VisualAivika Manual

David E. Sorokin <davsor@mail.ru>,
Yoshkar-Ola, Mari El, Russia

April 12, 2026

Contents

Chapter 1

Getting Started

VisualAivika is a visual simulation software tool. It is focused on System Dynamics and Discrete Event Simulation (DES). There is an easy-to-use diagram editor that allows creating nice looking Stock and Flow Maps (SFM) as shown in figure 1.1. These diagrams can be used for discrete event simulation models too.

There is also a simulation component that supports its own high-level modeling language. The model equations are displayed in a separate tab panels as demonstrated in figure 1.2. The user can switch between diagrams and equations.

There is the *Equation Editor*, where you can define integrals, arrays, random functions — look at figure 1.3. The editor is opened right after one of the entities is selected on the diagram.

Moreover, VisualAivika supports the Monte-Carlo simulation, which allows providing Sensitivity Analysis. There are means for plotting charts on the diagram to show the results of simulation in a form of Time Series, XY Chart and Deviation Chart as shown in figure 1.4. The results can be exported as CSV data files.

Futhermore, VisualAivika supports arrays and subscript. Figure 1.5 shows the chart that corresponds to some array.

Finally, before starting the simulation, the user can define the simulation specs as illustrated in figure 1.6.

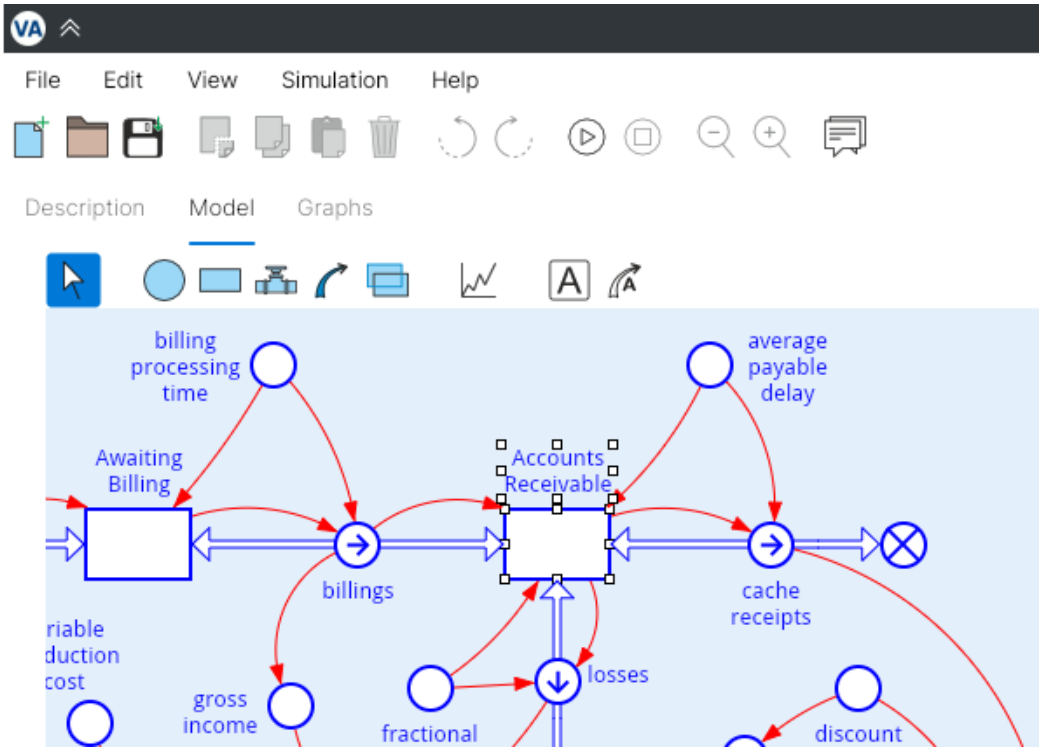

Figure 1.1: Editing the SFM diagram.

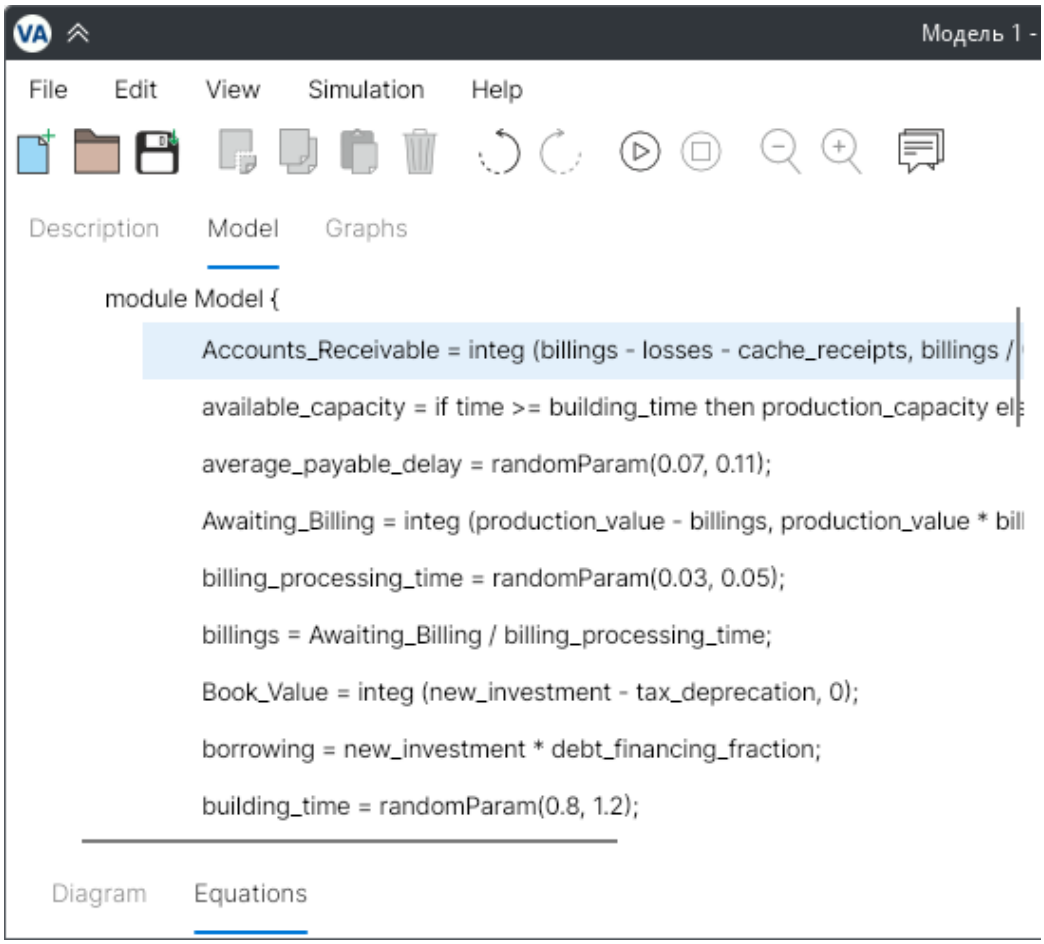

Figure 1.2: The *Equations* panel.

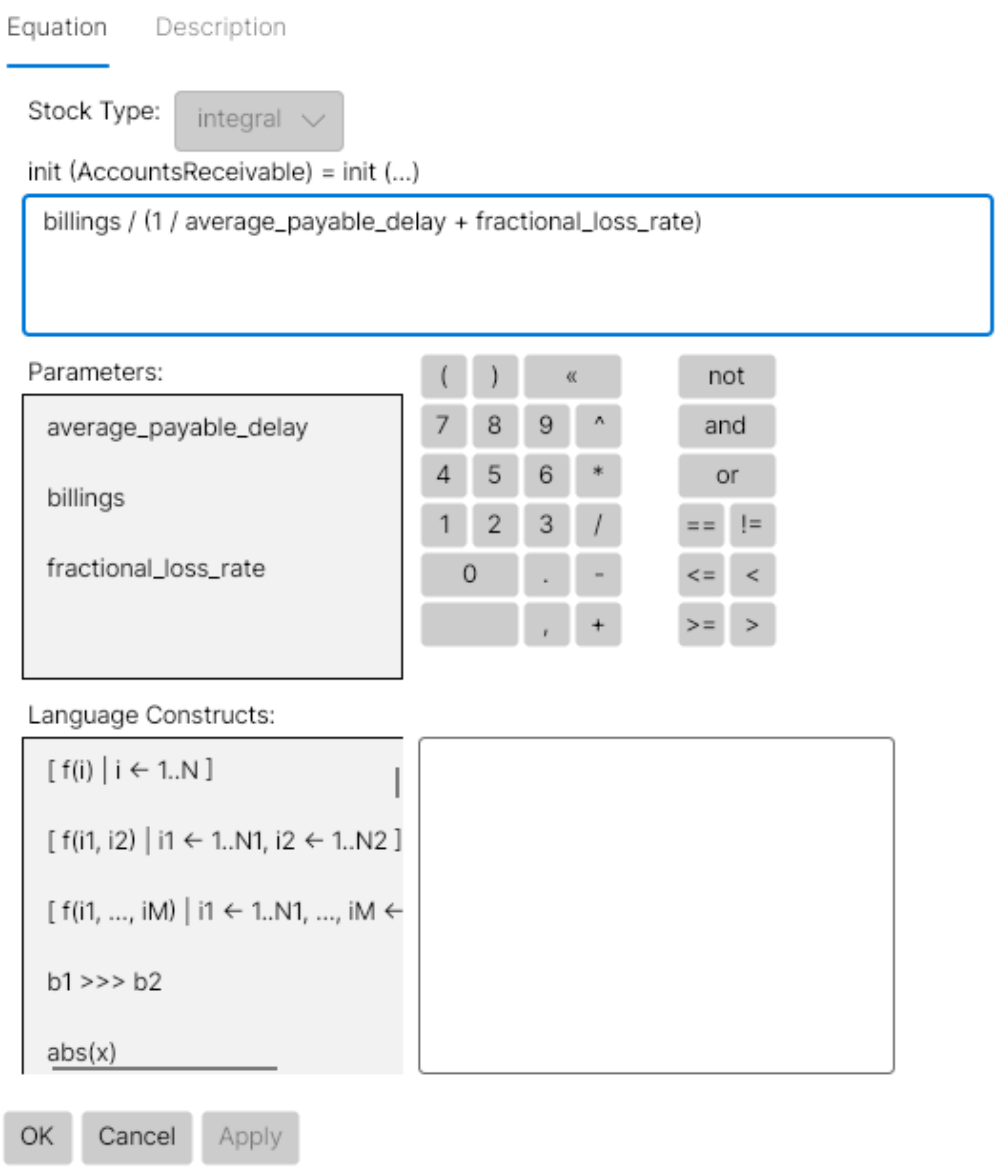

Figure 1.3: The *Equation Editor* panel.

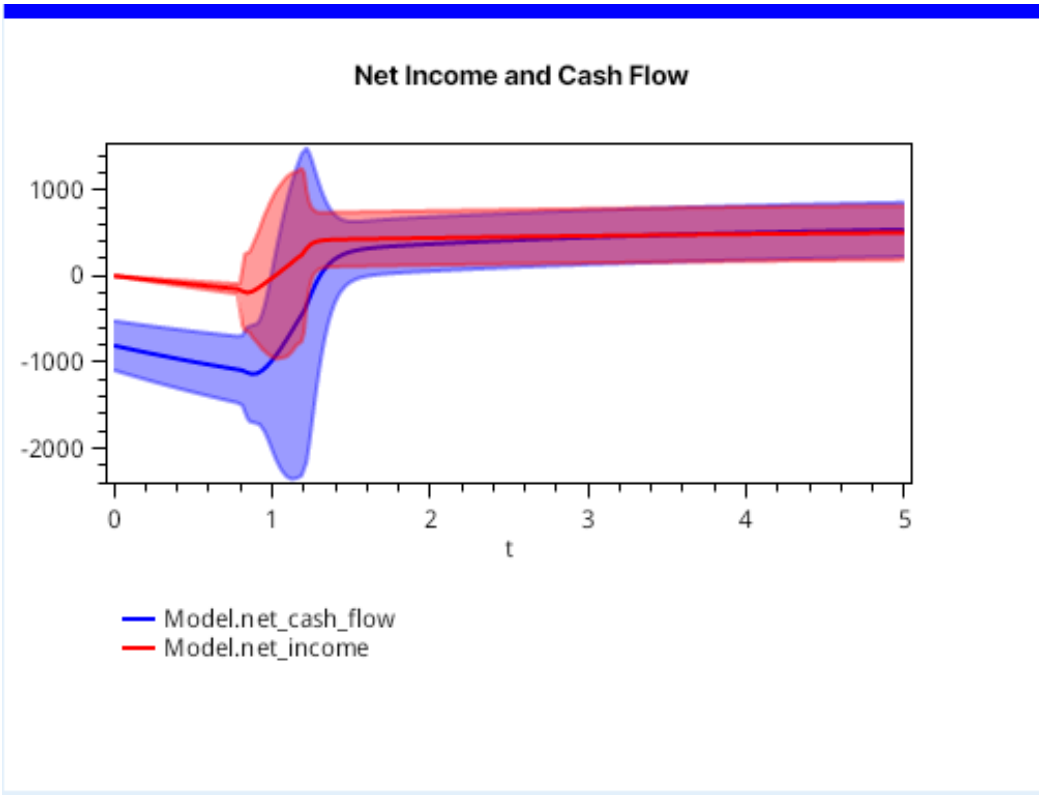

Figure 1.4: The charts for Sensitivity Analysis.

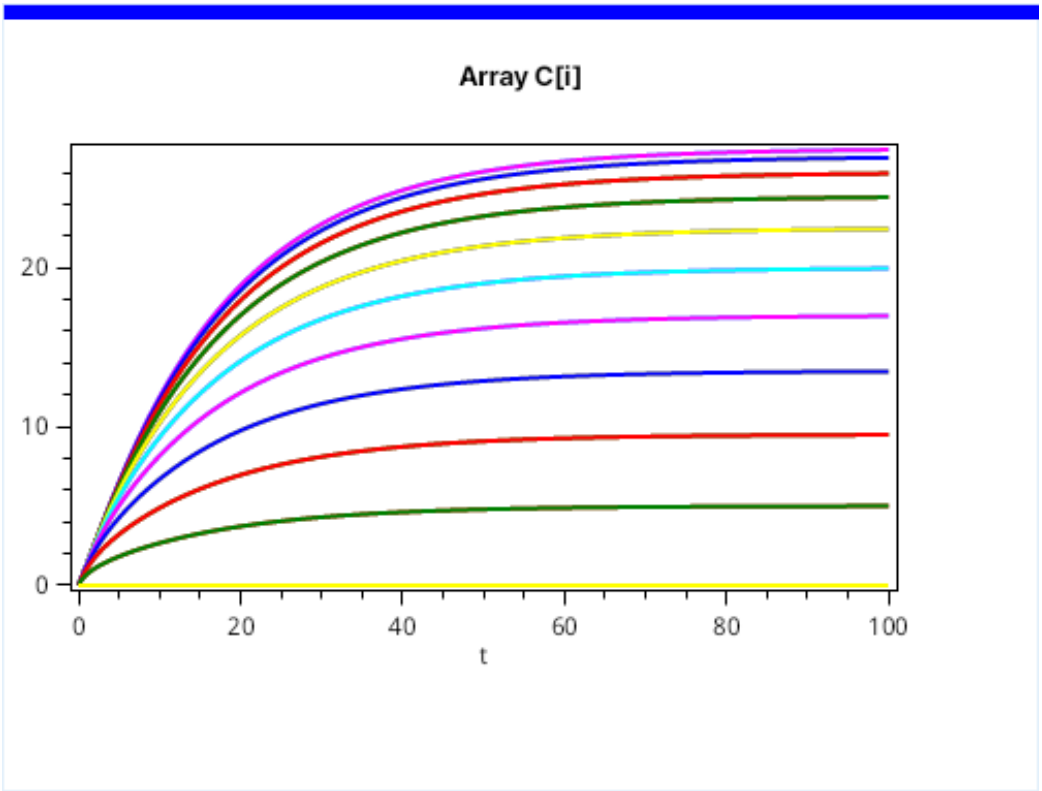

Figure 1.5: Plotting arrays on the chart.

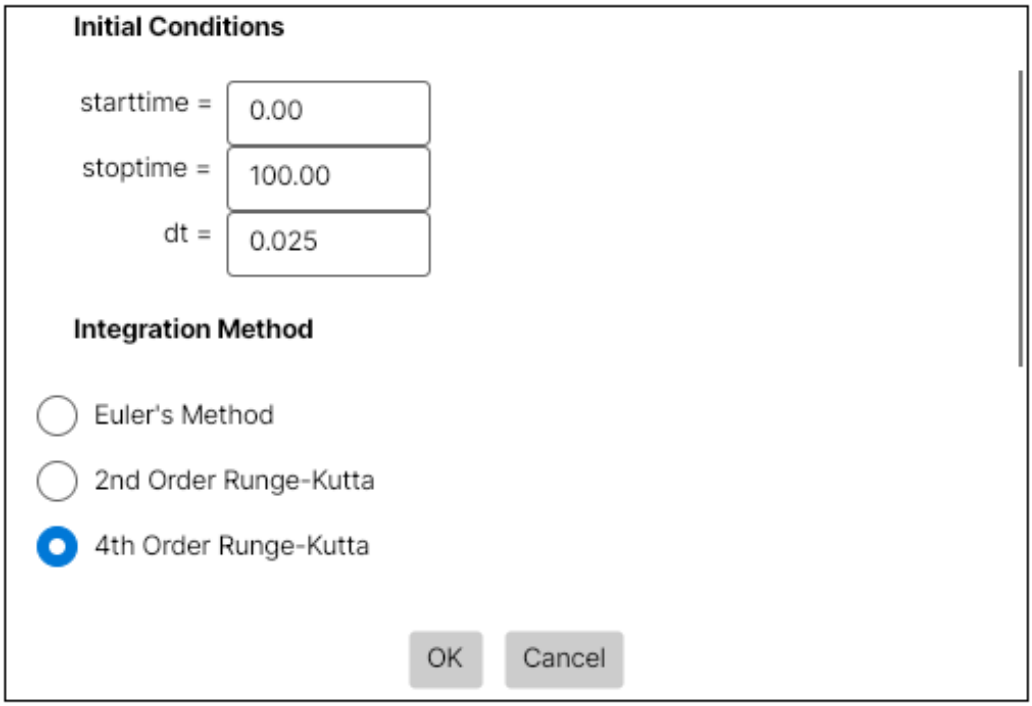

Figure 1.6: Defining the simulation specs.

Chapter 2

How to Create Simple Model

For example, we can reproduce a model from the 5-Minute Tutorial of Berkeley-Madonna[1] with the following equations.

$$\begin{aligned}
\dot{a} &= -ka \times a, \qquad a(t_0) = 100, \\
\dot{b} &= ka \times a - kb \times b, \qquad b(t_0) = 0, \\
\dot{c} &= kb \times b, \qquad c(t_0) = 0, \\
ka &= 1, \\
kb &= 1.
\end{aligned}$$

There are two ways how to define these equations. The first one is based on direct using integrals.

Create a new empty model and then define the following SFM Auxiliaries connected with help of SFM Links as shown in figure 2.1. To add new items, you can use the toolbar of the diagram panel.

Then click on the *Equations* tab. By clicking on the variable equations, complete the definition of the corresponding equations in the *Equation Editor*. For the first time, it is required to click on some equation to see the editor. Then the editor remains active and it is enough to click on each variable equation. In the end, you should define the following equations as shown in figure 2.2.

Now it is time to add the chart. Return to the diagram panel by clicking on the *Diagram* tab. Select the *Result Chart* element from the diagram toolbar and create a chart item on the same diagram. Here you should receive something similar to that one, which is displayed in figure 2.3. The

selected fragment on the displayed diagram is a part of the future chart element we will define.

Then click on the chart element to open the *Graph Editor*. In the editor you can add the following variables to the chart: A, B and C. It should look like figure 2.4. These variables are integrals.

Now we are ready to launch the simulation. There is the *Start Simulation* button on the main toolbar, which is displayed as arrow. It is shown in red in figure 2.5. Push the button to start running the simulation!

You should see the results similar to figure 2.6.

Probably, your chart is not smooth enough as it was illustrated in the figure. To fix this and improve the accuracy of plots, open the *Simulation / Simulation Specs* menu item and decrease the `dt` parameter value. Repeat the simulation run. Now you should see a more smooth and precise chart as it was shown in the figure earlier.

There is also another method of defining the ordinary differential equations, which is based on using the SFM Stocks and Flows.

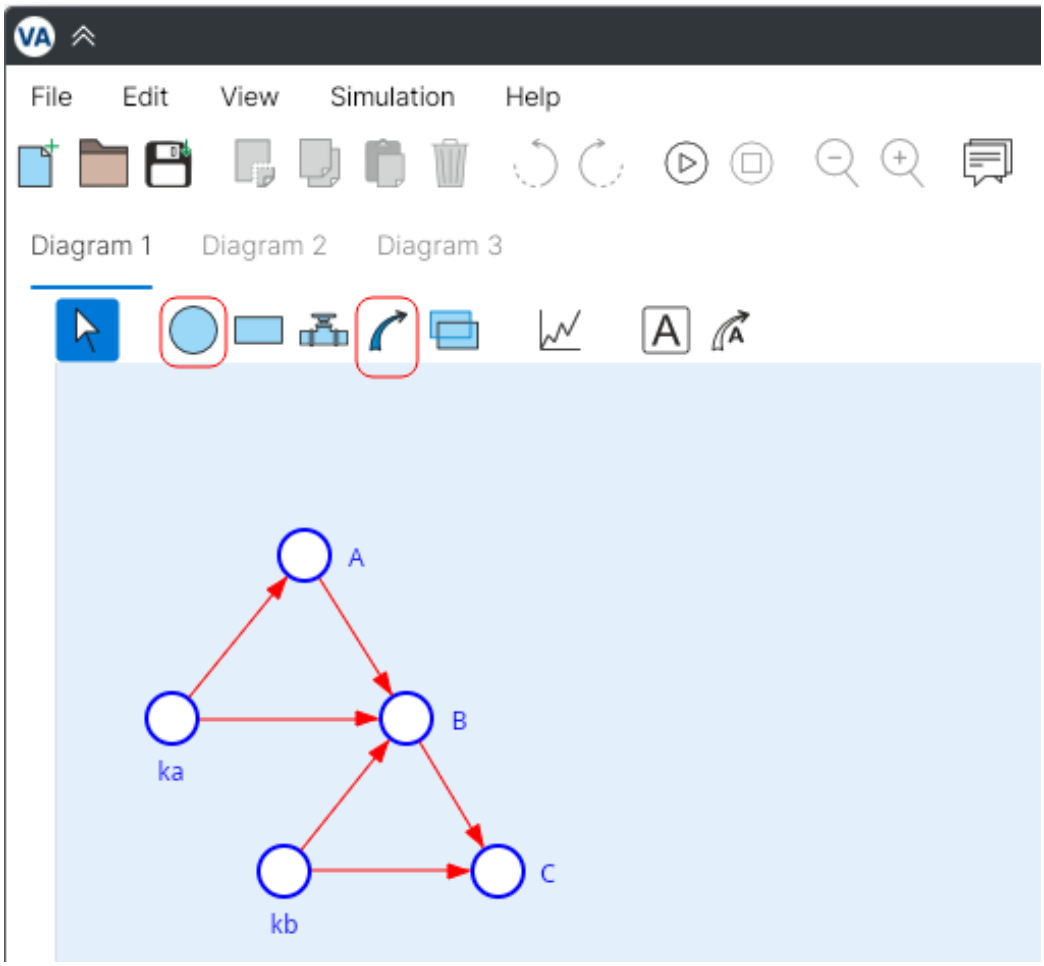

Figure 2.1: The diagram consisting of future integrals.

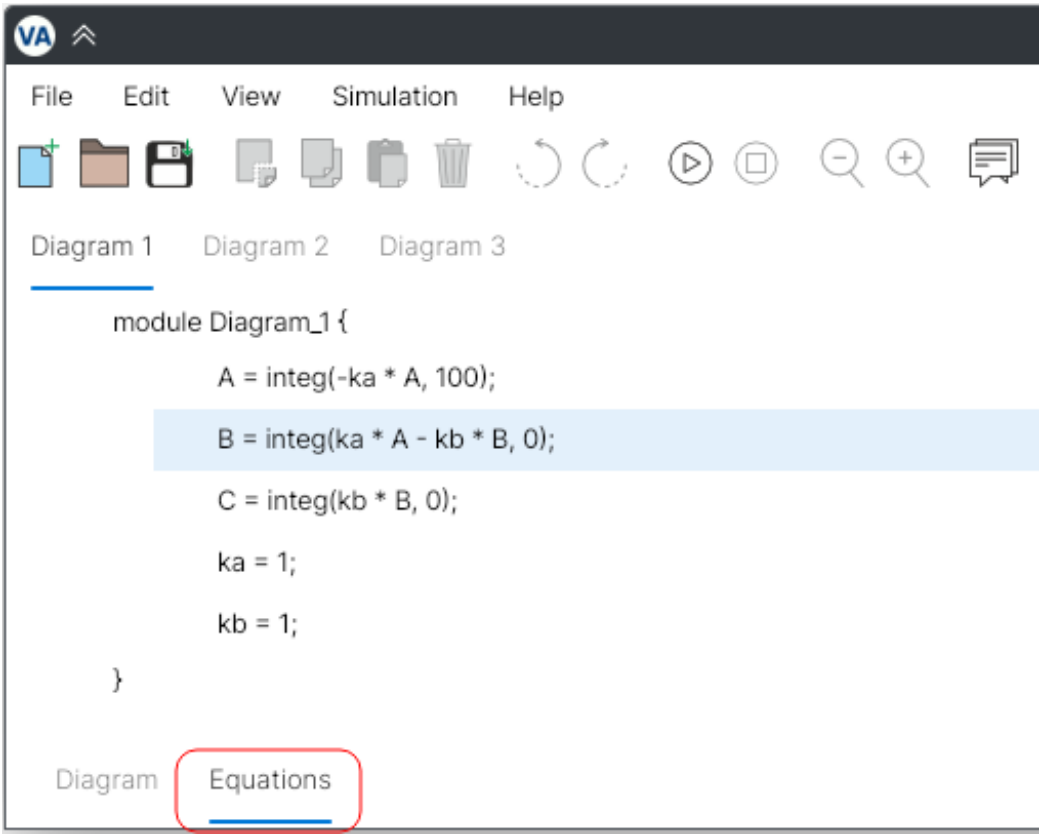

Figure 2.2: The equations consisting of integrals.

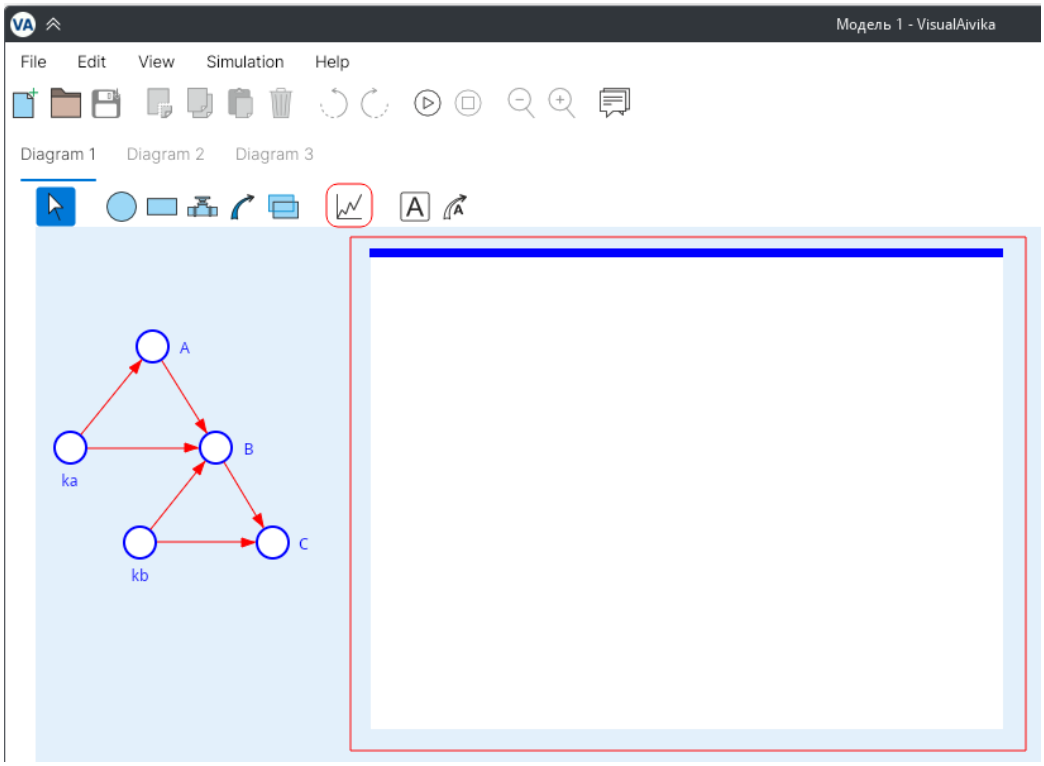

Figure 2.3: After the chart is added to the diagram.

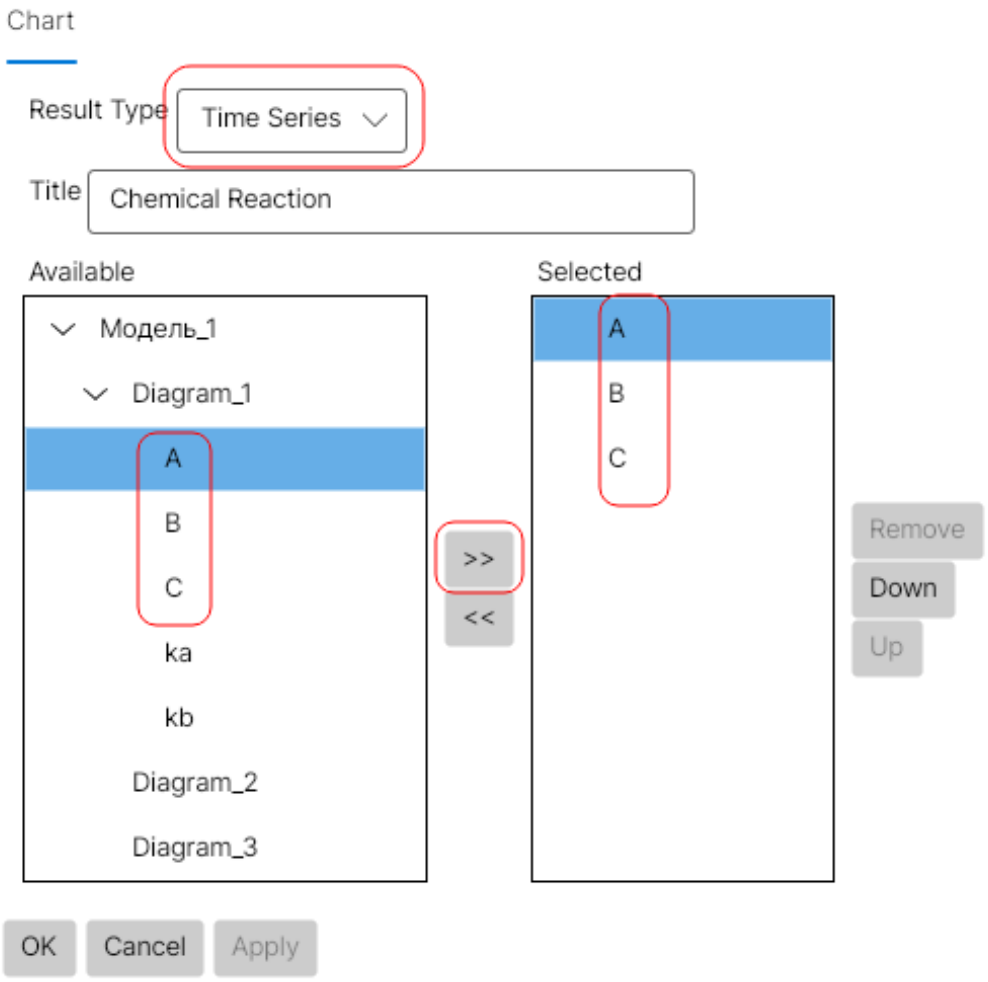

Figure 2.4: Add the variables of integrals to the chart for plotting.

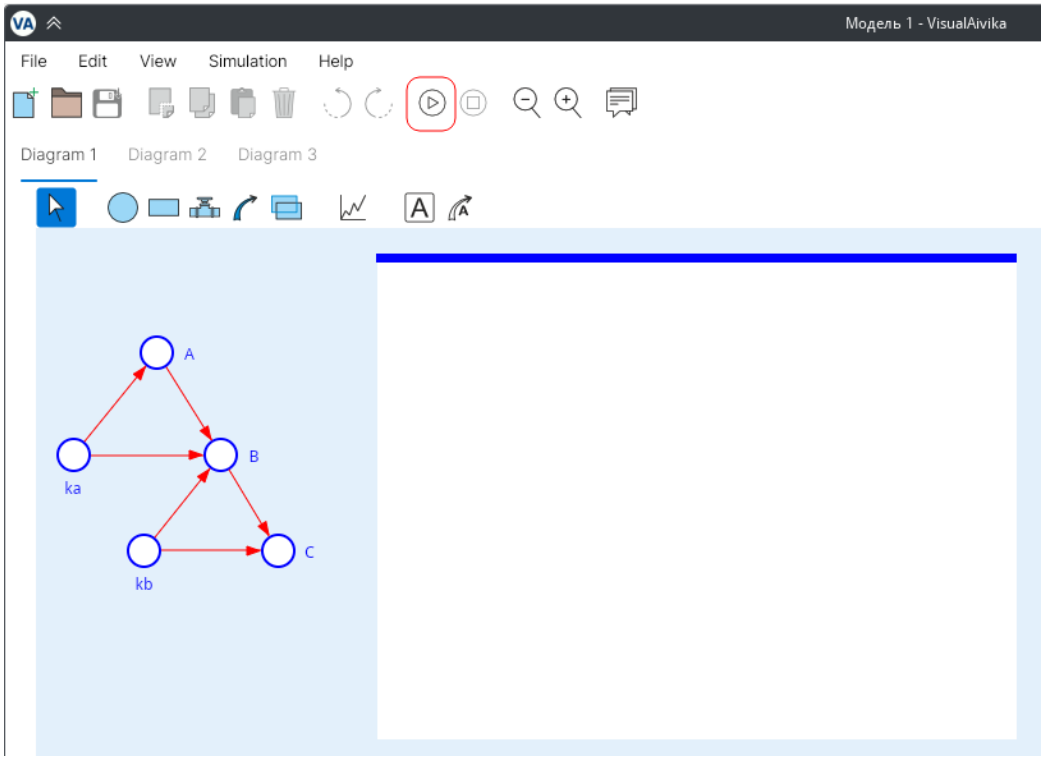

Figure 2.5: The *Start Simulation* button.

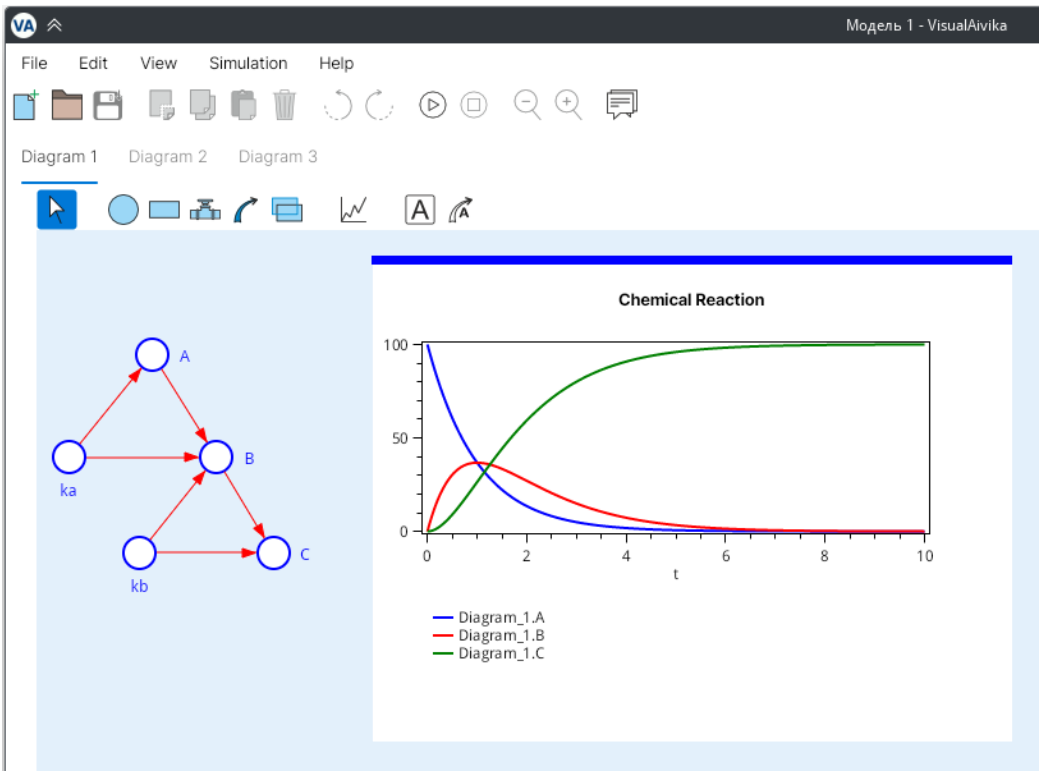

Figure 2.6: How the diagram looks like after the simulation has been finished.

Chapter 3

How to Create SFM Model

The SFM stock can be a reservoir. Then it becomes equivalent to the integral, but SFM flows become equivalent to derivatives. The direction of the SFM flow then shows the sign of the derivative, which can be positive or negative. The combination of all SFM flows connected to the specified SFM stock defines the corresponding sum of derivatives with possibly different signs.

The unidirectional SFM flow is always non-negative per se, while the bidirectional one may have any value. But the actual impact on the derivative for each connected integral depends on the combination of all factors: whether the SFM flow is bidirectional or unidirectional, whether it is inflow or outflow.

Fortunately, the *Equations* tab displays the corresponding equations in a mathematically oriented way, where it is easy enough to see the sign of each derivative and its actual impact.

To demonstrate the approach, we will use the simple model of exponential growth.

Create a new model and add the following SFM elements to the diagram by using the diagram component toolbar as shown in figure 3.1.

Click on the Cash stock to open the *Equation Editor*. Define the initial value equal to 1000 as illustrated in figure 3.2. This value will be the initial value of the integral that corresponds to the Cash stock.

Then select the Net Interest flow and change the *Flow Direction* property in the equation editor so that the property value would be *uniflow*. You should see something similar to figure 3.3.

It means that the corresponding SFM flow can have a non-negative

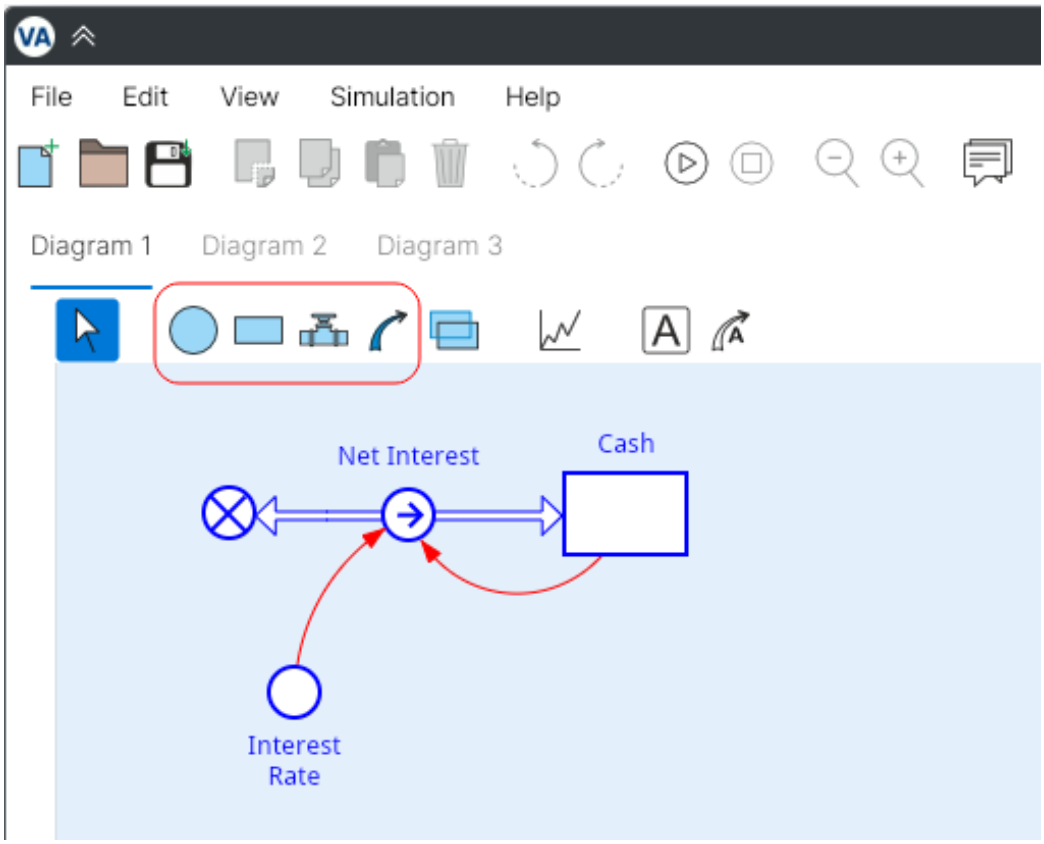

Figure 3.1: The initial SFM diagram for the exponential growth model.

value only, although the actual impact on the SFM stock may differ by the sign, which depends on the direction of the SFM flow too. Here the Net Interest entity is an inflow for the Cash stock. Hence the corresponding derivative has always a non-negative sign. The stock value must grow as we will see later.

After we made the Net Interest flow unidirectional, our diagram must slightly change. Please note to the fact that the corresponding flow element has one arrow now, while it had two arrows before, in the start and in the end. Now it has one arrow only in the end, where it connects to the Cash stock.

Now define the rest of equations so that they would look the same as shown in figure 3.4. Note that you should not enter the maximum function call for the Net Interest entity. It will be added automatically, for the flow is

defined as unidirectional. What you should enter for the flow is expression `Cash * Interest_Rate`.

The same equations could be defined directly with help of integrals, but here we use the SFM elements. The SFM Stock corresponds to the integral. The SFM Flow is related to the derivative.

If we add the chart element for the Cash stock according to the approach described in the previous chapter 2, then we will see the next chart as demonstrated in figure 3.5. Here the stop time was increased up to 36 in the *Simulation Specs* editor.

The point is that VisualAivika provides a comprehensive support of equations, with help of which we can build and run complex enough systems of ordinary differential equations. Moreover, VisualAivika has means for providing the Sensitivity Analysis with help of such equations. At the same time, VisualAivika allows you to create discrete event simulation models too, which can be combined with the ordinary differential equations.

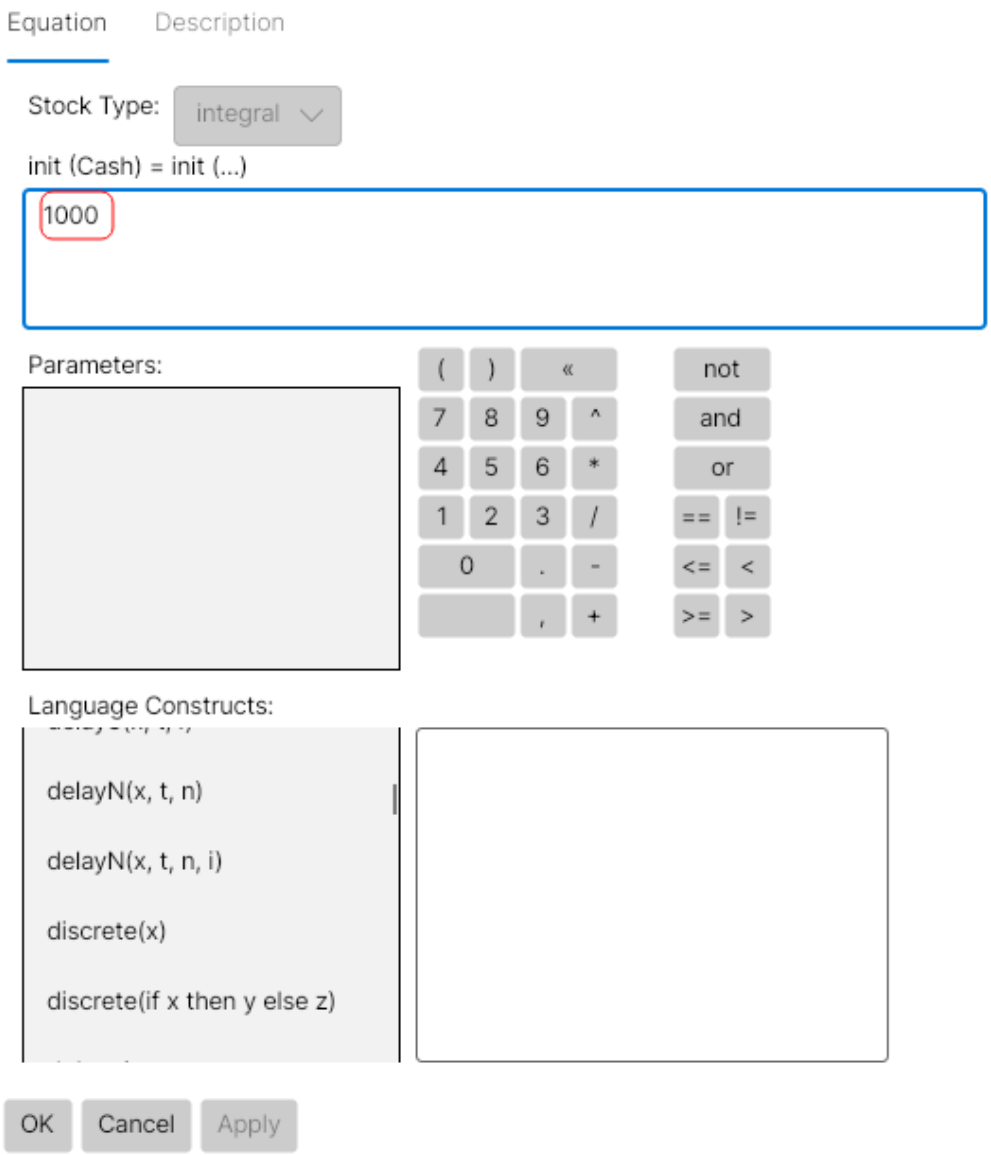

Figure 3.2: The SFM stock initial value.

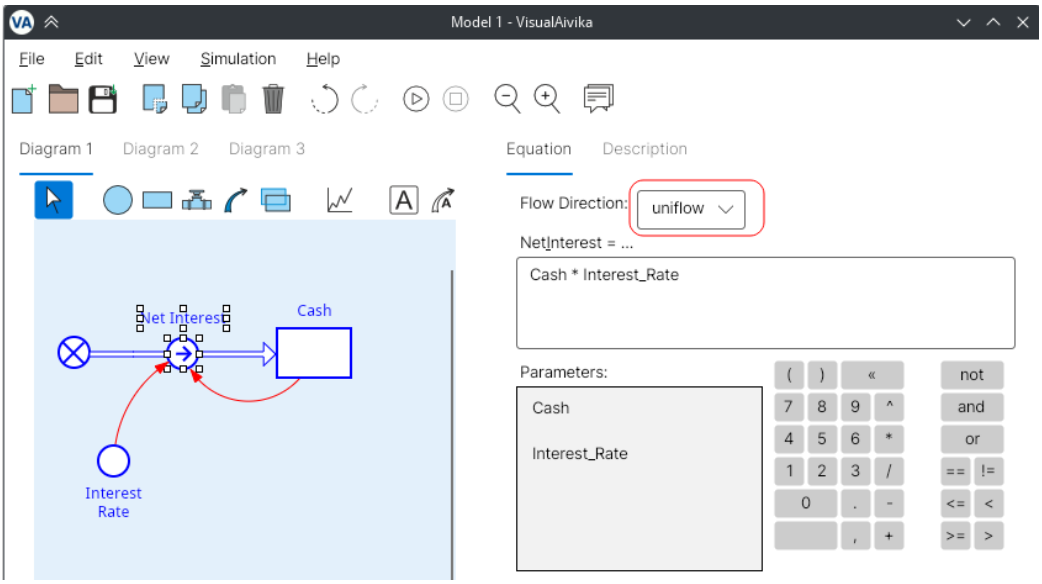

Figure 3.3: Making the SFM flow unidirectional.

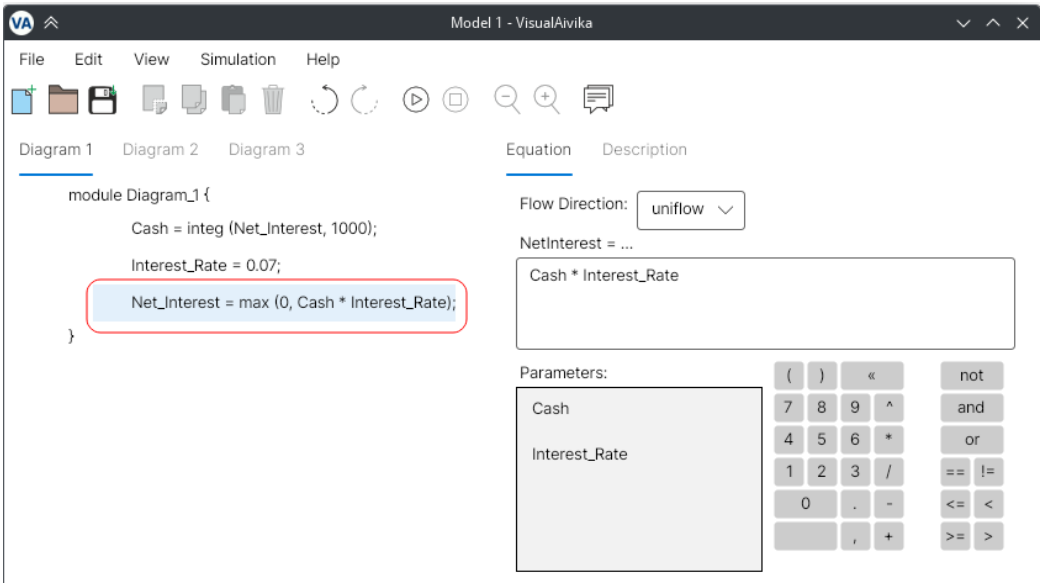

Figure 3.4: The exponential growth model equations.

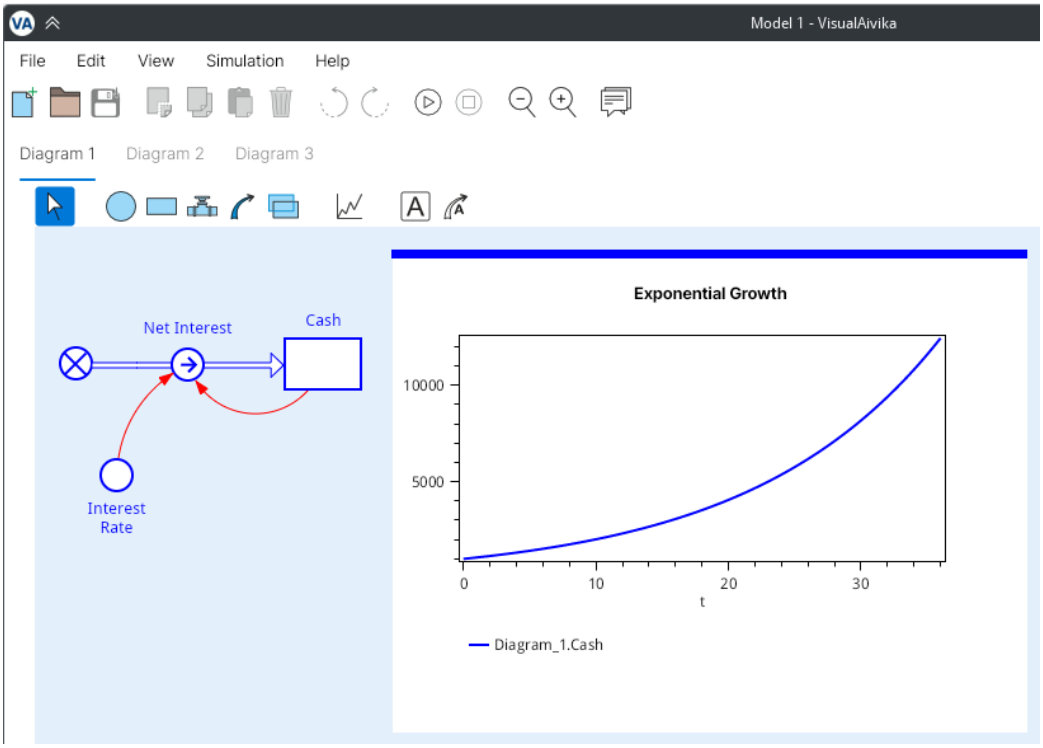

Figure 3.5: The exponential growth chart.

Chapter 4

How to Create Queueing System Model

To model the queueing systems, VisualAivika uses an approach similar to the GPSS modeling language[2], but the implementation is quite different, which has roots in functional programming. Like GPSS, VisualAivika allows using blocks, but here blocks are already *computations* that can be assigned to variables like integrals or numbers.

Actually at the time of writing this text, there was no yet special visual support in VisualAivika for blocks, but we can use the same SFM Auxiliaries that we used earlier in the previous chapters. We just assign the block computations to the SFM Auxiliary variables. As a consequence, the diagram consists of items that are connected in the direction opposite to the actual transact processing.

Figure 4.1 shows blocks, which are similar to the corresponding code in the GPSS language.

Code in GPSS

```
GENERATE 18,6
QUEUE JOEQ
SEIZE JOE
DEPART JOEQ
ADVANCE 16,4
RELEASE JOE
TERMINATE

GENERATE 480
```

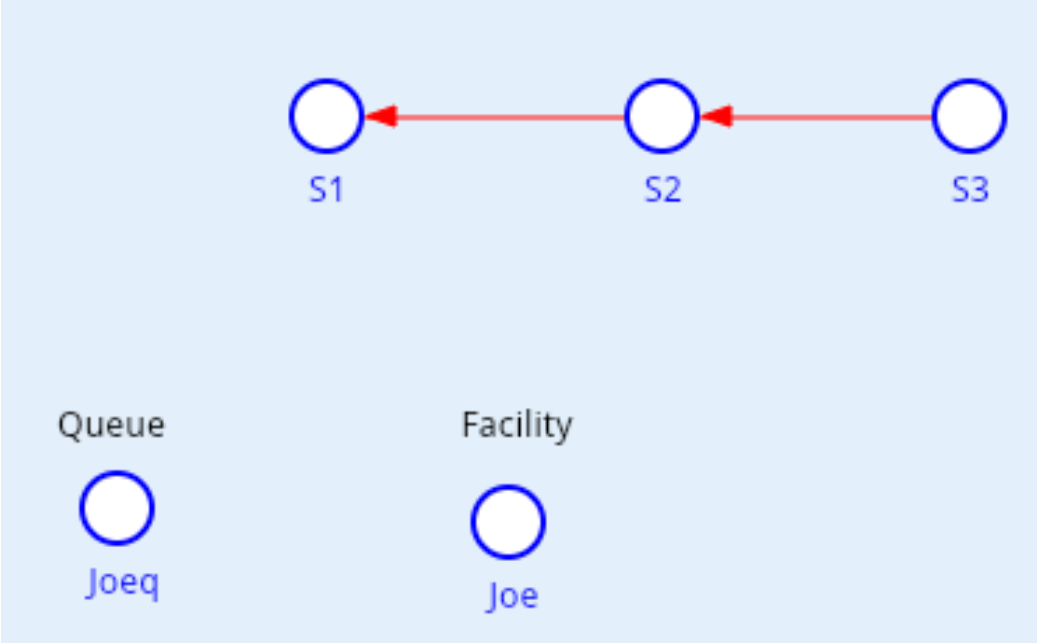

Figure 4.1: Block computations as SFM Auxiliaries.

```
TERMINATE 1

START 1
```

The corresponding equations in VisualAivika are provided in figure 4.2. Here we note that VisualAivika checks the diagrams and equations for consistency. The warnings in orange colour can be ignored, for we do not want to complicate the diagram.

The blocks are *composable* in VisualAivika. They can be connected to each other with help of the >>> operator to create new blocks and block chains. The block passes the transacts further, while the block chain either terminates the processing, or creates an infinite loop.

The provided equations are not complete yet. Now we should create a stream of random arrivals and then launch the entire simulation. For that, we add the stream and a special runner element to the diagram as shown in figure 4.3.

The complete equations are provided in figure 4.4. Here we note that there is a special `do!` operator to launch the transact processing by the

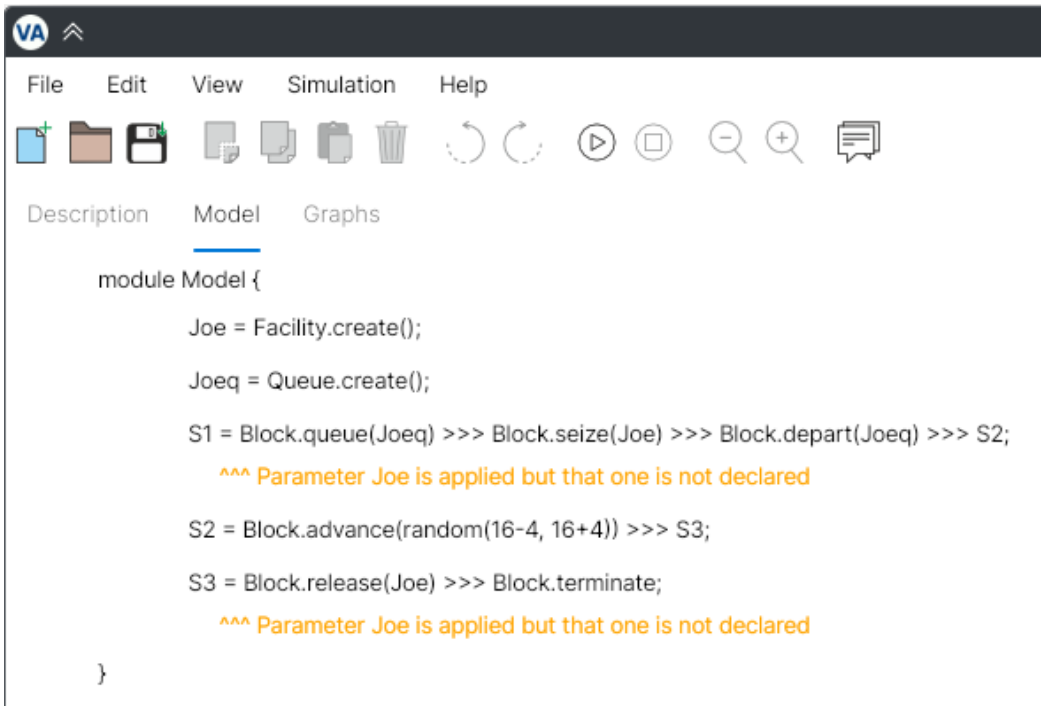

Figure 4.2: Equations for the block computations.

specified stream of random arrivals and block chain. The act of running is always explicit in the model.

Now we can define the stop time as 480 and make the integration time step small enough and equal to 2.5, for example.

Actually, the integration time step is not used in the queueing system model, but it affects to that how the charts are plotted and how the CSV tables are created. The chart plotting always happens in the integration time points, regardless of that whether the integration method is used or not. Therefore the `dt` parameter has to have some reasonable value.

Finally, it is time to add some output for the results of simulation. You can use chapter 6 as a reference material that describes different methods of displaying the results. However, there is one yet SFM element that can be useful now and that can decrease the cluttering on the visual diagrams.

Add a new diagram to the model and then add a new SFM Reference to that diagram as shown in figure 4.5. Let this reference be connected to the `Joe` variable from the main equations of diagram `Model`. Also add new

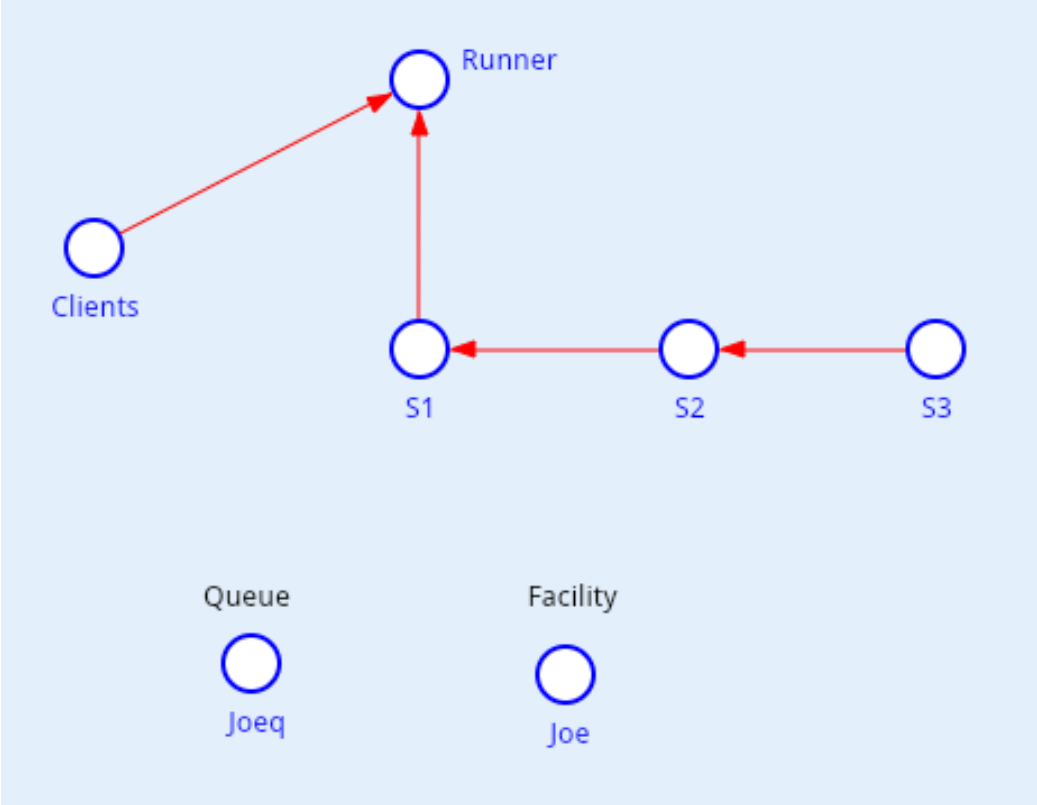

Figure 4.3: The complete model items.

items that are connected to the SFM Reference item. They will return the facility properties we want to display on the charts.

The corresponding equations for the properties are provided in figure 4.6. The functions applied are described in section 5.16.

For example, if we plot the Deviation Chart for the queue length property and its statistical sibling for 10000 simulation runs then we can get the followings results as shown in figure 4.7.

Note that the both deviation charts converge, for the random process becomes stationary fast enough[1]. Also we do not use the fact that the property is non-negative, but it has a wide enough spread because of high deviation.

In this model we did not use ordinary differential equations, but actu-

[1]The statistics reset can be added to one of the future versions of VisualAivika.

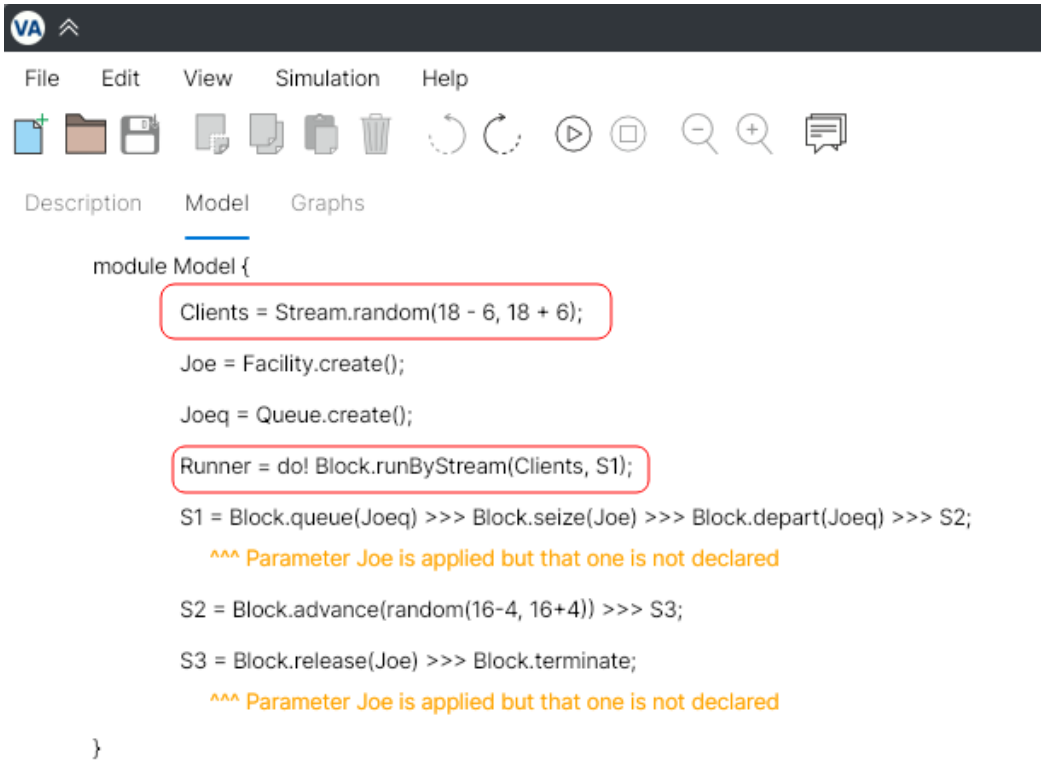

Figure 4.4: The complete model equations.

ally we could. VisualAivika does allow us to use the ordinary differential equations and discrete event simulation in one combined model.

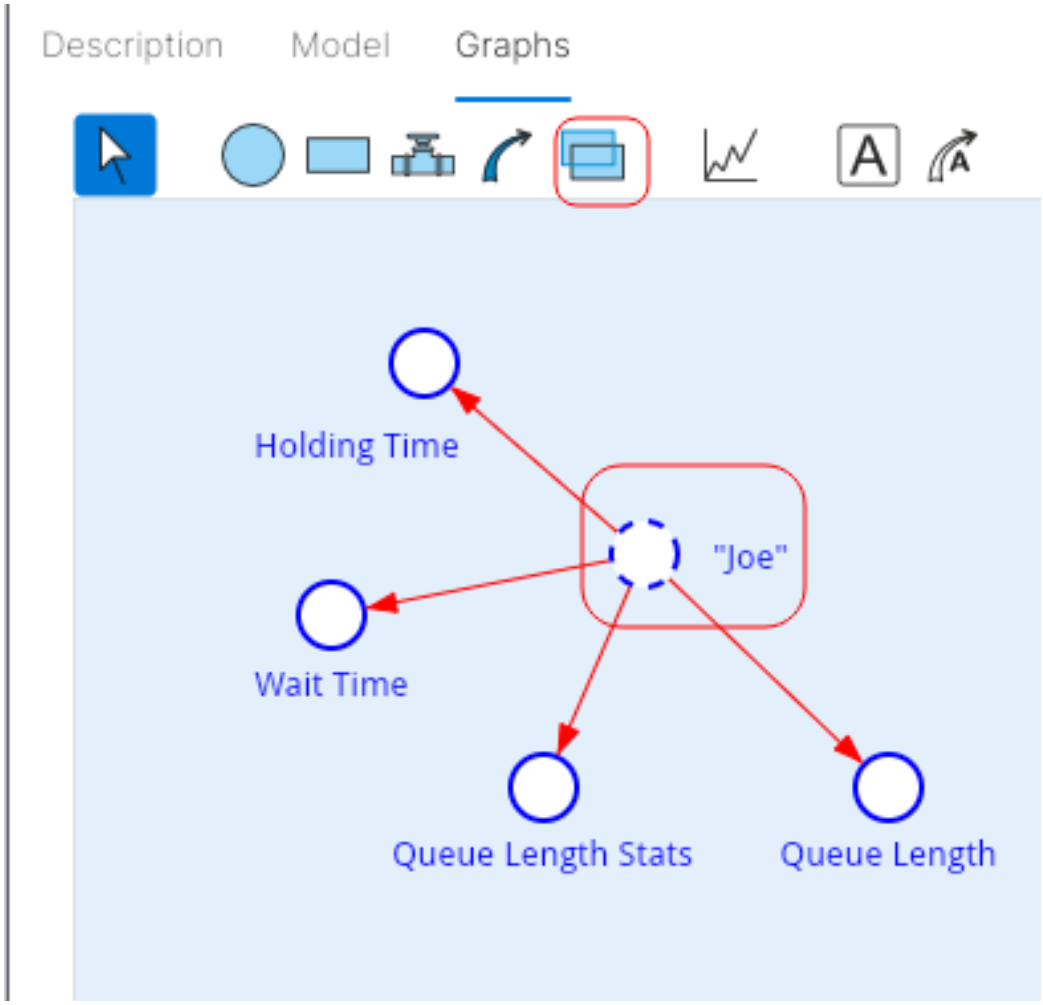

Figure 4.5: Use SFM Reference to decrease the cluttering on the diagrams.

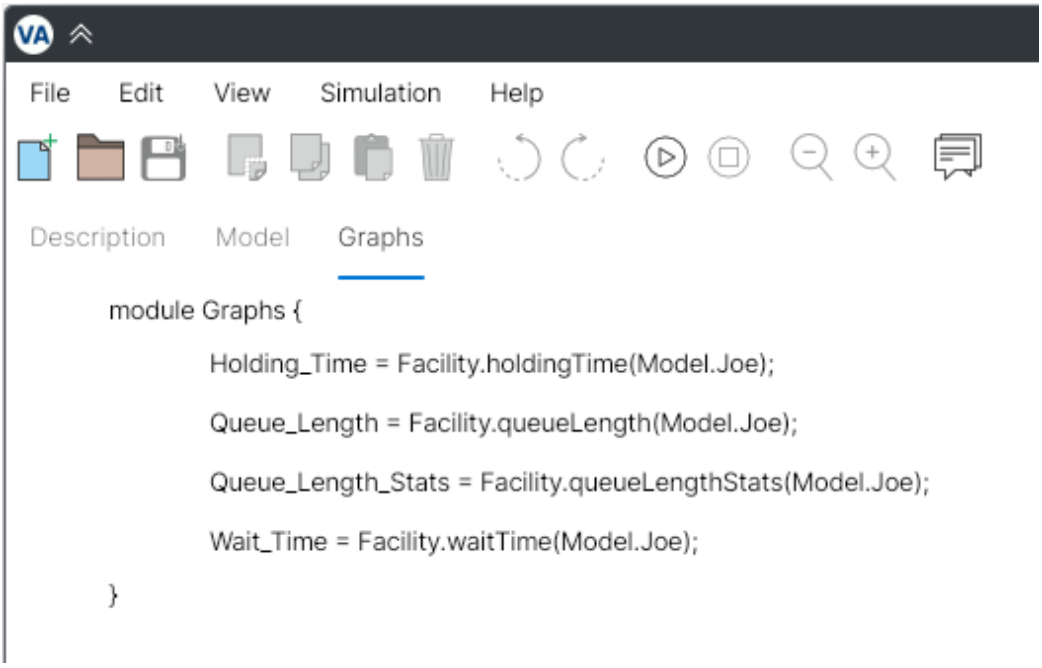

Figure 4.6: Extract the facility properties.

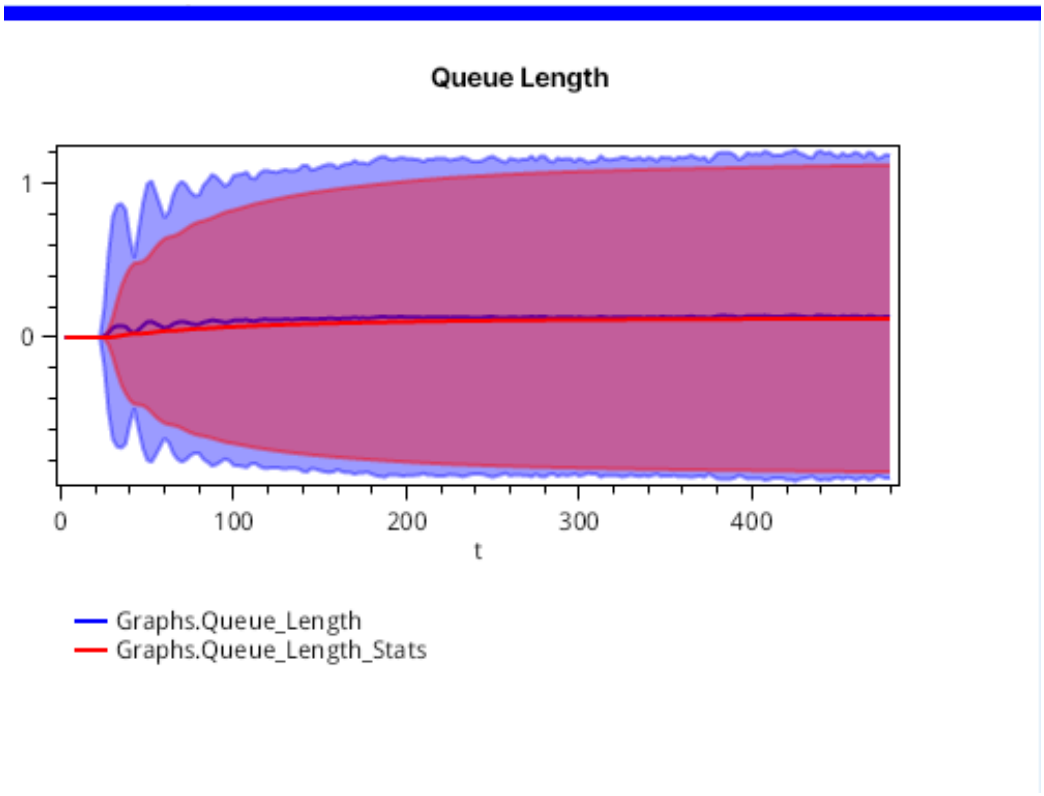

Figure 4.7: The statistical results of simulation.

Chapter 5

Modeling Language

The VisualAivika modeling language allows you to specify dynamic systems. The model can consist of stocks and auxiliaries. The stock variable can be an integral (reservoir). The auxiliary variable is a function of other variables and computations.

Currently, discrete event simulation entities are supported through the auxiliary variables only.

5.1 Simulation Specs

There are predefined variables that exist in each simulation:

- `starttime` defines the initial time;
- `stoptime` defines the final time;
- `dt` defines the integration time step;
- `time` returns the current integration time.

The first three constants like other simulation specs described in this section can be defined in the dialog window, which is opened by menu *Simulation / Simulation Specs*.

If your model contains only discrete event simulation entities, then the `dt` parameter can be made arbitrary. Nevertheless, its value reflects on that how charts are displayed, because values on the chart are sampled in the `dt` increments.

VisualAivika supports the following integration methods:

- Euler's method;
- 2nd order Runge-Kutta;
- 4th order Runge-Kutta.

Also it supports the following modes of generating random numbers:

- a simple mode turns on a standard .NET random number generator, which is weak but fast;
- a strong mode uses a cryptographic random number generator, which returns more random numbers but it is rather slow;
- a mode with the specified initial seed, which allows reproducing the same pseudo-random numbers for each run.

In addition, you can specify the number of simulation runs for the Monte-Carlo simulation, which is reflected by the following predefined variables:

- `runIndex` returns the current run index starting from 0;
- `runCount` returns the total number of simulation runs within the experiment.

These two built-in variables are useful for planning the simulation experiment, when providing the Sensitivity Analysis as shown in section 5.9 below.

5.2 Variables

The following list describes the variable types in VisualAivika.

- *Auxiliary*. Any variable that is defined as a function of other variables, or a constant.
- *Stock*. The dynamic variable in the model. At present, it can be a reservoir (integral) only.

- *Flow*. The variable that manages a stock. It can be either an inflow or outflow, bidirectional or unidirectional.
- *Table*. A table function which parameters are the x- and y- coordinates.
- *Range*. An index range for arrays.
- *Block*. A block computation that processes transacts and then returns transacts, either the same or modified ones.
- *Block Chain*. A block computation that processes transacts and then, either finishes the processing or has an infinite loop.
- *Generator Block*. A generator block computation that can start processing the transacts by the specified block chain.
- *Stream*. A stream of arrival events that come in the modeling system outside.
- *Action*. Some action such as launching the processing of transacts by the block chain.
- *Queue*. The queue entity.
- *Facility*. The facility is such a resource that may have only one owner.
- *Storage*. The storage is such a resource that can be borrowed by multiple transacts.
- *MatchChain*. The match chain can delay the transacts from the same assembly set.
- *Array*. An array of other variables.

The variable name in VisualAivika is case sensitive. The first symbol must be a letter or underscore. The next symbols can contain letters, digits and underscore in any sequence.

Example

```
Total_Resources, Scope, x, y
```

5.3 Equations

The VisualAivika modeling language allows you to define the model by writing a set of mathematical equations and expressions. The equations can be written in any order. The order of computation is determined based on dependencies between the variables.

VisualAivika uses standard algebraic expressions with the same rules of precedence as those used by Java, C/C++ and other common languages.

Also VisualAivika supports standard mathematical functions and has additional functions that make writing equations faster and simpler.

Example

```
y = integ (1 + 0.2 * y * sin (t) - 1.5 * t ^ 2, 0);
z = integ ((y - z)^2, 0);
```

Here the `integ` function returns an integral by the specified derivative and initial value.

Example

```
Adequacy_of_Control_Resources = Control_Resources / Desired_Control_Resources;
```

The *Equation Editor* automatically ends each equation by the semicolon symbol.

5.4 Operators

VisualAivika supports standard binary and unary operators that observe conventional precedence. The operators are shown in the tables below.

Table 5.1: Unary Operators

`not`	Logical inversion
`+ -`	Plus and minus

Table 5.2: Binary Operators

`^`	Arithmetic power
`* / + -`	Arithmetic operators
`< <= > >=`	Comparison
`== !=`	Equality and inequality
`and or`	Conjunction and disjunction
`>>>`	Block composition

The block composition operator requires some clarification.

Expression `b1 >>> b2` returns a block composition that processes transacts by the first block `b1` and then by the second block or block chain `b2`. If the latter argument `b2` is block, then the result is a block as well. Otherwise, if the latter argument `b2` is block chain, then the result is a block chain too.

5.5 Constants

VisualAivika defines the following constants.

Table 5.3: Built-in Constants

`true`	Logical true
`false`	Logical false
`pi`	The value of π
`infinity`	Represents a positive infinity
`nan`	Represents a value that is not a number

5.6 Functions

Here is a summary table of the basic predefined functions.

Table 5.4: Basic Functions

`abs (x)`	Returns the absolute value of `x`
`sqrt (x)`	Returns the square root of `x`
`int (x)`	Returns the largest whole number less than or equal to `x`

`round (x)`	Returns the number nearest to `x`
`power (x, y)`	Returns the same as x^y
`mod (x, y)`	Returns the remainder of `x/y`
`min (x1, x2, ...)`	Returns the minimum value of `x1`, `x2`, ...
`max (x1, x2, ...)`	Returns the maximum value of `x1`, `x2`, ...
`sum (x1, x2, ...)`	Returns the sum of `x1`, `x2`, ...
`prod (x1, x2, ...)`	Returns the product of `x1`, `x2`, ...
`mean (x1, x2, ...)`	Returns the average value of `x1`, `x2`, ...
`sin (x)`	Returns the sine of `x`
`cos (x)`	Returns the cosine of `x`
`tan (x)`	Returns the tangent of `x`
`arcsin (x)`	Returns the inverse sine of `x`
`arccos (x)`	Returns the inverse cosine of `x`
`arctan (x)`	Returns the inverse tangent of `x`
`arctan (y, x)`	Returns the inverse tangent of `(y/x)`
`sinh (x)`	Returns the hyperbolic sine of `x`
`cosh (x)`	Returns the hyperbolic cosine of `x`
`tanh (x)`	Returns the hyperbolic tangent of `x`
`sinwave (a, t)`	Returns the sine wave of amplitude `a` and period `t`
`coswave (a, t)`	Returns the cosine wave of amplitude `a` and period `t`
`exp (x)`	Returns e raised to power `x`
`log (x)`	Returns the natural (base e) logarithm of `x`
`log (x, y)`	Returns the logarithm of `x` in base `y`

The `sinwave` and `coswave` functions require some note. They are defined as follows.

Definition

```
sinwave(a, t) = a * sin(2 * pi * time / t);
coswave(a, t) = a * cos(2 * pi * time / t);
```

Table 5.5: Random Number Generators

`random (a, b)`	Returns the uniform random number between `a` and `b`
`randomInt (a, b)`	Returns the integer uniform random number between `a` and `b`
`triangular (a, m, b)`	Returns the triangular random number between `a` and `b` with median `m`
`normal (m, n)`	Returns the normal random number with mean `m` and deviation `n`
`exponential (m)`	Returns the exponential random number with mean `m`
`erlang (b, m)`	Returns the Erlang random number with scale `b` and integer shape `m`
`poisson (m)`	Returns the Poisson random number with mean `m`
`binomial (p, n)`	Returns the binomial random number on `n` trials of probability `p`

Table 5.6: Conditional Expression

`if x then y else z`	Returns `y` if `x` is true, otherwise returns `z`

Table 5.7: Miscellaneous Functions

`step (h, t)`	Returns 0 until time `t` and then returns `h`
`pulse (t, d)`	Returns the pulse of height 1 starting at time `t` with duration `d`
`pulse (t, d, p)`	Returns the pulse of height 1 starting at time `t` with duration `d` and period `p`
`ramp (s, t1, t2)`	Returns 0 until time `t1` and then slopes until time `t2` and then holds `s`

These functions have the following definition. They use the `discrete` operator described further. In short, the operator returns the value, which doesn't change except of the integration `dt` intervals regardless of the integration method used.

Definition

```
step(h, t) = discrete(if time + dt/2 > t then h else 0);

pulse(t, d) = discrete(if (time + dt/2 > t) and (time + dt/2 < t + d)
                       then 1 else 0);

pulse(t, d, p) = pulse(t + (if (p > 0) and (time > t)
                            then round((time - t) / p)*p else 0), d);

ramp(s, t1, t2) = discrete(if (time + dt/2 > t1)
                           then (if (time - dt/2 < t2)
                                 then s*(time - t1)
                                 else s*(t2 - t1))
                           else 0);
```

Table 5.8: Interpolation

`table ((x1, y1), (x2, y2), ...)`	Creates a table consisting of points `(x1, y1)`, `(x2, y2)`, ..., where the table can be used as a function later
`table ((x1, y1), (x2, y2), ...) (x)`	Lookup `x` in the list of points `(x1, y1)`, `(x2, y2)`, ... using linear interpolation
`step (t)`	Creates a discrete step-wise interpolation function based on the specified table `t`
`step (table ((x1, y1), (x2, y2), ...)) (x)`	Lookup `x` in the list of points `(x1, y1)`, `(x2, y2)`, ... using the discrete step-wise interpolation

The table function interpolates the argument according the specified table.

Example

```
Effect_of_Scope_Stability_of_Rules =
  table ((0, 0.5), (0.2, 0.535), (0.4, 0.585), (0.6, 0.675),
         (0.8, 0.82), (1, 1), (1.2, 1.21), (1.4, 1.35),
         (1.6, 1.42), (1.8, 1.47), (2, 1.5))
        (Scope);
```

Table 5.9: Integral Functions

`integ (d, i)`	Returns the integral of rate `d` and initial value `i`
`delay (x, t)`	Returns a first order exponential delay of `x` for time `t` conserving `x`
`delay (x, t, i)`	Returns a first order exponential delay of `x` starting with `i` for time `t` conserving `x`
`delay3 (x, t)`	Returns a third order exponential delay of `x` for time `t` conserving `x`
`delay3 (x, t, i)`	Returns a third order exponential delay of `x` starting with `i` for time `t` conserving `x`
`delayN (x, t, n)`	Returns an `n`'th order exponential delay of `x` for time `t` conserving `x`
`delayN (x, t, n, i)`	Returns an `n`'th order exponential delay of `x` starting with `i` for time `t` conserving `x`
`smooth (x, t)`	Returns a first order exponential smooth of `x` over time `t`
`smooth (x, t, i)`	Returns a first order exponential smooth of `x` over time `t` starting at `i`
`smooth3 (x, t, i)`	Returns a third order exponential smooth of `x` over time `t`
`smooth3 (x, t, i)`	Returns a third order exponential smooth of `x` over time `t` starting at `i`
`smoothN (x, t, n)`	Returns an `n`'th order exponential smooth of `x` over time `t`

`smoothN (x, t, n, i)`	Returns an `n`'th order exponential smooth of `x` over time `t` starting at `i`
`forecast (x, t, h)`	Forecasts for `x` over the time horizon `h` using an average time `t`
`trend (x, t, i)`	Returns the fractional change rate of `x` using the average time `t` and starting with `i`

Here the `delay`-like functions have the following idea, where the `delayN` functions are a generalization of this rule. If you will compare these functions with other simulation software tools, then please note that the `delayN` functions have no effect of the `discrete` operator that may implicitly present in some other tools. In case of need, this operator can be added in the equations manually.

Definition

```
D1=delay(x, t) if and only if
  D1=1/t * integ(x - D1, x*t);

D1I=delay(x, t, i) if and only if
  D1I=1/t * integ(x - D1I, i*t);

D3_3=delay3(x, t) if and only if
  D3_3=1/(t/3) * integ(D3_2 - D3_3, x*t/3);
  D3_2=1/(t/3) * integ(D3_1 - D3_2, x*t/3);
  D3_1=1/(t/3) * integ(x - D3_1, x*t/3);

D3I_3=delay3(x, t, i) if and only if
  D3I_3=1/(t/3) * integ(D3I_2 - D3I_3, i*t/3);
  D3I_2=1/(t/3) * integ(D3I_1 - D3I_2, i*t/3);
  D3I_1=1/(t/3) * integ(x - D3I_1, i*t/3);
```

For example, figure 5.1 demonstrates the first-order and third-order delay functions for the following input and time period.

Example

```
x=sin(time);
t=40*dt;
dt=0.025;
```

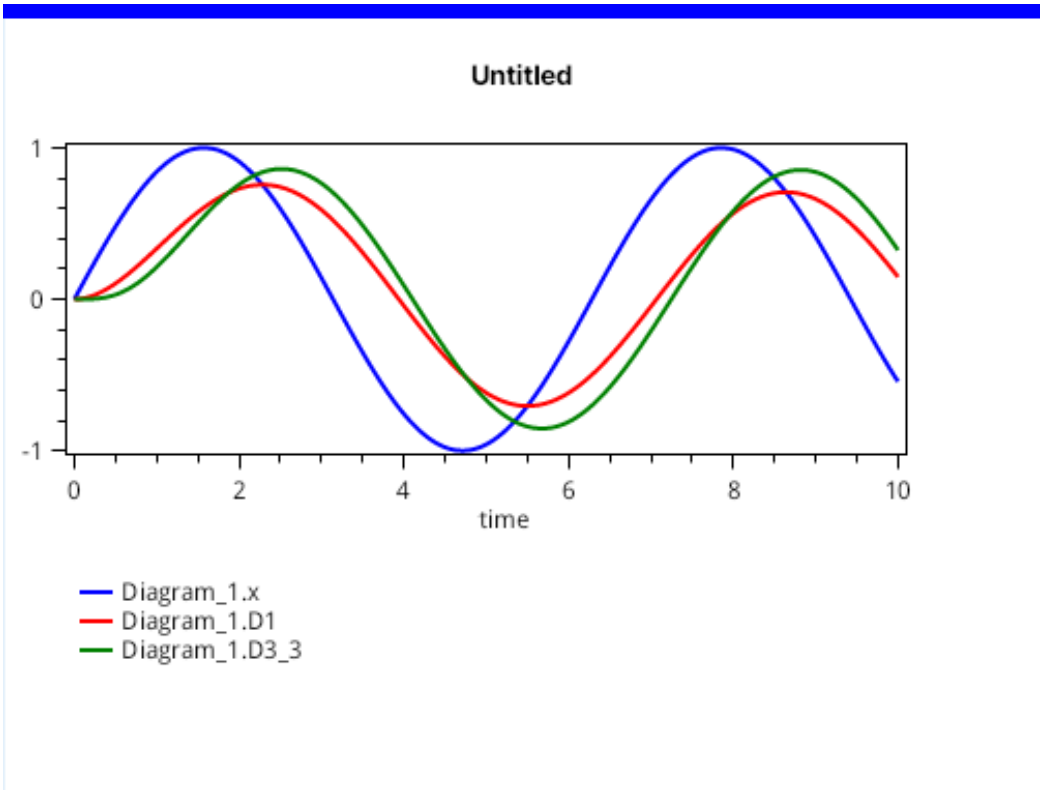

Figure 5.1: The first-order delay, `D1`, and third-order delay function, `D3_3`, for the specified input, `x`, and time period.

The `smooth`-like functions have the next idea, where the `smoothN` functions are a generalization of the following rule. If you will compare these functions with other simulation software tools, then please note that the `smoothN` functions have no effect of the `discrete` operator that may implicitly present in some other tools. In case of need, this operator can be added in the equations manually.

Definition

```
S1=smooth(x, t) if and only if
  S1=integ((x - S1)/t, x);

S1I=smooth(x, t, i) if and only if
  S1I=integ((x - S1I)/t, i);
```

```
S3_3=smooth3(x, t) if and only if
  S3_3=integ((S3_2 - S3_3)/(t/3), x);
  S3_2=integ((S3_1 - S3_2)/(t/3), x);
  S3_1=integ((x - S3_1)/(t/3), x);

S3I_3=smooth3(x, t, i) if and only if
  S3I_3=integ((S3I_2 - S3I_3)/(t/3), i);
  S3I_2=integ((S3I_1 - S3I_2)/(t/3), i);
  S3I_1=integ((x - S3I_1)/(t/3), i);
```

The `delay` and `smooth` functions behave different when the time period is changing.

The `forecast` and `trend` functions have the following definition, where the `init` operator returns the initial value of the expression at start time of simulation.

Definition

```
forecast(x, t, h) = x * (1 + (x / smooth(x, t) - 1) / t * h);

trend(x, t, i) = (x / smooth(x, t, init(x) / (1 + i * init(t))) - 1) / t;
```

Table 5.10: Financial Functions

`npv (s, r, i, f)`	Returns the Net Present Value (NPV) of stream `s` computed using the specified discount rate `r`, the initial value `i` and some factor `f` (usually 1)
`npve (s, r, i, f)`	Returns the Net Present Value End of period (NPVE) of stream `s` computed using the specified discount rate `r`, the initial value `i` and some factor `f`

In VisualAivika the financial functions have the following definition.

Definition

```
npv=npv(s, r, i, f) if and only if

  npv = (accum + dt * s * df) * f;
```

```
  df = integ(- df * r, 1);
  accum = integ(s * df, i);

npve=npve(s, r, i, f) if and only if

  npve = (accum + dt * s * df) * f;
  df = integ(- df * r / beta, 1 / beta);
  accum = integ(s * df, i);
  beta = 1 + r * dt;
```

Table 5.11: Simulation Operators

`discrete (x)`	It returns a value of `x`, which doesn't change except of the integration `dt` intervals regardless of the integration method used
`init (x)`	It returns the initial value of `x`

The `init` operator returns the initial value for the integral. Also it returns the `Block.identity` block for any block as well as `Block.terminate` for any block chain. Finally, it returns the `Stream.empty` computation for any stream.

Unlike other simulation software tools, VisualAivika has slightly non-standard operators `discrete` and `init`. They are related to the very simulation engine of VisualAivika. Any numeric expression can have the initial value. Also we can treat any numeric expression as something that would be updated as if the Euler's method was applied. Usually, you do not need to apply these operators in your models. Moreover, these two last operators are specific to VisualAivika and they are not transferrable between different simulation software tools.

5.7 Ranges

A range is defined by two integer expressions: the low index of the range and the high index of the range. The expressions must be delimited by two dots.

Example

```
N = 10;

I_Range = 1..N;
J_Range = 1..5;
```

The ranges can be defined on the diagram like other variables and then be used in the equations. They are needed for arrays.

5.8 Arrays

To define an one-dimensional array, you can use a special syntax:

[*Element* | *Index* <- *Range*]

Here the element expression can depend on the index which values will be received from the specified range.

This syntax is generalized for other dimensions too. For example, a two-dimensional array can be defined by adding another index:

[*Element* | *Index1* <- *Range1* , *Index2* <- *Range2*]

VisualAivika supports up to 5 dimensions.

To refer to an element of the one-dimensional array by the specified index, you can use a subscript like this:

Array[*Index*]

Similarly, the two-dimensional array element can be referenced by two indices:

Array2D[*Index1* , *Index2*]

The rule is generalized for other dimensions too.

Example

```
n = 51;

C = [ if (i == 0) or (i == n + 1) then 0 else M[i] / v | i <- 0..n+1 ];
M = [ integ (q + k*(C[i-1] - C[i]) + k*(C[i + 1] - C[i]), 0) | i <- 1..n ];

q = 1;
k = 2;
v = 0.75;
```

In this example C and M are one-dimensional arrays that have different indices. The C array has values `C[0]`, . . . , `C[n+1]`, while array M has values `M[1]`, . . . , `M[n]`.

By the way, the ranges could be defined as separate variables. For simplicity, here the ranges are defined within the arrays. The both methods are acceptable.

5.8.1 Functions for Arrays and Ranges

The following functions are defined for the arrays and ranges.

Table 5.12: Functions for Arrays and Ranges

`length (a)`	Returns the total element count for the specified range or array `a`
`length (a, d)`	Returns the element count for the specified array `a` and dimension `d` starting from 0
`low (a)`	Returns the low index for the specified range or one-dimensional array `a`
`low (a, d)`	Returns the low index for the specified array `a` and dimension `d` starting from 0
`high (a)`	Returns the high index for the specified range or one-dimensional array `a`
`high (a, d)`	Returns the high index for the specified array `a` and dimension `d` starting from 0
`min (a)`	Returns the minimal element of the specified array `a`

`max (a)`	Returns the maximal element of the specified array `a`
`sum (a)`	Returns a sum of the elements for the specified array `a`
`prod (a)`	Returns a product of the elements for the specified array `a`
`mean (a)`	Returns an average value for the specified array `a`

The last functions are aggregating. It is useful that we can create intermediate arrays to pass in them to these functions.

The following example is an equivalent to [Vensim 5 Reference Manual, page 30, example 3], from the documentation of Vensim[3].

Example

```
efficiency = prod(factor_efficiency);
US_population = sum(population);
revenue = [ sum([ sales[c, p] * price[p, b] | p <- product ]) |
              c <- country, b <- brand ];
```

Please note how an intermediate array is created, when summing the revenue.

The next example is related to the Theory of Games. It calculates a maximin and minimax by the specified matrix of dimension $n \times n$, respectively.

Example

```
max_min = max([ min([ A[i,j] | j <- 1..n ]) | i <- 1..n ])
min_max = min([ max([ A[i,j] | j <- 1..n ]) | i <- 1..n ])
```

Here the temporary arrays are created only once at the very beginning of the simulation run.

The arrays can define the block computations as well as resources and queues. Then we can access to them by index.

5.8.2 Array Initialization

Some arrays can be defined in a table form without direct using ranges and indices. There is a simplified syntax for that.

Example

```
A1 = [0, 1, 2, 3, 4, 5];
A2 = [[0, 1], [2, 3], [4, 5]];
A3 = [[[0, 1], [2, 3]], [[4, 5], [6, 7]]];
```

Only such arrays will always have indices starting from zero.

5.8.3 Plotting Arrays on Charts

The arrays can be plotted on charts. For example, the revenue array defined in section 5.8.1 through aggregation looks like figure 5.2.

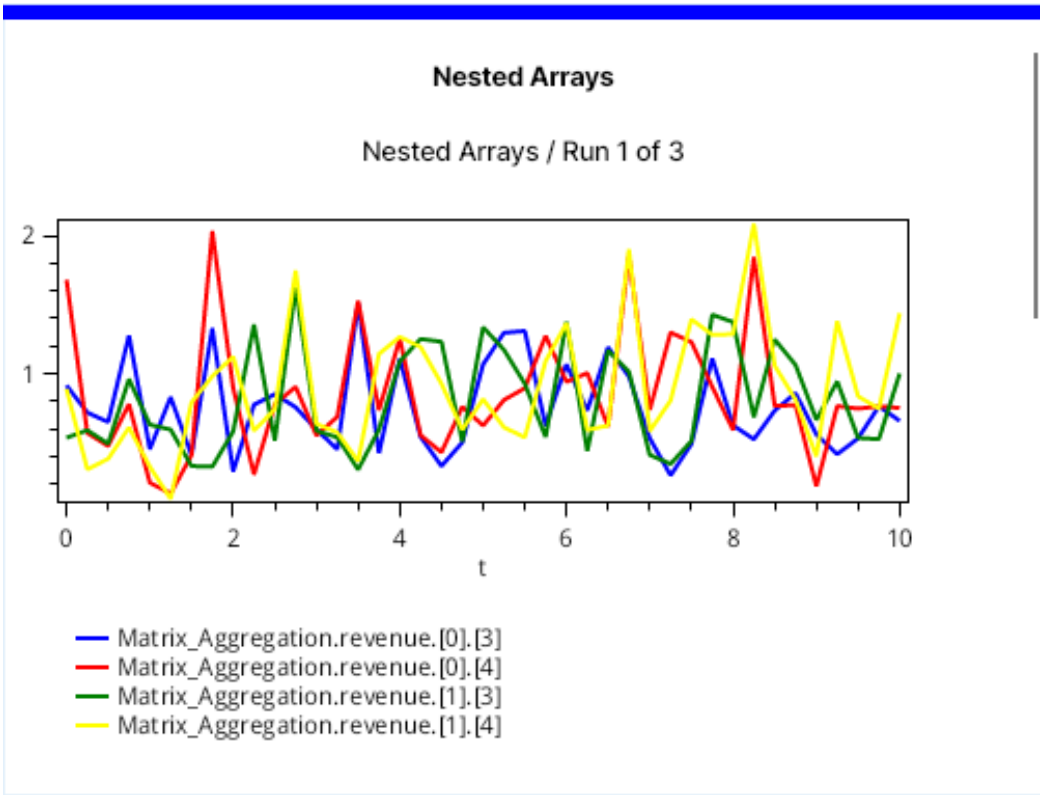

Figure 5.2: When plotting arrays on the chart, each element has its own subscript.

5.9 Sensitivity Analysis

As it was mentioned before, there are predefined variables `runIndex` and `runCount` that return the current run index, starting from zero, and the total run count for the Monte-Carlo simulation experiment, respectively.

It is important that the `runIndex` variable is constant within the simulation run, but then it is updated for another run. There are similar random parameters that have the same property. They are constant within the current run, but they are updated for another run.

Table 5.13: Random Parameters

`randomParam (a, b)`	Returns the uniform random parameter between `a` and `b`
`randomIntParam (a, b)`	Returns the integer uniform random parameter between `a` and `b`
`triangularParam (a, m, b)`	Returns the triangular random parameter between `a` and `b` with median `m`
`normalParam (m, n)`	Returns the normal random parameter with mean `m` and variance `n`
`exponentialParam (m)`	Returns the exponential random parameter with mean `m`
`erlangParam (b, m)`	Returns the Erlang random parameter with scale `b` and integer shape `m`
`poissonParam (m)`	Returns the Poisson random parameter with mean `m`
`binomialParam (p, n)`	Returns the binomial random parameter on `n` trials of probability `p`

Such parameters are useful for providing the Sensitivity analysis. They can be used in the equations.

By redefining some constants as random external parameters, we can test the model for stability. For that, VisualAivika supports the Monte-Carlo simulation to provide the Sensitivity Analysis. The results of this

analysis can be displayed on charts like figure 5.3, where the *Deviation Chart* option is used for the *Graph Element*.

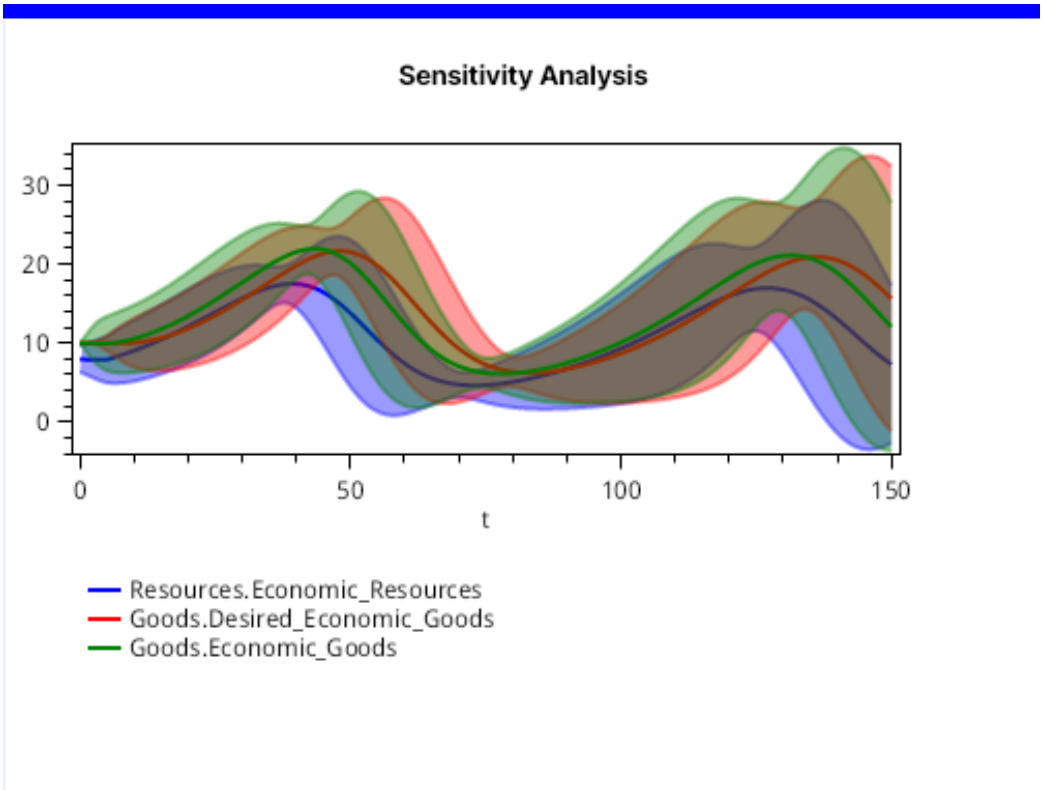

Figure 5.3: The chart shows how stable is the model relative to changes of the external random parameters.

Furthermore, combining the array initialization syntax and built-in variables `runIndex` and `runCount`, we can specify sampling for the external parameters. An idea is quite simple. We define an array with values and then refer to the array by the run index, probably, bound by the array length.

Example

```
A = [[1, 2, 3],
     [2, 3, 1],
     [3, 1, 2]];
```

```
Sample = mod(runIndex, length(A, 0));

X1 = A[Sample, 0];
X2 = A[Sample, 1];
X3 = A[Sample, 2];
```

5.10 Transacts

The block computations process transacts. Most of the blocks take expressions that can have access to the transact attributes by name, which can be an arbitrary identifier.

Table 5.14: Examples of the Transact Attributes

`transact.attribute`	Access to the "attribute" attribute of the transact
`transact.ABC`	Access to the "ABC" attribute of the transact

The transact can have an arbitrary number of attributes.

When splitting the transact into copies, the changes of transact attributes do not affect other transacts from the same assembly set.

5.11 Stream of Arrival Events

Before creating transacts by the generator block, the corresponding arrival events have to come in the simulation model outside. Such events are defined with help of the `Stream` computation.

Table 5.15: Stream Computation

`Stream.empty`	Returns an empty stream that does not produce arrival events
`Stream.take(s, n)`	Takes the specified number `n` of arrival events from the stream `s`

`Stream.random(a, b)`	Returns a stream of arrival events with uniform random delays between `a` and `b`
`Stream.randomInt(a, b)`	Returns a stream of arrival events with integer uniform random delays between `a` and `b`
`Stream.triangular(a, m, b)`	Returns a stream of arrival events with triangular random delays between `a` and `b` with median `m`
`Stream.normal(m, n)`	Returns a stream of arrival events with normal random delays with mean `m` and deviation `n`
`Stream.exponential(m)`	Returns a stream of arrival events with exponential random delays with mean `m`
`Stream.erlang(b, m)`	Returns a stream of arrival events with Erlang random delays with scale `b` and integer shape `m`
`Stream.poisson(m)`	Returns a stream of arrival events with Poisson integer random delays with mean `m`
`Stream.binomial(p, n)`	Returns a stream of arrival events with binomial random delays on `n` trials of probability `p`

To create the specified number of arrival events, say 15, at the start time of simulation, you can create a stream by the following example.

Example

```
S = Stream.take(Stream.random(0, 0), 15);
```

Here we create an infinite stream and then take only 15 initial arrival events from it.

5.12 Generator Block

The generator block computation can take the stream of arrival events, take the block chain and then starts generating and processing of transacts. Usually, this happens implicitly, when applying the `Block.runByStream` function, but the information about the generator blocks should be provided, nevertheless.

Table 5.16: Generator Block

`GeneratorBlock.byStream(s)`	Returns a new generator block by the specified stream `s` of arrival events

5.13 Block Computation

The transact processing is performed within blocks. The block can delay the transacts, change them and finally destroy. The blocks are composable in that sense that we can create a chain of blocks, which can be even infinite or have loop-backs in case of need.

The composion operator looks like `b1 >>> b2`, where the transacts are initially processed by block `b1` and then by block or block chain `b2`.

The difference between the ordinary block and block chain is that the block chain either terminates the processing or has an infinite loop. On the contrary, the block passes the transact further, possibly by modifying one of its attributes.

The block computations can be branched with help of the `Block.select` operator, or a few block computations can be united into one computation with help of the composition operator.

The block composition is opposite to the direction in which the transacts are handled!

Table 5.17: Basic Block Computation

`Block.select(if x then y else z)`	Returns a block chain that processes transacts by the block chain `y` if condition `x` is true, otherwise processes them by the block chain `z`
`Block.terminate`	Returns a block chain that finishes the processing of transacts
`Block.identity`	Returns a block that just passes input transacts further
`Block.transfer (b)`	Returns a block chain that redirects the processing of transacts to the specified block chain `b`. It allows creating loop-backs
`Block.assign (transact.attribute <- v)`	Returns a block that assigns the specified transact attribute to value `v`, when processing the transact
`Block.advance (v)`	Returns a block that holds every transact for the specified time interval `v`, which can be an expression that can depend on the transact attributes
`Block.priority(p)`	Returns a block that assigns a new priority `p` to transacts, where `p` can be an expression that can depend on the transact attributes

The following example creates a block chain that holds every transact for 10 time units, then holds for the time interval calculated from the expression and then finally terminates.

Example

```
B = Block.advance(10) >>>
      Block.advance(20 + transact.SomeDelay) >>>
      Block.terminate;
```

Below is another example that defines the simplest infinite loop. The `Block.transfer` function allows the simulator to cut the circular dependency. Otherwise, the equation could not be resolved.

Example

```
B = Block.transfer(B);
```

To save the current modeling time in the arbitrary `ArrivalTime` attribute, we can use the next example. Then the transact is passed to the specified block chain.

Example

```
B = Block.assign(transact.ArrivalTime <- time) >>> B2;
```

Here is a small trick. If the block chain consists of multiple blocks then it makes sense to name them with help of the same identifiers, which will differ by some suffix that will indicate to the step number.

Example

```
B1 = Block.advance(10) >>> B2;
B2 = Block.advance(20) >>> B3;
B3 = Block.advance(30) >>> B4;
B4 = Block.terminate;
```

Then such equations will appear in the *Equations* tab sequentially one by another. The equations are sorted by their name.

Finally, the next example shows how to create two branches `B2` and `B3` from the same computation, where the decision depends on the random number generator. Here we could check the available resources, for example.

Example

```
B = Block.select(if random(0, 1) < 0.3 then B2 else B3);
```

On the contrary, if we have two block computations `B4` and `B5`, then we can redirect the both of them to the same block chain `B` as illustrated below.

Example

```
B8 = B4 >>> B;
B9 = B5 >>> B;
```

The both transact processing paths will merge into one.

5.14 Running Blocks

Provided some block chain, we can run the processing of transacts by that block chain. For that we need a generator block, which can be created by the stream of arrival events.

Table 5.18: Running Block Computation

`Block.run(g, b)`	Returns an action that can be a subject of applying the `do!` operator to run the processing of transacts by block chain `b`, where the transacts are prepared by the generator block `g`
`Block.runByStream(s, b)`	Returns an action that can be a subject of applying the `do!` operator to run the processing of transacts by block chain `b`, where the transacts are received from stream `s`

Here the latter function is just a shorthand for calling the former one with help of `Block.run(GeneratorBlock.byStream(s), b)`.

The both functions return an action that should be performed yet with help of the `do!` operator, which makes the execution explicit in the equations.

The following example illustrates the simplest complete model, but which processes nothing.

Example

```
A = do! Block.runByStream(Stream.empty, Block.terminate);
```

Here the empty arrival stream is processed by the termination block chain. Nothing happens. But it demonstrates the main idea.

Below is another example.

Example

```
S = Stream.exponential(5);
B = Block.advance(3) >>> Block.terminate;
A = do! Block.runByStream(S, B);
```

The `do!` operator can be applied within array items too.

Example

```
S = [ Stream.exponential(i) | i <- 5..10 ];
B = [ Block.advance(i - 2) >>> Block.terminate | i <- 5..10 ];
A = [ do! Block.runByStream(S[i], B[i]) | i <- 5..10 ];
```

5.15 Queue

VisualAivika introduces some entities for gathering statistics. Some of such entities is a queue. The transact may enqueue and then dequeue. VisualAivika counts each of these operations.

Table 5.19: Queue Constructor

`Queue.create()`	Creates a new queue instance

To enque and dequeue transacts, we have to use the corresponding block computations. These computations have an optional increment and decrement parameters, which are equal to 1 by default.

Table 5.20: Queue Operations

`Block.queue(q)`	Returns a block that enqueues the transacts in queue `q` with the content increment equal to 1
`Block.queue(q, n)`	Returns a block that enqueues the transacts in queue `q` with the specified content increment `n`
`Block.depart(q)`	Returns a block that dequeues the transacts from queue `q` with the content decrement equal to 1
`Block.depart(q, n)`	Returns a block that dequeues the transacts from queue `q` with the specified content decrement

For example, we can enqueue the transacts, delay them and then dequeue. Usually, we also can try to acquire some resource right after enqueueing, but the resources are described further.

Example

```
Q = Queue.create();
B = Block.enqueue(Q) >>>
      Block.advance(12 + random(0, 10) + transact.ABC) >>>
      Block.depart(Q) >>> B2;
```

At any modeling time we can receive the information about the queue properties, for example, to plot them on the chart such as *Deviation chart*.

Table 5.21: Queue Properties

`Queue.content(q)`	Returns the queue content value at the current modeling time
`Queue.contentStats(q)`	Returns the queue content statistics summary at the current modeling time. The statistics value is time persistent (the timing statistics)

`Queue.enqueueCount(q)`	Returns the enqueue count value for queue `q` at the current modeling time
`Queue.zeroEntry-EnqueueCount(q)`	Like the former function, but returns such the enqueue count, where there was no delay between dequeueing from the queue and enqueueing
`Queue.waitTime(q)`	Returns the queue wait time statistics summary at the current modeling time. The statistics value is based upon observations (the sampling statistics)
`Queue.nonZeroEntry-WaitTime(q)`	Like the former function, but returns such the statistics summary, where there was a delay between dequeueing from the queue and enqueueing

For example, we create a temporary variable `WaitTime` to save the statistics, which can be already added to the deviation chart.

Example

```
Q = Queue.create();
WaitTime = Queue.waitTime(Q);
```

5.16 Facility

Another kind of entities supported by VisualAivika is a facility. The facility is a resource that may have only one owner. The facility can be seized, preempted, released and returned. When preempting the facility, the old owner can be transferred to another block chain to proceed with its execution. It allows us to model quite a complex behaviour.

Table 5.22: Facility Constructor

`Facility.create()`	Creates a new facility instance

To affect the facility by transacts, there are the corresponding block computations. Some of these computations may have optional parameters. The most complex behaviour is inherent in the `Block.preempt` function.

Table 5.23: Facility Operations

`Block.seize(f)`	Returns a block that tries to seize facility `f`. Later, the facility must be released by the transact
`Block.release(f)`	Returns a block that releases facility `f`, which was seized by the transact earlier
`Block.preempt(f)`	Returns a block that tries to preempt the specified facility `f`. Later, the facility must be returned by the transact. See below
`Block.preempt(f, priorityMode=f1, removalMode=f2)`	Returns a block that tries to preempt the specified facility `f`, where the priority mode and removal mode flags `f1` and `f2` are optional. By default, the both boolean flags are turned off. See below
`Block.preempt(f, priorityMode=f1, removalMode=f2, transfer(b) via transact.attribute)`	Like the former function, but if the transact is preempted then it will be transferred to the block chain `b`, where the specified transact attribute will contain the remaining time, during which the transact would had to be hold yet in the delay. See below

`Block.preempt(f, transfer(b) via transact.attribute)`	A more simple form of the former function with default mode flags
`Block.return(f)`	Returns a block that returns facility `f`, which was preempted by the transact earlier

The most popular use case is to count the queue statistics, when trying to acquire the facility, simulate some activity with help of the delay and then release or return the facility. Here the transact will be blocked in case of the resource sufficiency, which will increase the wait time for the queue.

Example

```
Q = Queue.create();
F = Facility.create();
B = Block.queue(Q) >>>
      Block.seize(F) >>>
      Block.depart(Q) >>>
      Block.advance(random(10, 20)) >>>
      Block.release(F) >>> B2;
```

Like it was true for the queue, we can request for the facility properties and statistics during the simulation at any modeling time.

Table 5.24: Facility Properties

`Facility.isInterrupted(f)`	Returns a boolean flag indicating whether the facility `f` is interrupted at the current modeling time
`Facility.count(f)`	Returns the facility count value at the current modeling time
`Facility.countStats(f)`	Returns the facility count statistics summary at the current modeling time. The statistics value is time persistent (the timing statistics)

`Facility.captureCount(f)`	Returns the capture count value for the facility `f` at the current modeling time
`Facility.utilisation-Count(f)`	Returns the utilisation count value for the facility `f` at the current modeling time
`Facility.utilisation-CountStats(f)`	Returns the facility utilisation count statistics summary at the current modeling time. The statistics value is time persistent (the timing statistics)
`Facility.queueLength(f)`	Returns the queue length for the facility `f` at the current modeling time
`Facility.queueLength-Stats(f)`	Returns the facility queue length statistics summary at the current modeling time. The statistics value is time persistent (the timing statistics)
`Facility.totalWait-Time(f)`	Returns the total wait time for the facility f at the current modeling time
`Facility.waitTime(f)`	Returns the facility wait time statistics summary at the current modeling time. The statistics value is based upon observations (the sampling statistics)
`Facility.totalHolding-Time(f)`	Returns the total holding time for the facility f at the current modeling time
`Facility.holdingTime(f)`	Returns the facility holding time statistics summary at the current modeling time. The statistics value is based upon observations (the sampling statistics)

As before, we can save any of these properties in some variable and

then display the result.

We can request for any of the properties during simulation. For example, we can test whether the facility is interrupted currently. If the facility is interrupted then we can terminate the transact processing as shown below in the example.

Example

```
Q = Queue.create();
F = Facility.create();
B = Block.select(if Facility.isInterrupted(F) then Busy else B2);
B2 = Block.preempt(F, priorityMode=true,
       transfer(Add) via transact.P5) >>> B3;
...
Busy = Block.terminate;
```

Also here the preempted transact, the old owner of the facility, will be redirected to block chain `Add`, where the transact attribute with name "P5" will contain the remaining time value, during which the transact would had to be hold yet in the delay.

Section A in the Appendix provides some technical details of that how the facility preemption is implemented in the simulator.

5.17 Storage

Another kind of resources supported by VisualAivika is a storage. The storage has a capacity. Also the storage can be borrowed by multiple transacts, while it has the available content.

Table 5.25: Storage Constructor

`Storage.create(n)`	Creates a new storage instance by the specified capacity `n`

To use the storage, there are the corresponding block computations. Some of these computations may have optional parameters.

Table 5.26: Storage Operations

`Block.enter(s)`	Returns a block that tries to enter storage `s` with the content decrement equal to 1. Later, the storage must be left by the transact
`Block.enter(s, n)`	Returns a block that tries to enter storage `s` with the specified content decrement `n`. Later, the storage must be left by the transact
`Block.leave(s)`	Returns a block that leaves storage `s` with the content increment equal to 1. The storage had to be entered into by the transact earlier
`Block.leave(s, n)`	Returns a block that leaves storage `s` with the specified content increment `n`. The storage had to be entered into by the transact earlier

When trying to enter the storage, if the content is not sufficient then the transact is blocked. But this behavior is much simpler than the facility seizing or preemption. Another difference is that the content changes can differ from 1.

The usual approach is to count the queue statistics, when trying to enter the storage, simulate some activity with help of the delay and then leave the storage. Here the transact will be blocked in case of the resource sufficiency, which will increase the wait time for the queue.

Example

```
Q = Queue.create();
S = Storage.create();
B = Block.queue(Q) >>>
      Block.enter(S) >>>
      Block.depart(Q) >>>
      Block.advance(random(10, 20)) >>>
      Block.leave(S) >>> B2;
```

The storage has its own properties. We can request the storage for them at any modeling time.

Table 5.27: Storage Properties

`Storage.capacity(s)`	Returns the storage capacity
`Storage.content(s)`	Returns the storage content value at the current modeling time
`Storage.contentStats(s)`	Returns the storage content statistics summary at the current modeling time. The statistics value is time persistent (the timing statistics)
`Storage.useCount(s)`	Returns the total number of cases for storage `s`, when the content had been decreased
`Storage.usedContent(s)`	Returns the total used content value for storage `s` at the current modeling time
`Storage.content-Utilisation(s)`	Returns the storage content utilisation value at the current modeling time
`Storage.content-UtilisationStats(s)`	Returns the storage content utilisation statistics summary at the current modeling time. The statistics value is time persistent (the timing statistics)
`Storage.queueLength(s)`	Returns the queue length for the storage `s` at the current modeling time
`Storage.queueLength-Stats(s)`	Returns the storage queue length statistics summary at the current modeling time. The statistics value is time persistent (the timing statistics)

`Storage.totalWaitTime(s)`	Returns the total wait time for the storage `s` at the current modeling time
`Storage.waitTime(s)`	Returns the storage wait time statistics summary at the current modeling time. The statistics value is based upon observations (the sampling statistics)
`Storage.averageHolding-Time(s)`	Returns the average holding time for the storage `s` at the current modeling time

We can save any of these properties in some variable and then display the result.

5.18 Assembly Set

Every transact created by the Generator Block has its own assembly set. At the same time, during simulation the transacts can be splitted into multiple copies, but each copy will belong to the same assembly set. Then such transacts can be assembled into one, or they can delayed before all them will be gathered together.

Also we can create so called a Match Chain, so that some transact could be delayed until another one from the same assembly set would match the common match chain instance.

Table 5.28: Match Chain Constructor

`MatchChain.create()`	Creates a new match chain instance

There are the following block computations to use the assembly set that will be common for the whole group of transacts, which were splitted earlier from the same transact.

Table 5.29: Assembly Set Operations

`Block.split(n, b)`	Returns a block that splits every transact into `n` copies, each of them is transfered to the specified block chain `b`. Every copy will belong to the same assembly set of the source transact
`Block.assemble(n)`	Returns a block that assembles `n` transacts from the same assembly set in one transact, after they were splitted earlier
`Block.gather(n)`	Returns a block that delays and gathers `n` transacts from the same assembly set, after they were splitted earlier. After `n` transacts are gathered, they continue their execution
`Block.matchChain(c)`	Returns a block that delays the transact until another transact from the same assembly set will call a similar block with the same match chain `c`

The following example demonstrates how we can split and then gather the transacts.

Example

```
Worker = Facility.create();

S2 = Block.seize(Worker) >>>
       Block.advance(random(8 - 3, 8 + 3)) >>>
       Block.split(1, S2) >>>
       Block.release(Worker) >>>
       Block.priority(1) >>>
       Block.gather(24) >>> S3;
```

It captures the facility to make transact copies with delay. Then the priority increases and 24 transacts are gathered. Actually, in this example

only one source initial transact is used, but the example creates a plenty of its copies.

5.19 Nested Modules

The model can contain modules that can be nested. Each module defines its own name space for variables.

VisualAivika uses modules to keep variables from the same diagram in a separate module.

Example

```
// the global variable
A = 1;

module M1 {

    // variable M1.B
    B = A + 1;

    module M2 {

        // variable M1.M2.C
        C = 2 * B;

        // use the qualified name to M1.B
        D = C - M1.B;
    }
}
```

Chapter 6

Displaying Results

The *Result Chart* element from the diagram toolbar allows you to see the simulations results in a preferred way. Figure 6.1 shows the corresponding element on the toolbar. Then the result output view can be a chart, or table with CSV data, or something else. The element specifies exactly how the results will be represented on its view. The diagram can contain an arbitrary number of chart elements.

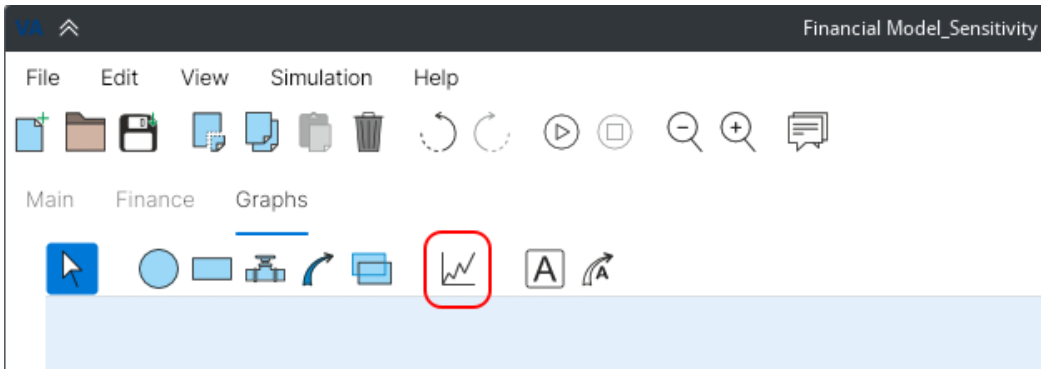

Figure 6.1: The Result Chart element on the diagram toolbar.

6.1 Deviation Chart

The Deviation Chart view is the most universal view, which is applicable both to single and multiple runs. If the run is single, then the Deviation

Chart becomes similar to the Time Series chart as described in section 6.2. But if the Monte-Carlo simulation experiment is used, then the Deviation Chart shows the trend and confidence intervals by the 3-sigma rule.

Figure 6.2 illustrates how we can select the corresponding Deviation Chart view on the *Chart* editor. This editor is opened right after you select any Result Chart element on the diagram. Here you should define the *Result Type* field value equal to *Deviation Chart* and then click on the *Apply* button, which is not shown on the figure.

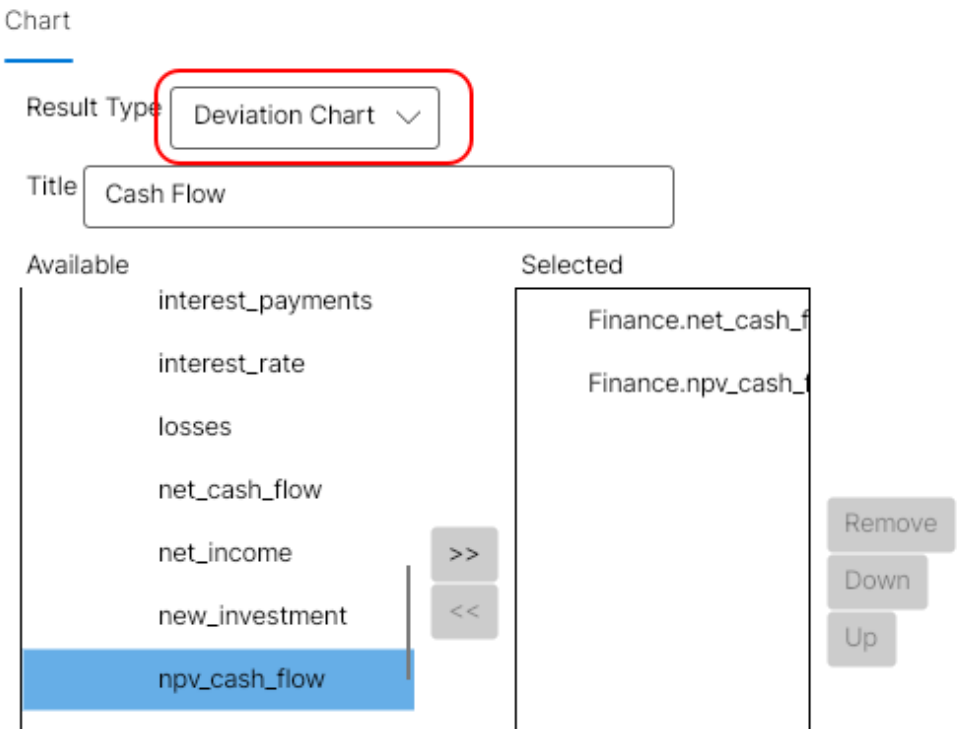

Figure 6.2: Selecting the Deviation Chart view.

Figure 6.3 shows how the deviation chart can look on the diagram after the simulation is finished.

The Deviation Chart view can show the statistics values too. For example, it can show the queue wait time or facility content statistics. Here we suppose that the statistics values are distributed equally for different simulation runs, but these runs are independent from each other.

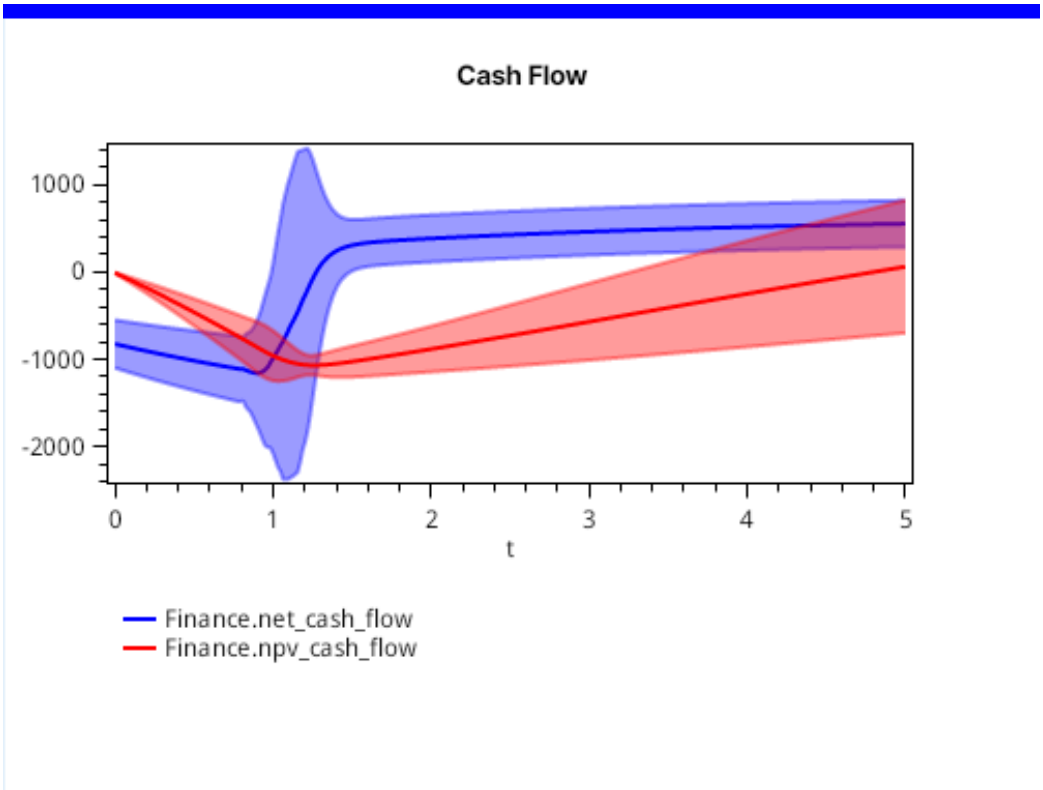

Figure 6.3: The Deviation Chart example.

6.2 Time Series

The Time Series chart displays the ordinary chart, where the series depend on the modeling time. If the Monte-Carlo method is used, then the corresponding number of charts will be plotted, by one for each run. Hence, please use the Time Series chart only for single run, or for the Monte-Carlo method with a small number of runs!

Like shown in figure 6.2, to dislay the Time Series chart, now it is necessary to select the *Result Type* field equal to *Time Series*, which is selected by default.

Figure 6.4 shows how the Time Series chart can look on the diagram for a single simulation run.

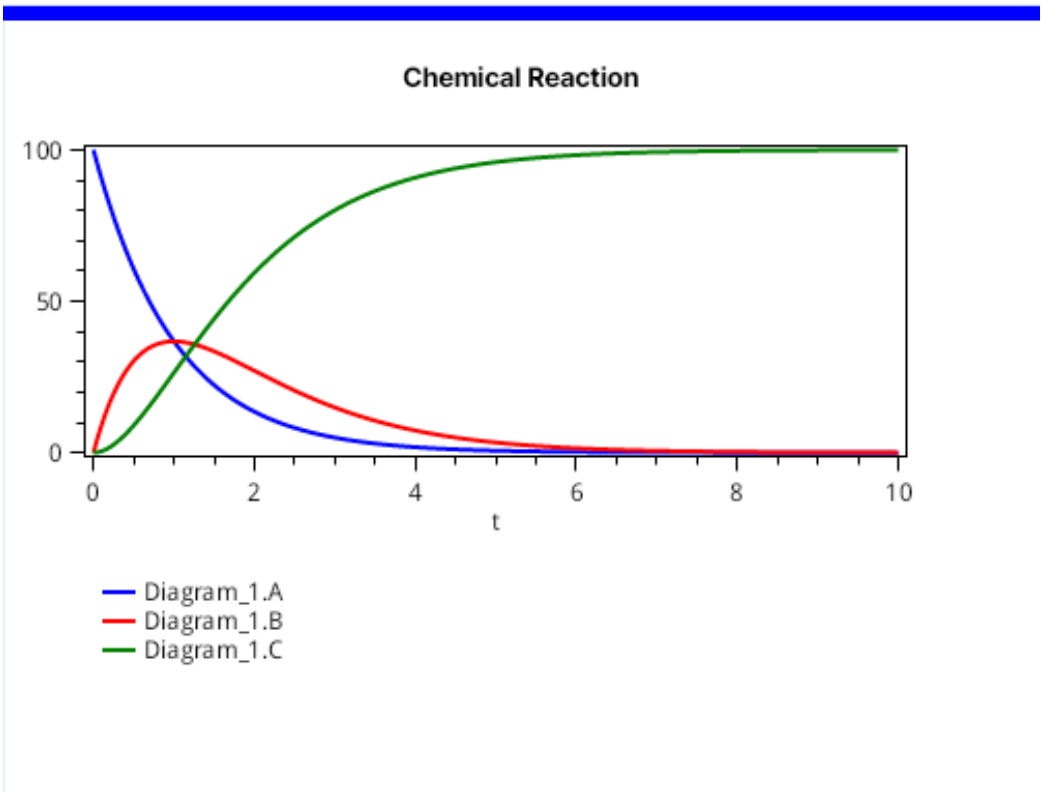

Figure 6.4: The Time Series example.

6.3 XY Chart

The XY Chart is similar to the Time Series chart described in the previous section 6.2. Only the first selected series becomes the X coordinate for the chart. As before, if the Monte-Carlo method is used, then the corresponding number of charts will be plotted, by one for each run. Hence, please use the XY Chart view only for single run, or for the Monte-Carlo method with a small number of runs!

Like shown in figure 6.2, to dislay the XY chart, now it is necessary to select the *Result Type* field equal to *XY Chart*.

Figure 6.5 shows how the XY Chart can look on the diagram for a single simulation run.

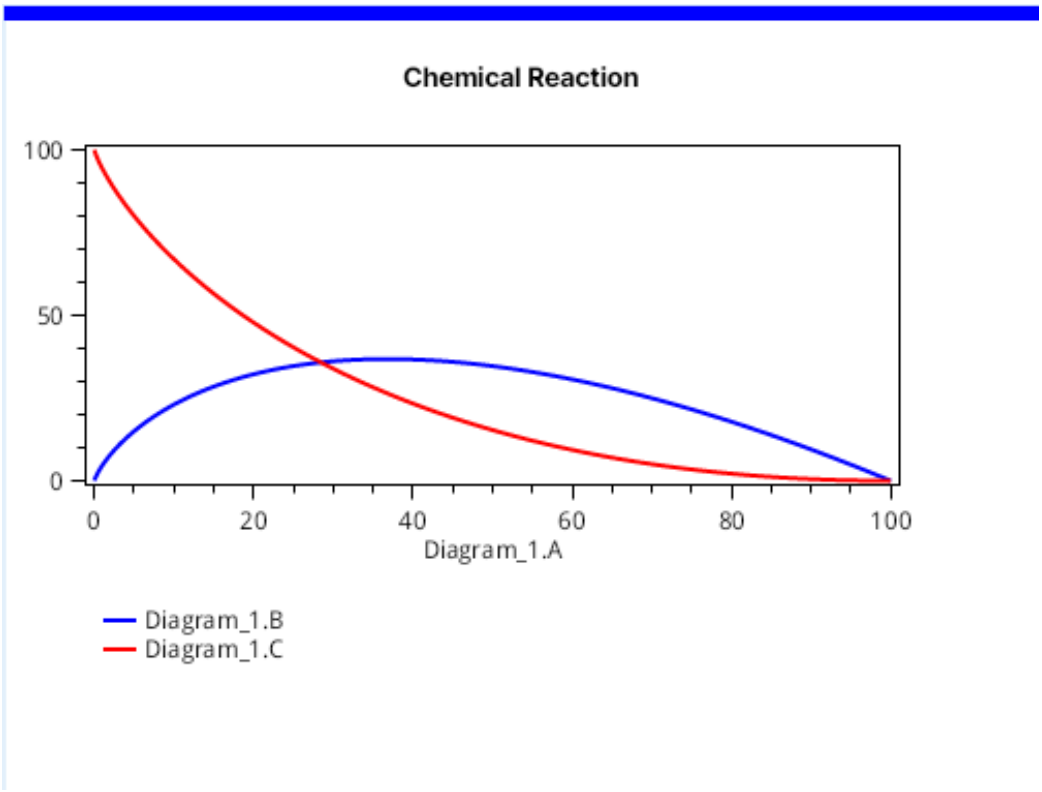

Figure 6.5: The XY Chart example.

6.4 CSV Table

The CSV Table view is destined for exporting CSV data to other applications, or saving the data in the file. A text block component is displayed, from which you can copy the corresponding CSV data. If the Monte-Carlo method is used, then the corresponding number of components will be created, by one for each run. Hence, please use the CSV Table view only for single run, or for the Monte-Carlo method with a small number of runs!

Like shown in figure 6.2, to dislay the CSV table, it is necessary to select the *Result Type* field equal to *Table*.

Figure 6.6 shows how the CSV Table can look on the diagram.

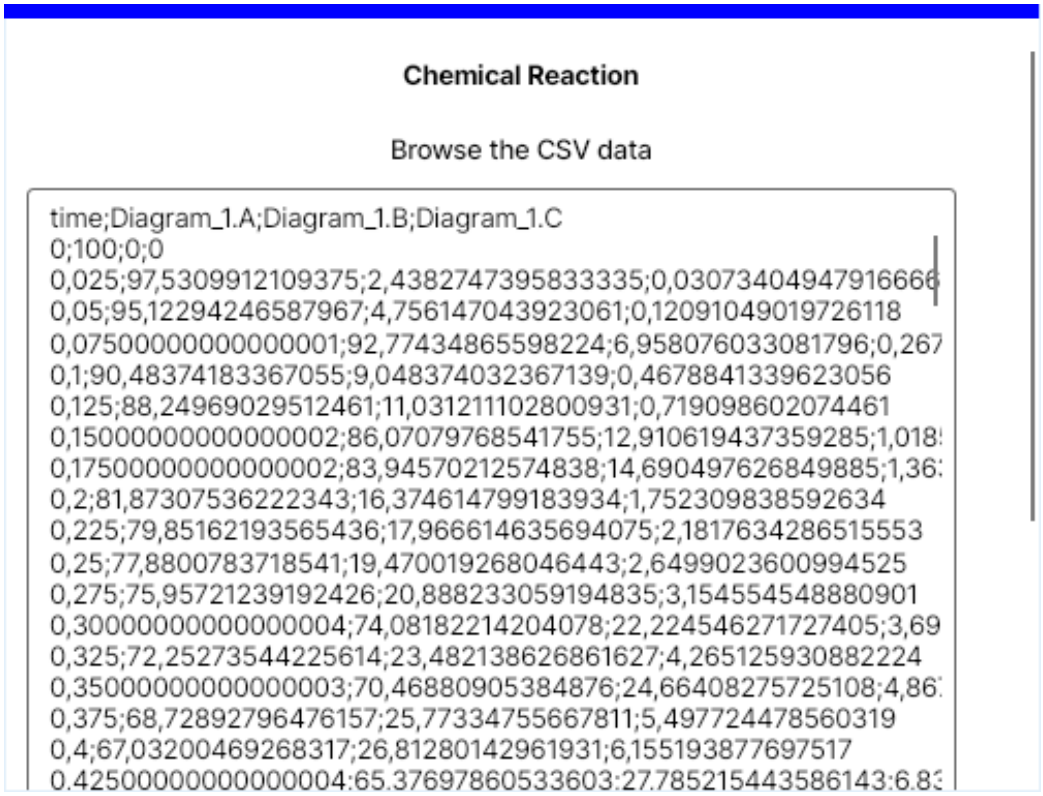

Figure 6.6: The CSV Table example.

6.5 Last Values

The view of Last Values is destined for displaying the series values at the final time point of simulation. If the Monte-Carlo method is used, then the corresponding number of components will be created, by one for each run. Hence, please use the Last Values only for single run, or for the Monte-Carlo method with a small number of runs!

Like shown in figure 6.2, to dislay the Last Values, it is necessary to select the *Result Type* field equal to *Last Values*.

Figure 6.7 shows how the corresponding element can look on the diagram.

When displaying the statistics summary, this element shows also the minimal and maximal values as well as the average and deviation. Moreover, the element is able to display all the properties together for the queue,

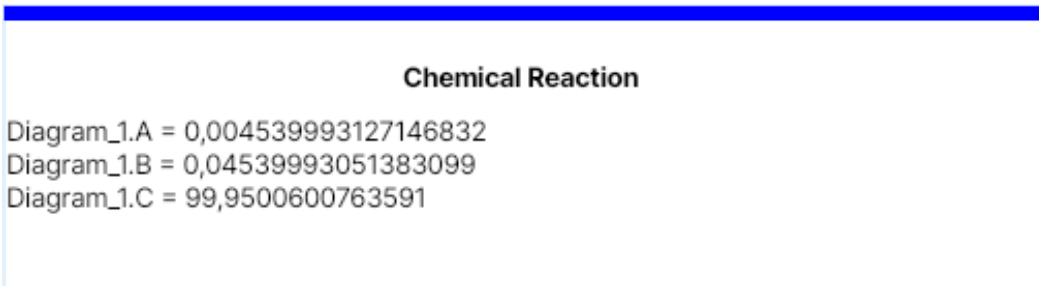

Figure 6.7: The example of representing the Last Values.

facility and storage.

6.6 Last Value Histogram

The Last Value Histogram view is destined for plotting the histogram by series values at the final modeling time, when using the Monte-Carlo method with multiple runs.

Like shown in figure 6.2, to dislay the Last Value Histogram, it is necessary to select the *Result Type* field equal to *Last Value Histogram*.

Figure 6.8 shows how the corresponding element can look on the diagram.

6.7 Last Value CSV Table

When applying the Monte-Carlo method, the Last Value CSV Table view collects data at final time points, and it is destined for exporting the data to other applications, or saving the data in the file. A text block component is displayed, from which you can copy the corresponding CSV data.

Like shown in figure 6.2, to dislay the Last Value CSV Table, it is necessary to select the *Result Type* field equal to *Last Value Table*.

Figure 6.9 shows how the corresponding element can look on the diagram.

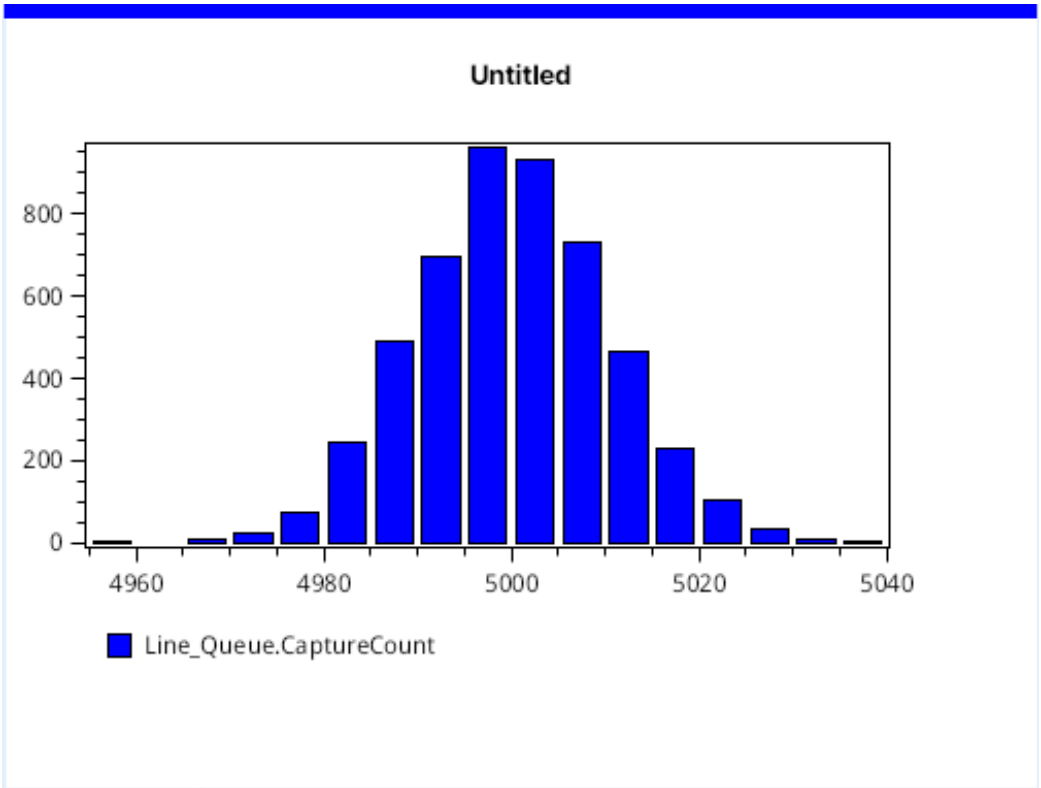

Figure 6.8: The Last Value Histogram example.

6.8 Last Value Statistics Summary

When applying the Monte-Carlo method, the Last Value Statistics Summary view collects data at final time points and presents the results on the corresponding component.

Like shown in figure 6.2, to dislay the Last Value Statistics Summary, it is necessary to select the *Result Type* field equal to *Last Value Statistics*.

Figure 6.10 shows how the corresponding element can look on the diagram.

There is one subtle thing related to displaying the `count` parameter for the time persistent statistics. Before displaying the time persistent statistics, the latter is transformed to a similar representation based upon observations, where the `count` parameter becomes to show a completely different value, while the rest parameters remain correct. Such a trans-

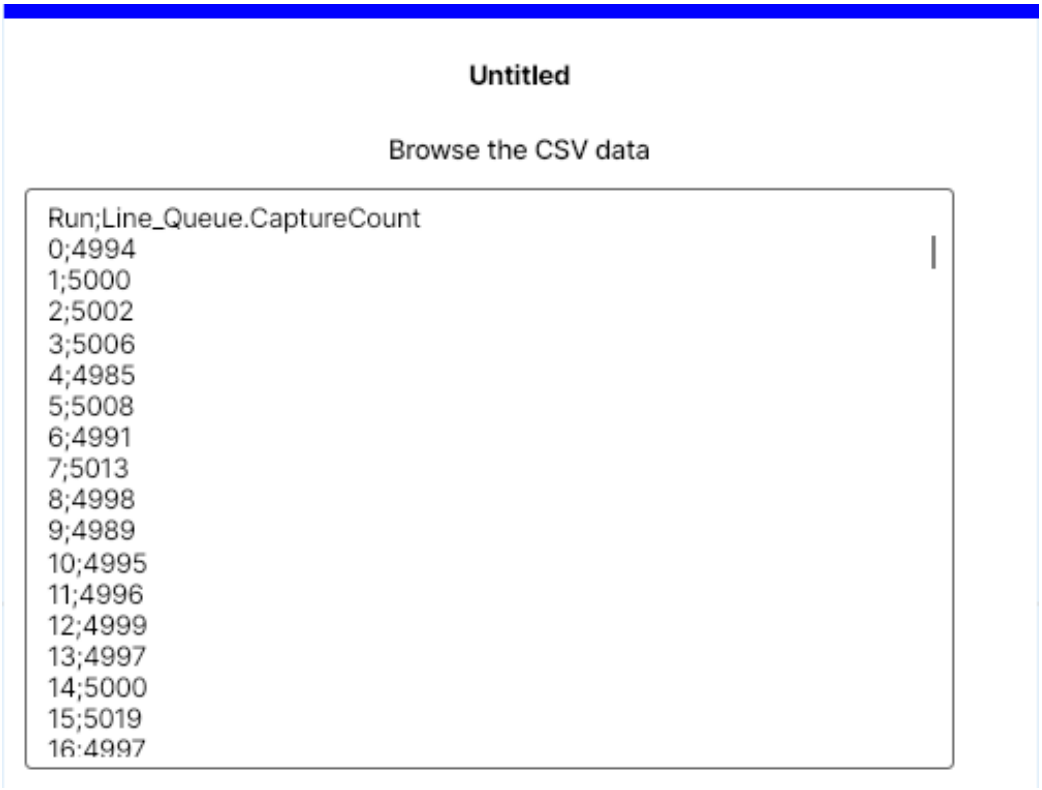

Figure 6.9: The Last Value CSV Table example.

formed statistics is called *normalized* in VisualAivika. So, this element does not show the `count` parameter for the time persistent statistics. In other cases, the `count` parameter is what it is meant to be.

For example, the mentioned figure 6.10 displays two statistics summary objects. The first one is an ordinary observation-based sampling statistics. The second block corresponds to the transformed version of the time persistent statistics, where the `count` parameter is a result of normalization.

Here it is important to note that Bessel's correction is not applied to the normalized version of the statistics.

Untitled

Graphs.Queue_Length	
mean	0,16899999999999996
deviation	0,3749394345485409
minimum	0
maximum	1
count	1000
Graphs.Queue_Length_Stats	
mean	0,1876680168267531
deviation	0,4122502542915388
minimum	0
maximum	3
count	100000
is the statistics normalised?	true

Figure 6.10: The Last Value Statistics Summary example.

Chapter 7

Exporting .NET Applications

The Equation Compiler version of VisualAivika allows you to export your simulation models as .NET applications. Such exported applications will depend on the programming library code, which is available in the source form under the corresponding licences. It means that the exported applications can be built and run without VisualAivika itself. VisualAivika can be used for modeling and exporting only.

This feature is especially useful if you are going to extend your simulation models with help of arbitrary .NET code, which can be involved in the simulation. Please refer to the corresponding chapter 8 for more detail.

To export the simulation model as the .NET application, please select menu *Simulation / Export As .NET Application*. Then select the directory, which VisualAivika will save the application in. VisualAivika can ask you to save the model before it, if required.

The resulting .NET application will be exported in the simplest form, when it can dump the results in the final time point. But the code itself is general enough. It can be adapted for different purposes. Please read the manual *IronAivika: Simulation Library for .NET* for more detail.

To run the test application, you should switch to the corresponding directory, where the application was exported to. Then you can enter in the console[1]:

```
$ dotnet build
$ dotnet run
```

It works if you will see something similar to

[1]The prompt sign will be different on Windows.

```
----------

// time
t = 10

Diagram_1.A = 220,19641241130046
```

Here it shows the final value for some variable `Diagram_1.A`. You can see another output for your own simulation model.

This is plasticine, from which you can sculpt any figure.

Chapter 8

External Modules

VisualAivika allows you to extend your simulation models by calling arbitrary .NET programming code within simulation. This is especially useful if you are going to export your simulation models as .NET applications, which was described in chapter 7.

To develop and use external modules, please read the manual *VisualAivika Manual: Creating External Modules*.

Example

```
F =
  [<assembly("lib/DelayBlockFunction.dll")>]
  [<class(VisualAivika.Demo.DelayBlockFunction)>]
  extern fun (A: Block): Block;

Y1 = F(Block.advance(10) >>> Block.terminate);
```

Here the `DelayBlockFunction` class type can be defined in F# or C#. It must be located in the `DelayBlockFunction.dll` assembly.

The external modules can define functions and modules. The latter ones can have multiple input and output parameters. The function provided in the example is just a particular case for the module.

Appendix A

Facility Implementation Details

These sections are provided as a reference material for more deep understanding of that how the model is simulated.

A.1 Facility Preemption

Below is described the logic behind the `Block.preempt` computation.

The optional parameter `priorityMode` specifies whether the Priority or Interrupt mode (default) is used. The `transfer` parameter can define a block chain, where the preempted transact is passed to in case of preemption. The `removalMode` parameter specifies, whether the Remove mode is used (not used by default).

Conceptually, the `Block.preempt` computation tries to mimic the behaviour of the `PREEMPT` block from the GPSS modelling language, although the implementation in VisualAivika is based on completely different principles that have roots in functional programming.

The facility may have one owner only, i.e. a transact that owns the facility, or the facility has no owner at all. Moreover, the facility has three queues: *Delay chain*, *Interrupt chain* and *Pending chain*. The Delay chain and Pending chain tend to be similar to FIFO, while the Interrupt chain is close to LIFO, but these queues use the priorities. The FIFO queues store the corresponding transact computations along with transacts themselves.

The algorithm applied by the `Block.preempt` computation is as follows.

1. If the facility had no owner then the current transact becomes the owner, but as the owner which is marked as *non-interrupting*.

2. Otherwise, if the `priorityMode` parameter is `false` (default) and the current owner is *interrupting* then the current transact is marked as interrupting and with its computation is added to the Pending chain of the facility with the transact priority.

3. Otherwise, if the `priorityMode` parameter is `true` and the transact priority is lower (less) than the owner's priority, then the current transact is marked as interrupting and with its computation is added to the Delay chain of the facility with the transact priority.

4. Otherwise, if the `removalMode` parameter is `false` (default) then the current transact displaces the owner. The current transact is marked as interrupting. The previous owner is preempted and added to the Interrupt chain with its priority.

   Next time, when the previous owner will return from the Interrupt chain and if the `transfer` parameter is not specified (default) then that transact will capture the facility again and proceed with its execution from the block, where it was preempted.

   Otherwise, if the `transfer` parameter specifies a block chain, then the returned owner will be transferred to the corresponding block chain, where the specified transact attribute will assign to the time interval remained to hold the transact, starting from time at which the previous owner was preempted.

5. If other cases fail, and hence `removalMode` is specified explicitly as `true`, then the current transact becomes a new owner and it is marked as interrupting. The previous owner is preempted and transferred to the block chain returned by the specified `transfer` parameter, which must define a block chain computation. The time remained to hold the previous owner is passed through the corresponding transact attribute. Now this `transfer` parameter is obligatory. No Interrupt chain is used. The previous owner is removed at the current modelling time.

A.2 Facility Seizing

The algorithm for the `Block.seize` computation is as follows.

1. If the facility has another owner already, then the transact computation is blocked until that other owner releases or returns the facility. In such a case, the current transact is marked as non-interrupting and with its computation is added to the Delay chain of the facility with the transact priority.

2. Otherwise, if the facility had no owner then the current transact becomes the owner of the facility. This owner is marked as non-interrupting and the transact proceeds with its execution.

A.3 Facility Release

The `Block.release` computation must match the `Block.seize` computation. In the rest, the releasing compution is similar to the facility return procedure described next.

A.4 Facility Return

The `Block.return` computation must match `Block.preempt`. In the rest, the facility releasing and returning computions have a similar behaviour.

The current transact must be the facility owner. The algorithm of selecting another owner is as follows.

1. If the Pending chain of the facility is not empty and the top transact computation is not cancelled then the corresponding transact is removed from the queue and it becomes the owner. The Pending chain is close to FIFO but uses priorities.

   If the top computation was cancelled then the transact is also removed from the queue, but the algorithm is repeated from start again.

2. Otherwise, if the Interrupt chain of the facility is not empty and the top transact has a computation which is not cancelled, then the corresponding transact is removed from the queue and it is selected as a new owner.

   Now if the transact was interrupted together with the `transfer` parameter then this parameter is used to transfer the transact computation to the corresponding block chain returned by the `transfer`

parameter. The previous computation is cancelled and a new one is created instead, which is already started with the specified block chain. The Interrupt chain is close to LIFO but uses priorities.

If the `transfer` parameter was not specified then the transact proceeds with its execution from the computation at which the transact was preempted.

If the top computation was cancelled then the transact is also removed from the queue, but the algorithm is repeated from start again.

3. Otherwise, if the Delay chain of the facility is non-empty and the top transact computation is not cancelled then the corresponding transact is removed from the queue and it becomes the owner. The Delay chain is close to FIFO but uses priorities.

   If the top computation was cancelled then the transact is also removed from the queue, but the algorithm is repeated from start again.

4. If all other cases fail then all three queues are empty and hence the facility has no owner.

Bibliography

[1] Berkeley Madonna. http://www.berkeleymadonna.com, 2024. Accessed: 20-July-2024.

[2] Thomas Schriber. *Simulation using GPSS*. Wiley, 1974.

[3] Vensim Software. http://vensim.com, 2024. Accessed: 20-July-2024.