Chapter 7. Designing state-based behavior: statecharts

7.1.   What is a statechart?

A *statechart* (an extended version of *state diagram*) is a visual construct that enables you to define event- and time-driven behavior of various objects (agents). Statecharts are very helpful in simulation modeling. They are used a lot in agent-based models, and also work well with process and system dynamics models.

Statecharts consist of *states* and *transitions*. A state can be considered as a "concentrated history" of the agent and also as a set of reactions to external events that determine the agent's future. The reactions in a particular state are defined by transitions exiting that state. Each transition has a *trigger*, such as a message arrival, a condition, a timeout, or the agent arrival to the destination. When a transition is taken ("fired") the state may change, and a new set of reactions may become active. State transition is atomic and instantaneous. Arbitrary actions can be associated with transitions and with entering and exiting states.

AnyLogic supports a version of UML statecharts, ("UML state machine", n.d.) which, in turn, is the adapted version of David Harel statecharts. AnyLogic statecharts have composite states (states that contain other states), history states, transition branching, and internal transitions. Orthogonal states are not supported, but you can define multiple statecharts for an agent that will work in parallel.

Example 7.1: A laptop running on a battery

Let us consider an example: a laptop running on a battery. When the laptop is on and the user is working, i.e. is pressing the keyboard keys and moving the mouse, the laptop stays in the **On** state, see Figure 7.1. However, after 5 minutes of user inactivity, the laptop turns off the screen to save power. This fragment of the laptop behavior is modeled by two transitions exiting the **On** state: a timeout transition **5 min** and a loop transition triggered by **Key or mouse** (although the external loop transition does not change the statechart state, it resets the timeout: 5 minutes are calculated since the last entry to the **On** state, i.e. last user action). If the user touches the keyboard or mouse while the screen is off, the screen turns back on, hence the **Key or mouse** transition from **Screen off** back to **On**. If, however, the user remains inactive for 10 minutes more, the laptop switches to the **Sleep** mode to further minimize the battery usage. This is modeled by the transition **10 min**. In the **Sleep** state, the laptop does not react to the mouse and the keyboard; therefore, there are no corresponding transitions from that state. To wake the laptop up again,

you need to press the power button ⏻, which triggers the transition from **Sleep** to **On**.
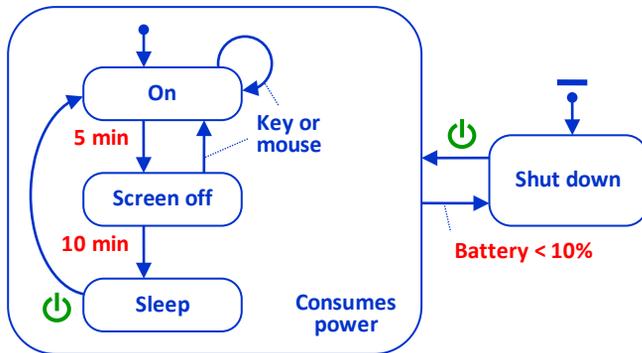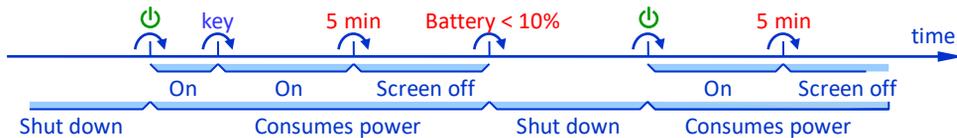

Figure 7.1 Statechart of laptop running on battery

In any of the three states (**On**, **Screen off,** or **Sleep)** the laptop consumes the battery power. When the battery level falls to 10% the laptop is forced to shut down, regardless of the state. How do we model this? We, of course, can draw three transitions (one from each state) triggered by a condition **Battery < 10%**, and this is what we would do if we were using a classical "flat" state machine. However, statecharts offer a much better way to define such "interrupt-like" reactions common to a group of states: you can draw a *composite state* around the group and define reaction at the composite state level. The transition **Battery < 10%** exiting the composite state **Consumes power** will bring the statechart to the **Shut down** state no matter in which of the three inner *simple states* it occurs.

In the **Shut down** state the laptop will only react to the power button, so there is only one transition from there to **Consumes power**. As the latter is a composite state, we need to specify the initial simple inner state where the statechart gets after entering the composite state. In our case this is **On**, so **On** is marked with the *initial state pointer*. Similarly, we need to mark the initial state at the topmost level of the statechart, either **Consumes power** or **Shut down**. You can see the *statechart entry point* that points to the **Shut down** state.

How do statecharts differ from flowcharts and stock and flow diagrams?

In simulation modeling in general, and in AnyLogic in particular, we use other diagrams that look similar to statecharts (have boxes connected with arrows), but have different meaning, or semantics. For instance: process flowcharts, action charts, and stock and flow diagrams (see Section 5.1). It is worth considering the difference between these diagrams:

Statecharts define internal states, reactions to external events, and the corresponding state transitions of a *particular agent*: a person, a physical device, an organization, a project, etc. The (simple) states of the statechart are *alternative*: at any given moment in time the statechart is in *exactly one simple state*.

Process flowcharts (composed of the Process Modeling Library blocks) define sequences of operations performed over agents. During the simulation there may be multiple agents in different places of a flowchart, so the "state" of the flowchart is actually spread across the whole diagram.

Stock and flow diagrams are a fundamental part of system dynamics (see Chapter 5) models. Stocks are accumulations and the state of the system is defined by the values of all stocks. Compared to statecharts, all stocks and all flows are simultaneously active.

Action charts are a graphical way to define algorithms or calculations. The arrows define the control flow for a calculation. The action chart does not persist in time: it executes logically instantaneously from start to end, and it does not have a notion of state. Action chart can be considered as a graphical form of a Java function.

7.2.   Drawing statecharts

The statechart building elements are located in the **Statechart** palette, see Figure 7.2. Since statecharts are a frequent tool for defining an agent behavior, the whole set of elements is also duplicated in the **Statechart** section of the **Agent** palette.

You can create statecharts by dragging the elements to the graphical editor. States and transitions also support drawing mode (explained below). While you draw a statechart, its structure is permanently validated, and inconsistencies found are displayed in the **Problems** view.
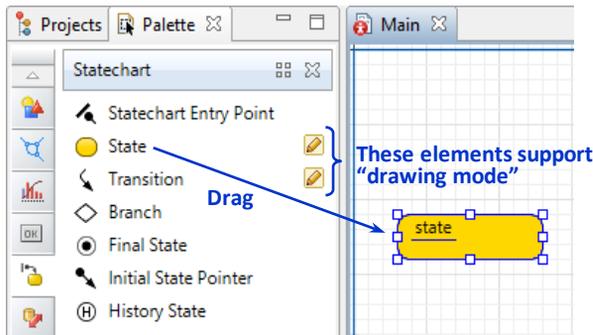
Figure 7.2 The Statechart palette

Simple states

Drawing a statechart typically starts with drawing states, giving them names, and adjusting their sizes and colors.

▶ **To create a state in the drawing mode:**
1. Double-click the **State** element in the **Statechart** palette or double-click the pencil icon on its right-hand side.
2. Press the mouse button in the graphical editor where the upper left point of the state should be placed and hold the button down.
3. Drag the cursor to the bottom right point of the state.
4. Release the mouse button.

The in-place editor of the state name opens automatically just after you create a state. And, of course, you can change the state name any time later.

▶ **To change the state name:**
1. Double-click the state name in the graphical editor.
2. Type the new name in the in-place text editor.

Alternatively, you can change names of statechart elements in the properties view. The position of the state name relative to the state can be adjusted.

▶ **To adjust the position of the state name:**
1. Select the state by clicking it.
2. Drag the state name.

The statechart can be made more informative if you paint its states with different colors.

▶ **To change the color of the state:**
1. Select the state.
2. Choose a new color in the **Fill color** property of the state.

Transitions

You can drag the **Transition** element to the graphical editor, connect it to states, and edit its shape, or you can draw a transition point by point in the "drawing mode".

▶ **To draw a transition point by point (in the drawing mode):**

1. Double-click the **Transition** element in the **Statechart** palette.
2. Click in the graphical editor at the position of the first point of the transition.
3. Continue clicking at all but the last points.
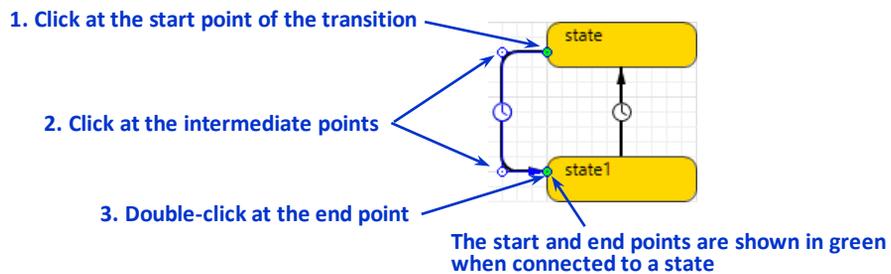4. Double-click at the last point position.

1. Click at the start point of the transition

2. Click at the intermediate points

3. Double-click at the end point

The start and end points are shown in green when connected to a state

Figure 7.3 Drawing a transition point by point

After a transition has been created, you can edit its shape by moving, adding, or removing points.

▶ **To edit the transition shape:**

1. Select the transition.
2. To add a new point, double-click the transition where the point must be added.
3. To delete a point, double-click it.
4. To move a point, drag it to the new position.

A transition gets automatically connected to a state if its start or end points lie on the state boundary. You can check if the transition is connected by selecting the transition and looking at the start and end points: when connected, the point is highlighted with green. If a point is not connected to a state, the transition is highlighted with red, and a "Hanging transition" or "Element is not reachable" error is displayed in the **Problems** view. You can double-click that error message and the transition will be displayed and selected. To fix the error drag the transition point to the state boundary.

The icon on the transition shows the transition's trigger type (🕐 timeout, ◣ rate, ⑦ condition, ✉ message, ⊞ arrival), see Section 7.3, "State transitions: triggers, guards, and actions".

The position of the icon can be adjusted.

▶ **To adjust the position of the transition's trigger icon:**
1. Select the transition by clicking it.
2. Drag the trigger icon.

By default, the transition name is not displayed in the graphical editor and at runtime. You can change this.

▶ **To display the transition name:**
1. Select the transition.
2. Check the **Show name** checkbox in the transition properties.

In most cases you would need to adjust the position of the transition name. To do this, drag the name while the transition is selected.

Statechart entry point

Each statechart *must have exactly one* statechart entry point – an arrow that points to the initial top-level state (a top-level state is a state that is not contained in any composite state). You may have noticed that until you add the entry point, the error "Element is not reachable" is displayed for all statechart elements. The errors will disappear once you create an entry point.
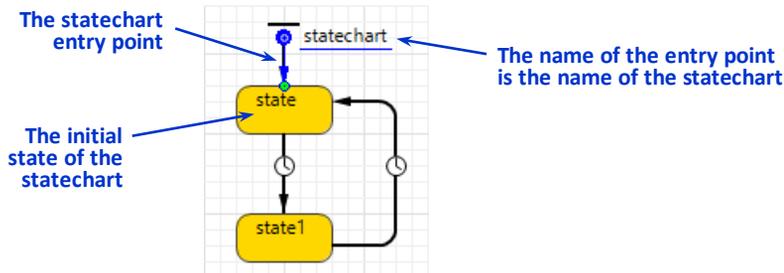

Figure 7.4 The statechart entry point

▶ **To create a statechart entry point:**
1. Drag the **Statechart Entry Point** from the **Statechart** palette to the graphical editor.
2. Drag the end point of the arrow to the boundary of the initial top-level state. The end point will show in green when connected.

The name of the statechart entry point is the name of the statechart.

Do not confuse the statechart entry point with the initial state pointer. The latter is used to mark the initial state within a composite state.
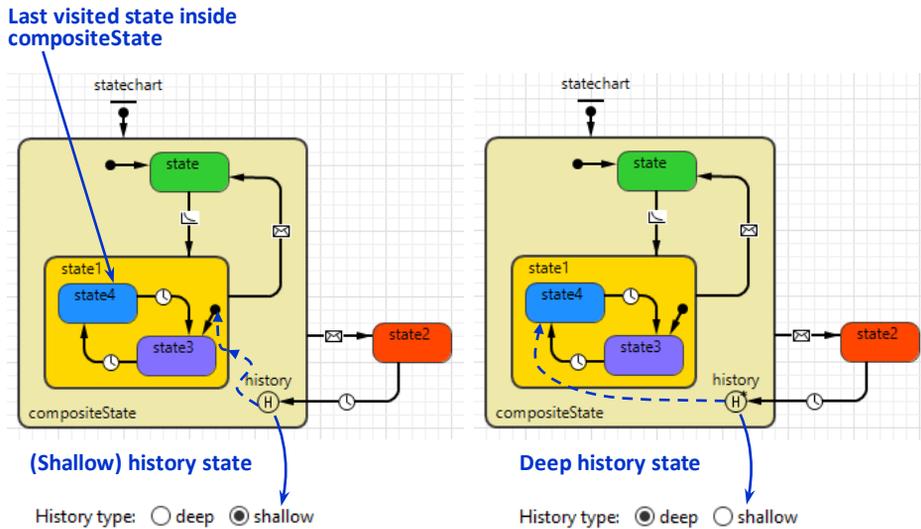
Composite states

A composite state is created in the same way as a simple state. You can drag the **State** to the graphical editor and then resize it so that it contains other states, or you can draw it around other states in the drawing mode. The graphical editor prevents you from drawing intersecting states, so the state structure is always strictly hierarchical.

Figure 7.5 Composite state

When a state becomes composite, its fill color is set to transparent. You can change it by setting the **Fill color** field in the state properties. In most cases (namely, each time there is a transition or a pointer pointing to the composite state), you need to identify the default initial state inside the composite state, i.e. one level down in the state hierarchy. This is done by the **Initial State Pointer** that can be found in the same **Statechart** palette.

Transitions may freely cross the composite state boundary both inbound and outbound. An inbound transition can lead to an internal state other than the default initial state (see Figure 7.5).

History state

History state is a pseudo-state – it is a reference to the *last visited state inside a composite state*. History states are used to model a return to the previously interrupted activity.

Figure 7.6 History state

History state can be located only inside a composite state. There are two types of history states: *shallow* and *deep*. Shallow history state remembers the last visited state on the same level of hierarchy, and deep history state remembers the last visited simple state, which may be at a deeper level. You can switch between shallow and deep in the history state properties.

Consider the statechart in Figure 7.6. Assume the statechart was in **state4** when the transition took it to **state2**. The transition from **state2** points to the history state; therefore, when entering the **compositeState** that way the statechart will return to the last visited state within the **compositeState**. If the history is shallow (on the left) it refers to **state1**, which is on the same level as the history state. Therefore, the statechart will enter **state1** and then go to the default state inside **state1** (**state3**). If the history is deep, the statechart will fully restore its state on the moment of leaving the **compositeState** (returning to **state4**).

Final state

Final state is a state that terminates the statechart activity. Final state may be located anywhere – inside composite states or at the top level. There may be any number of final states in a statechart. A final state may not have any outgoing transitions.

Consider the statechart in Figure 7.7. The final state is inside **state1**, and upon entering it, the statechart deactivates all transitions that were active.
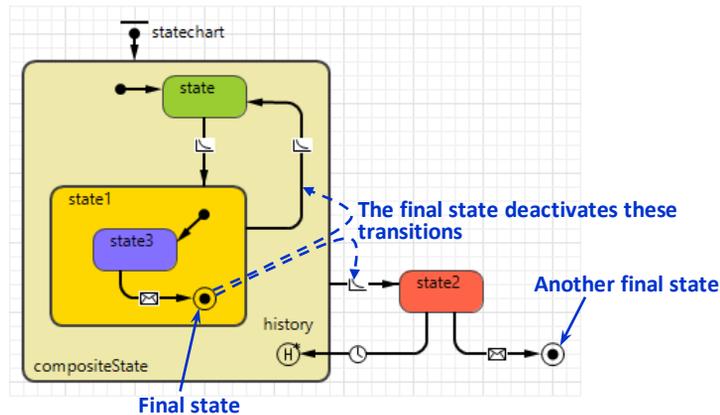
Figure 7.7 Final state

7.3.   State transitions: triggers, guards, and actions

Which transitions are active?

As you know, the transitions from the current state of the statechart define how the statechart will react to the external events and conditions. Consider Figure 7.8.
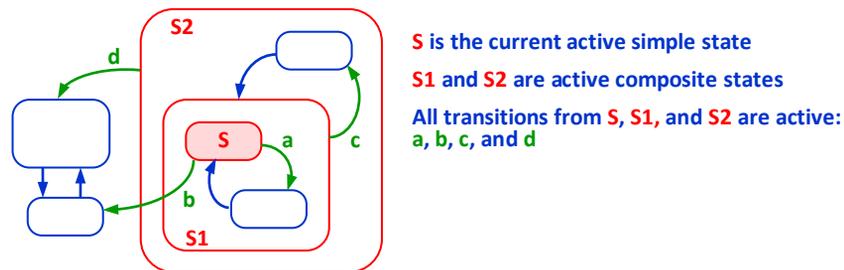

S is the current active simple state

S1 and S2 are active composite states

All transitions from S, S1, and S2 are active: a, b, c, and d

Figure 7.8 Active states and active transitions

At any moment during the statechart lifetime there is exactly one current (or *active*) simple state. In our case, it is **S**. **S** is located inside the composite state **S1**, which, in turn, is inside **S2**. Both **S1** and **S2** are also active states.

You can test whether a state is active by calling the statechart function **isStateActive(<state name>)** or the agent function **inState(<agent type name>.<state name>)**. You can find out the current simple state by calling the statechart function **getActiveSimpleState()**.

All transitions exiting any active state are *active* (they may be triggered). In the case of the statechart in Figure 7.8, these are transitions **a**, **b**, **c**, and **d**.

Trigger types

In the Table below we explain all possible ways to trigger a statechart transition, what different trigger types are used for, and how they work.

Trigger type	Primary use	Transition fires
Timeout ⏱	**Timeout**: change state if other awaited events do not occur within the specified time interval. **Delay**: stay in a state for a given time, then leave.	After a specified time interval counted from the moment the statechart enters the direct "source" state of the transition (i.e. the state on whose boundary the transition start point is located). The timeout expression is calculated one time when the statechart enters that state. The expression can be stochastic as well as deterministic.
Rate ⌐	Sporadic state change with a known mean time. In agent-based models used to represent sporadic decisions made by an agent under a certain, possibly variable, influence (purchase decisions, adoption of ideas, etc.).	Same as timeout, but the time interval is drawn from an exponential distribution parameterized with the given rate. For example, if the rate is 0.2 per hour the timeouts will have mean values of $1/0.2 = 5$ hours. If a change occurs in the agent while the rate transition is active (namely, if **onChange()** function is called), the rate expression is re-evaluated and, if it gives a new result, the transition is rescheduled using a new exponential distribution.
Condition ⑦	Monitor a condition and react when it becomes true. For example: buy if the stock price falls below a certain threshold; launch a missile if the aircraft is closer than 5 miles, etc.	Once a given condition becomes true. The condition is an arbitrary boolean expression and may depend on the states of any elements in the whole model with continuous as well as discrete dynamics. In most cases you can assume the condition is constantly monitored while the transition is active. For more information see Section 8.2.

Message ✉	React to messages received by the statechart or by the agent from outside. The messages can model communication between people or organizations, commands given to a machine, physical products, electronic messages, etc.	Upon reception of a message that matches the template specified in the transition properties. The ways of sending a message to a statechart as well as the rules of message processing are described below.
Arrival ⚑	React to arrival. Can be used in agents moving in continuous or GIS space.	When the agent arrives at the destination point (assuming its movement was initiated by **MoveTo** block of the Process Modeling Library or by calling agent's **moveTo()** function). This is implemented as a message of a special type sent to the statechart by calling the function **fireEvent()**.

Timeout expressions

The expression in the **Timeout** field of the transitions triggered by timeouts is interpreted as time interval in given time units. You set the time units using the drop-down list to the right of the field: milliseconds, seconds, minutes, etc.

Please note that choosing the timeout time units larger than a week may lead to the transition occurrence on a wrong date. If you choose **months**, the month duration will always be 30 days, regardless of the actual month duration, which can be 28, 29, 30 or 31 days. For example, the timeout of **3 months** will occur in 90 days. Timeouts specified in **years** are calculated in the same way: 1 year always means 365 days.

For timeouts measured in time units larger than a week, and also for correct handling of daylight saving time (see Section 16.2), you should use the function:

    double toTimeout(int timeUnit, int amount)

where **timeUnit** can be **YEAR** and **MONTH** as well as **WEEK**, **DAY**, etc. This function returns the time interval between the current time and the time in the future, which is the **amount** of given time units later.

To use this technique, you must set the same timeout time units that you have chosen as your model time units in the model properties.

If you wish a transition to fire *at the same time and date but 15 years later than the statechart comes to a state*, the transition timeout expression should be:

   toTimeout(YEAR, 15)

Sometimes you want the transition to fire *15 years later than a given date* (typically in the past) that you stored in the **lastDate** variable of type **Date**. In this case, the timeout expression should be:

   dateToTime(lastDate) + toTimeout(YEAR, 15) – time()

Here we first convert the **lastDate** into model time, then add 15 years timeout and subtract the current time (remember that the expression is calculated when the statechart enters the transition's source state). If you want the timeout transition to fire *exactly at 9AM on the nearest Monday,* you can create and use the function described in Section 16.2:

   timeOnNearestDayOfWeek(MONDAY, 9, 0, 0) - time()

The same applies to timeout events (see Section 8.2).

More information about the model time, time units, and usage of calendar can be found in Chapter 16, "Model time, date and calendar. Virtual and real time".

Transitions triggered by messages

You can send messages to a statechart, and the statechart can react on message arrival. A message can be an arbitrary object: a string, a number, an **Agent**, etc. You can set up the transition to check the message type and content. Several examples of transition triggered by a message are given in Figure 7.9.

Transition triggered by the option FIRE of the option list AlarmType

Triggered by:	Message ⌄
Message type:	AlarmType ⌄
Fire transition:	○ Unconditionally
	◉ On particular message
	○ If expression is true
Message:	FIRE

Transition triggered by the string "Will you marry me?"

Triggered by:	Message ⌄
Message type:	String ⌄
Fire transition:	○ Unconditionally
	◉ On particular message
	○ If expression is true
Message:	"Will you marry me?"

Transition triggered by an integer that is greater than 100

Triggered by:	Message ⌄
Message type:	int ⌄
Fire transition:	○ Unconditionally
	○ On particular message
	◉ If expression is true
Expression:	msg > 100

Transition triggered by any message at all

Triggered by:	Message ⌄
Message type:	Object ⌄
Fire transition:	◉ Unconditionally
	○ On particular message
	○ If expression is true
Action:	

Transition triggered by an agent whose name starts with "Company"

Triggered by:	Message ⌄
Message type:	Agent ⌄
Fire transition:	○ Unconditionally
	○ On particular message
	◉ If expression is true
Expression:	msg.getName().startsWith("Company")

Figure 7.9 Examples of transitions triggered by messages

The message type is checked first. The properties of the transition allow specifying of several standard types (numeric – **int** or **double**, **String**, **boolean**) or a custom type in the field **Class name**. **Object** is the most general type; if **Object** is specified, any message will pass the type filter. After the type check, the transition is triggered by the message either unconditionally, or if it equals the given object, or if a given expression evaluates to true. In the expression, you can refer to the message as **msg** (see the last example in Figure 7.9). Some recommendations on using the message types are given in the Table below.

Message type	Primary use
Option list	There is a finite number of commands or notifications you wish to send to a statechart. For such messages, you would use an option list with several options (you create it in the **Projects** view, via the **New** item of the context menu). For example, a statechart that models a coffee machine may react to the following option set defined with a **CoffeeType** option list: **ESPRESSO**, **CAPPUCCINO**, **LATTE**.
String	You need to send a textual information and the messages do not carry any additional data. For example, a statechart that models a trader may react to the following message: **"Sell your stock!"**.
Numeric (**int** or **double**)	You need to notify a statechart about a certain quantity and the statechart always knows what this quantity means. For example, a statechart that models a behavior of a broker who watches the price of a particular stock can receive price updates in the form of double type messages.
Object of custom type	Messages contain several pieces of information for example: the sender's address, the destination address, the command ID, the parameters, etc. For such messages, you would typically define a new agent type with several fields.

Sending messages to a statechart

A message can be sent to a statechart by calling either the function **fireEvent()** or the function **receiveMessage()**; the ways the messages are then processed by the statechart are different.

If **receiveMessage(<message>)** is called, the statechart looks through its currently active transitions and, if the message matches a transition trigger, the transition is scheduled for execution. If the message matches more than one transition, all matched transitions will be scheduled, but later on firing of one transition may cancel the others. If the message does not match any of the triggers, it is discarded.

If the message was sent to a statechart by **fireEvent(<message>)**, it is added to the *message queue* and, at the same time, the statechart tries to match each active transition with *each* message in the queue. If a match is found, the message is consumed by the transition and all messages before it are discarded. The queue does not persist in time: *if the model time is advanced, all unconsumed messages are lost*. However, the statechart may make several steps taking zero time reacting to several messages. Figure 7.10 shows how the message queue is handled. Initially, all messages came to the statechart at the same model time **t**. The messages **x** and **y** were discarded because they did not match any transition. The message **c** was discarded because non-zero time elapsed since it has been added to the queue.
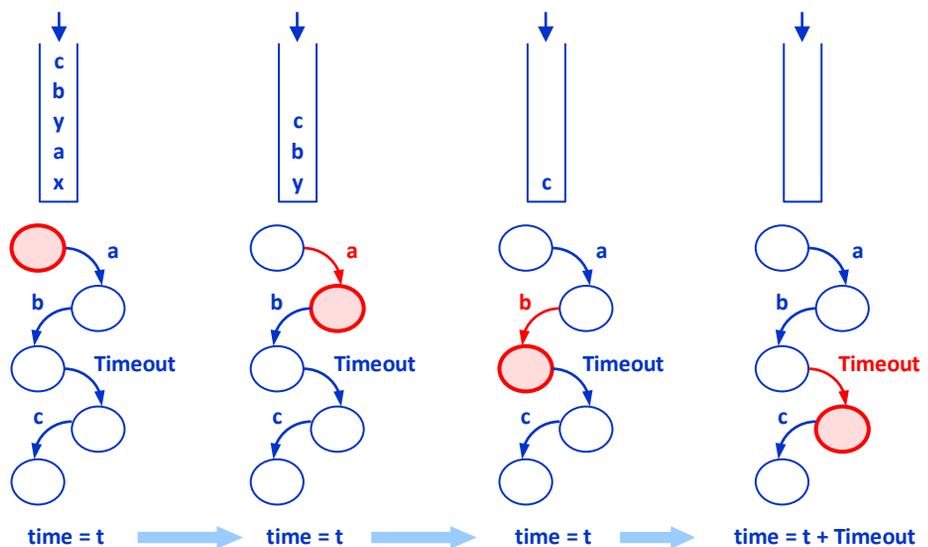
Figure 7.10 Statechart message queue

The messages received by an agent (see Section 3.8) can be forwarded to its statecharts. When forwarding, the agent uses the **fireEvent()** function.

▶ **To forward the agent's incoming messages to statechart(s):**

1. Open the graphical diagram of the agent and pan it down by right-dragging until you see the **connections** element, see Figure 7.11. The element defines how the messages received by the agent are processed.
2. Select the **connections** element and expand the **Communication** section of its properties.
3. In the **Forward message to** list, select the statecharts where you wish to forward the messages.
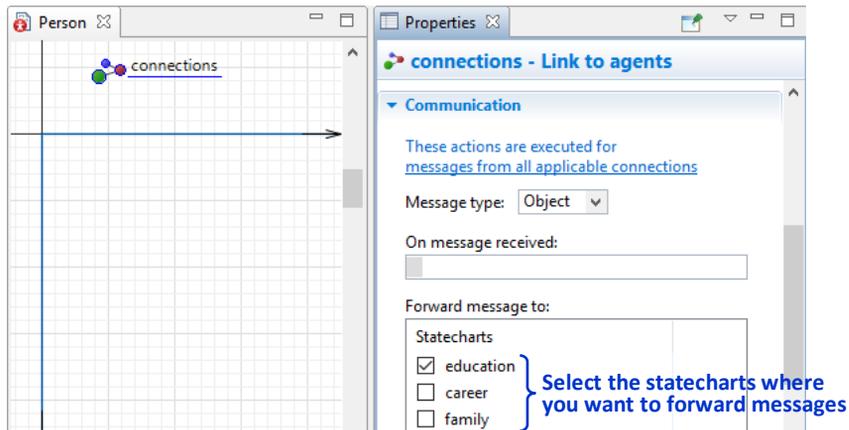
Figure 7.11 Forwarding the agent's incoming messages to statecharts

Guards of transitions

When the transition is ready to fire, it performs one final check – it checks its *guard* and, if the guard evaluates to **false**, the transition is *not taken*. Thus, guards can impose additional conditions on state transitions. Guard is an arbitrary Boolean expression and can refer to any element in the model.

Consider an example: a car engine that would not start if the gearbox is not in the **Parked** position. The model of the car can have two statecharts (see Figure 7.12): one for the engine and another for the gearbox. The transition **start** of the **engine** statechart triggered by the **"Start/Stop"** message will only be taken if the **gearbox** statechart is in the **Parked** state.
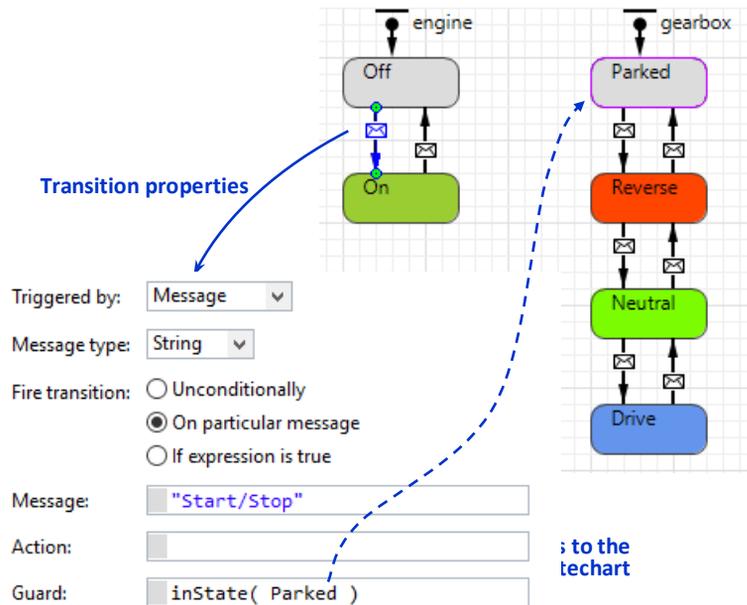
Figure 7.12 Using guards: car engine state transition depends on the gearbox state

If the transition was ready to fire but did not fire because the guard evaluated to **false**, then:

- **Timeout** transitions get deactivated
- **Rate** transitions get deactivated until the rate is changed
- **Condition** transitions continue monitoring the condition
- Transitions triggered by a **message** continue waiting for another message
- Transitions triggered by **arrival** continue waiting for the next arrival

Do not confuse guards and conditions. The condition that triggers the transition is constantly monitored while the transition is active. The guard is only checked when the transition is ready to fire, i.e. when its trigger is available.

Transitions with branches

A transition may have *branches,* and can bring the statechart to different states depending on conditions. Those conditions are evaluated, and the target state is chosen immediately after the transition has been triggered and its guard evaluated to **true**, so firing of a transition with branches can be considered as instant and atomic.

The number and configuration of decision points and branches can be arbitrary. Each decision point is created using the **Branch** element from the **Statechart** palette.

The branches after the decision points are created and edited in the same way as transitions, but their properties are different. Each decision point *must have exactly one default branch*. The **Default** branch is drawn with a dashed line. Other branches exiting a decision point should be set as **Conditional**. The conditions of conditional branches should be *complete and unambiguous*. If during the transition firing two or more conditions evaluate to **true**, one of the corresponding branches is chosen nondeterministically. The default branch is taken if none of the conditions evaluate to **true**.

The statechart in Figure 7.13 is set up to test a number that it receives as a message. The transition from **WaitNumber** state is triggered by a message of type **int**. The transition branches into three: one for negative numbers, one for zero, and the third one for everything else (default). Obviously, a positive number will cause the statechart to follow the third branch, which, in turn, has two exits: one for even numbers and another (default) – for odd. The expression **X % 2** returns a remainder of **X** divided by **2** (see Section 10.5, "Expressions").
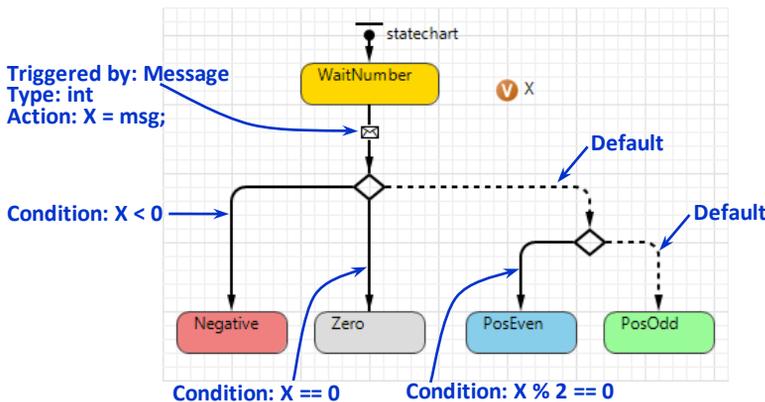
Figure 7.13 A transition with two decision points and five branches

Frequently, when a transition triggered by a message has branches, the conditions of the branches depend on the message content. As the message received is not available outside the first section of the transition where the trigger is defined, you may need to save the message in a variable, as in the example above.

Internal transitions

With the help of *internal transitions* you can define some activity the statechart performs while being in a state without exiting that state. Consider a cell phone user who is a frequent traveler. While abroad, a person makes phone calls using roaming, and the frequency of those calls is less than the frequency of calls at the home location. We can model this behavior as a statechart with two states: **AtHome**

and **Abroad**. Let the person spend from 2 to 5 weeks at home and from 3 to 7 days abroad. This is modeled by the timeout transitions **flyOut** and **flyBack** that limit the time the user stays in each of the two states.
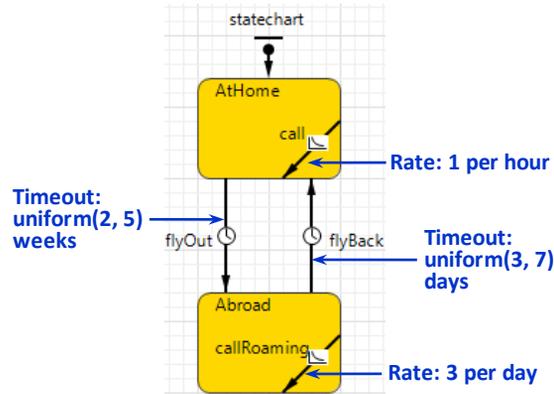

Figure 7.14 A statechart of a cell phone user with internal transitions

At home, the person makes on average of one call per hour. We can model this by creating an internal transition **call,** that fires at the rate **1 per hour**. Graphically, the transition **call** is *entirely inside* the state **AtHome**, which tells AnyLogic th at the transition should be treated as internal and its firing does not cause exit and re-enter of the state **AtHome**. Therefore, the 2- to 5-week timeout is *not reset* by the phone calls. Similarly, the **callRoaming** transition does not affect the duration of the trip abroad defined by the transition **flyBack**.

> The transition is counted as internal if it exits and enters the same state and all its intermediate points are inside that state.
>
> Firing of an internal transition interferes neither with other internal transitions nor with other state transitions that may be defined inside the same state.

Order of action execution

All statechart elements (states, transitions, decision points, transition branches, etc.) may have actions associated with them. The actions are executed as control is passed from one state to another along a transition. To understand the *order of action execution,* consider the statechart in Figure 7.15.
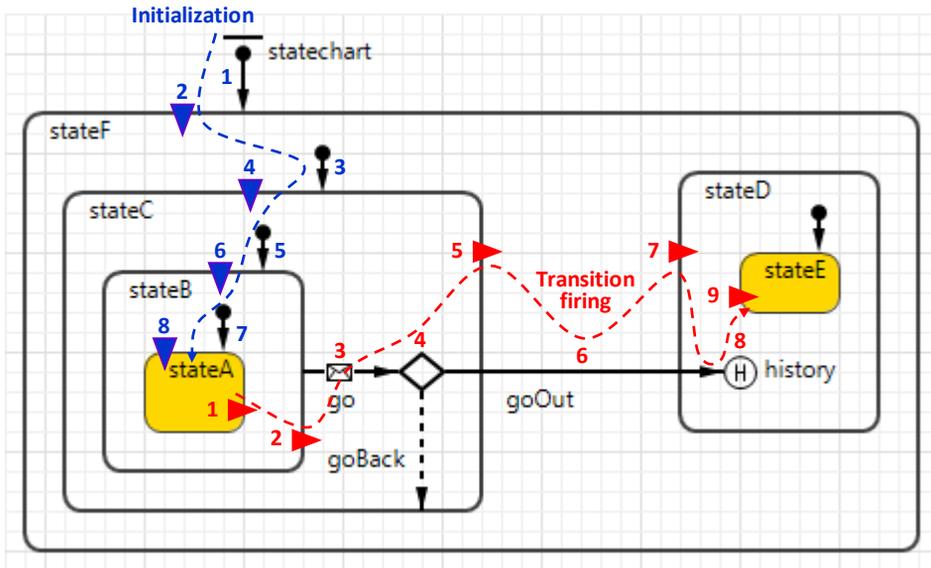
Figure 7.15 The order of action execution in a statechart

During the initialization, the statechart executes first the action of its entry point and then entry actions of the states and actions of initial state pointers down the state hierarchy all the way to the initial simple state (see the initialization path and action numbering in Figure 7.15).

During the transition firing, the statechart first exits the current simple state (in our case **stateA**), then all states up the state hierarchy to the transition source state (**stateB**). After that, the action of the transition is executed, and then the statechart enters the transition target state (in our case this is a **history** state, which points to **stateE**). If the target state is composite, the statechart follows the initial state pointers down to the next simple state. If the transition has decision points and branches, their actions are executed in the natural graphical order (i.e., they can be mixed with exit and entry actions of states (see the action numbering in Figure 7.15). If the transition with its source and target states is fully contained in a composite state (like **stateF**), that state and all states above it are not considered as exited or entered.

Synchronous vs. asynchronous transitions

Subject to the input data available, sometimes it makes sense to use *synchronous state transitions* (i.e., the transitions where the decision to change state is made in several attempts at regular time intervals - on "ticks"). Consider a model of alcohol use by an individual. Under a certain age, the person does not use alcohol at all, then

starts drinking. The input data may be in the form of a table where initiation probability is provided for each age, see the Table below.

Age	Probability of alcohol initiation	Age	Probability of alcohol initiation
11	0.00	18	0.23
12	0.01	19	0.17
13	0.05	20	0.11
14	0.05	21	0.05
15	0.12	22	0.03
16	0.14	23	0.01
17	0.18	24	0.00

Suppose in the model there is a statechart with two states: **NeverUser** and **User**. How do we design a transition from one to another? Obviously, this can be a timeout transition with time drawn from a certain distribution. We do not have that distribution explicitly and could construct it from the data we have (which might be an interesting analytical exercise). However, instead we can create a synchronous transition and use the input data directly.

In Figure 7.16 the transition exiting the state **NeverUser** is taken every year (assuming the time unit (see Section 16.1) is year). Then the decision is made: go to the state **User** with the probability taken from the table function according to the current age, otherwise return to **NeverUser**.

The age in this model equals the model time as the statechart is created at time 0 and the time unit is year, hence **time()** is provided as an argument when accessing the table function. In general, this may be not true (for example, if the statechart belongs to an agent that may be created in the middle of the model runtime). In that case, you should remember the year the agent was born in the agent's local variable and calculate the age as **time() - <year born>**.
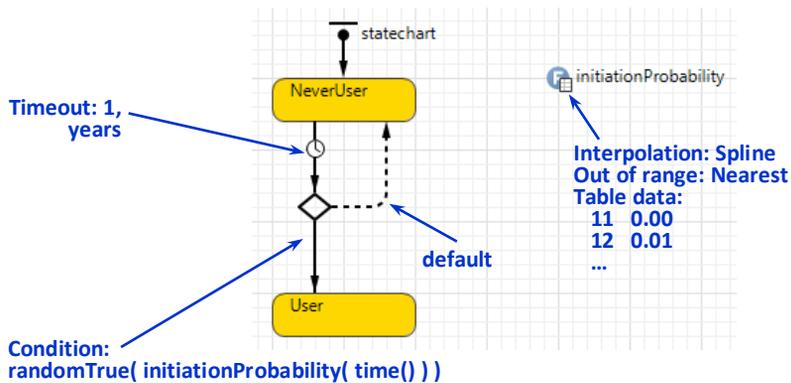
Figure 7.16 Alcohol initiation modeled by a synchronous state transition

The table function is set up in such a way that it returns the nearest value when the argument is out of range. Therefore, for ages under 11 and ages over 24, the initiation probability is zero.

7.4.   Statechart-related API

The API related to statecharts is contained in the classes **Statechart**, **Agent**, and **Transition**, and also in the interface **IStatechartState**.

These two functions send messages to the statechart. They are discussed in detail in Section 7.3.

- **boolean receiveMessage(Object msg)** – posts a message to the statechart without putting it to the queue. Returns **true** if the message matched the trigger of at least one transition, otherwise returns **false**.
- **fireEvent(Object msg)** – adds a message to the statechart queue.

To find out the current state of the statechart, use these functions:

- **boolean isStateActive(IStatechartState state)** – returns **true** if a given state (simple or composite) is active, otherwise returns **false**.
- **IStatechartState getState()** – returns the currently active simple state of the statechart.
- **Set<IStatechartState> getFullState()** – returns the currently active composite states of the statechart, including the current simple state. The returned set is unordered.

You can ask the agent where the statechart belongs (not the statechart itself!), whether the given state is currently active:

- **boolean inState(IStatechartState state)** – returns **true** if the agent is in the specified state, i.e. exactly in the state for a simple state and in one of its inner states for a composite state.

The statechart transitions expose these functions:

- **boolean isActive()** – returns **true** if the transition firing is scheduled, and **false** otherwise.
- **double getRest()** – returns the time in the model time units remaining to the scheduled transition firing, or **infinity** if the firing is not scheduled.
- **double getRest(TimeUnits units)** – returns the time in given time units remaining to the scheduled transition firing. Use one of the constants **YEAR**, **MONTH**, **WEEK**, **DAY**, **HOUR**, **MINUTE**, **SECOND**, **MILLISECOND** as the **units**.

The states provide the following functions:

- **boolean isSimpleState()** – returns **true** if this state is a simple state, and **false** if it is a composite state.
- **Set<IStatechartState> getStatesInside()** – returns unordered set of all, simple and composite, states inside this state.
- **boolean containsState(IStatechartState simpleState)** – returns **true** if this state is composite and it contains the **simpleState** state directly or at a deeper level, and returns **false** otherwise.
- **IStatechartState getContainerState()** – returns the immediate container state of this state or **null** if this state is at the topmost level.

During the code generation, the state names become elements of Java enumeration that implements **IStatechartState** interface, and transitions – objects of one of the subclasses of **Transition**. They are all defined at the level of the agent; therefore, two states or transitions cannot have the same name even if they are in different statecharts. When referring to the states or transitions from outside the agent, you should prefix them with the agent name. For example, to test if a statechart of the agent **car** is in the state **Parked** you should write:

    car.inState(car.Parked)

Such expressions are used, when, for example, you collect statistics on a population of agents each having a statechart inside.

7.5.   Viewing and debugging the statecharts at runtime

At runtime, AnyLogic highlights the active states of statecharts, the transitions that are scheduled to fire or have just fired. For active timeout and rate transitions, the remaining time to firing is displayed.
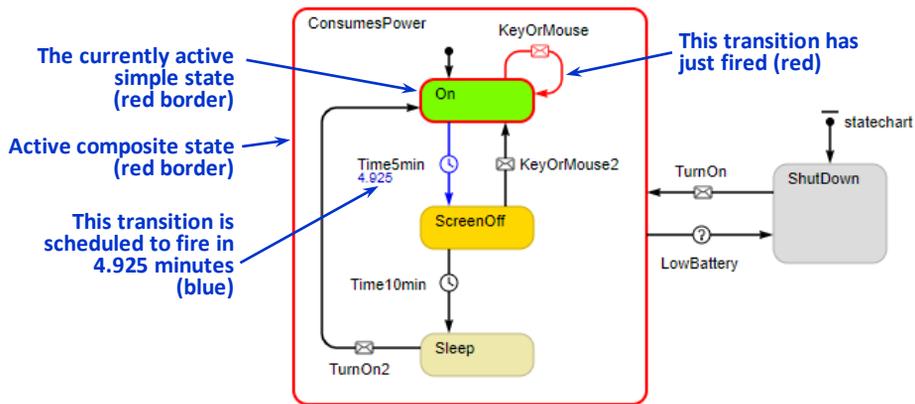
Figure 7.17 Statechart highlighting at runtime

Just like events, the scheduled statechart transitions can be observed in the **Events** panel (see Chapter 8, "Discrete events and Event model object" to find out how to open the **Events** panel). For example, for the statechart state shown in Figure 7.17 above the **Events** panel will display transition **Time5min** that is scheduled and, in a different list, the transition **LowBattery** that is constantly monitoring the battery condition.
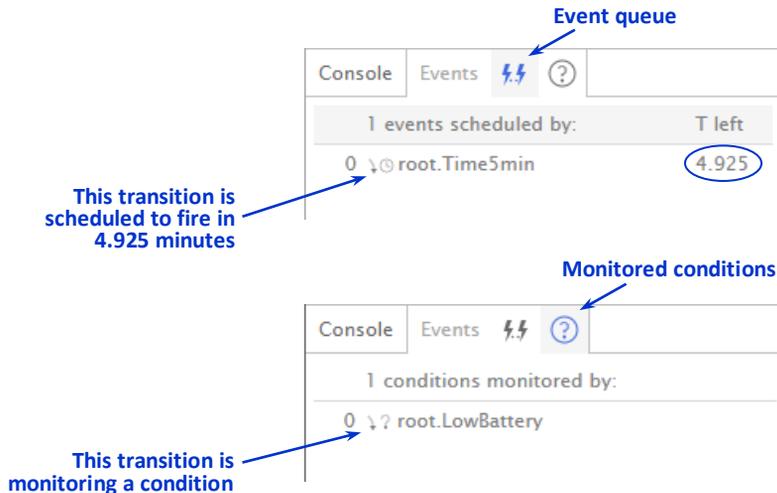

Figure 7.18 Events panel displays the scheduled transitions and transitions that monitor conditions

To further debug the statecharts, you can use the step-by-step execution mode, the breakpoints, and also you can write to the model log (see Example 8.1: "Event writes to the model log every time unit") from statechart actions.

7.6.   Statecharts for people's lives and behavior

Example 7.2: Life phases

The statechart for an individual's life phases described here is a modified version of the one from the (Wallis, Paich & Borshchev, 2004). At the topmost level, the individual can be either **Alive**, or **Dead**, see the two highest level states. If alive, the individual can be in one of four different life phases: **Child**, **Adult**, **MidAge**, or **Senior**.

As in general, the individuals can appear in the model at different ages (e.g. as a result of immigration), the statechart may initialize in either of the states, depending on the age. (Note that the statechart entry point points not to a state directly but to a decision point with four branches.) The transitions between the life phases are triggered by stochastic timeouts. For example, the transition from **Adult** to **MidAge** happens when the person is *around 49*, which is modeled by the timeout **normal(49,5) – age()**, where **normal(49,5)** is a normally distributed age with mean 49 and standard deviation 5, and **age()** is the age the person became **Adult** (remember that the timeout expressions are evaluated at the moment the statechart gets into the transition source state, in this case **Adult**).

The **Adult** life phase is further decomposed to describe family-related behavior. The decisions in this section of the statechart (to have family or not, how long to wait before the first kid, how many kids to have, etc.) are also stochastic and may depend on the gender, the level of education, the cultural norms, etc. If this statechart is inside an agent in an agent-based model, the act of childbearing may result in a new agent added to the model, who may inherit the characteristics of the parents. The event of death then may delete the agent from the model.
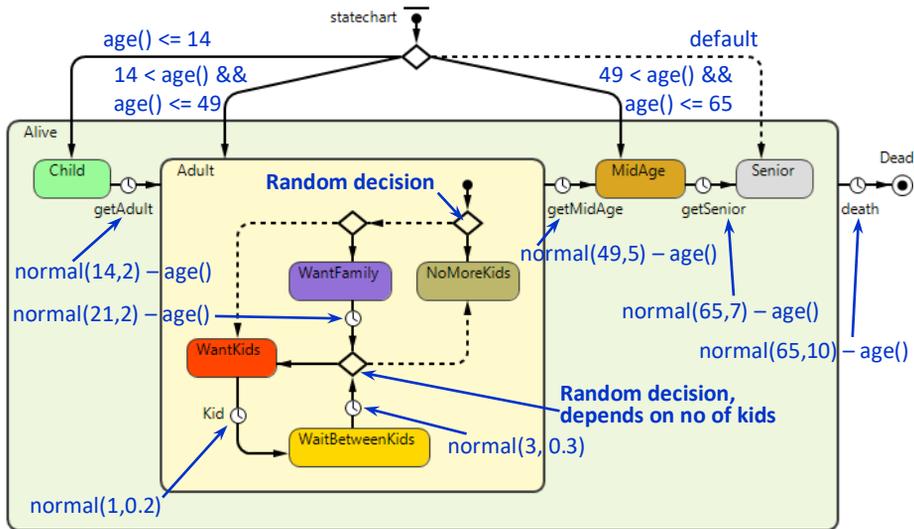
Figure 7.19 Statechart for life phases

A model of an individual may have other statecharts or other behavior "threads" that can communicate with the life phase statechart. For example, the education, the employment, the purchase behavior, etc. may depend on the life phase and may, in turn, affect the decisions made in the life phase statechart.

Example 7.3: Adoption and diffusion

The statechart in this example is a fundamental construct for all individual-based (agent-based) models where people are first being influenced to adopt an idea, a product, etc., and then, having adopted it, spread it by word of mouth. A very similar statechart is used in the models of diffusion of infectious diseases (see Example 7.4: "Disease diffusion").

We distinguish between two states of a person: **PotentialAdopter** and **Adopter**. People are sensitive to advertising (by which we mean any non-personal information sources) and to word of mouth, hence the two transitions from the first state to the second. The **Advertising** transition fires after a stochastic timeout, which models random (and comparatively low) sales/adoptions caused by advertising. **AdEfficiency** is the corresponding parameter. The **WordOfMouth** transition is triggered by a message received from another person. Having received the message **"Good stuff!",** the person adopts the "stuff" with a certain probability – **AdoptionFraction** – otherwise, returns to the state **PotentialAdopter**.

Although any firing of **WordOfMouth** resets the previously scheduled **Advertising** transition, we should not bother: the exponentially distributed timeout (which, in

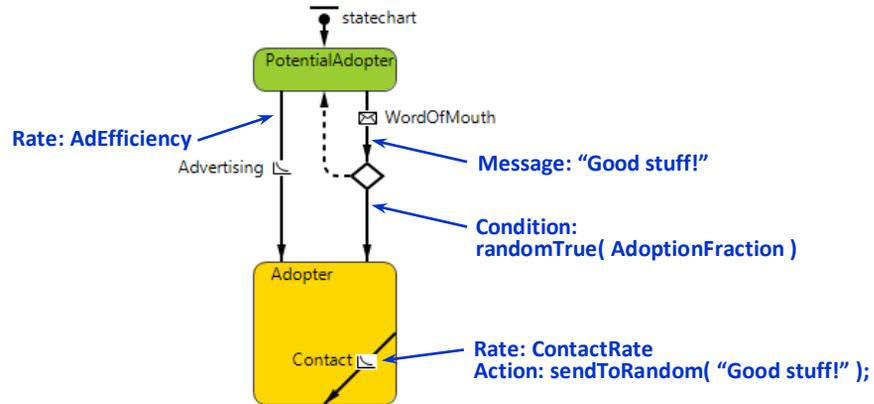fact, is a rate transition) survives any number of resets without violation of distribution function.

Figure 7.20 Statechart for adoption and diffusion

In the **Adopter** state, the person periodically contacts other people (randomly picked from the whole population, from personal contacts, neighbors, etc.) and tells them how good the stuff he has adopted is. This is modeled by the internal transition (see Section 7.3) **Contact** being executed in a loop at **ContactRate**.

Example 7.4: Disease diffusion

The statecharts used in disease diffusion models are similar to the one for product/idea adoption described in the previous section. We will consider a fairly general case. The disease only spreads from one person to another (therefore, as opposed to the adoption model, there is no "advertising" channel). Having been infected, the person goes through several stages with different degrees of infectivity and different contact rates. Finally, having recovered from the disease, the person becomes immune (insensitive to infection).
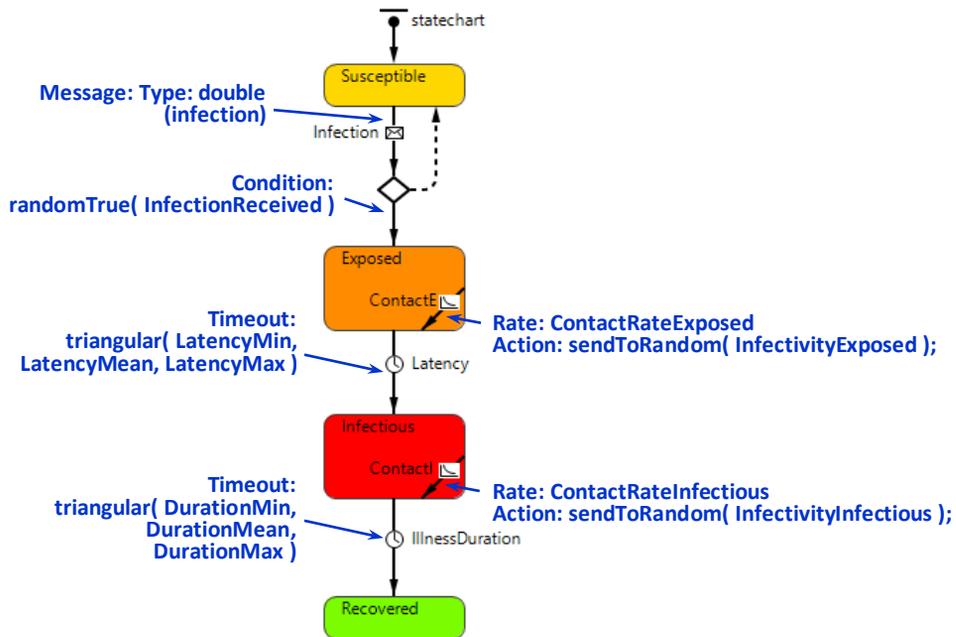
Figure 7.21 Statechart for agent-based SEIR model

The statechart in Figure 7.21 can be used in an agent-based SEIR (Susceptible Exposed Infectious Recovered) model of diffusion of an infectious disease. Initially, the person is **Susceptible** (can be infected by contacting other people). The infection passed during a contact can be of different degrees; therefore, a double value 0…1 is used as the infection message. Having received an infection, the person becomes **Exposed** with the corresponding probability, otherwise the person remains **Susceptible**. The **Exposed** state corresponds to the disease latency period when the symptoms have not yet developed, so the person continues contacting other people at a regular rate **ContactRateExposed** (see internal transition **ContactE**). During each contact, the person passes infection, although its doze is comparatively low: **InfectivityExposed**. After the latency period, the person becomes ill (state **Infectious**) and limits his contacts to the minimum (**ContactRateInfectious**). However, each contact results in a serious doze of infection being passed: **InfectivityInfectious**. Both latency period duration and illness duration are triangularly distributed. The recovered person becomes immune and (as long as there is no transition from **Recovered** back to **Susceptible** in this statechart) immune forever.

Example 7.5: Purchase behavior with a choice of two competing products
This statechart is yet another extension of the generic statechart for adoption and diffusion (see Example 7.3: "Adoption and diffusion"). It can be used in agent-based models of consumer markets. The additional features captured are:

- The competition between two companies (or between your company and all others).
- The limited usage time of the product and the need for repeated purchases.
- The potential limited availability of product (e.g., because of supply chain problems).
- Some sort of brand loyalty and switching behavior.

There are two competing products on the market: A and B. Initially, the consumer is in the **PotentialUser** state, where a decision to buy that type of product has not been made yet. Either by advertising or by word of mouth, the consumer becomes convinced and wants to buy a product of a particular brand: states **WantA** and **WantB**. (This happens in the same way as in the generic adoption statechart, although the advertising effectiveness may be different for different brands.) If the product of the corresponding brand is available (the stocks can be modeled e.g. by system dynamics stocks **RetailerStockA** and **RetailerStockB**, by variables, or by **Queue** block of the Process Modeling library), the consumer immediately proceeds to the **UserA** or **UserB** state, otherwise the consumer waits. In this example, waiting for a particular brand is limited to 2 days. After that, the consumer gives up (his loyalty "expires"), and is ready to buy any brand that is available (state **WantAnything**).
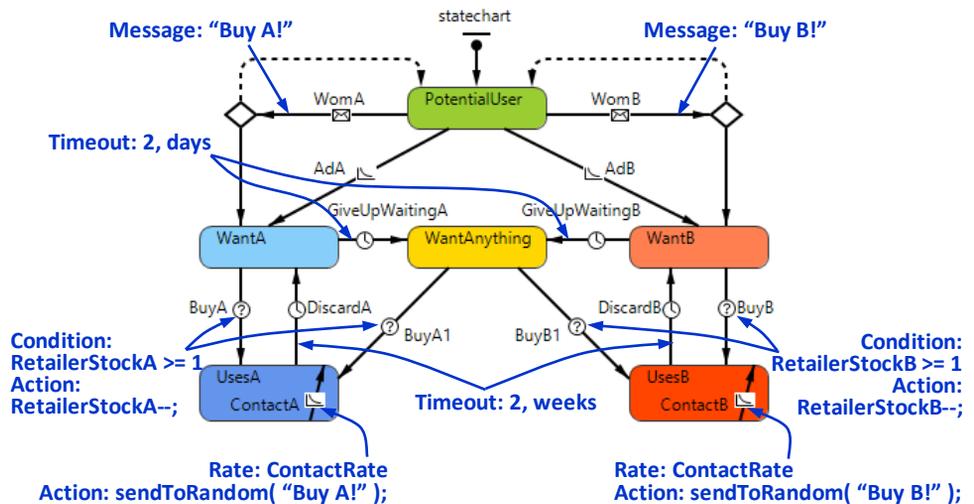

Figure 7.22 Statechart for consumer's choice of two competing products

The product's usage time is two weeks, and after that period the product is discarded and the consumer needs to buy another one, preferably of the same brand (transitions **DiscardA** and **DiscardB** bring the consumer back to **WantA** and **WantB** states).

? One of the assumptions in this model is that after buying a particular brand the consumer stays with it and ignores the advertising and word of mouth for the competing brand, so the only reason for switching is unavailability of the currently chosen brand. How would you modify the statechart to drop this assumption?

7.7.   Statecharts for physical objects

Example 7.6: Generic resource with breakdowns and repairs

Sometimes, agent-based models include agents that are used and shared by other agents and can be treated as resources (similarly to resources in process models). A very generic statechart for such resource is shown in Figure 7.23. The resource can be either **Operational** or **OutOfOrder**. The failures occur sporadically, and the time between failures is frequently modeled as exponentially distributed with a certain known mean value (*MTBF* – Mean Time Between Failures); therefore, we can use a rate transition with the rate **1/MTBF**. Similarly, for the repair time we can use another rate transition with the rate **1/MTTR** (*MTTR* – Mean Time To Repair).
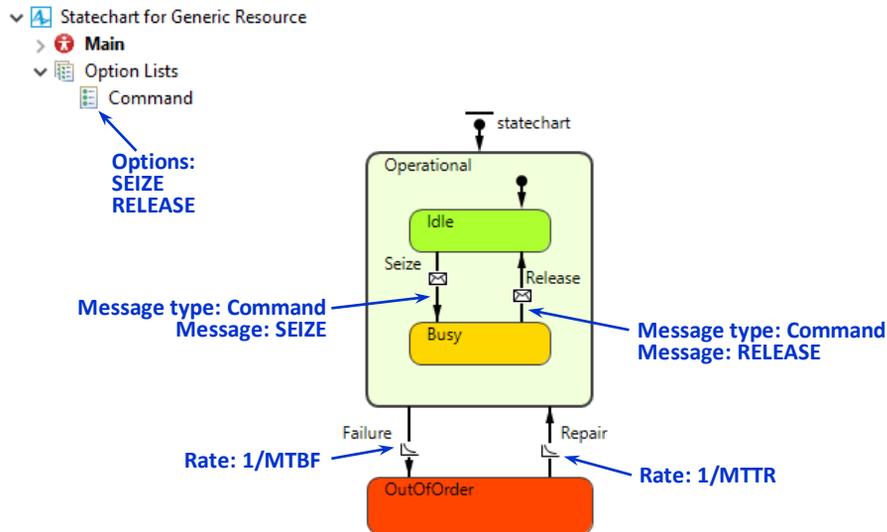

Figure 7.23 Statechart for generic resource

Seize and release operations can be modeled by, for example, message-triggered transitions toggling the statechart between **Idle** and **Busy** states. The seize/release messages are defined as the options of the **Command** option list.

? While the failure can occur both in **Idle** and **Busy** states, the **Repair** transition always brings the resource to the **Idle** state. Is there a simple way to restore the state?

Example 7.7: Delivery truck

Statecharts are extensively used to model operation of various physical objects: vehicles, cranes, elevators, machines, infrastructure elements, etc. Consider a delivery truck, which delivers goods to clients from a warehouse. If the truck is modeled as an autonomous agent, it makes a lot of sense to define its behavior in the form of a statechart like the one in Figure 7.24.
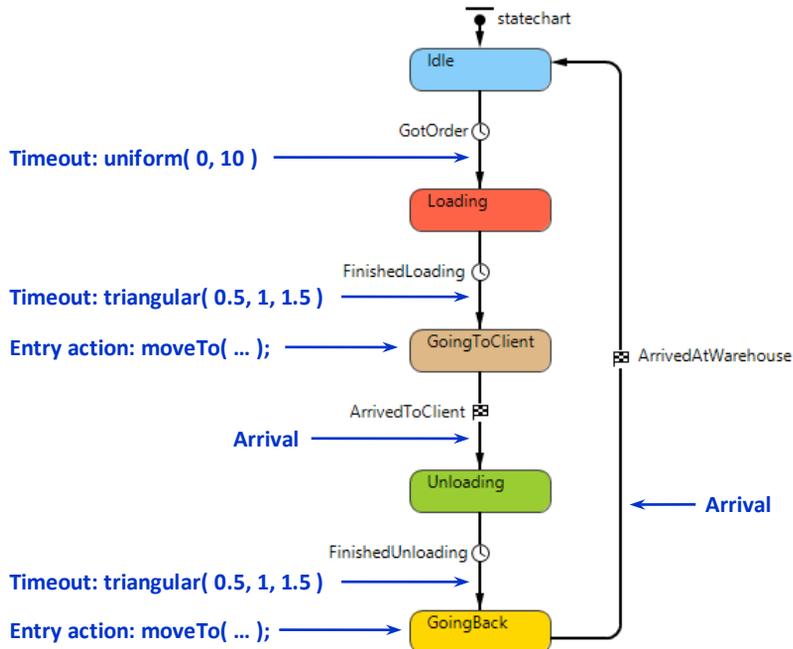

Figure 7.24 Statechart for delivery truck

Initially, the truck is in the **Idle** state (parked at the warehouse) and waits for a delivery order. Once the order is received, it starts loading, which takes 1 hour on average. Having been loaded, the truck departs to a client (see the call of agent's function **moveTo(…)** in the entry action of **GoingToClient** state). The transition **ArrivedToClient** is triggered by arrival. Unloading and returning to the warehouse is modeled in a similar way.

Example 7.8: Aircraft maintenance checks

In the agent-based models of fleets of aircrafts, rail cars, trucks (or, in general, any pools of objects that are subject to periodic maintenance, such as houses, water or electricity delivery infrastructure, etc.) statecharts can be efficiently used to model the maintenance rules and the resulting availability of the objects. Consider an aircraft. The periodic maintenance checks must be done after a certain amount of time or usage. The most common checks are:

- **ACheck** – due every month or each 500 flight hours; done overnight at the airport gate.
- **BCheck** – due every 3 months; also done overnight at the gate.
- **CCheck** – due every 18 months or each 9000 flight hours; performed at a maintenance base or at a hangar and takes about 2 weeks.
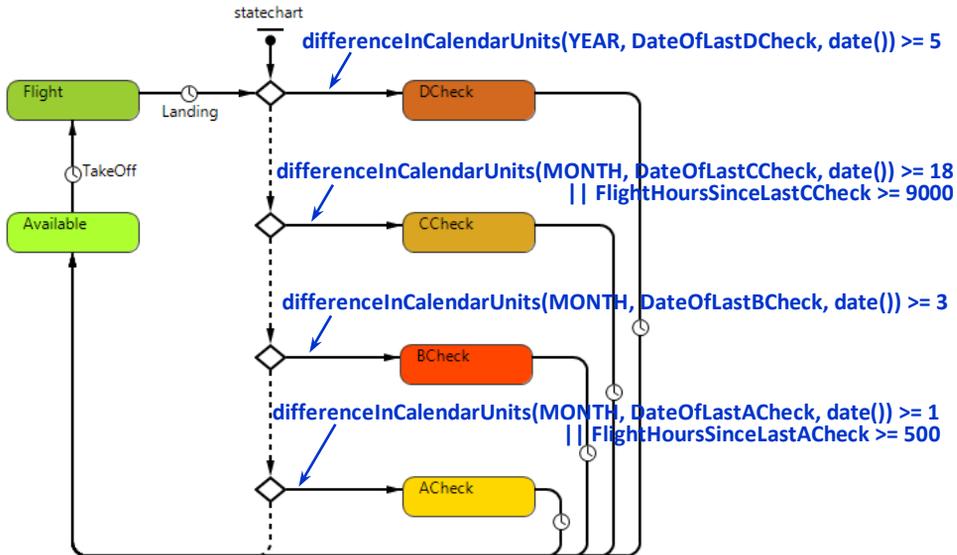- **DCheck** – due every 5 years; performed at a maintenance base and takes 2 months.

Figure 7.25 Statechart for aircraft maintenance checks

A possible statechart for aircraft maintenance is shown in Figure 7.25. When not undergoing maintenance, the aircraft is either **Available** (ready to fly), or in the **Flight** state. The transitions **TakeOff** and **Landing** can be triggered by timeouts or by actual commands, depending on how you wish to model the aircraft operation.

Upon each landing, the statechart determines if the aircraft is due for maintenance, starting from the "heaviest" DCheck. The condition in a branch to **DCheck** state is:

```
differenceInCalendarUnits( YEAR, DateOfLastDCheck, date() ) >= 5
```

Here, **date()** is the current model date, and **DateOfLastDCheck** is assumed to be a variable of type **Date,** where the date of last D Check is stored (this can be done in an exit action of the **DCheck** state). The function **differenceInCalendarUnits()** called with the first parameter set to **YEAR** returns the number of years between the two given dates. If that condition is false, the statechart evaluates the next one – for CCheck:

```
differenceInCalendarUnits( MONTH, DateOfLastCCheck, date() ) >= 18 ||
FlightHoursSinceLastCCheck >= 9000
```

This condition has two parts: one based on time since the last CCheck and another – based on the flight hours since the last CCheck, which are stored in a variable **FlightHoursSinceLastCCheck** and are updated on each landing (remember that **||** is Java syntax for OR). The other two conditions for BCheck and ACheck are defined similarly. It is important that when a maintenance task is performed, it resets the dates and flight hours for itself and for all lighter maintenance types. The transitions outgoing the maintenance states represent maintenance times: **2** weeks, **15** hours, etc.

? In the statechart, the maintenance conditions are only checked upon each landing; therefore, the aircraft will not miss the time-based check only if it flies regularly. If, however, there are significant idle periods, the need for time-based checks can potentially be detected too late. How would you modify the statechart to drop the assumption of flight regularity?

7.8. Statecharts for products and projects

Example 7.9: Product life cycle, including NPD
Statecharts can be used to model the lifecycle of the company's projects or products, (e.g. in the agent-based models built to support portfolio management). A very generic statechart for a product lifecycle including NPD (New Product Development) phase is shown in Figure 7.26. At the top level, we distinguish between **NewProductDevelopment** and **InMarket** states. NPD in turn breaks down into four simple states ("New product development", n.d.) (these may be different for different types of products):

- **FrontEnd** – opportunity identification, idea generation and screening, business analysis.
- **Prototyping** – producing a physical prototype of the product
- **Testing** – testing the product prototype in typical usage situations, making adjustments.
- **Development** – planning and implementing engineering operations, quality management, supplier collaboration, etc.

The transitions between these states have the meaning of successful accomplishment of the corresponding phase and proceeding to the next one. Technically, they can be stochastic timeout transitions where distribution of phase durations is based on expert knowledge, or may depend on other parts of the model (e.g. on resource availability). At any time during the NPD phase, the process can be suspended and then resumed, or killed. These are the company decisions, which may depend on many different factors.
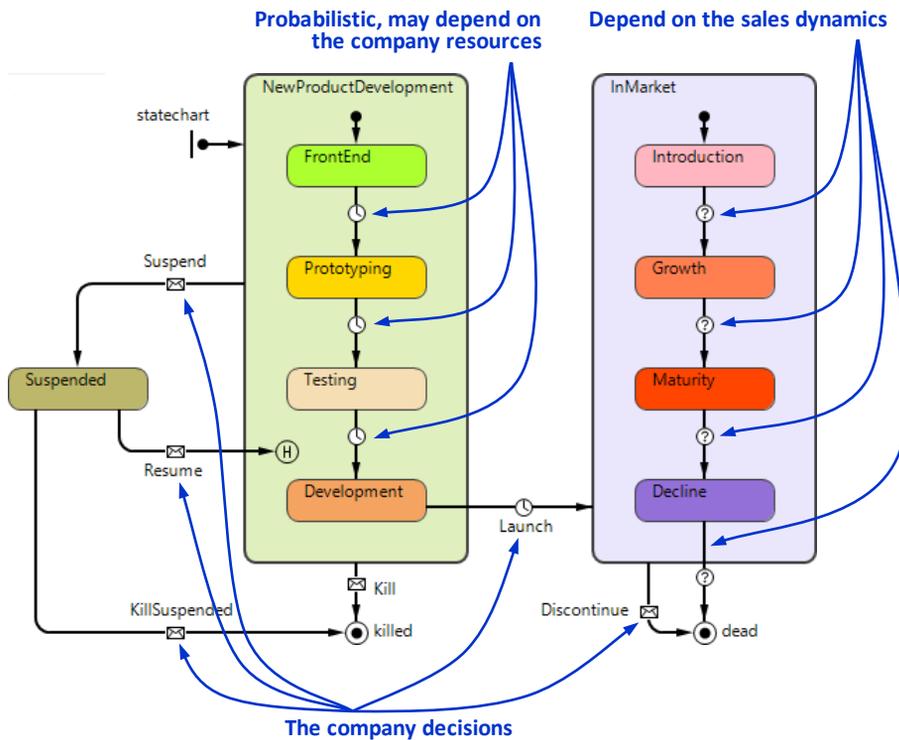
Figure 7.26 Statechart for product life cycle

When the development phase is successfully completed, the product is launched onto the market. The **InMarket** state, in turn, is broken down into another four states (QuickMBA.com, 2010):

- **Introduction** – sales are low, the company is building the product awareness, advertising costs are high.
- **Growth** – strong growth in sales, little competition, the company is building a market share, promotion is aimed at a broader audience.
- **Maturity** – the strong growth in sales diminishes, competition may appear with similar products: the objective is to defend the market share.
- **Decline** – sales are declining because of market saturation, or the product becomes technologically obsolete: the marketing support is typically reduced.

Again, at any moment, the company can discontinue the product (although it is typically done at the decline phase). The transitions between these states typically depend on the sales and market dynamics, which are modeled outside of the statechart.

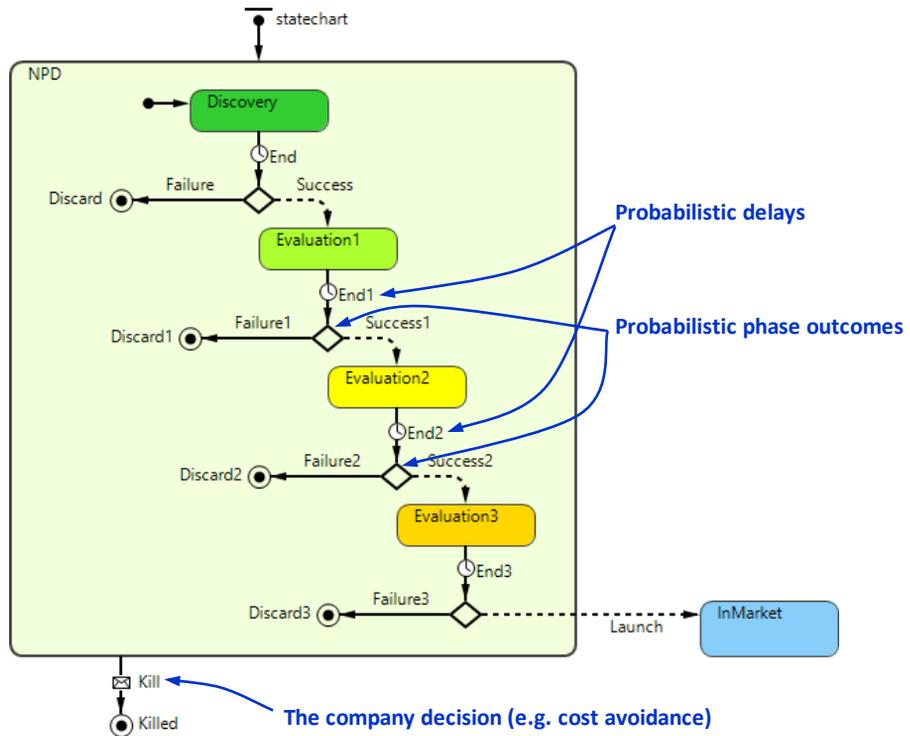**Example 7.10: Pharmaceutical NPD pipeline**

Figure 7.27 Statechart for pharmaceutical NPD pipeline

Depending on the industry, the product lifecycle structure may be slightly different from the generic one described in the previous example. For example, the pharmaceutical and biotechnological NPD pipelines require that the product is put through three clinical evaluation phases before it can be launched. (Solo & Paich, 2004) suggest using a statechart like the one in Figure 7.27.

The probability distributions used to model the durations of each phase (the timeout transitions: **End**, **End1**, etc.) are parameterized using the company's statistics and expert knowledge, as well as the probabilities of success and failure. You can also associate the resource consumption rate with each phase.