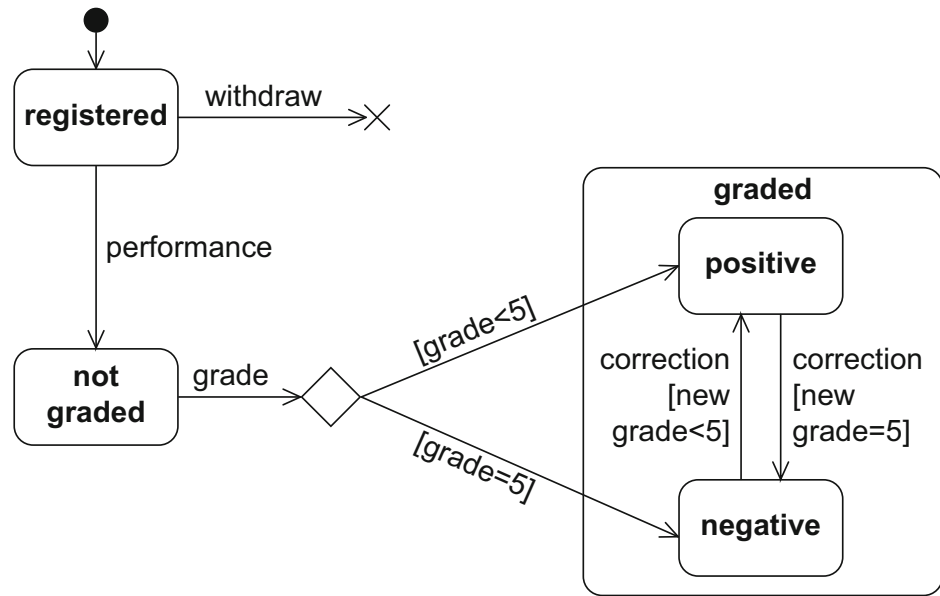


Figure 5.8
States of a student's course
participation



Terminate node



The *terminate node* is represented in the diagram with a large X. If a terminate node is reached in a flow, the state machine terminates and the modeled object ceases to exist. In [Figure 5.8](#), a specific course participation object for a certain student is deleted if it is in the state registered and the event withdraw occurs.

Final state ...



... is a "real" state

The only other “real” state—that is, a non-pseudostate—in addition to the states discussed in Section 5.1 is the *final state*. The final state has at least one incoming edge and no outgoing edges. In a diagram, it is represented by a small circle containing a solid circle. It marks the end of the sequence of states (see also Section 5.5). The object can remain in a final state permanently. Note that the final state must not be confused with the terminate node, where the modeled object is deleted! For a detailed explanation of entry and exit points, see Section 5.5.3.

5.3 Types of State Transitions

Internal transition

*Entry activity
and
exit activity*

As already mentioned, there are two types of state transitions, namely internal transitions and external transitions. *Internal transitions* represent the reaction to an event that triggers an activity but not a state transition. As there is no change in state, no entry or exit activities are executed either. Entry and exit activities are modeled with the same notation as any other internal transition. However, they require the keywords *entry* and *exit* instead of the name of the triggering event in order to specify that the respective activity is executed when the system or object enters or exits the state. Internal transitions are modeled within states.

When the system or object exits one state and enters another as a reaction to an event, the transition is called an *external transition*. First, the exit activities of the source state, then the activities of the transition, and finally the entry activities of the target state are executed as part of the state transition. A *self-transition* is a special type of external transition in which the source state and target state are identical.

Figure 5.9 shows examples of internal and external transitions.

External transition

Self-transition

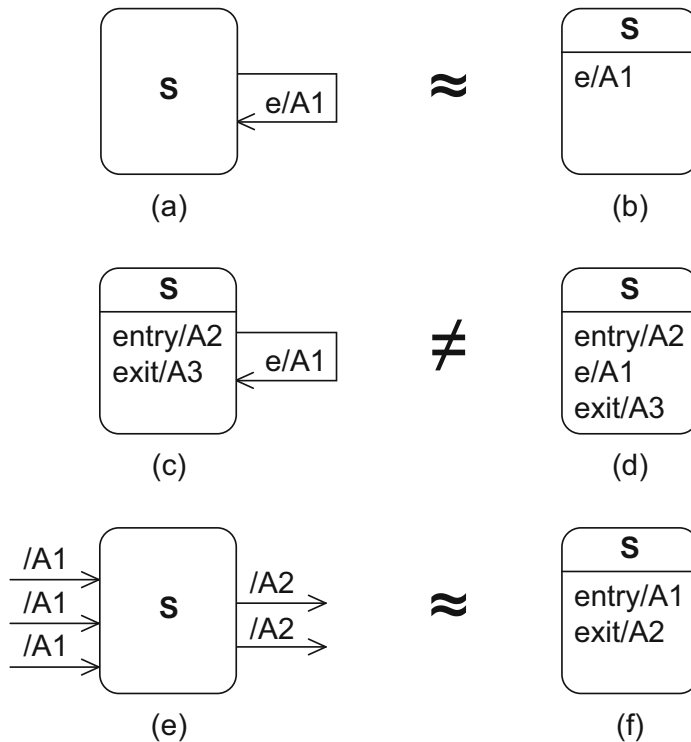


Figure 5.9
Examples of internal and external transitions

For example, Figure 5.9(b) shows an internal transition, Figure 5.9(e) an external transition, and Figure 5.9(a) a self-transition. The models in Figure 5.9(a) and Figure 5.9(b) show the same behavior: in both models, when event *e* occurs, activity *A1* is executed; in both models, before event *e* occurs, the system was in state *S* just as it was after the processing of event *e*. Figures 5.9(c) and 5.9(d) are in no way equivalent, as in 5.9(c), whenever event *e* occurs, the system exits state *S* and thus exit activity *A3* is executed, followed by *A1*, and finally, when the system again enters state *S*, entry activity *A2* is executed. In contrast, in 5.9(d), event *e* does not trigger the exit and entry of state *S*, which is why no entry and exit activities are executed. If the same activity is modeled for all incoming transitions of a state, the execution of this activity can be modeled as an entry activity of the state instead. In the same way, activities for outgoing transitions can be modeled as an exit activity. Therefore, Figures 5.9(e) and 5.9(f) are semantically equivalent.

5.4 Types of Events

UML defines various types of events, with the most important being the signal event, call event, time event, change event, any receive event, and completion event. The *signal event* is used for asynchronous communication. In this case, a sender sends a signal to a receiver and does not wait for an answer. The receiver is the modeled object and the sender can be another object or the same object as the receiver. The receipt of the signal is processed as an event. The name of the event corresponds to that of the signal and arguments can be specified. For example, right-mouse-down or send sms(message) are signal events.

Signal event:
event name(arg1,arg2)

Call events are operation calls. The name of the event corresponds to the name of an operation including parameters, for example, occupy(user,lectureHall) or register(exam).

Call event:
opName(par1,par2)

Time events enable time-based state transitions. The specified time can be relative—based on the time of the occurrence of the event in the state currently active—or absolute. Relative time events consist of the keyword after and a time span in parentheses, for example, after(5 seconds). Absolute time events are modeled with the keyword when and a time in parentheses, for example, expressions like when(time==16:00) or when(date==20150101) indicate absolute time events.

Time event:
after(period)
when(time)

You can use a *change event* to permanently monitor whether a condition becomes true. A change event consists of a boolean expression in parentheses and the preceding keyword when. Examples of change events are when(registrations==number of seats) or when($x > y$). The event occurs as soon as the value of the logical expression changes from false to true. It is lost—just like every other event—if, for example, a guard prevents the event from being processed. However, it can only occur again when the value of the boolean expression changes from false to true again, meaning that the expression must have been false in the meantime. In [Figure 5.10](#), the system is in the state course execution. As soon as semester end changes from false to true, the system checks whether grades are available. If this is the case, there is a state change to certificates issued. If no grades are available, the system remains in the state course execution and the change event is lost. Even if the guard [grades available] becomes true at a later point in time, there can be no transition. The system does not check the guard again until semester end has changed to false and then true again. This expresses that certificates can only be issued at the end of a semester, and then only if grades are available.

Change event:
when(boolExpr)

It is important to stress here that events of the type change event must not be confused with the guards for transitions. The system checks the boolean expression of a change event constantly and the event can trig-

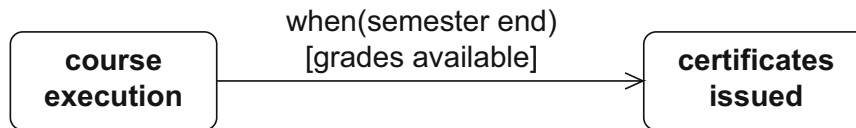


Figure 5.10
Example of a change event

ger a state transition in the instant it becomes true. In contrast, a guard is only evaluated when the related event occurs. Therefore, a guard can never trigger an event itself. In [Figure 5.11](#) a small example illustrates this difference. A student can be in one of two states, namely attending lecture or leisure time. In (a), the student listens to the lecture for 90 minutes. After 90 minutes, there is a state transition to the state leisure time. In contrast, in (b), the student listens until the end of the lecture, as it is only when the event lecture ended occurs that the system checks whether the lecture has already lasted for 90 minutes. If we model the content as shown in (a), this means that leisure time begins for the student after exactly 90 minutes. According to model (b), leisure time begins whenever the lecturer finishes the lecture—but at the earliest after 90 minutes, as the guard is only true then. Note that this model assumes that the lecture is never shorter than 90 minutes.

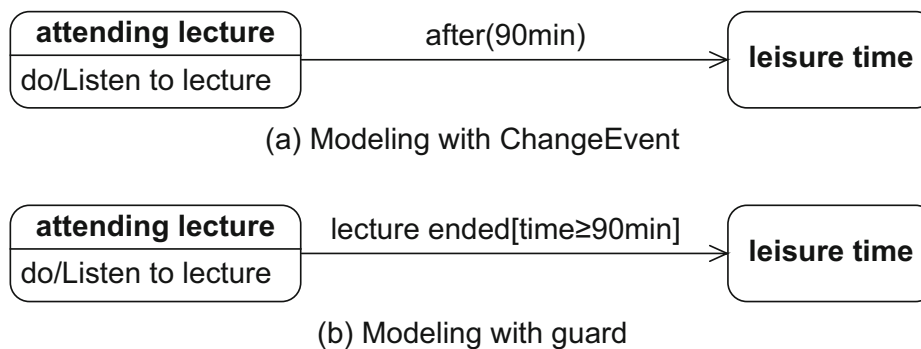


Figure 5.11
Lecture and leisure time

You can use an *any receive event* to specify a type of “else” transition. For this type of event, the keyword *all* is attached to a transition as an event which occurs when any event occurs that does not trigger another transition for the active state. In [Figure 5.12](#), the system changes from state S1 to state S2 if event e1 occurs. If e2 occurs, there is a transition to state S3. If any other event occurs, the system changes to S4.

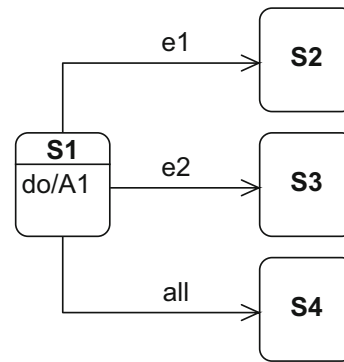
A *completion event* takes place when everything to be done in the current state is completed. This includes entry and do activities as well as the completion of nested states, if there are any (see next Section). If a state has an outgoing transition without any event specified, the completion event triggers this transition.

Any receive event:
all

Completion event

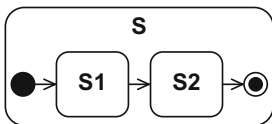
Figure 5.12

Transition with any receive event



5.5 Composite States

Composite state ...



... consists of substates

*Arbitrary nesting depth
of substates*

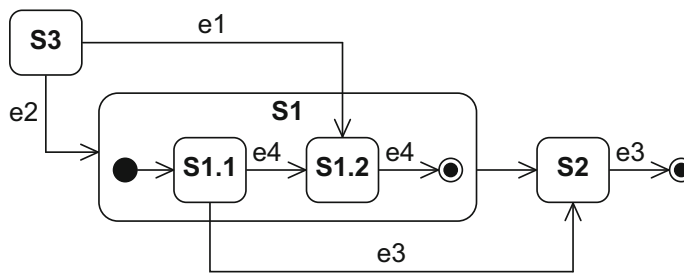
A simple state consists only of internal and external transitions and entry, do and exit activities, if there are any. It has no further substructure. A *composite state*, also referred to as a *complex state* or a *nested state*, is a state that contains multiple other states as well as pseudostates. The states contained within a composite state are referred to as its *substates*. A composite state can have an initial state. A transition to the boundary of this composite state can be understood as an implicit transition to the initial state of the composite state. If multiple states are nested within one another, that is, if a composite state contains further composite states, which in turn also contain further composite states, and so on, the life cycle of a new object always begins at the outermost initial state. The same applies for the final state. If a composite state has a final state, a transition that leads to this final state creates a completion event of the composite state in which the final state is located. Alternatively, transitions can lead to or away from a substate of a composite state.

Figure 5.13 shows examples of how a composite state can be entered or exited. If an object is in state S3 and event e2 occurs, composite state S1 becomes active and the initial state of S1 is entered. This triggers the immediate transition to state S1.1. However, if e1 occurs while the object is in S3, state S1.2 becomes active. If the object is in state S1.2 and e4 occurs, the object exits the higher level state S1, the assigned completion transition is executed, and the corresponding target state S2 is activated. However, if e3 occurs while the object is in state S1.1, the object immediately changes to state S2 and does not reach S1.2.

If e3 occurs while the object is in state S1.2, the system remains in S1.2 and the event is lost because it is neither consumed within S1.2, nor is the event specified on a transition originating from S1.2 or the states it is contained in.

Synonyms:

- *Composite state*
- *Complex state*
- *Nested state*

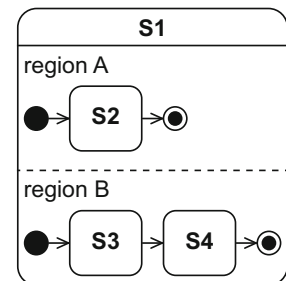
**Figure 5.13**

Example of the entry into and exit from a composite state

5.5.1 The Orthogonal State

If a composite state is active, only one of its substates is active at any point in time. If you want to achieve concurrent states, a composite state can be divided into two or more regions, whereby one state of each region is always active at any point in time. This type of composite state is called an *orthogonal state*. Each region of an orthogonal state can have an initial state. A transition to the boundary of the orthogonal state then activates the initial states of all regions. Each region can also have a final state. In this case, the completion event of the higher level state is not created until the final state is reached in all regions. If an orthogonal state is not to be entered or exited via its initial states and final states, the parallelization and synchronization nodes presented briefly in Section 5.2 are required. The incoming edge of the parallelization node may show events, guards, and activities, but at the outgoing edges, only activities are permitted. Every outgoing edge must target a substate of a different region of the same orthogonal state. Conversely, all edges that end in a synchronization node must originate from substates of different regions of the same orthogonal state. The outgoing edge of a synchronization node may show events, guards, and activities, but at the incoming edges, only activities are permitted.

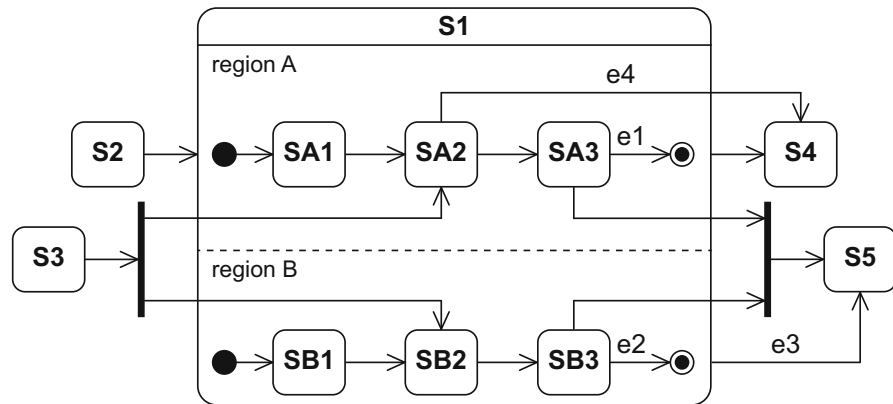
Figure 5.14 shows an example of how an orthogonal state can be entered or exited. If S1 is entered via state S2, the initial states of the two orthogonal regions region A and region B are activated. However, if S1 is entered via the transition that proceeds from S3, SA2 and SB2 are activated. There are four different ways of exiting state S1. If the final state has been reached in both regions, a completion event is created and there is a completion transition to S4 via the “bare” edge. If event e3 occurs while the object is in any substate of S1, any ongoing activities in S1 are terminated, the object exits all substates of S1, and there is an immediate transition to S5. If all activities in SA3 and SB3 were completed before events e1 and e2 occurred, there is a transition to S5. Event e4 offers the final opportunity to exit S1. If the system is in state SA2 and event e4 occurs, any ongoing activities in S1 are terminated,

Orthogonal state

the object exits all substates of S1, and there is a transition to state S4. This takes place regardless of which state of region B the object was in at the time the event e4 occurred.

Figure 5.14

Example of the entry into and exit from an orthogonal state



5.5.2 Submachines

If multiple state machine diagrams share parts with the same behavior, it is not practical to model the same behavior multiple times, because this would make the models difficult to maintain and reuse. In this situation, the recommendation is to reuse parts of state machine diagrams in other state machine diagrams. To do this, you model the behavior that is to be reused in a *submachine* accessed from another state machine diagram by a *submachine state*. A submachine is a special type of composite state. The name of the submachine state takes the form state:submachine state. In addition, you can optionally annotate the submachine state with a refinement symbol. If a submachine state is modeled in a state machine diagram, as soon as the submachine state is activated, the behavior of the submachine is executed. This is equivalent to calling a subroutine in programming languages. If there is a transition to the boundary of the submachine state, the initial state of the referenced submachine is activated. If a final state is reached in the submachine, the state of the calling state machine diagram that the transition from the submachine state leads to is activated. Figure 5.15 shows the states that a student can take when participating in a specific course, whereby the modeling of the states positive and negative has been transferred to the submachine grade, which is referenced in the state graded.

Submachine

Submachine state



Refinement symbol



Submachine \cong
subroutine

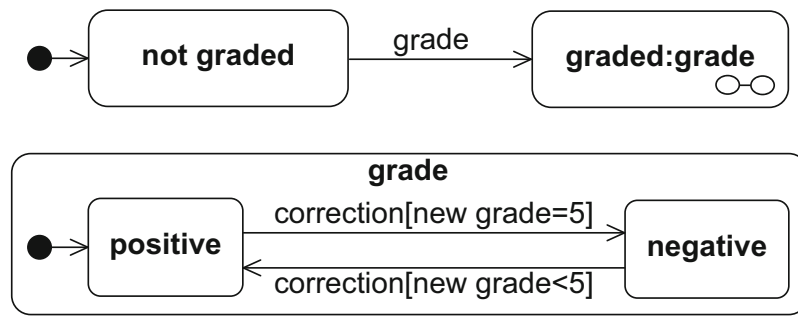


Figure 5.15
The states of a student in a course

5.5.3 Entry and Exit Points

If a composite state shall be entered or exited via a state other than the initial and final states, you can model this using *entry* and *exit points*. An entry point is modeled by a small circle at the boundary of the composite state and has a name that describes the entry point. The entry point has a transition to the state where the execution should begin. If an external transition leads to this entry point, the execution can be started with the desired state without the external transition having to know the structure of the composite state. If the composite state is not to be ended as usual when the final state is reached but instead through the ending of another state, you can model exit points in the same way. An exit point is denoted at the boundary of the composite state by a small circle containing an X and has a name that describes the exit point. If an external transition has the exit point as source state, this relates to the alternatively determined final state but without the external transition having to know the structure of the composite state. Entry and exit points are therefore a type of encapsulation mechanism. In practice, they are used in particular when modeling and using submachines.

Figure 5.16(a) shows a modification of the example from Figure 5.13. Instead of the transition leading directly to S1.2, an entry point is used. In the same way, S1.1 is exited via an exit point. Figure 5.16(b) shows the external view of S1. The entry and exit points are visible as interfaces to S1 but the detailed structure of S1 remains invisible for external transitions.

Entry point



Exit point



Composite state with entry and exit point

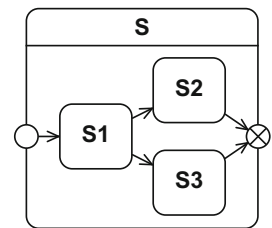


Figure 5.16

Example of entering and exiting a composite state with entry and exit points (see Fig. 5.13)

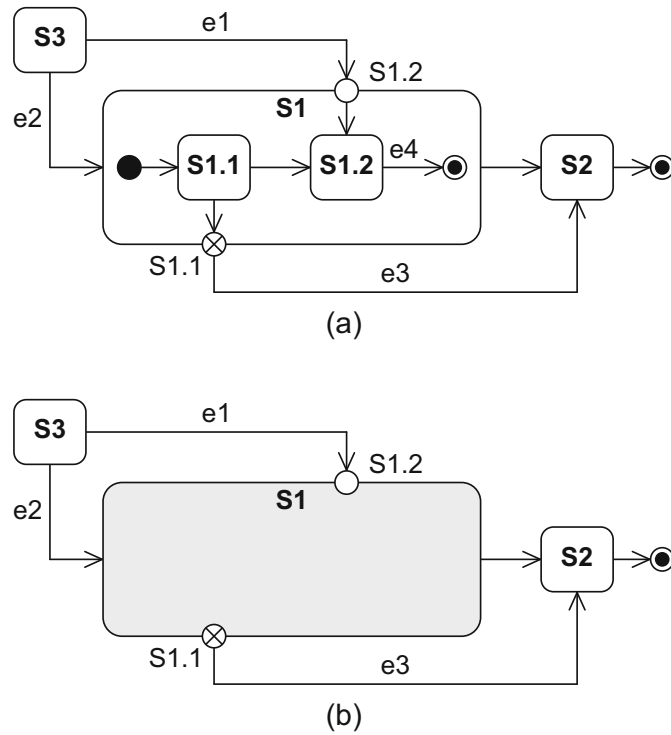
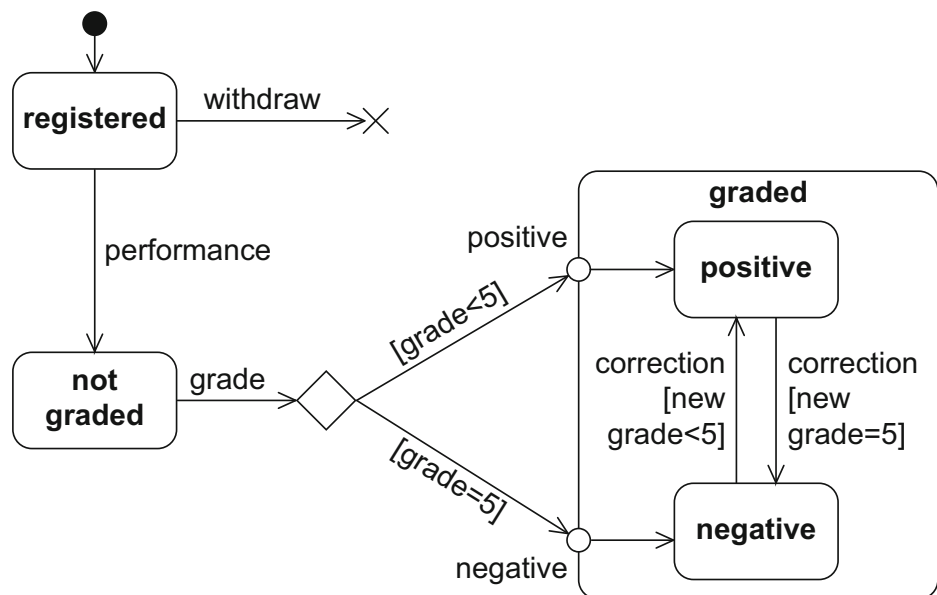


Figure 5.17 shows a modification of the example from Figure 5.8, in which the substates of the state graded are entered via entry points. Thus outside the state graded, no internal details of graded have to be known.

Figure 5.17

Modeling of the example from Figure 5.8 with entry points



5.5.4 The History State

History states are used when, after an external transition that leads away from a composite state, the system is to return to the same substate that was active before the transition occurred. The history state remembers which substate of a composite state was last active. If a transition from outside leads to the history state, the history state activates the “old” substate and all entry activities are conducted sequentially from the outside to the inside of the complex state. A history state can have any number of incoming edges but only one outgoing edge. The outgoing edge must not have any events or guards. Its target is the substate that is to be active if the composite state was never active before and there is therefore no “last active substate”, or if the composite state was recently exited in the “standard way” via the final state being reached.

There are two types of history states: the *shallow history state* and the *deep history state*. Every composite state may have a maximum of one shallow history state and one deep history state. The shallow history state restores the state that is on the same level of the composite state as the shallow history state itself. In contrast, the deep history state notes the last active substate over the entire nesting depth.

Figure 5.18 illustrates the difference between the shallow and the deep history states with an example.

History state

Shallow history state



Deep history state

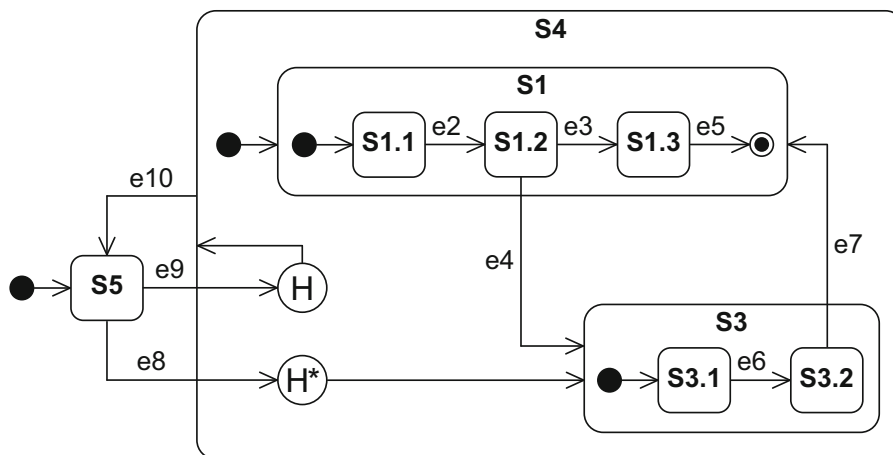


Figure 5.18
Shallow history state and
deep history state

Let us assume that the object is in state S1 and there in substate S1.2 when e10 occurs and forces a transition to S5. The shallow history state remembers that the object was previously in state S1 because S1 is on the same level as the shallow history state itself. In contrast, the deep history state remembers that the last active state was actually S1.2, as it knows the position over the entire nesting depth. If event e8 occurs, the deep history state activates state S1.2. However, if e9 occurs instead, the

shallow history state activates state S1 and, therefore, the initial state of S1 is active, which immediately activates the first real state, S1.1. Let us now assume that state S4 in the life cycle of our object has never been active and the object is currently in state S5. If e8 now occurs, the deep history state activates state S3 and thus implicitly S3.1, as the outgoing edge of the deep history state points to S3. If e9 occurs instead, the edge of the shallow history state points to the boundary of S4 and therefore the initial state of S4 is activated. In turn, this activates S1, which, via its initial state, activates the first real state, S1.1.

Figure 5.19 shows the states that a student takes during a study program. Initially, a study program is inactive. If the tuition fees have been paid (and thus the student has registered for the study program), the study program becomes active. Tuition fees must be paid at the beginning of every semester. If this does not happen, the study program becomes inactive again. During the course of an active study program, the student progresses through the levels bachelor, master, and doctorate. If the student does not pay the tuition fees for a particular semester—for example, because the student wants to take a break for one semester—after this semester, it should be possible for the student to return to the stage of the study program that was reached before the break. The deep history state ensures that this is possible.

5.6 Sequence of Events

In a final example, we will illustrate the relation of events, guards, and activities in states and in state transitions. Special attention is given to the order in which activities are executed.

Figure 5.20 shows an abstract example of a state machine diagram. Depending on which events occur, there are different state transitions. The variables x , y , and z are set to different values during the execution of certain activities. We will use the example to solve the following question: What state is the state machine in after the occurrences of the events e2, e1, e3, e4, e1, and e5 (in that given order) and what values are the variables x , y , and z set to?

At the beginning, the state machine is in state A, whereby before the entry into state A, the variable x was assigned the value 2. When the state machine enters state A, variable z is set to the value 0. The system now remains in state A until event e2—the first event in this specific example—occurs. As soon as e2 occurs, the state machine exits state A. When the state machine exits A, the value of z is increased by the value 1; z is therefore 1. There is a transition to state C. As part of this transition, the value of z is multiplied by 2; z is therefore 2. When

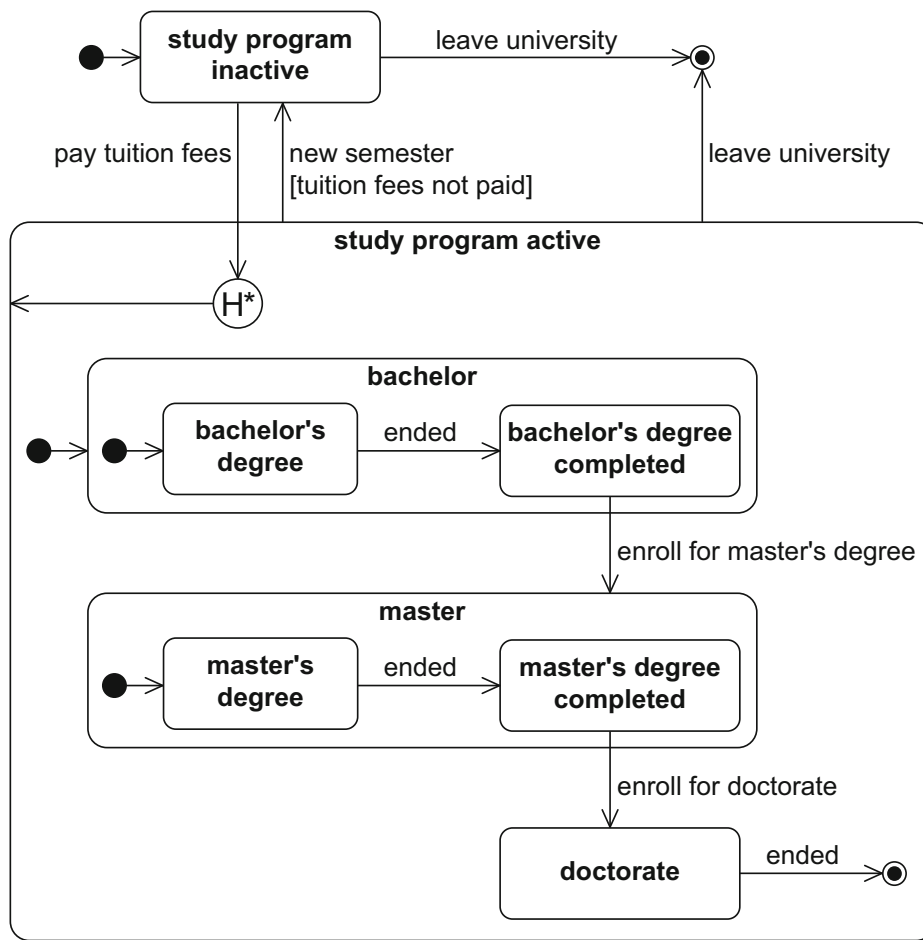


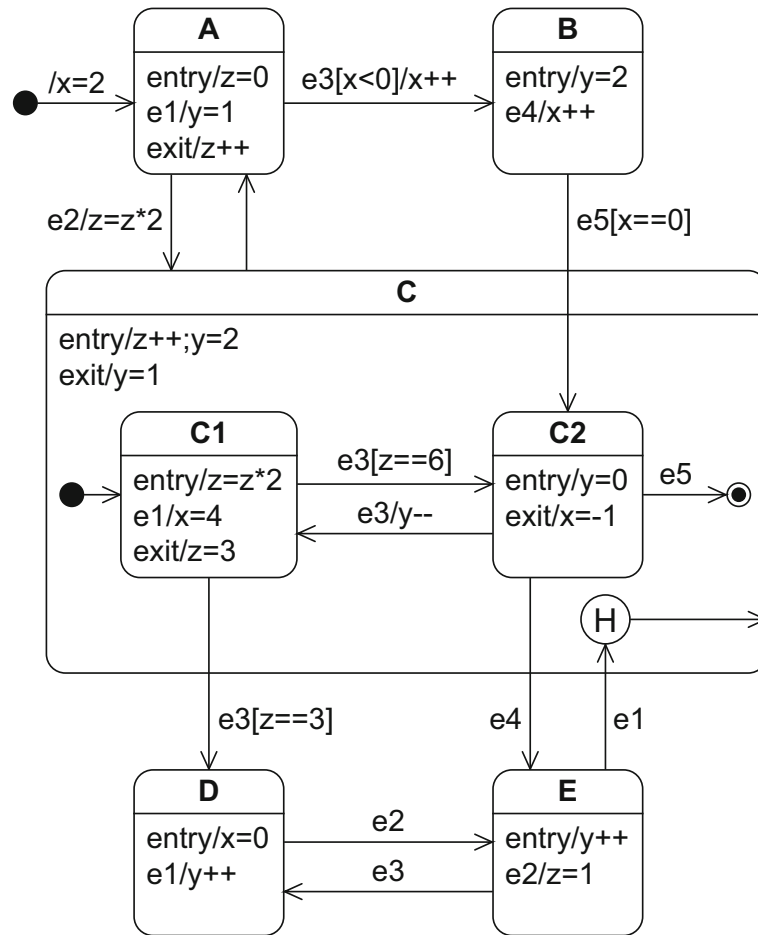
Figure 5.19
States of an academic education

the state machine enters composite state C, z is increased by 1 again and now has the value 3. Furthermore, y is set to the value 2. The initial state of composite state C leads directly to state C1; when the state machine enters C1, z is again multiplied by 2 and now has the value 6.

If e_1 now occurs, the state machine remains in state C1, as this event occurrence “only” triggers an internal transition and is processed within C1. The variable x is set to the value 4. Then e_3 occurs, and the system checks which value z has at this point in time. As z currently has the value 6, the guard $[z==6]$ is true. When the state machine exits C1, z is set to the value 3 and there is a state transition to state C2. When the state machine enters C2, y is set to 0. The next event in the sequence is e_4 , and therefore the state machine exits C2 and the exit activity of C2 is executed; x therefore becomes -1. The state machine then exits composite state C and this state’s exit activity is executed. The variable y is set to the value 1. When the state machine then enters E, y is increased by the value 1. The variable y therefore becomes 2. The occurrence of event e_1 makes the state machine exit state E. The history state returns to the last active substate of C, that is, to C2. As a result of the execution

Figure 5.20

State machine diagram to demonstrate a sequence of events



of the entry activities of C, the value of z increases from 3 to 4 and y is set to the value 2. The execution of the entry activity of C2 means that y is overwritten with the value 0. The last event in this example is $e5$. This event leads the state machine to the final state of composite state C. When the state machine exits C2, x is set to the value -1, which is irrelevant as x already has this value. There is an edge that leads away from state C where no event is specified at this edge. The completion event created by the ending of C thus leads to a completion transition to state A via this “empty” edge. When the state machine exits C, y is set to the value 1. When the state machine then enters A, z is set to 0. Therefore, after the events $e2$, $e1$, $e3$, $e4$, $e1$, and $e5$, our state machine is in state A, x has the value -1, y is 1, and z has the value 0. [Table 5.1](#) summarizes the individual steps.

Event	State entered	x	y	z
<i>Start</i>	A	2		0
e2	C1		2	6
e1	C1	4		
e3	C2		0	3
e4	E	-1	2	
e1	C2		0	4
e5	A	-1	1	0

Table 5.1

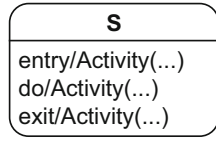
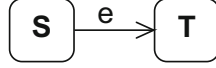



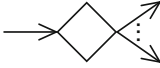
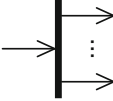
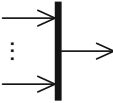

State changes and variable assignments for x , y , and z after the occurrence of the individual events

5.7 Summary

A state machine diagram can be used to show the states in which a system or an object can find itself during its “life cycle”, that is, from its creation to its destruction. The diagram also shows the conditions under which the transitions between these states occur. Events and activities triggered by these events can be modeled in the diagram. You can also specify guards that must apply for an event to trigger related activities or a state transition. Additional concepts allow you to model more complex state machine diagrams. Parallelization and synchronization nodes, as well as orthogonal states, enable you to model simultaneously active states and chains of states. The shallow and deep history states, as well as entry and exit points, allow a defined entry into transitive nested substates of composite states. The most important elements of the state machine diagram are summarized in [Table 5.2](#).

Table 5.2

Notation elements for the state machine diagram

<i>Name</i>	<i>Notation</i>	<i>Description</i>
State		Description of a specific “time span” in which an object finds itself during its “life cycle”. Within a state, activities can be executed on the object.
Transition		State transition e from a source state S to a target state T
Initial state		Start of a state machine diagram
Final state		End of a state machine diagram
Terminate node		Termination of an object’s state machine diagram
Decision node		Node from which multiple alternative transitions can proceed
Parallelization node		Splitting of a transition into multiple parallel transitions
Synchronization node		Merging of multiple parallel transitions into one transition
Shallow and deep history state		“Return address” to a substate or a nested substate of a composite state