Chapter 8. Discrete events and Event model element

8.1. Discrete events

The terminology

For the sake of clarity, we need to establish clear definitions for the terms used in this book to avoid confusion with other uses the reader may have come across. The terms *discrete event modeling* or *discrete event simulation* are commonly used for the modeling method that represents the system as a process, i.e. a sequence of operations being performed over agents such as customers, parts, documents, etc. These processes typically include delays, usage of resources, and waiting in queues. Each operation is modeled by its start event and end event, and no changes can take place in the model in between any two discrete events. The term *discrete* has been in general use for decades to distinguish this modeling method from *continuous time methods*, such as system dynamics (see Chapter 5).

With the emergence of agent-based modeling (see Chapter 3) the term "discrete event modeling" in its traditional sense created confusion since in most agent-based models actions are also associated with discrete events, but there may be no processes, or resources. Therefore, throughout this book we will be using the term *process modeling* for the modeling method where agents use resources and wait in queues, and the term *discrete event* for the more general idea of approximating the reality by instant changes at discrete time moments.

Discrete events: approximation of real-world continuous processes

The dynamics of the world around us appear to be continuous: there are no instant changes – everything takes non-zero time, and there are no atomic changes – every change can be further divided into phases. For example, an airplane landing includes: descending, touching the ground, slowing down along the runway, and taxiing to the gate. An employee leaving a company must: look for a new job, send out a resume, get interviewed, get an offer, and so on. However, depending on your level of abstraction, "airplane lands" or "employee leaves" can be considered as instant events. Their component detail may not be relevant.

In discrete event modeling we only consider important moments in the system's lifetime, treat them as instantaneous and atomic *events*, and abstract away from anything that goes on between two contiguous events – see Figure 8.1. These are some examples of events:

- A customer enters the supermarket
- A truck arrives to the warehouse bay

- A project is finished
- A patient recovers from a disease
- Inventory level falls below the threshold
- The product price is reduced by 30%

Though virtually nonexistent in the natural world instantaneous atomic events can be observed in some artificial environments such as computer systems. For example, a credit card approval by a bank server, arrival of an SMS message to a cell phone, submitting a web form, etc.

All dynamics (i.e. all changes) in a discrete event model, be it a process model or an agent-based model, are associated with events. Events are *instantaneous* (the execution of an event takes zero time) and *atomic* (event execution cannot be interrupted by, combined, or interwoven with another event). Events can schedule other events; the event of airplane takeoff may schedule the event of landing. Events may be simultaneous and be scheduled to occur at the same time. In that case events are *serialized* by the simulation engine, i.e. executed in some order.

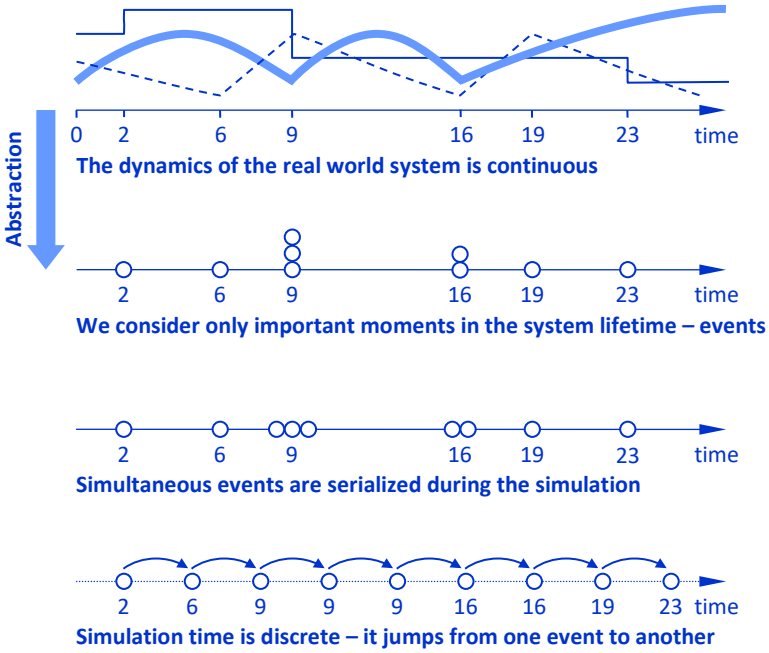

Figure 8.1 Discrete event modeling and simulation

Since nothing happens in between two subsequent events, the time in discrete event simulation is discrete: it jumps from one event to another.

Discrete event management inside AnyLogic engine

Consider how AnyLogic simulation engine executes a discrete event model. The engine maintains the *event queue* – the structure that contains all scheduled events. Look at the event queue example in Figure 8.2. Having executed the event **a** the engine advances the model clock to the group of simultaneous events **{b,c,d}**. Let the engine decides to execute **b** first (below we will consider how exactly the engine orders the simultaneous events). After **b** the engine chooses **d** and executes it (the model clock shows the same time). As a result of **d** two scheduled events get cancelled: **c** and **f**. As long as **c** is no longer in the event queue, the clock jumps to **e**. **e** gets executed and schedules a new event **i** after the group **{g,h}**. The clock is advanced to **{g,h}**, and the engine goes on.

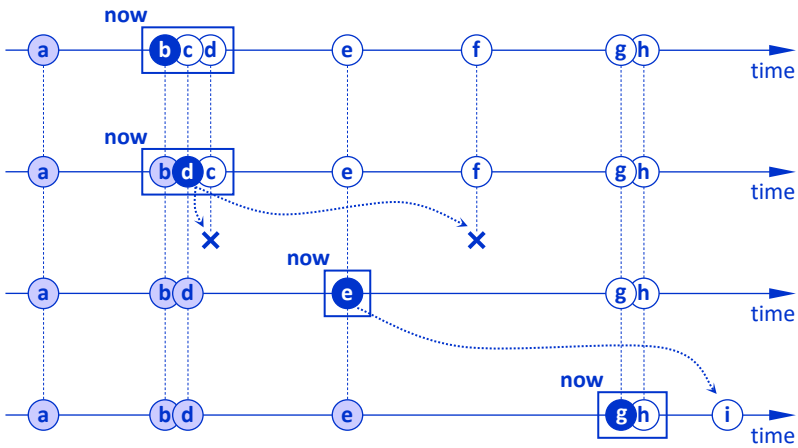

Figure 8.2 Event queue in AnyLogic simulation engine

There are two modes in which the simulation engine can serialize (impose order on) simultaneous events. It can follow a deterministic order or choose events randomly.

Two deterministic orders are supported: *FIFO* (first in, first out) and *LIFO* (last in, first out). In FIFO mode, the engine follows the order of event scheduling – it chooses the event that has been scheduled earlier than the events simultaneous with it. LIFO mode implements reverse scheduling order and chooses the most recently scheduled event from the simultaneous alternatives.

Executing a deterministic order is faster than a random order, but if your model is sensitive to the simultaneous event ordering the random mode will ensure you simulate a wider spectrum of possible scenarios. We suggest you build models to be insensitive to low-level event ordering. But in any case, you can set up the serialization type at the experiment level:

▶ **To set the ordering mode for simultaneous events:**

1. Select the experiment and open the **Advanced** section of its properties.
2. Depending on what you want, select the required option from the **Selection mode for simultaneous events** drop-down list: **Random**, **FIFO (in the order of scheduling)**, or **LIFO (in the reverse order of scheduling)**.

8.2. Event – the simplest low-level model element

In this section we will introduce the simplest element of AnyLogic modeling language called *event* (sometimes also called *static event* in contrast with dynamic event, see Section 8.3.

Please do not confuse *event elements* used by the modeler and *discrete events* in the simulation engine described in the previous section. A single event element can generate one or several discrete events during the simulation, which you can consider as its instances. Besides event elements, discrete events can be scheduled by statecharts (see Chapter 7).

The Event element provides a way to schedule a discrete event (see Section 8.1) or a sequence of discrete events directly into the simulation engine event queue. Theoretically, the event element is sufficient to build all kinds of discrete event models. In practice, however, modelers use higher level constructs such as the Process Modeling Library blocks or statecharts as well as fairly low-level events. Events are used mainly in the following cases:

Use case for events	Event type
Generate arrivals, births, etc. in non-process models, e.g. in agent-based models. (In process models you use the **Source** block for that.)	Rate or cyclic timeout with stochastic recurrence time
Perform periodic actions, e.g.: daily patient review, annual budget planning, assignment of tasks at the beginning of the working day, scan the area every second, adjust direction every hour, replenish account every month, etc.	Cyclic timeout with deterministic recurrence time
Perform sporadic actions: move to a new house on average every five years, change mind on average every year, change demand approximately every month, etc.	Rate or cyclic timeout with stochastic recurrence time

Schedule delayed action “outside” the main action flow (which may be e.g. a process flowchart, or a statechart): the door closes in five seconds, the product is launched into the market in one month, the new train arrives with one minute interval, etc.	User-controlled timeout
Wait on a condition, e.g. inventory level reaching a threshold, or enemy aircraft is within the triggering zone.	Condition
Periodically collect custom statistics, calculations, or write output.	Cyclic timeout with deterministic recurrence time
Emulate external influence at particular time moments, e.g. demand increase, or commodity price change.	Timeout in the “occurs once” mode
Emulate user actions such as slowing the model down, zooming in, changing view, pausing, etc.	Timeout in the “occurs once” mode
Do something immediately, but in a different discrete event (because you wish to finish the current event first, for example).	User-controlled timeout with time = 0
Do something at time 0, but after the model is fully initialized and the startup code is executed.	Timeout in the “occurs once” mode with occurrence time = 0

To create an event:

1. Open the **Agent** palette.
2. Drag the **Event** element from the palette to the graphical editor.

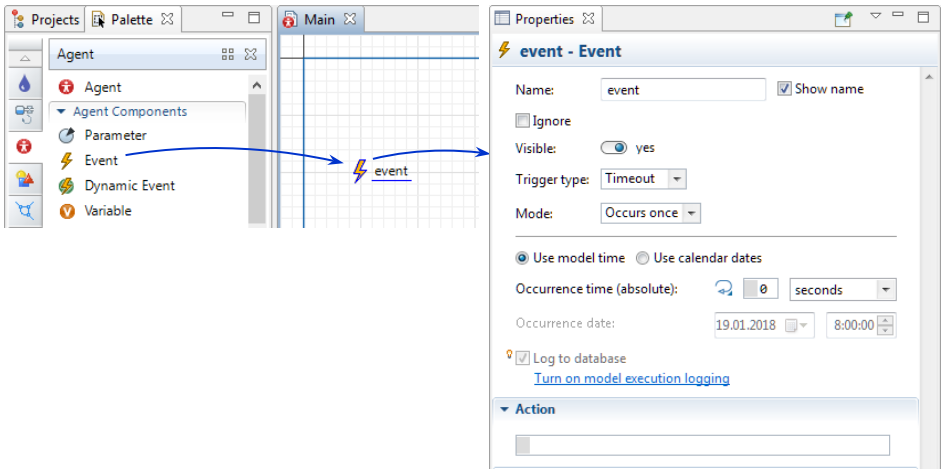

Figure 8.3 Creating a new event

The event has the **Trigger type** (**Timeout**, **Rate**, or **Condition**) property and mode of operation. Event also has the **Action** field where you can specify what happens when the event occurs. Events may be of the following types (see also Figure 8.4):

- *Timeout event that occurs once* at the specified calendar time or model time.
- *Cyclic timeout event* that occurs periodically with a certain recurrence time. You can also specify the time of the first occurrence.
- *User-controlled timeout event* that occurs after its **restart()** function is called.
- *Rate event* that occurs sporadically with exponentially distributed inter-occurrence times (Poisson stream of events).
- *Condition event* that occurs when a given condition becomes true.

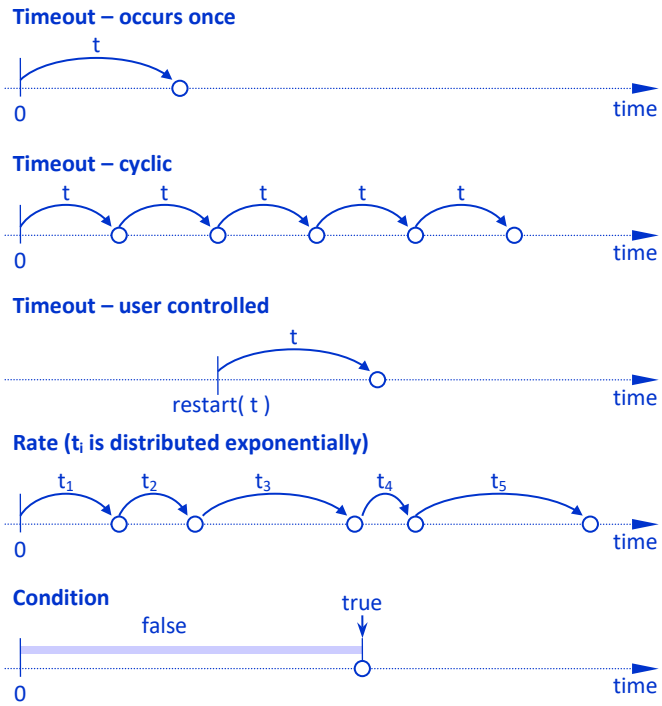

Figure 8.4 Event types (static events)

Below we will consider several examples of events.

Example 8.1: Event writes to the model log every time unit

In the first example we will create the simplest model of a queuing system and use a cyclic timeout event to write the status of the system to the model log. The system will consist of an agent generator, a queue, and a server.

Follow these steps:

1. Click the **New** button on the toolbar. In the **New Model** wizard enter the model name. A new model is created, and the editor of its **Main** agent opens.
2. Open the **Process Modeling Library** palette and create a flowchart as shown in Figure 1.5. Drag four blocks from the library palette into the graphical editor and connect them in the following sequence: **Source** – **Queue** – **Delay** – **Sink**. Place the blocks close enough to each other and they will connect automatically.
3. Run the model. Make sure the agents go through the system.
4. Go back to the graphical editor of **Main** and open the **Agent** palette.
5. Drag the **Event** element from the palette and drop it anywhere below the flowchart. Set the name of the event to **trace**.

6. In the event's properties set the **Mode** to **Cyclic** and leave the default value of the **Recurrence time**: **1**.
7. Expand the **Action** properties section and type there the following code:
   traceln(time() + ": In queue: " + queue.size() + " In service: " + delay.size());
8. Run the model and in parallel watch the model log in the developer panel's console. Open it by clicking the **Toggle Developer panel** control in the model window.

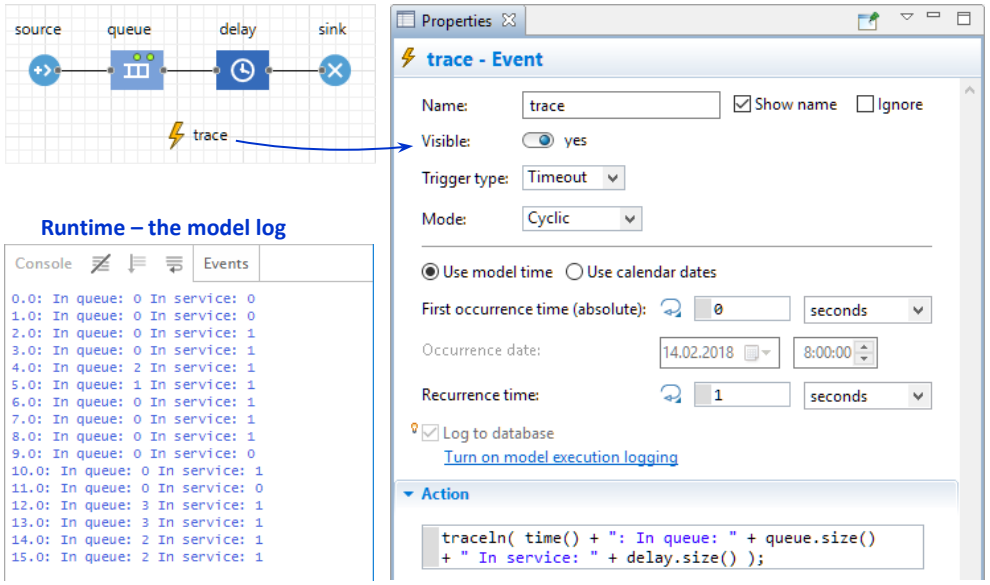

Figure 8.5 Recurring timeout event is used to trace the model state

By setting the mode to **Cyclic** you make the event recurring and you can specify the recurrence time. The default value is **1**. The function **traceln()** writes to the model log, which you can view in the **Console** displayed in the developer panel. The argument of the **traceln()** function is a string. In this example it is composed of string constants (like **" In service: "**) and numeric values returned by the function **time()** and the functions of the flowchart blocks.

You can write arbitrary expressions in the **Recurrence time** field, not just a constant. The expression will be evaluated dynamically after each event occurrence. Therefore, the inter-occurrence time can be made variable. You can make it stochastic by using probability distribution functions in the expression.

Example 8.2: Event generates new agents

In the following example the event generates new agents; one agent every time unit on average. We will create an agent-based model, but our agent population will be initially empty.

Follow these steps:

1. Create a new model.
2. Drag the **Agent** element from the **Agent** palette to the graphical editor.
3. On the first page of the **New agent** wizard, click the **Population of agents** option.
4. On the next page of the wizard set the **Agent type name** to **Person**. The **Agent population name** will automatically change to **people**. Click **Next**.
5. On the next page switch to the **2D** option and select the **Person** shape from the list below as the agent's animation. Click **Next**.
6. On the next page of the wizard click the **Next** button to skip it and proceed further.
7. Select the **Create initially empty population, I will add agents at the model runtime** option. Click **Next**.
8. On the last page of the wizard, leave the default agent space settings (the continuous space of 500*500 pixels). Select the **Apply random layout** option to randomly distribute agent animations in the defined space. Click **Finish**.
9. Select the population presentation shape (to the right of the **people** agent population, see Figure 1.6). Expand the **Advanced** section of its properties and select the **Draw agent with offset to this position** option. Now this shape will define the upper left point of the **people** population space.
10. Run the model. Make sure there are no agents in the population.
11. Go back to the editor of **Main** and open the **Agent** palette.
12. Drag the **Event** element from the palette and drop it below the **people** agent population. Set the name of the event to **newAgent**.
13. In the event properties set the **Trigger type** to **Rate** and leave the default value of **Rate**: **1 per second**.
14. In the **Action** section of the event properties type: **add_people();** (remember to use code completion available at Ctrl+Space, see Section 10.4).
15. Run the model. You should see agents appearing on each event occurrence.

The trigger type **Rate** = 1 means that the event will occur sporadically on average once per time unit, and the inter-occurrence times will be distributed exponentially. The function **add_people()** is generated automatically if there is an agent population with name **people**.

Figure 8.6 Event of Rate type dynamically adds new agents to the model

Events triggered by a condition

Event may be triggered by a *condition*. Condition is an arbitrary expression and may depend on the states of any elements in the whole model with continuous as well as discrete dynamics: system dynamic variables (see Section 5.1), statecharts (see Chapter 7), variables (see Section 10.3), Process Modeling Library blocks, etc.

The condition of such event is tested by the simulation engine when anything happens in the *same agent* (the same applies to the statechart transitions triggered by condition, see Section 7.3):

- Another event occurs
- A statechart transition is taken
- A parameter is changed
- A control of the agent has been accessed by the user
- If agent arrives to the destination point
- If agent receives a message

In addition, if the model contains any continuous dynamics (i.e. if there are any system dynamics variables in *any* of the agents), the conditions of *all* agents are tested on each integration step.

In most cases you should assume the condition is monitored "all the time" and the event is triggered as soon as it becomes true. However, if the condition depends on *other agents*, it will not necessarily be tested when "remote" discrete changes take place there. Then, to avoid missing the exact moment of time when the condition

becomes true, you should explicitly call the function **onChange()** of the agent where the condition event is defined each time such change occurs.

After the event gets executed by the engine, it gets deactivated and stops monitoring its condition. This default behavior is useful because it prevents infinite event loops cases where the condition remains true after the event occurs. If you need the event to continue monitoring you should call its function **restart()** in the event action.

It is important to understand that some dynamics in the model that may seem to be continuous and is *animated as continuous* (such as movement of an agent in the network) is in fact modeled as two discrete events: one in the beginning and another one in the end. Therefore if you specify a condition like **agent.distanceTo(otherAgent) <= 50**, it may not be tested at the right time and you may miss the moment. To fix this you need to test the condition periodically by using e.g. a cyclic timeout event.

Example 8.3: Event waits on a stock reaching a certain level

In this example we will create a very simple system dynamics structure: a stock and a constant incoming flow filling the stock. The event triggered by condition will close the inflow when the stock reaches the level of 100 units. This example demonstrates one of the methods you can use to link continuous system dynamics model components with discrete ones.

Follow these steps:

1. Open the **System Dynamics** palette and drag the **Stock** element to the graphical editor.
2. Drag the **Flow** element and drop it on the left of the stock so that the right end of the arrow gets connected to the stock. This creates an inflow, see Figure 1.7.
3. In the properties of **flow** check the **Constant** checkbox. Then set the field **flow =** to **10**.
4. Open the **Agent** palette and drag the **Event** element to the graphical editor. Name the event **stockAt100**.
5. In the properties of the event set:
   Trigger type: **Condition**
   Condition: **stock >= 100**
   Action: **flow = 0;**
6. Run the model. Observe the value of the stock.

Since we want to control the value of the **flow** from outside the system dynamics model, we need to mark it as a constant. This tells AnyLogic that the value of the **flow** will not be continuously evaluated as a formula. The other important thing is that the condition of the event is not **stock == 100** but **stock >= 100**. The latter condition is correct because the value of the stock is integrated and grows in small discrete steps, therefore it is possible that it never gets exactly equal to 100, so that the condition **stock == 100** may never become true!

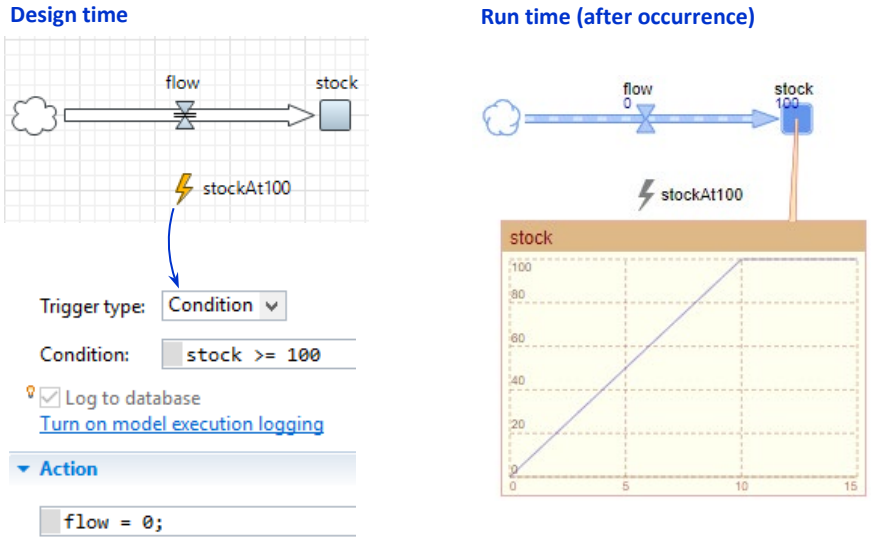

Figure 8.7 Event of Condition type triggered by a system dynamics variable

Please note that after the event is executed it stops monitoring the condition. This default behavior makes a lot of sense in this particular model where the condition remains true after the event occurrence. If it continued to monitor, it would be triggered infinitely in a loop not allowing the simulation time to progress.

An obvious question would be: but what if we want the event to be activated again when the stock falls below the 100 level? One of the solutions would be to create another event **stockBelow100** triggered by condition **stock < 100** and write in its action field **stockAt100.restart();**. At the same time, you should add the line **stockBelow100.restart();** to the action field of **stockAt100**.

Example 8.4: Automatic shutdown after a period of inactivity

We will model automatic shutdown of a device, e.g. a laptop, done after a period of user inactivity. We will use two events: a rate event for sporadic user actions, and a manually controlled timeout event for the shutdown. Each user action will immediately wake up the laptop and reactivate the shutdown timeout.

Follow these steps:

1. Create a new model. In the **New Model** wizard, set **minutes** as the **Model time units**.
2. Open the **Agent** palette and drag the **Event** element to the graphical editor. Name the event **userAction**.
3. Drag another event from the palette and name it **shutdown**.
4. In the properties of the **userAction** event set:
   Trigger type: **Rate**
   Rate: **0.1** **per minute**
   Action: **shutdown.restart();**
5. In the properties of the **shutdown** event leave the **Trigger type** default value **Timeout** and set:
   Mode: **User control**
   Timeout: **10** **minutes**
6. Open the **Presentation** palette, drag the **Oval** element and drop it anywhere beside the events. In the **Appearance** section of the oval properties set the **Line color** of the oval to **No color**.
7. Click the **Fill color** property's icon  to switch to the dynamic value editor. Type **shutdown.isActive() ? limeGreen : gold** in the **Fill color** field.
8. Run the model. Watch the times to the next occurrences of events and the color of the circle.

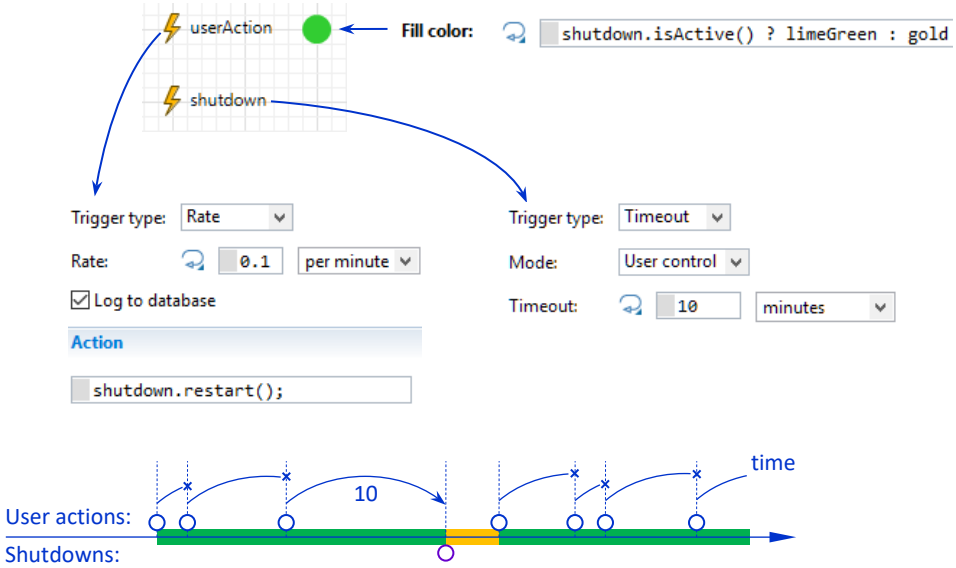

Figure 8.8 Rate event models user actions, and timeout event models automatic shutdown

The rate event **userAction** in this model occurs irregularly at exponentially-distributed time intervals on average 0.1 times per minute (i.e. every 10 minutes). The **shutdown** event is triggered in the "user control" mode by a timeout of 10 minutes. It means that the **shutdown** event will not be activated until someone calls the **restart()** function. Every time the **userAction** event occurs, the **shutdown** timeout is restarted. If it happens before the previous timeout expires, the previous timeout is cancelled, and a new one starts from the beginning. Therefore, the **shutdown** event can only occur if the time between two subsequent user actions exceeds 10 minutes.

The expression for the fill color of the circle tests whether the shutdown timeout is active and displays green or yellow color.

Example 8.5: Event slows down the simulation on a particular date

Suppose your model has some initialization or warm-up period that is not interesting to your audience, and you wish to skip it during the demonstration. You can use events to set up a demonstration scenario so that the model starts in very fast virtual time mode and, on a certain date, slows down to real time simulation mode.

Follow these steps:

1. Repeat the steps 1-2 of Example 1.1: "Event writes to the model log every time unit" (or just copy the flowchart from there).

2. In the **Projects** view click the **Simulation: Main** experiment of the model. The properties of the experiment are shown in the **Properties** view.
3. Open the **Model time** section of the properties and set the **Start date** to January 1, 2020. Make sure that the model is not going to stop (the **Stop** is set to **Never**).
4. In the same section, set the **Execution mode** to **Virtual time (as fast as possible)**.
5. Open the editor of the **Main** agent type where the flowchart is drawn.
6. Open the **Agent** palette and drag the **Event** element to the graphical editor. Name the event **slowDown**.
7. In the event properties leave the **Trigger type** set to **Timeout** and **Mode** set to **Occurs once**.
8. Select the **Use calendar dates** option.
9. Set the **Occurrence date** to February 1, 2020.
10. Expand the **Action** section of the event properties and type the following code there: **getEngine().setRealTimeMode(true);**
11. Run the model. Open the developer panel and watch how fast the first month is simulated (the model date is shown at the top of the panel). Then the model switches to the real time mode.

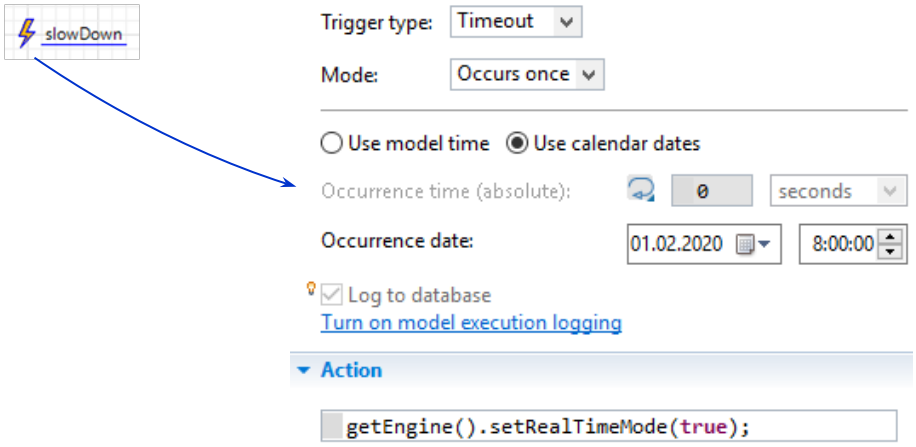

Figure 8.9 Event slows down the simulation on a particular date

In this example we are using calendar to define both the simulation start date and the event occurrence date (alternatively you could set **31 days** in the **Occurrence time (absolute)** field). The event **slowDown** occurs once and remains inactive after that. You can use such events to perform any one-time actions in the model.

Even if the event is specified with **Occurs once** option it can be scheduled again after occurrence by calling its **restart(time)** function.

Event API

Just like everything in AnyLogic, events are Java objects and expose their API to the modeler. Here is a summary of the functions:

- **reset()** – cancels the currently scheduled occurrence of the event, if any. In case the event is a cyclic timeout or a rate, the cycle would not resume until the **restart()** function is called.
- **restart()** – cancels the currently scheduled event, if any, and schedules the next occurrence according to the event mode.
- **restart(double timeout)** – cancels the currently scheduled occurrence of the event, if any, and schedules the new occurrence in time **timeout**. **timeout** is defined in model time units. If the event is a cyclic timeout or a rate, it will then continue occurring at the original timeout/rate.
- **restart(double timeout, TimeUnits units)** – similar to **restart(double timeout)** except that **timeout** is defined in the specified time units, e.g. **restart (10, SECOND)**.
- **suspend()** – cancels the currently scheduled occurrence of the event, if any, and remembers the remaining time to that occurrence so that it can be resumed by calling **resume()**. If the event is not scheduled at the time of calling **suspend()**, the subsequent resume will result in nothing.
- **resume()** – reschedules the previously suspended event in the remaining time.
- **double getRest()** – returns the time remaining to the next scheduled occurrence of the event in model time units. If the event is not scheduled, the function returns **infinity**.
- **double getRest(TimeUnits units)** – similar to **getRest()** except that the time is returned in the specified time units, e.g. **getRest(MINUTE)** returns the remaining time in minutes.
- **boolean isActive()** – returns **true** if the event is currently scheduled, and **false** otherwise.

8.3. Dynamic events

Imagine you are modeling product delivery by the postal service. After you send the product it gets delivered in 2 to 5 days and you ship several products per day, so multiple products may be in the delivery stage simultaneously. Which AnyLogic construct would you use? If delivery is a part of the process model you can use a **Delay** block of the Process Modeling Library. However, if there is no process (e.g. if this is an agent-based model) you can use a lower level (and a more lightweight) construct – *dynamic event*. Dynamic event allows you to schedule an arbitrary number of occurrences in parallel, and each occurrence can be parameterized.

To create a dynamic event:

1. Open the **Agent** palette.
2. Drag the **Dynamic Event** element from the palette to the graphical editor.

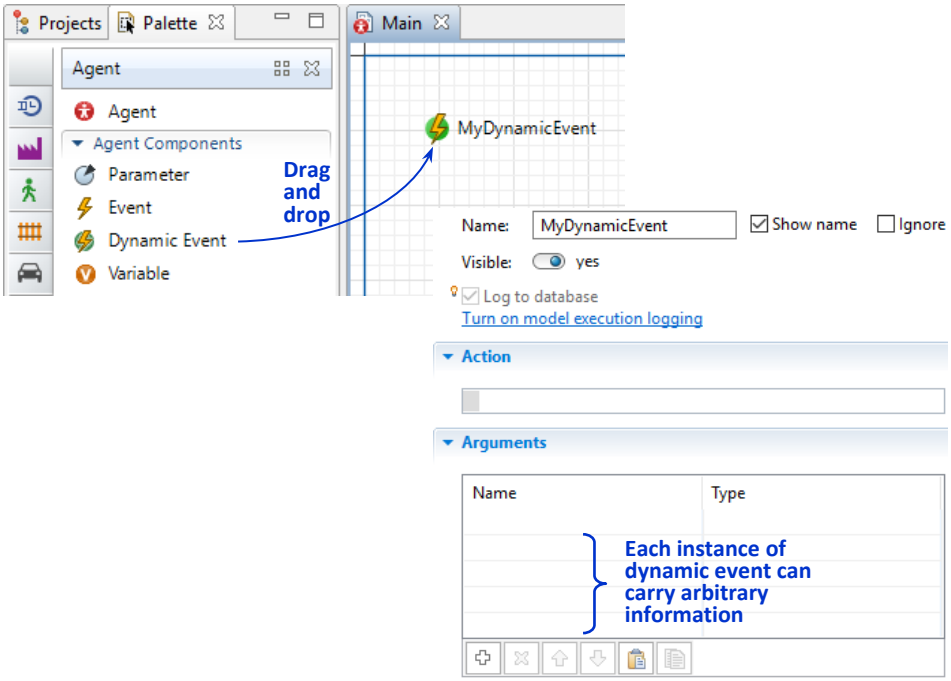

Figure 8.10 Creating a new dynamic event

Just like the static event described in the previous section, the dynamic event has an action field where you can define what should be done when the event occurs. However, unlike the static event, which persists throughout the simulation run, the dynamic event can be considered as a template, or a class, whose instances are created when you schedule an event and are destroyed immediately after occurrence. An instance of a dynamic event only exists while it is scheduled. Each instance can carry arbitrary information, and that information can be accessed when the action is executed upon event occurrence. For example, if dynamic events are used to model product delivery, each event instance can carry the product being delivered.

To schedule a dynamic event, you need to call an auto-generated function of the following syntax:

```
create_<Dynamic event name>( <time interval>, <parameter1>, ... )
```

The first argument in the function call is the time in which you want the event to occur. It is followed by the parameter values – one for each parameter defined in the event properties. The time interval can be deterministic as well as stochastic.

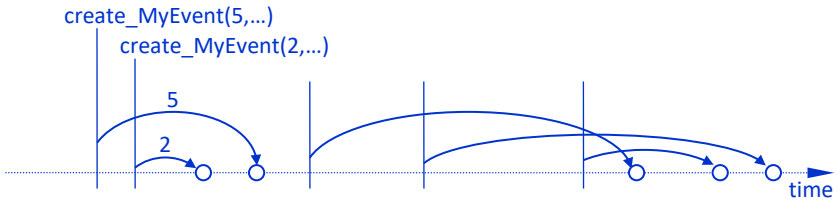

Figure 8.11 Dynamic events – multiple instances can be scheduled in parallel

Since the dynamic event that you create in the editor is actually a *class* of events, its name is capitalized according to Java writing convention.

Example 8.6: Product delivery

Let's model product delivery. The product in this simple model will be identified by a string. The delivery time will have a triangular distribution (see Section 15.1) with a minimum of 2, a maximum of 5, and a mode of 3 days. The shipment will be invoked by the user clicking a button, and when the product is delivered we will write a confirmation line into the model log.

▶ **Follow these steps:**

1. Open the **Agent** palette and drag the **Dynamic Event** element to the graphical editor. Set the name of the dynamic event to **Delivery**.
2. In the **Arguments** section of the dynamic event properties add a parameter **product** of type **String**.
3. In the **Action** section type: `traceln( time() + " Delivered " + product + "!" );`
4. Open the **Controls** palette and drag the **Button** element nearby the dynamic event.
5. In the button properties set the **Label** to **Ship**.
6. In the **Action** section of the button properties type:

```
traceln( time() + " shipping A..." );
create_Delivery( triangular(2,3,5), "A" );
```

7. Run the model. Try clicking the button multiple times. Open the developer panel and watch the model log in the **Console**.

The dynamic event **Delivery** has one parameter of type **String**. It means that each instance of **Delivery** will have a string attached to it. When the user clicks the button, a new instance of the dynamic event is scheduled with the string "A" attached to it.

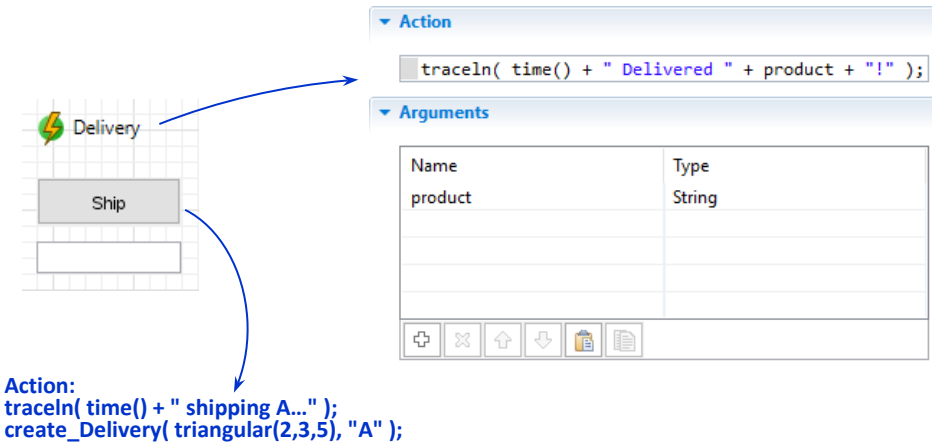

Figure 8.12 Dynamic event models delivery of a product

Let's extend the example a little. Instead of always shipping just one product "A" we will provide the user the ability to specify the product type in the edit box. We will also observe the scheduled dynamic events in the events panel.

Add a text field for the product name

8. Drag the **Edit box** item from the **Controls** palette under the button as shown in Figure 8.13.
9. Select the button and change its **Action** to:

```
String product = editbox.getText();
traceln( time() + " shipping " + product );
create_Delivery( triangular(2,3,5), product );
```

10. Run the model.
11. Type something in the edit box and click the button. Open the developer panel and watch the model log.
12. In the developer panel, switch from the **Console** to the **Events** panel, see Figure 8.13.
13. Try clicking the button multiple times and watch the events panel.

The product type is now taken from the edit box, whose function **getText()** returns a **String**. The **Events** panel displays all events (not just dynamic) currently scheduled in AnyLogic simulation engine. Each event is identified by the element that originated the event. **root.Delivery** means the root (top-level) agent of the model (the instance of **Main**) and **Delivery** dynamic event within that agent.

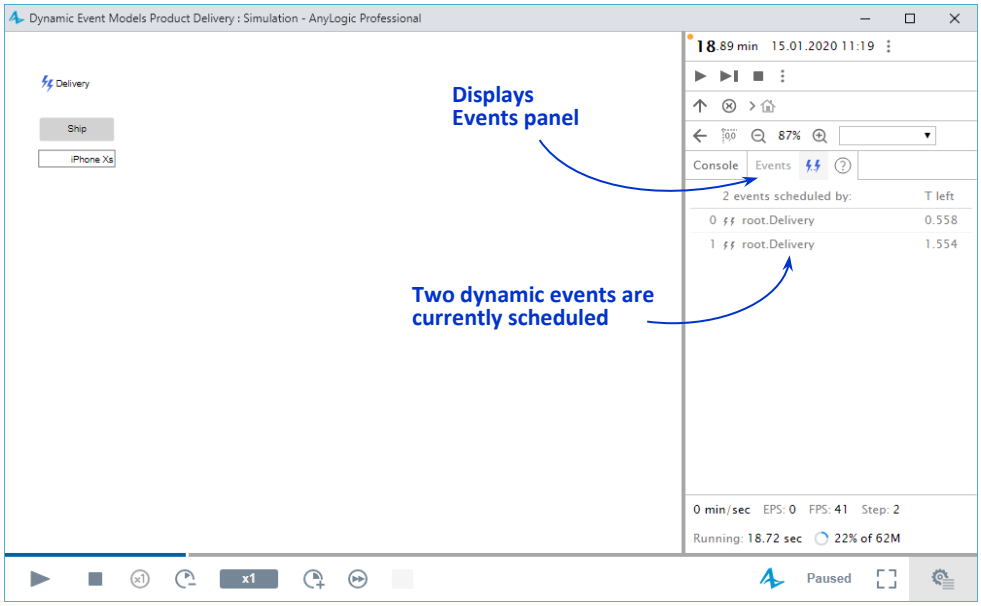

Figure 8.13 Scheduled dynamic events can be seen in the Events view

API related to dynamic events

Dynamic event items that you drag from the palette to the graphical editor become a Java *class* (an inner class of the agent type), and each call of **create_<Dynamic event name>()** creates a new object – an *instance* of that class. Each instance of a dynamic event has the following API:

- **reset()** – cancels and destroys the instance of the dynamic event.
- **double getRest()** – returns the time remaining to the occurrence of the event in model time units.
- **double getRest(TimeUnits units)** – returns the time remaining to the occurrence of the event in the specified time units.

To call the functions of a dynamic event instance you need to remember the instance. You can do it when you create it:

```
Delivery deliveryA = create_Delivery( 25, "A" );
...
double remainingTime = deliveryA.getRest();
```

You can retrieve the list of all dynamic event instances scheduled in an agent by calling the **Agent**'s function

```
Set<DynamicEvent> getDynamicEvents()
```

The function returns a set of all dynamic event instances of all classes defined within this agent. To find out the class of an event you can use the **instanceof** operator, see Section 10.2, "Classes".