Chapter 16. Model time, date and calendar. Virtual and real time

Time is the central axis in the dynamic simulation models we are building. The models are full of various references to time: delays, arrival times, service times, rates, timeouts, schedules, dates, speeds, etc. This chapter explains what model time is and how the user can work with it.

16.1. The model time

Model time is the virtual (simulated) time maintained by the AnyLogic simulation engine. The model time has nothing to do with the real time or the computer clock (although you can run the model in a scale to real time, see Section 16.3).

In AnyLogic, the model time takes Java **double** type values (real numbers with double precision). The model clock is advanced in steps: while the engine is executing a discrete event model, the model time jumps from one event to another (see Section 8.1); if a continuous-time model is being executed, the time steps are typically smaller and have equal size.

As all events in AnyLogic are instantaneous and indivisible, the model time does not progress during event execution, no matter how long it takes to complete all the computations associated with the event.

The current model time can be obtained during execution of the model by calling:

- **double time()** – returns the current model time.

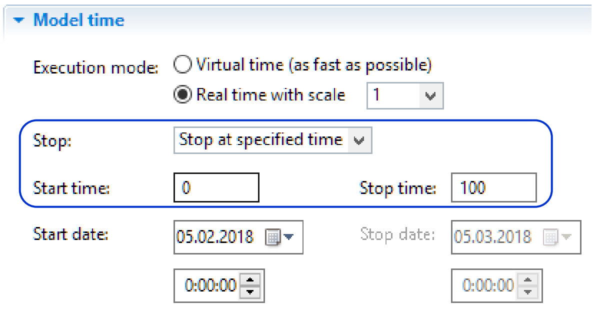
At the beginning of a simulation run, the time typically equals 0, although you can change it. You can set the *stop time*, the time at which you wish the simulation run to be terminated.

▶ To set the start and stop times of a simulation run:

1. Select the experiment and expand its **Model time** properties section.
2. Set the initial value of the model clock in the **Start time** field (by default, it is 0, and typically you do not need to change it).
3. If you wish the model to stop at a specific time, set **Stop** to **Stop at specified time** and enter the time in the **Stop time** field.
4. If you do not want the model to stop at a specific time, set **Stop** to **Never.**

Setting **Stop** to **Never** does not mean the simulation will run infinitely long. It can be terminated in several other ways: programmatically, using a stop condition, or when the engine detects there are no more dynamics left in the model.

You can set the stop date instead of the stop time (see Section 16.2).


Model time

Execution mode: ( ) Virtual time (as fast as possible) (●) Real time with scale 1

Stop: Stop at specified time

Start time: 0 Stop time: 100

Start date: 05.02.2018 Stop date: 05.03.2018

0:00:00 0:00:00

Figure 16.1 Setting the start and stop times

Time units

To establish the correspondence between the model time and real-world time where the system being modeled lives, we need to define the *time units*. The type of time unit depends on the time scale of the activities you are modeling. For example, if you are modeling a call center where the call durations are measured in seconds or minutes, you may set the time units to seconds or minutes. If you are modeling a supply chain, where manufacturing and shipping times are measured in days, days would be the right choice.

To set the time units:

1. Select the model (topmost) item in the **Projects** tree and set the time units in the **Model time units** property.

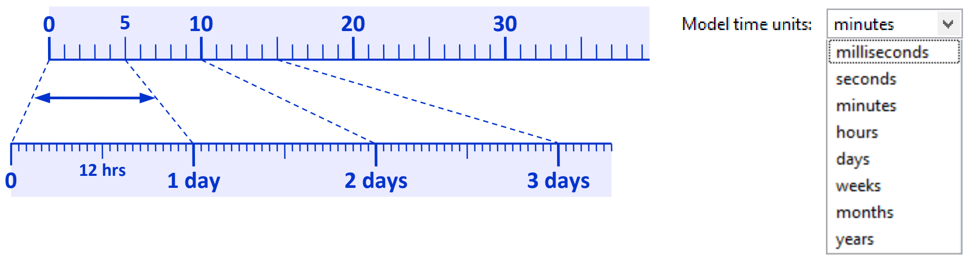


Figure 16.2 Setting the time units

The available time units are given in the Table below. The time unit constants are contained in the enumeration **TimeUnits**.

Time units	AnyLogic constant	Value
milliseconds	**MILLISECOND**	1
seconds	**SECOND**	1000
minutes	**MINUTE**	60*1000
hours	**HOUR**	60*60*1000
days	**DAY**	24*60*60*1000
weeks	**WEEK**	7*24*60*60*1000
months	**MONTH**	30*7*24*60*60*1000
years	**YEAR**	365*30*7*24*60*60*1000

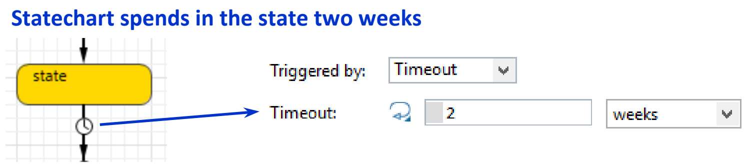
AnyLogic offers the following function to obtain the model time units. (It is the function of the engine; therefore, if it is called from an experiment or an agent, it must be accompanied by the "**getEngine().**" prefix):

- **TimeUnits getTimeUnit()** – returns the current time unit, namely a constant from the Table above.

Any time unit longer than a week is not a constant: a month may have 28, 29, 30 or 31 days, and a year can have 365 or 366 days. If you choose, say, months as the time unit in your model, then 1 month in the model will always have 30 days. In this case the calendar months will obviously differ slightly from months based on time units and the longer the period, the bigger the error. The same happens when you choose years as your model time units: every year in the model will be 365 days long.

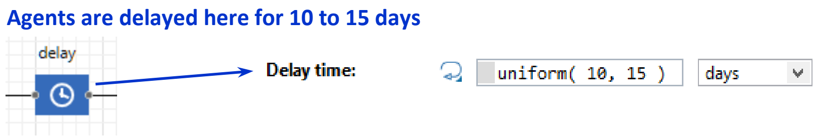
Developing models independent of time unit settings

Most model elements accept time (**double** type values measured in given time units) as their parameters: event and transition timeouts, delay and interarrival times in the Process Modeling Library blocks, etc. When you specify a time interval in some property, you have to specify time units for this interval to the right of the field, see Figure 16.3.



Triggered by: Timeout

Timeout: 2 weeks



Delay time: `uniform( 10, 15 )` days

Figure 16.3 Specifying time intervals in the properties of model elements

Most time-related functions of model elements have the dedicated argument to specify units for the given time interval as well, e.g. the **restart(double timeout, TimeUnits units)** function of the event. However, when you work with time in the code of your functions or in the actions of your model elements, the intervals are specified in model time units, and there is no explicit way to specify other units. Let's say the time unit in your model is hours. What if you need to schedule something to happen in 2 days? Or how would you define a duration of 5 minutes? Of course, you could write 48 and 5.0/60. But a much better solution is to use the special functions that return the value of a given time interval with respect to the current time unit settings:

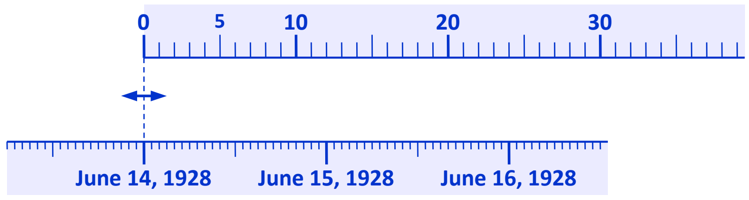
- **double millisecond()** – returns the value of a one-millisecond time interval.
- **double second()** – returns the value of a one-second time interval.
- **double minute()** – returns the value of a one-minute time interval.
- **double hour()** – returns the value of a one-hour time interval.
- **double day()** – returns the value of a one-day time interval.
- **double week()** – returns the value of a one-week time interval.

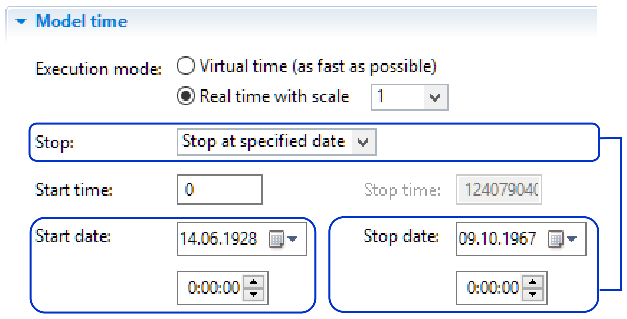
For example, if the time unit is hours, **minute()** will return 0.0166, and **week()** will return 168.0. Thus, instead of remembering what the current time unit is and writing 48 or 5.0/60, you can simply write **2*day()** and **5*minute()**. You can also combine different units in one expression: **3*hour() + 20*minute()**.

What is probably even more important about these functions is that the expressions using them are completely independent of the time unit settings: the expressions always evaluate to the correct time intervals. Therefore, we recommend always using multipliers such as **minute()**, **hour()**, **day()**, etc. when you work with time in functions and actions of model elements: this way, you can freely change the time units without changing the model.

16.2. Date and calendar

The start point of the simulation is always tied to a particular date. This is also done in the experiment's **Model time** properties section.




Model time

Execution mode: ○ Virtual time (as fast as possible)
● Real time with scale 1

Stop: Stop at specified date

Start time: 0 Stop time: 12407904(

Start date: 14.06.1928 0:00:00 Stop date: 09.10.1967 0:00:00

Figure 16.4 Setting the start and stop dates

To set the simulation start date:

1. Select the experiment and expand its **Model time** properties section.
2. Use the **Start date** control to set the start date.

By default, the *start date* is set to the date when the model is created.

If **Stop** is set to **Stop at specified time**, the end date of the simulation will be:

Start date + Time unit * (Stop time – Start time)

Alternatively, you can set the stop date explicitly:

To set the simulation stop date:

1. In the same properties section, set **Stop** to **Stop at specified date.**
2. Use the **Stop date** control to set the stop date.

The stop time will automatically recalculate.

You may need to efficiently manipulate dates and convert dates into time values and vice versa. AnyLogic provides a rich API for that purpose.

Finding out the current date, day of week, hour of day, etc.

The date in AnyLogic is stored in the form of the Java class **Date**. **Date** is composed of the year, month, day of month, hour of the day, minute, second and millisecond. To find out the current date, you should call:

- **Date date()** – returns the current model date.

A number of functions return particular components of the current date (and all those functions also have the form with parameter **<function name>(Date date)**, in which case they return the component of a given, not current, date):

- **int getYear()** – returns the year of the current date.
- **int getMonth()** – returns the month of the current date: one of the constants **JANUARY**, **FEBRUARY**, **MARCH**, …
- **int getDayOfMonth()** – returns the day of the month of the current date: 1, 2, …
- **int getDayOfWeek()** – returns the day of the week of the current date: one of the constants **SUNDAY**, **MONDAY**, …
- **int getHourOfDay()** – returns the hour of the day of the current date in 24-hour format: for 10:20 PM, will return 22.
- **int getHour()** – returns the hour of the day of the current date in 12-hour format: for 10:20 PM, will return 10.
- **int getAmPm()** – returns the constant **AM** if the current date is before noon, and **PM** otherwise.
- **int getMinute()** – returns the minute within the hour of the current date.
- **int getSecond()** – returns the second within the minute of the current date.
- **int getMillisecond()** – returns the millisecond within the second of the current date.

Consider a model of a processing center that operates from 9 AM to 6 PM on weekdays. The following function returns **true** if the center is currently open and **false** otherwise:

```
boolean isOpen() {
  int dayofweek = getDayOfWeek();
  if( dayofweek == SUNDAY || dayofweek == SATURDAY )
    return false;
  int hourofday = getHourOfDay(); //will be in 24-hour format
  return hourofday >= 9 && hourofday < 18;
}
```

Do not confuse the functions **hour()**, **minute()**, … with the functions **getHour()**, **getMinute()**… While **hour()** returns the duration of the hour in model time units, **getHour()** returns the current hour of the day.

Constructing dates. Converting the model date to the model time and vice versa

To create a **Date** object from its components (year, month, etc.), use the following function:

- **Date toDate(int year, int month, int day, int hourOfDay, int minute, int second)** – returns the date in the default time zone with the given field values.

To modify the given date, you can call:

- **Date addToDate(Date date, TimeUnits timeUnit, double amount)** – returns the date obtained by adding the given amount of specified time units to the original date (with respect to daylight saving time).
- **Date dropTime(Date date)** – returns the same date but with the time being set to 00:00:00.000.

To convert between a given model time and the model date, you can call:

- **Date timeToDate(double t)** – converts a given model time to the model date with respect to the start date, start time and model time unit settings; returns **null** if the time is infinity.
- **double dateToTime(Date d)** – converts a given model date to the model time with respect to the start date, start time and model time unit settings.

To find out how many days, months or years are contained between two given dates or model times, you can use the following two functions (one of the constants **YEAR**, **MONTH**, **WEEK**, **DAY**, **HOUR**, **MINUTE**, **SECOND**, **MILLISECOND** should be used as the **timeUnit**):

- **double differenceInCalendarUnits(TimeUnits timeUnit, Date date1, Date date2)** – returns the number of given time units between the two dates.
- **double differenceInCalendarUnits(TimeUnits timeUnit, double time1, double time2)** – returns the number of given time units between the two model times.

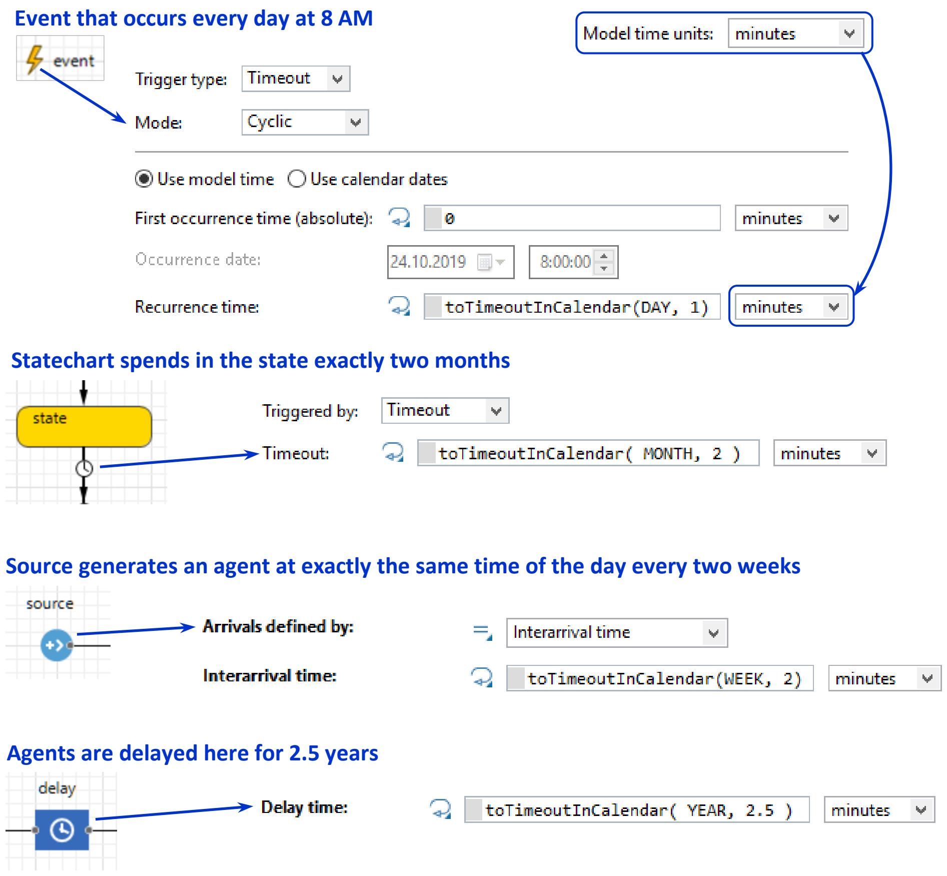
Specifying timeouts and delays in days, months, years

In simulation models, you sometimes need to schedule something to happen, for example, at same time next month or in 2.5 years. There is a simple way to obtain the corresponding timeout value that can be used in an event, statechart transition, **Source** or **Delay** block:

- **double toTimeoutInCalendar(TimeUnits units, double amount)** – returns the amount of time (in model time units) from the current time to the (current time + a given number of days, months, years, etc.) with respect to daylight saving time. Use one of the constants **YEAR**, **MONTH**, **WEEK**, **DAY**, **HOUR**, **MINUTE**, **SECOND**, **MILLISECOND** as the **units**.

Because of *daylight saving time,* the time interval from 8 AM today to 8 AM tomorrow is not always **24 * hour()** or **1 * day()**! Therefore, if you wish, for example, an event to occur at 8 AM every day, you should set the recurrence time to **toTimeoutInCalendar(DAY, 1)**.

Important: since **toTimeoutInCalendar()** function returns a value in model time units, you must choose your model's time units from the drop-down list to the right of the field where you call **toTimeoutInCalendar()**, see Figure 16.5.


Event that occurs every day at 8 AM
event
Model time units: minutes
Statechart spends in the state exactly two months
state
Source generates an agent at exactly the same time of the day every two weeks
source
Agents are delayed here for 2.5 years
delay

Trigger type: Timeout
Mode: Cyclic
(●) Use model time ( ) Use calendar dates
First occurrence time (absolute): 0 minutes
Occurrence date: 24.10.2019 8:00:00
Recurrence time: `toTimeoutInCalendar(DAY, 1)` minutes

Triggered by: Timeout
Timeout: `toTimeoutInCalendar( MONTH, 2 )` minutes

Arrivals defined by: Interarrival time
Interarrival time: `toTimeoutInCalendar(WEEK, 2)` minutes

Delay time: `toTimeoutInCalendar( YEAR, 2.5 )` minutes

Figure 16.5 Using toTimeoutInCalendar() function in events, statecharts and Process Modeling Library blocks

16.3. Virtual and real-time execution modes

AnyLogic can execute the simulation model in two modes, *virtual time* and *real time* on a given scale. Virtual time is the "natural" execution mode when the simulation engine executes the model as fast as possible. The model time progresses unevenly and not continuously relative to real time; see Figure 16.5. In discrete event models, the model clock may instantly jump to the next event or may stall at one point while several simultaneous events are being executed. The model execution rate may appear more continuous if the model contains continuous-time dynamics (as in system dynamics models): in that case, the model is driven by the numeric solver, which makes small time steps that are more or less even. The computational complexity of events and equations obviously affects the speed of the execution of the model.

The virtual time mode is used when simulation performance is important and animation of the model dynamics is not needed, in particular in optimization, sensitivity analysis, parameter variation, Monte Carlo and other experiments where the model is run multiple times. System dynamics modelers also use the virtual time mode as they are typically interested more in the output graphs of the simulation than in the simulation process itself.

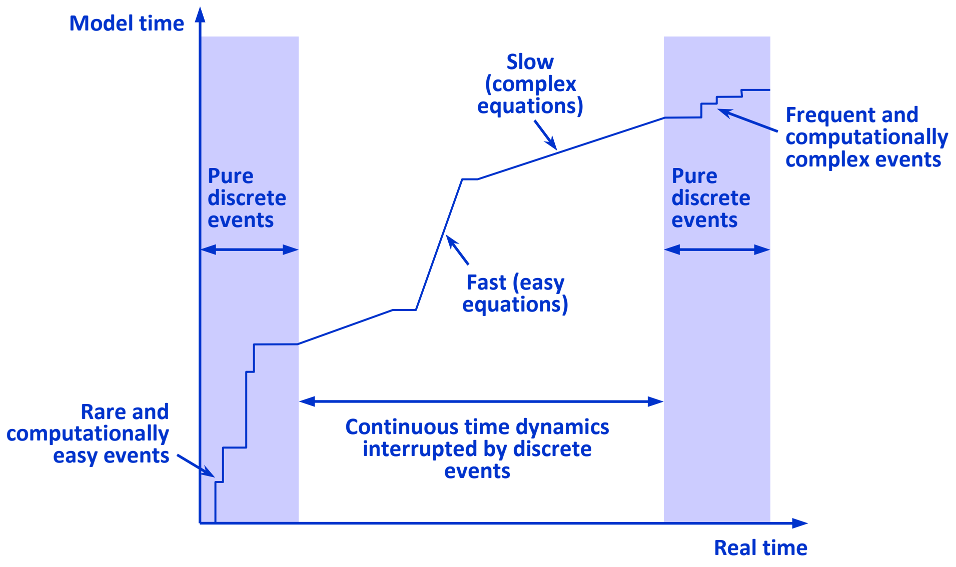

Model time
Slow (complex equations)
Frequent and computationally complex events
Pure discrete events
Pure discrete events
Fast (easy equations)
Rare and computationally easy events
Continuous time dynamics interrupted by discrete events
Real time

Figure 16.6 Virtual time ("natural") execution mode

In the real-time mode, the engine tries to keep to a given scale, say 10 model time units (e.g., 10 simulated weeks) per 1 real second. If the model's computational complexity is not too high, the engine will periodically put itself in the "sleep" state and wait for the correct real time to execute the next event or make the next step in the numeric calculations. Sometimes, though, the engine is unable to keep a given time scale because of too-frequent or too-complex events or because of a large system of equations and/or a too-small time step. Then the engine will work as fast as possible until it finds the next opportunity to maintain the real-time scale. Thus, the only thing the model can guarantee with respect to the real time is that the *model execution will never go faster than requested*; see Figure 16.6.

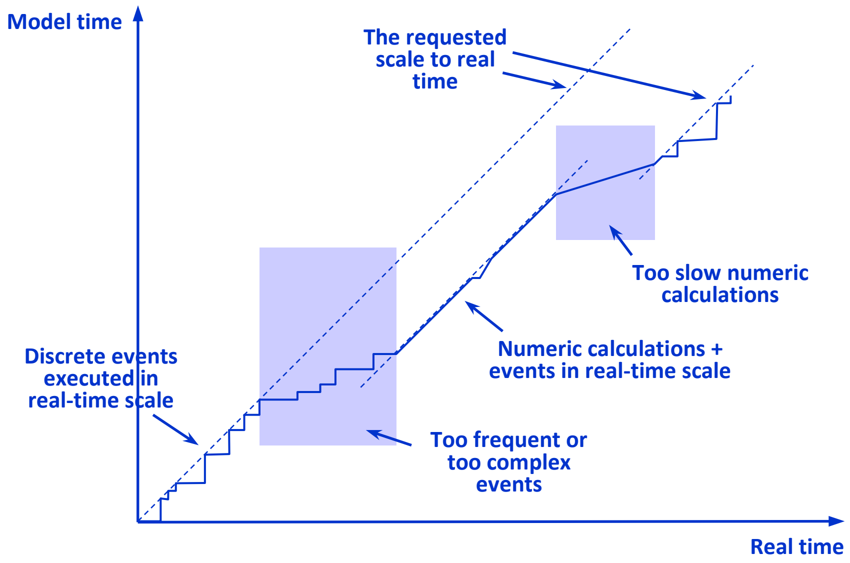
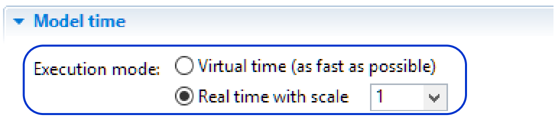

Model time
The requested scale to real time
Too slow numeric calculations
Numeric calculations + events in real-time scale
Discrete events executed in real-time scale
Too frequent or too complex events
Real time

Figure 16.7 Model execution in scale to real time

You can set the desired execution mode in the simulation experiment's **Model time** properties section (for other experiment types, the virtual time mode is assumed). Later on, you can use the controls in the model window to change the mode during runtime or do it programmatically.

▶ To set the default execution mode:

1. Select the simulation experiment and expand its **Model time** properties section.
2. Set the **Execution mode** to either **Virtual time (as fast as possible)** or **Real time with scale.** The available scales range from 1/500 to 500 model time units in one real second.


Model time

Execution mode: ○ Virtual time (as fast as possible) [✓ ● Real time with scale] 1

Figure 16.8 Setting the default execution mode

▶ To change the execution mode at runtime using the controls:

1. Locate the controls at the bottom of the model window.

2. Click to set the real-time mode with the default scale (1 model time unit per 1 real second). Click or to slow down or speed up execution of the model. Click for the virtual time mode.

Execution mode API

For programmatic control of the execution mode, AnyLogic offers the following functions of the engine (they must be accompanied by the "**getEngine().**" prefix):

- **setRealTimeMode(boolean on)** – sets the virtual (**on** is **false**) or real (**on** is **true**) mode of the model execution.
- **boolean getRealTimeMode()** – returns the current execution mode (**true** if real time, **false** if virtual time).
- **setRealTimeScale(double scale)** – sets the scale for the real-time mode (but does not change the current mode). The **scale** is the number of model time units to be simulated per real second.
- **double getRealTimeScale()** – returns the current scale setting of the real-time mode.

In Chapter 8 you can find the example of programmatic control of the execution mode (Example 8.5: "Event slows down the simulation on a particular date").