Chapter 15. Randomness in AnyLogic models

The world we live in is non-deterministic. If today it took you 35 minutes to get to work, tomorrow it may be an hour, so when you set off in the morning you never know exactly how long it will take. You may know that on Friday evening on average 30 people come to your restaurant for dinner, but the time the first customer comes in tells you nothing about when you should expect the next one. John Smith who works for you as a salesman may have excellent skills, but when he is dealing with a particular prospect you never know for sure if he will be able to close the deal. When your company starts a new R&D project, you always hope it will bring you revenue in the end, but you always know it can fail. If you contact a person with flu you can get infected or you can successfully resist it. You are alive today, but nobody knows if you will be alive tomorrow.

Uncertainty is an essential part of reality and a simulation model has to address it to reflect this reality correctly. The only way of doing that is to *incorporate randomness into the model*, i.e. include the points that would give random results each time you pass them during the model execution.

This chapter describes possible ways to create sources of randomness in the model; namely the probability distributions, the random number generators and where and how you can use them in different kinds of models.

15.1. Probability distributions

Suppose you are modeling a business process in a bank, in particular the operation of opening a new bank account. You know from your observations that it takes a minimum of 10 minutes, most likely 20 minutes, and a maximum 40 minutes, but you did not bother to make serious measurements. How do you model the delay time associated with this operation? AnyLogic offers you a set of *probability distribution functions* that will return random values distributed according to various laws each time you call them. For the purposes of this example we can use the function **triangular(min, mode, max)** with these parameters:

triangular(10, 20, 40)

If you call that function several times you will get a sequence of results like these:

Result	11.555	18.592	30.945	24.867	21.346	31.423	22.741	28.350

and so on. As you can see, the results appear to be random and more or less consistent with your observations. To explore the function **triangular()** more

thoroughly you can call it very many times and build a histogram of the results distribution. The histogram will look like the one in Figure 15.1.

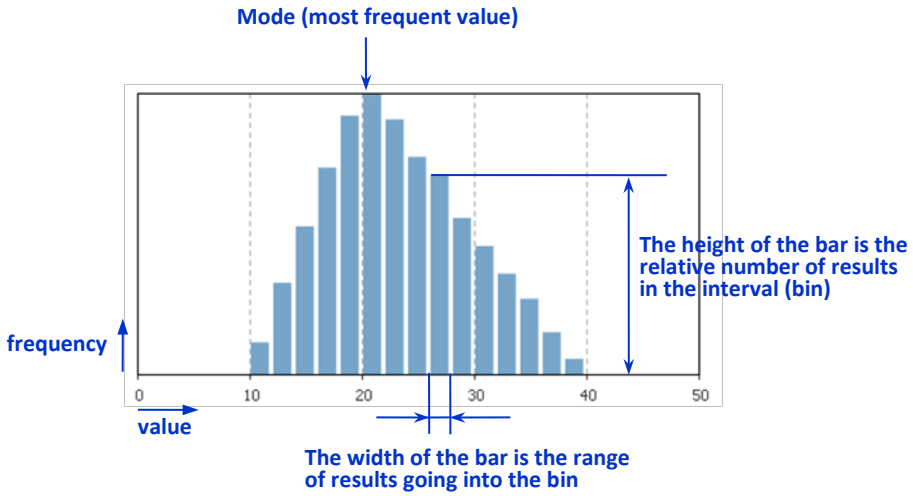


Figure 15.1 The distribution of 10,000 results returned by the triangular(10, 20, 40) function

The shape of the distribution (also called the *probability density function*, *PDF*) is a triangle with a minimum at 10, a maximum at 40 and a peak at 20 – the most frequent value also known as the *mode*. Indeed, the function **triangular(min, mode, max)** draws its results from the Triangular probability distribution with a given minimum, maximum, and most frequent values. We recommend using the triangular distribution function if you have limited sample data as in our bank example.

Depending on the type of your model you put a call to a probability distribution function, for example, into the **Delay time** parameter of a **Delay** or **Service** block, or into the **Timeout** field of a transition exiting the corresponding state, see Figure 15.2. Each agent passing the block (or coming to the state) will get a new sample of the distribution.

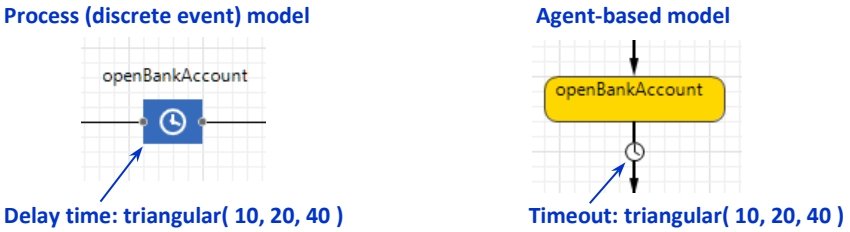
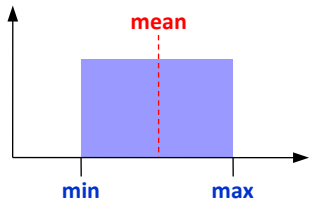
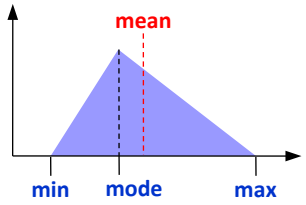
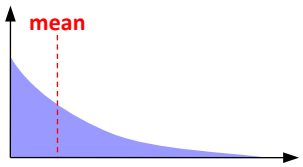
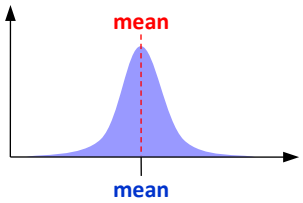
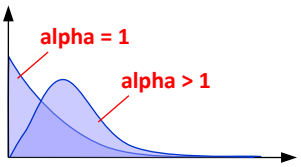


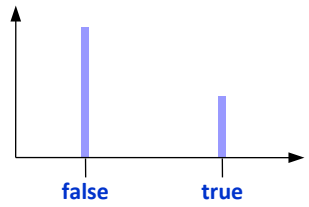
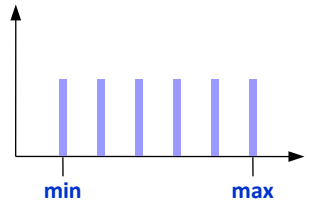
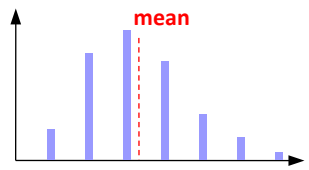
Figure 15.2 Using a probability distribution to model the duration of opening a bank account

Probability distribution functions

All randomness in AnyLogic models is based on calls to probability distribution functions. In total, AnyLogic supports about 25 distributions and offers over 50 corresponding functions. In the Table below we describe the most frequently used distributions. The complete information can be found in *Advanced Modeling with Java, AnyLogic functions* (The AnyLogic Company, 2019).

Distribution name, PDF form, and AnyLogic functions*,**	Primary use
Uniform  **uniform()** **uniform_pos()** **uniform(max)** **uniform(min, max)**	You know the minimum and the maximum values know nothing about how the values are distributed in between (i.e. you do not know if there are any values more frequent than others and assume a constant likelihood of a value being in any place between min and max). Used, for example, to generate coordinates of agents that are evenly spread over a rectangular area.
Triangular  **triangular(min, max)** **triangular(min, mode, max)** **triangular(min, max, mode)** **triangular(min, max, left, mode, right)**	You know the minimum, the maximum, and have a guess about the most likely (modal) value. Used, for example, for service times, travel times, or, in general, for the duration of operations in conditions of limited sample data (too few samples to build a meaningful distribution shape).

Exponential  **exponential()** **exponential(lambda)** **exponential(lambda, min)** **exponential(min, max, shift, stretch)**	Describes the times between events in a Poisson process, i.e. when events occur independently at a constant average rate. Used as the inter-arrival time for input streams of customers, parts, calls, orders, transactions or failures in process models. In agent-based models it is used as timeout for rate transitions (see Section 7.3) that model independent events in agents that are known to occur at a certain global average rate.
Normal  **normal()** **normal(sigma)** **normal(sigma, mean)** **normal(min, max, shift, stretch)**	Gives a good description of data that tend to cluster around the mean. For example, the height of an adult male person, the observation error in an experiment, etc. Note that the normal distribution is unbounded on both sides, so if you wish to impose limits (e.g. to avoid negative values) you must either use its truncated form, or use other distributions such as Lognormal, Weibull, Gamma or Beta.
Gamma  **gamma(alpha, beta)** **gamma(alpha, beta, min)** **gamma(min, max, alpha, shift, stretch)**	A distribution bounded on the lower side. If **alpha** (the shape parameter) is 1 it reduces to the exponential distribution; for larger values of **alpha** it starts at 0, then has a peak and decreases monotonically. Used to model, for example, lifetimes, lead times or personal income data.

Random boolean  **randomTrue(p)** **randomFalse(p)**	Used to make a random decision between two alternatives with a given probability. For example, in process models used to divide the flow of agents into two, for example, economy and business class passengers or regular and urgent orders. In agent-based models may be used in transition branches to model, for example. success or failure, and so on.
Discrete uniform  **uniform_discr(max)** **uniform_discr(min, max)**	Used to model a finite number of outcomes that are equally probable, or when you have no knowledge about which outcomes are more likely to occur. Example: a person (an agent) chooses a friend to communicate an idea. Note that both the minimum and maximum values are included in the set of possible results, so a call of **uniform_discr(3, 7)** may return 3, 4, 5, 6, or 7.
Poisson  **poisson(lambda)** **poisson(min, max, mean, shift, stretch)**	Discrete distribution describing the number of events occurring in a fixed period if these events occur independently and at a constant rate (**lambda**) Used to model, for example, the number of defects in a product or the number of calls in an hour, etc.
* There are many more probability distribution functions in AnyLogic. The full list with descriptions can be found in *Advanced Modeling with Java, AnyLogic functions* (The AnyLogic Company, 2019). ** Note that each distribution function has also a form **name(..., Random r)** which enables you to use a custom random number generator (see Section 15.3).	

As you can see, in general there may be more than one function for a distribution: a short form that assumes default parameter values, and longer forms that allow you to tune the distribution for your particular problem. For example:

- **normal()** – the Normal distribution with mean at 0 and sigma (standard deviation) = 1.
- **normal(sigma)** – mean at 0, custom standard deviation.
- **normal(sigma, mean)** – both mean and standard deviation can be customized.
- **normal(min, max, shift, stretch)** – same as above (**shift** is **mean**, **stretch** is **sigma**), but truncated to return values between **min** and **max**.

The latter form is provided for compatibility with Vensim™. *Truncation* of a distribution is performed in the following way: a draw from the original (not truncated) distribution is made, if the sample is outside the [min..max] interval, another try is made and so on until the sample is within the specified bounds.

Distribution fitting

If you are choosing the distribution without sufficient information you can follow the advice in the previous section. However, if you have a data set (a set of observations) which well characterizes the random values in the system you are modeling, you can choose the right distribution by *fitting* it to the data set. *Distribution fitting* is the process of finding the analytical formula of a distribution that describes the random value as closely as possible to a given data set. There are various fitting heuristics and goodness-of-fit tests (e.g. the *Kolmogorov-Smirnov test*) ("Kolmogorov–Smirnov test", n.d.) and a number of software packages that would automatically perform distribution fitting and suggest one or several analytical distributions.

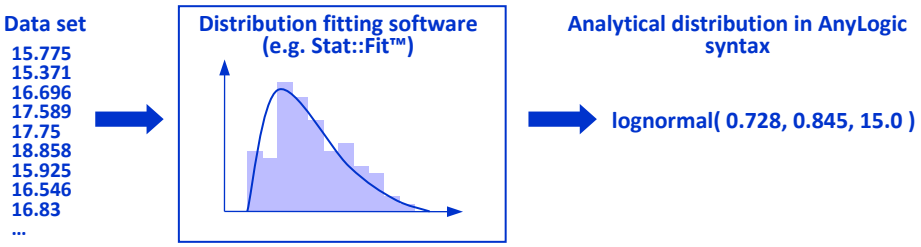


Figure 15.3 Input and output of distribution fitting software

Distribution fitting software would typically give you comprehensive statistics on your data set, display its histogram along with the PDF of the fitted distributions and rank the distributions according to several goodness-of-fit tests.

Should you be choosing distribution fitting software, it is recommended to use the one that directly supports the syntax of AnyLogic probability distribution functions, e.g. Stat::Fit (Geer Mountain Software Corporation, 2002). The output of such software can be directly copied into an AnyLogic model.

Custom (empirical) distributions

It may happen that no standard probability distribution can well fit the set of observations. In this case you can create a *custom* (also called *empirical*) distribution and use it in your model.

Example 15.1: Custom distribution of age groups

Suppose you have a population of people in your agent-based model and you want their ages to follow the observed data pattern.

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**. Click **Next**.
5. On the next page of the wizard leave the default animation. Click **Next**.
6. Create a parameter to store the agent's age value. In the **Parameters** table, click the **<add new...>** cell. In the fields on the right, name the parameter **age** and set its type to **int**. Click **Finish**. The **people** agent population is created in **Main**.
7. Open the **Agent** palette and drag the **Custom Distribution** element to the **Main** diagram. Name it **ageDistribution**.
8. Set the **Discrete** type for the distribution since we will use it to draw integer age values for the agents.
9. Set the **Define using** option to **Ranges** and define the age groups the population is comprised of. Create six age groups and define the percentage of total population for each of them. Enter the following values (they are taken from the (United States Census Bureau, 2019), see Figure 1.4).
10. Expand the **Preview** properties section to observe the histogram which is automatically generated based on the provided distribution.

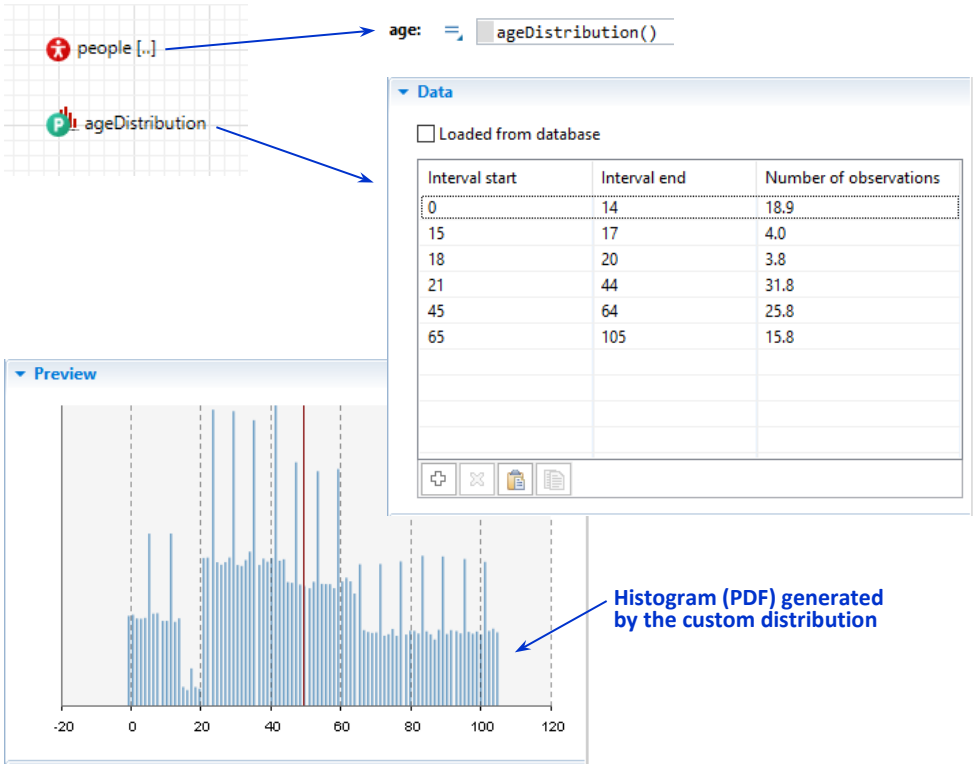


Figure 15.4 Custom (empirical) distribution of ages

If you have the historical data stored in a spreadsheet or a database, you can import data to the AnyLogic model's database and then simply select the **Loaded from database** option and specify the database table containing the required data.

If you have the data in a text editor, you can copy the data to the clipboard in e.g. tab-separated format and then paste the data by clicking the **Paste from Clipboard** button below the table in the **Data** properties section.

11. The custom distribution is ready to use. On the **Main** diagram, select the **people** agent population and place the distribution call in its **age** property: **ageDistribution()**
    This will draw age values from the empirical distribution.
12. Run the model and double-click the population to view its contents. You will see the diagram of the first agent of the population.
13. Open the developer panel and use the **people** navigation control to switch to the subsequent agents and observe their age values. As you can see, the values are assigned randomly, following the pattern determined by our source data.

15.2. Sources of randomness in the model

There are stochastic and deterministic models. A *deterministic model* has no internal randomness and, being run with the same set of input parameters, always drives the system through the same trajectory (the sequence of state changes) and gives the same output results. A *stochastic model* has internal sources of randomness, and each run (even with the same parameters) may give a different trajectory and output.

In general, there are more stochastic models than deterministic, especially among process (discrete event) models and agent-based models. System dynamics models are mostly deterministic. This is explained by abstraction level: process and agent-based models typically deal with individual objects, whose behavior has variations, while system dynamics deals with aggregates of large numbers of objects where individual variations are replaced (consumed) by averaging.

For a stochastic model you may need to perform multiple runs to get a meaningful picture of the system's behavior. The deterministic models are also often run multiple times with random variation of input parameters. A series of simulation runs with any kind of randomness (internal, at the level of input parameters, or both, see Figure 15.6) is called *Monte Carlo simulation*. The results of Monte Carlo simulation are statistically processed and typically have the form of probabilities, histograms, scatter plots or envelope graphs, etc.

Alternatively to performing multiple runs you can explore a stochastic model in a single run if each point of randomness is passed many times in a loop and the model enters a stochastically stable mode.

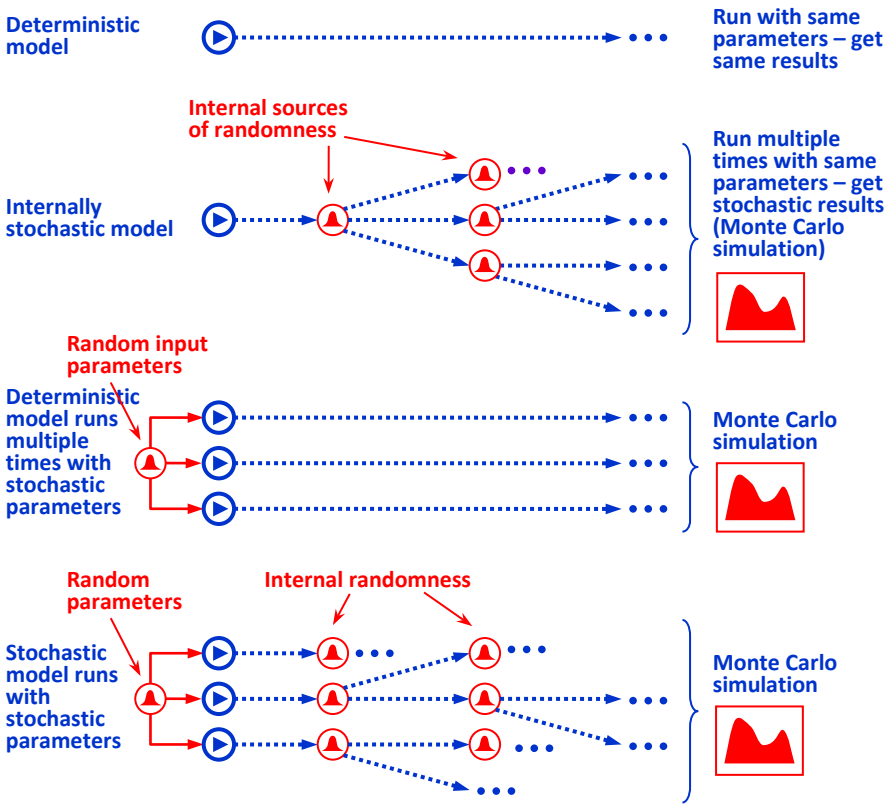


Figure 15.5 Sources of randomness inside and outside the model. Monte Carlo simulation

In this section we focus on internal sources of randomness in different types of AnyLogic models.

Randomness in process models

Randomness is an essential part of virtually any process model, be it a model of a manufacturing site, call center, warehouse, hospital, airport or a bank. The durations of operations, the arrivals of clients, orders or patients, the human decisions and errors, the equipment failures, the delivery times, etc. all vary randomly from one instance to another. Therefore, the process flowcharts typically contain plenty of calls to probability distribution functions.

By default, the fundamental process flowchart blocks of the Process Modeling Library have built-in randomness. These are (see Figure 15.7):

- **Source**: generates new agents with exponentially distributed inter-arrival time.

- **Delay** (and **Service** block based on it): has triangularly distributed delay time.
- **SelectOutput** (and its 5-exit version **SelectOutput5**): routes agents to different outputs with equal probability.

This means that, unless you explicitly eliminate the randomness from these blocks, any process model you build is stochastic.

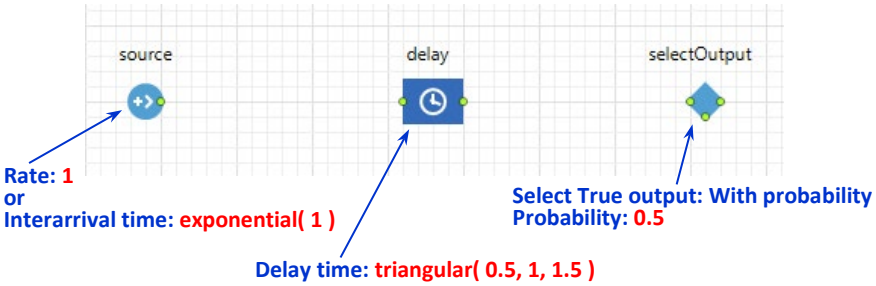


Figure 15.6 Randomness in the Process Modeling Library blocks

You may have noticed that if you run the simplest queuing model built of four blocks **Source**-**Queue**-**Delay**-**Sink** with default parameters for a longer period of time, **Queue** overflow will occur. This is caused by the fact that mean values of agent inter-arrival time and delay time both equal 1; see Figure 15.7. In these conditions the length of the queue has no limited mean.

Other typical sources of randomness in process models are randomly distributed properties of agents and resource units. For example, in the model of a call center an agent type **PhoneCall** may have a field **complexity** with a randomly assigned value and the call center operators (resource units of type **Operator**) may have different random skills in different groups; see Figure 15.8. Then the time it takes an operator to answer the call and the algorithm of call redirection may depend on the complexity and skill values.

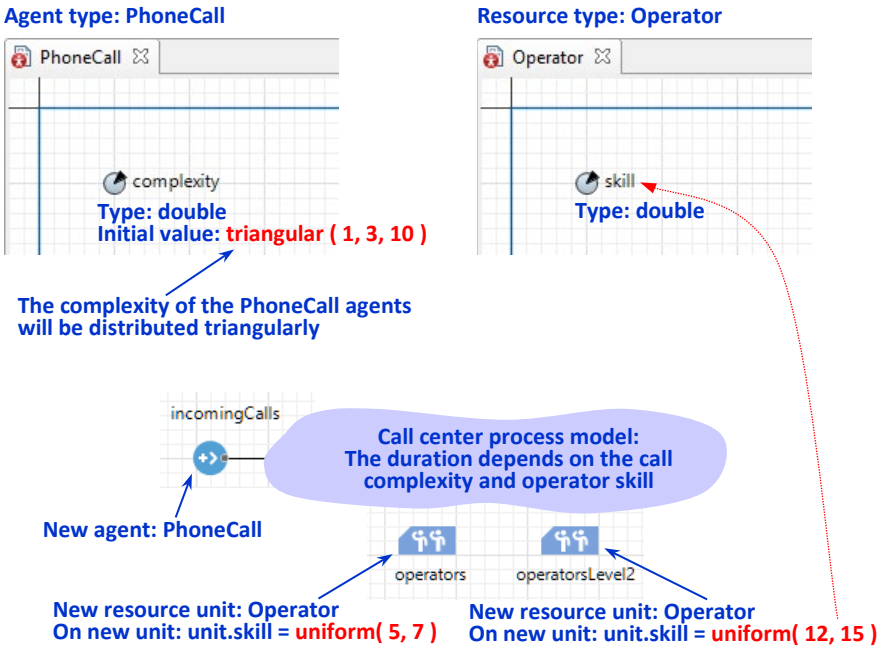


Figure 15.7 Random properties of agents and resource units in a call center model

Randomness in agent-based models

As well as process models, most agent-based models are stochastic too. In particular, randomness may be present in:

- The initial locations of the agents (if space is used)
- The network of agent connections
- The properties of agents
- The agent behavior; in particular in agent communication

A random layout is frequently used to evenly distribute the agents over a rectangular area; this is done using uniform distribution. **Random**, **Small world** and **Scale free** networks are constructed using link probabilities. Probability distributions are often used to set up the randomly distributed parameters over a population of agents, like the **income** parameter in Figure 15.9. While communicating with other agents a random friend or any random agent may be chosen. The transitions in the agents' statecharts may fire at random times and have nondeterministic results.

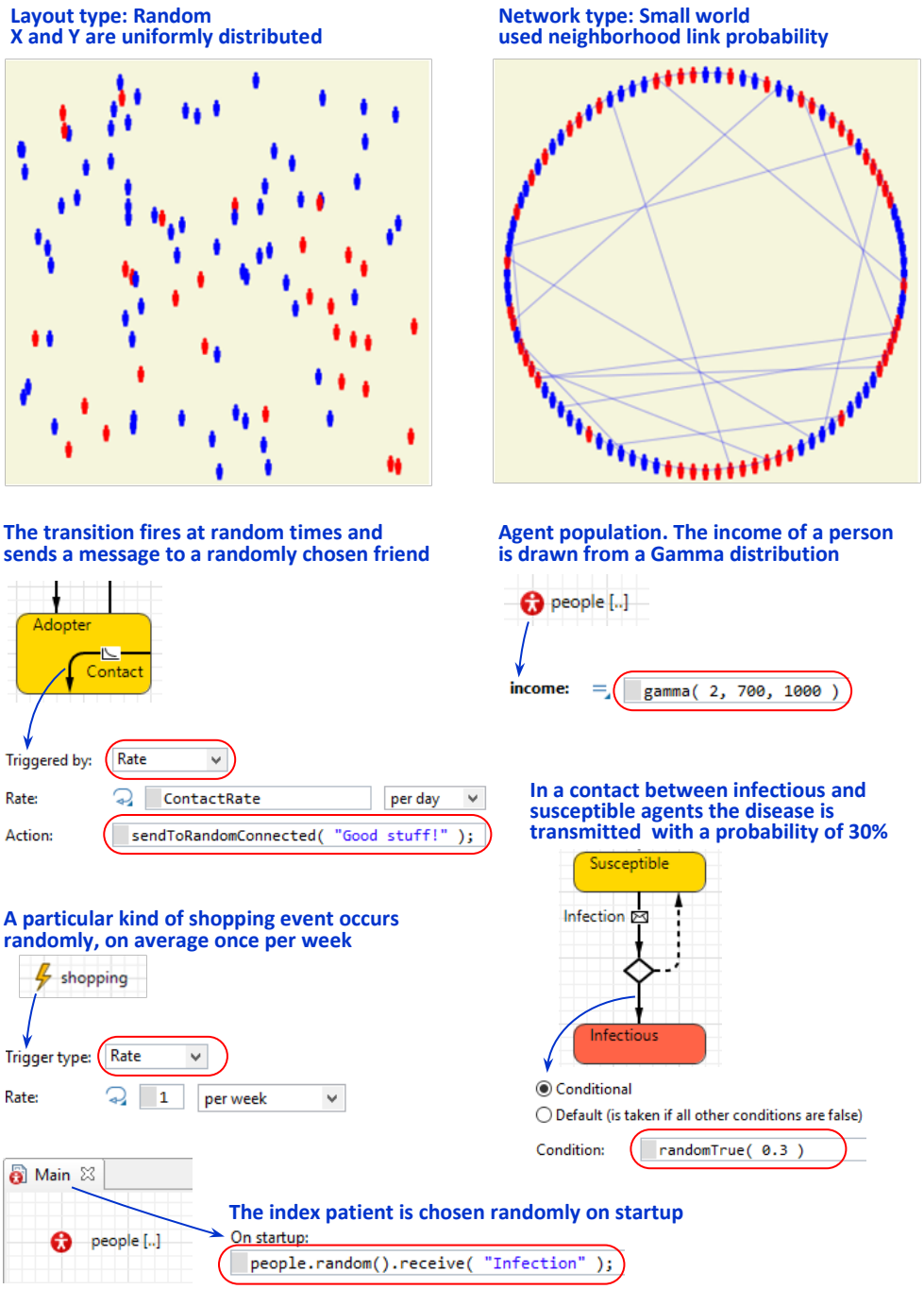


Figure 15.8 Randomness in agent-based models

Example 15.2: Agents randomly distributed within an area bounded by a polyline

Suppose in an agent-based model the agents need to be randomly distributed in an area bounded by a closed polyline or a curve (which may mean a city limit, for example). As the standard layouts only distribute agents over rectangular areas or rings only, you will need to create your own layout.

In this example we will randomly distribute agents within a closed polyline using the polyline's function **randomPointInside()**.

▶ **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**. 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 **Finish**.
6. The **people** agent population is created in **Main**. Open the population properties and set the **Initial number of agents** to **200**.
7. Open the **Presentation** palette, double-click the **Polyline** element and draw a polyline like the one in Figure 1.10 by clicking at node points. Use double-click to finish drawing.
8. Click the empty space of the graphical editor to display the properties of the **Main** agent type.
9. Expand the **Agent actions** section of the **Main** properties and type the following code in the **On startup code** field:

```
for ( Person p : people ) { //for each agent
  p.setLocation( polyline.randomPointInside() ); //set the location of the agent
}
```

10. Run the model. All agents should be randomly distributed within the polyline.

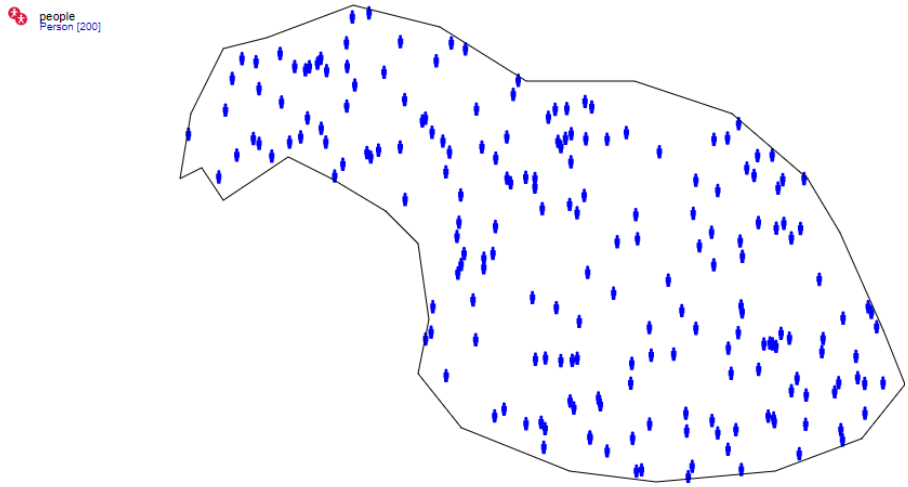


Figure 15.9 Agents randomly distributed within an area bounded by a polyline

In the startup code we are setting initial locations for the agents. In the **for** loop we iterate through all agents of the **people** population. For each agent we obtain a random point inside the closed polyline and place the agent at this point.

Example 15.3: Agents randomly distributed within an area bounded by a curve

The previous example demonstrates a simple way of distributing agents in an area bounded by a closed polyline. As the **randomPointInside()** function is not supported by curves, you will need to define a different algorithm.

Follow these steps:

1. Create a new model. Drag the **Agent** element from the **Agent** palette to the graphical editor.
2. On the first page of the **New agent** wizard, click the **Population of agents** option.
3. 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**.
4. On the next page switch to the **2D** option and select the **Person** shape from the list below as the agent's animation. Click **Finish**.
   A new agent-based model is created, and the editor of its **Main** agent type opens. The **people** agent population is created in **Main**. This population contains 100 agents of the type you have created (**Person**).
5. Open the **Presentation** palette, double-click the **Curve** element and draw a curve like the one in Figure 1.11 by clicking at node points. Use double-click

to finish drawing. Make sure the curve is fully inside the area starting at (50,200) and having the width of 550 and height of 400 pixels.

6. In the curve properties select the **Closed** checkbox.
7. Click the empty space of the editor to display the properties of the **Main** agent type.
8. In the **Agent actions** section of the properties write the following code in the **On startup** field:

```
for( Person p : people ) { //for each agent
  double x, y;
  do { //obtain a random point within the rectangle bounding the curve
    x = uniform( 200, 750 );
    y = uniform( 50, 450 );
  } while( ! curve.contains( x, y ) );
  p.setXY( x, y ); //set the coordinates of the agent
}
```

9. Run the model. All agents should be randomly distributed across the area bounded by the curve.

The algorithm above differs from the one used in the previous example to find a point within the polyline bounds. We first generate a pair of random numbers **x** and **y** so that **x** is within 200..750 and **y** is within 50..450. Then we test if the point (**x**,**y**) is contained in the curve. If the random point is not contained inside the curve, another point is generated. Sooner or later the point will be within the curve bounds. Once this happens, we set up the agent coordinates and proceed to the next agent. The performance of this algorithm (i.e. the percent of successful tries) depends on the ratio of the area inside the curve to the enclosing rectangular area where we generate points.

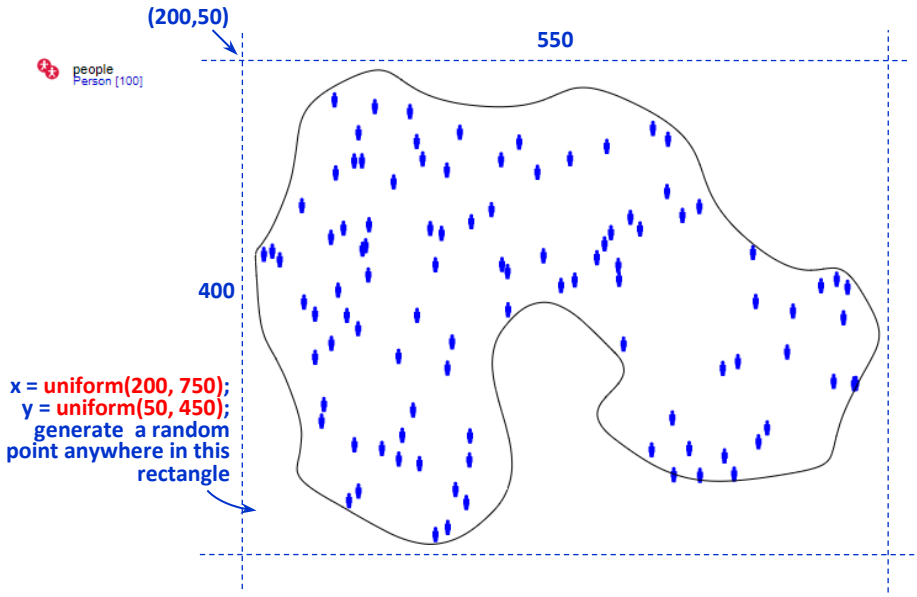


Figure 15.10 Agents randomly distributed within an area bounded by a curve

Randomness in system dynamics models

A system dynamics model built of standard elements, i.e. stocks, flows and feedback loops has no internal randomness (is deterministic) unless you explicitly insert it into the model. When doing that please follow the guidelines below.

In AnyLogic each new call to a probability distribution function generates a new value. Therefore you should not use those functions in the formulas of system dynamics models (see Chapter 5): the numeric solver evaluates the formulas *several times in one time step* and will be confused if it gets different results.

To model values randomly changing in time in system dynamics models you should create a variable or SD constant and assign a random value to it at each time step (or less frequently, depending on the model logic) using, for example, a cyclic event (see Section 8.2).

Example 15.4: Stock price fluctuations in a system dynamics model

Suppose in a system dynamics model you wish to have a variable for a stock price that randomly changes every day. A daily change is random and is not larger than 2 in either direction.

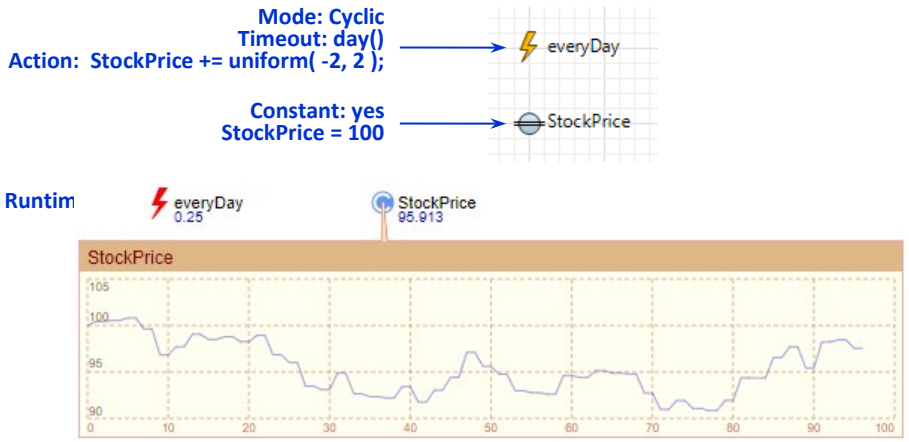


Figure 15.11 Random fluctuations of a stock price in a system dynamics model

Follow these steps:

1. Create a new model. In the **New Model** wizard, set **days** as the **Model time units**.
2. Open the **System Dynamics** palette and drag the **Dynamic Variable** element to the **Main** graphical diagram. Set the name of the variable to **StockPrice**.
3. In the properties of **StockPrice** select the **Constant** checkbox and set the value to **100**.
4. Open the **Agent** palette and drag the **Event** element to the graphical editor. Set the name of the event to **everyDay**.
5. In the event properties set **Mode** to **Cyclic** and set the **Recurrence time** to **1** day.
6. Expand the **Action** section of the event properties and type the following code in the field: **StockPrice += uniform(-2, 2);**
7. Run the model. Click the **StockPrice** and watch the changes in the inspect window (switch it to the chart mode).
8. You can now use the random variable **StockPrice** in any formula in the system dynamics model.

By marking the **StockPrice** as **Constant** we just tell AnyLogic that the value specified in its properties (in our case **100**) *should be treated as an initial value only and should not be treated as a formula and evaluated by the numeric solver*. Instead we are explicitly assigning a new value to the **StockPrice** every day by the event **everyDay**. (As an alternative to a constant system dynamics variable we could also use a variable.)

The uniform distribution was used as we do know the bounds of a daily change but do not know if any changes within those bounds are more likely than others.

Randomness in AnyLogic simulation engine

Besides the sources of randomness at the model level as discussed above, there is only one internal source in AnyLogic simulation engine, namely the ordering of simultaneous events. This topic is extensively addressed in the Chapter 8, "Discrete events and Event model object", here we would like to mention that:

- The engine uses the same default random number generator as the probability distribution functions do, and
- You can turn that randomness on and off:

To set the ordering mode for simultaneous events:

1. Select the experiment and expand the **Randomness** 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)**.

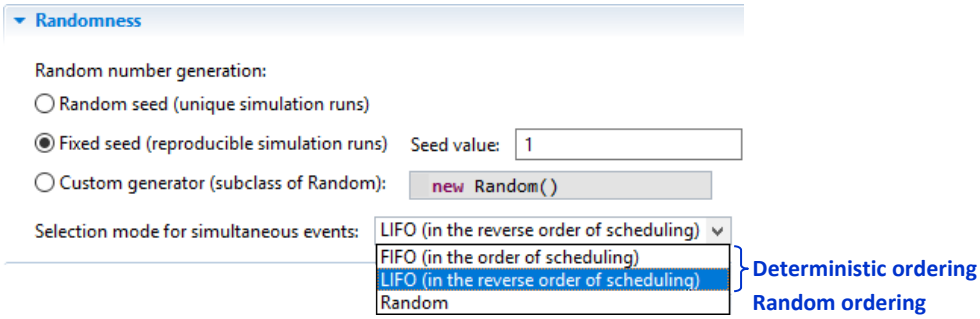


Figure 15.12 You can turn random ordering of simultaneous events on and off

15.3. Random number generators. Reproducible and unique experiments

The computer (or, at least, the processor executing a program) is a completely deterministic device: its next state is fully determined by the current state. So, when we talk about randomness in simulation models, did you ever wonder where that randomness comes from?

The truth is that, unless a software application accesses an external physical random number generator, there is no real randomness in it; however, a computational random number generation may be used to create pseudo-randomness.

Random number generators

A *random number generator* (*RNG*) is a device that generates a sequence of numbers that lack any pattern, i.e. appear random ("Random number generator," n.d.). There are two types of RNGs: physical and computational. *Physical RNGs* have been known from ancient times: dice, coin flipping, shuffling of playing cards, roulette wheel and other techniques. The modern ones use atomic and subatomic physical phenomena, such as radioactive decay or thermal noise, or (like (Random.org, 1998-2019)) atmospheric noise, or radio noise. Physical RNGs generate "true" random numbers, i.e. those that are completely unpredictable.

Computational RNGs are based on deterministic algorithms that produce long sequences of apparently random results, which are in fact completely determined by a shorter initial value, known as a *seed*, and are periodic. They are therefore called *pseudo-random number generators*. Pseudo-random number generators can be used instead of "true" ones in many applications; in particular, in the majority of simulation models. Moreover, their predictability feature is used to create reproducible stochastic experiments.

By default, all probability distribution functions in AnyLogic, the Process Modeling Library blocks, the random transitions and events, the random layouts and networks and the AnyLogic simulation engine itself – in other words, all randomness in AnyLogic, is based on the *default random number generator*. The default random number generator is an instance of the Java class **Random**, which is a *Linear Congruental Generator* (LCG). The LCG is one of the oldest and best known pseudo-random generators. It is very simple: the stream of random numbers is determined by the recurrence relation:

$$X_{n+1} = (aX_n + c) \bmod m$$

where a is a multiple, c is increment, and m is modulus. The period of LCG is at most m, and in the class **Random** $m = 2^{48}$. The initial value X_0 is a seed.

If for any reason you are not satisfied with the quality of **Random**, you can:

- Substitute the AnyLogic default RNG with your own RNG.
- Have multiple RNGs and explicitly specify which RNG should be used when calling a probability distribution function.

To substitute the default RNG with your own RNG:

1. Prepare your custom RNG. It should be a subclass of the Java class **Random**, e.g. **MyRandom**.
2. Select the experiment and expand the **Randomness** section of its properties.
3. Select the radio button **Custom generator (subclass of Random)** and in the field on the right write the expression returning an instance of your RNG, for

example:
new MyRandom() or **new MyRandom(1234)**

Figure 15.13 Setting a custom random number generator as default RNG

The initialization of the default RNG (provided by AnyLogic or by you) occurs during the initialization of the experiment and then before each simulation run.

In addition, you can substitute the default RNG at any time by calling:

setDefaultRandomGenerator(Random r)

However, you should be aware that before each simulation run the generator will be set up again according to the settings in the **Randomness** section of the experiment properties.

▶ **To use a custom RNG in a particular call of a probability distribution function:**

1. Create and initialize an instance of your custom RNG. For example, it may be a variable **myRNG** of class **Random** or its subclass.
2. When calling a probability distribution function, provide **myRNG** as the last parameter, for example:
   uniform(myRNG) or **triangular(5, 10, 25, myRNG)**

If a probability distribution function has several forms with different parameters, some of them may not have a variant with a custom RNG, but the one with the most complete parameter set always has it.

The seed. Reproducible and unique experiments

Although pseudo-random number generators do not produce "truly random" streams of numbers, they have one feature which is very important in simulation modeling: having been initialized with a particular seed they generate exactly the same sequence of numbers each time. This enables you to create *reproducible experiments* with stochastic models, which would be impossible with a "true" RNG. Reproducibility, i.e. the ability to run the model along the same trajectory of state changes, is useful when you are debugging the model, or when you wish to demonstrate a particular scenario.

In AnyLogic you have two options for the standard RNG, see Figure 15.14: you can choose **Random seed** to run unique experiments, or **Fixed seed** to run reproducible

experiments. The seed is set during the initialization of the experiment and then at the beginning of each simulation run (replication). If a custom RNG is provided, AnyLogic at those points just sets the default RNG to what is specified in the corresponding field.

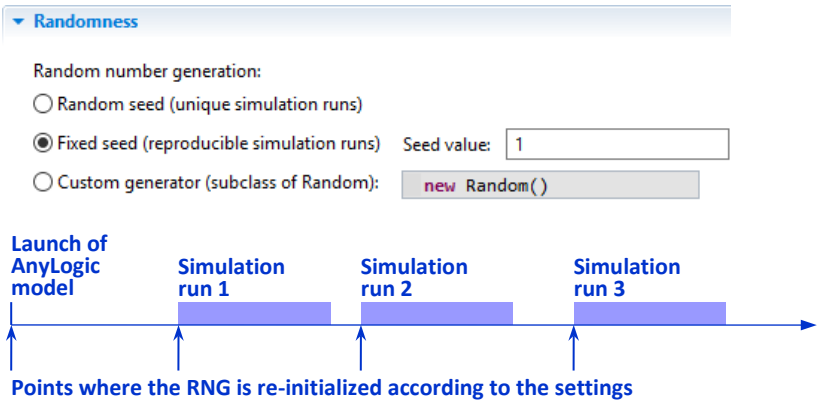


Figure 15.14 AnyLogic RNG seed settings and the points when they are applied

A valid question is: where is the "random seed" taken from? When the random seed option is chosen, AnyLogic calls the default constructor of Java class **Random()**, which sets the seed of the random number generator to a value very likely to be distinct from any other invocation of this constructor. In the earlier versions of Java the computer system time was used as a seed in that case.

Example 15.5: Reproducible experiment with a stochastic process model

We will create the simplest queuing system: a source of agents, a queue, a delay with capacity 1, and a sink block. The model will have two sources of randomness: the arrival times of the agents and the delay time: under the default settings the inter-arrival times are distributed exponentially and the delay times triangularly. (See Section 15.2, subsection "Randomness in process models".) We will record the agent exit times and compare them for different runs.

Create and run the model with default seed settings (fixed seed)

1. Create a new model. In the **New Model** wizard, leave the default **Model time units: seconds**.
2. Open the **Process Modeling Library** palette and create a flowchart as shown in Figure 1.15. 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. Select the **sink** block. In the **On enter** field of its **Actions** properties section type: `traceln( time() );` – each time the agent exits the model we will write the current time to the model log.
4. Click the empty space of the graphical editor to display the properties of the **Main** agent type.
5. In the **Agent actions** section of the **Main** properties write the following code in the **On destroy** field:
   `traceln( "------" );`
   This way we will separate outputs of different model runs in the Console.
6. In the **Projects** tree select the **Simulation** experiment. In the **Model time** section of the experiment properties set:
   Stop: **Stop at specified time**
   Stop time: **20**
7. Run the model up to completion. Open the developer panel and observe the model log in the **Console**: there should be about 20 records.
8. Click the **Stop** control in the model window and run the model again. Another portion of records should appear in the **Console**. Compare the outputs. They should be exactly the same.

You see that under the default settings of AnyLogic RNG the runs of the stochastic model produce the same results.

The important thing is that the results will be the same every time and everywhere: you may export your model, send it to a client, or upload it to the AnyLogic Cloud – and anybody who runs it will observe exactly the same behavior.

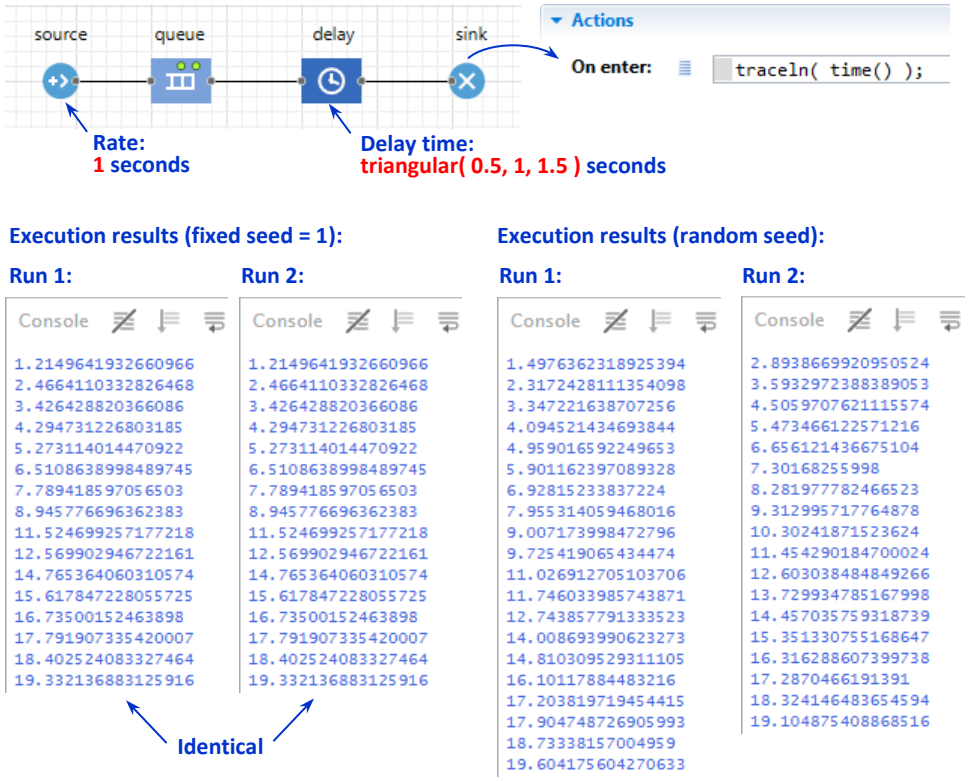


Figure 15.15 Reproducible and unique experiments with a stochastic process model

Now let us change the RNG settings.

Change the RNG settings to run unique experiments

9. In the **Projects** tree select the **Simulation** experiment. In the **Randomness** section of its properties choose **Random seed (unique simulation runs)**.
10. Run the model and observe the model output in the **Console**.
11. Run the model again and compare the outputs. The agent exit times differ from the ones in the first run.
12. Close the model window and run the model again. The output should be different again.

You see that with random seed each run of the stochastic model is unique.