

A first start in 4DScript



A First start in 4DScript

Simulation Software / TUTORIAL

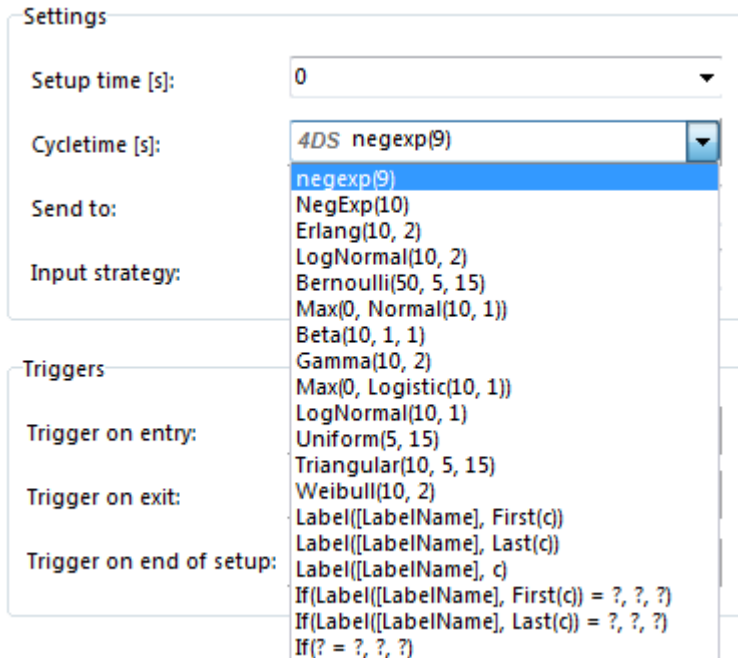
Papendorpseweg 77, 3528 BJ Utrecht, The Netherlands www.incontrolsim.com

1 Table of Contents

| | | |
|------|--|-------|
| 1. | Table of Contents | 1 |
| 2. | Introduction | 2-3 |
| 3. | Getting started | 4 |
| 3.1. | A close look at predefined logics | 4-5 |
| 3.2. | Experimenting with the Interact window | 5-6 |
| 3.3. | Getting help | 6 |
| 4. | Basics of 4DScript | 7 |
| 4.1. | Basic 4DScript | 7-10 |
| 4.2. | Control structures | 10-11 |
| 4.3. | Strings | 11-12 |
| 5. | Enterprise Dynamics scripting | 13 |
| 5.1. | Probability distribution | 13-14 |
| 5.2. | Atom statistics | 14 |
| 5.3. | Atom references | 14-18 |
| 5.4. | Labels | 18-19 |
| 5.5. | Flow control | 19-20 |
| 5.6. | Examples code in standard fields | 20-22 |
| 5.7. | Debugging | 22 |

2 Introduction

4DScript is the programming language of Enterprise Dynamics®. If you are building a model you can use this programming language in many of the properties that you can set on atoms of your model, for example the *Cycletime* and *Send to* property of a Server. Edit fields into which you can enter 4DScript code are indicated with a 4DS marker. Clicking such a field opens a 4DScript editor, see Picture 2.1.



Picture 2.1: 4DScript code options for the Cycletime property of a Server

Usually such 4DS fields do provide a list of commonly used 4DScript code. For example, two of the predefined codes of the *Cycletime* property are


4DScript code

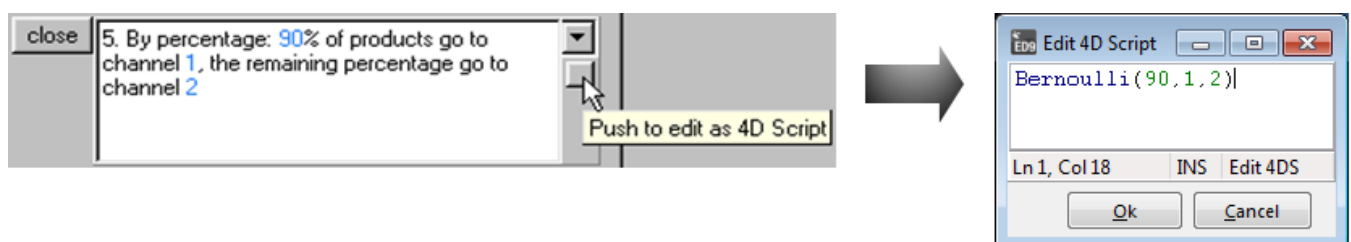
```
Uniform(5, 15)
```

4DScript code

```
Label([LabelName], Last(c))
```

At this moment you do not have to understand the above example codes.

 Note that the *Send to* field does not show the 4DS marker but this is a field in which you can enter 4DScript. This field contains a list of predefined logics, a description is given what the logic does. If you have selected a logic you can push the small button on the right hand side to edit the 4DScript of the logic, see Picture 2.2



Picture 2.2: Opening 4DScript editor of a predefined logic

This tutorial is for those people who want to learn the basics of the 4DScript programming language and how to use it in Enterprise Dynamics. You do not necessarily need to have any previous knowledge of other programming languages, but you need basic modeling skills in Enterprise Dynamics. If you already have some programming

A first start in 4DScript

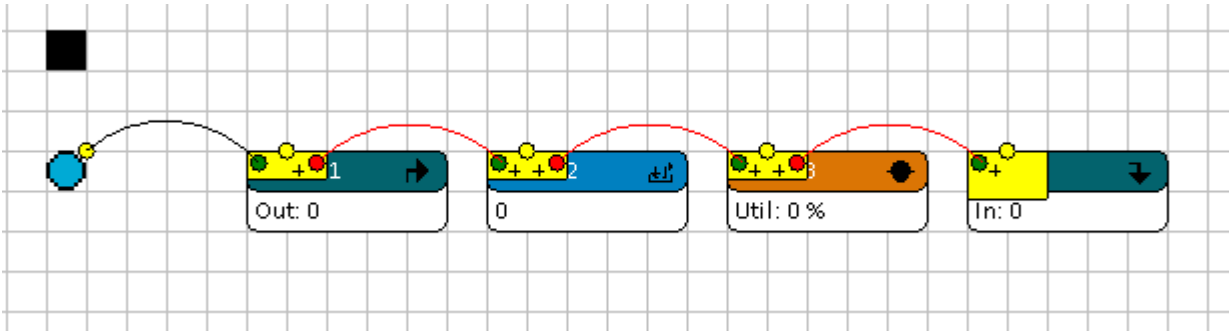
skills, you can use this tutorial as a review of concepts.

In this tutorial you will learn the basic of the 4DScript language such that you can understand and use the predefined logics, alter the parameters in the functions they use and write simple code yourself.

3 Getting started

3.1 A close look at predefined logics

We first start with building a simple model to demonstrate the use of some predefined 4DScript logics. Start with a new model and drag in this order a Source, Queue, Server and Sink atom in the model. Make sure that channel are connected correctly, see Picture 3.1.



Picture 3.1: Single server model

Note that all of these atoms have trigger fields. For example, on the general page of the Source atom you can find a *Trigger on Creation* and a *Trigger on Exit*. The Queue and the Server have a *Trigger on Entry* and a *Trigger on Exit* field. These trigger fields can be used to make something happen the moment a product either enters or leaves that atom. In this example we will use it to change the color of the products. Open the dialog of the Queue atom. Note that the trigger fields both display the 4DS marker at the beginning of the field that indicates that they can contain 4DScript code. If you click the drop-down triangle on the right side of the field, a list is shown containing some predefined 4DScript codes. Select for the exit trigger the predefined code `Icon(i) := ?` and change the question mark to 23. The code in the *Trigger on exit* should be

Trigger on exit: Change the icon of a product

```
Icon(i) := 23
```

Run the model and check that the icon of the product in the Queue is different from the icon of the product in the Server. The moment the product leaves the Queue its icon changes to icon number 23. You can find the icon belonging to this number in the **Resource Manager** ('Resource Manager 2D Icons' in the on-line documentation).

Another property that you can change using 4DScript is the *cycletime* of a Server. Open the dialog of the Server. Note that this field also begins with the 4DS marker. Open the drop down list and select the predefined code for a uniform distribution. The code in the *cycletime* field should be

Cycletime: uniform distribution

```
Uniform(5, 15)
```

The uniform distribution has two parameters, the minimum and the maximum value of the distribution. Run the model again for 100 hours. Check with the Summary Report that products stay on average about 10 seconds in the Server.

You can also click on the *cycletime* field a 4DScript editor appears. Remove the code and replace it with the value 9:

Cycletime: deterministic (seconds)

```
9
```

Run the model. Now products will stay exactly 9 seconds in the Server. The *cycletime* field requires you to enter the cycletime in seconds, to easily enter minutes you can use the 4DScript word `mins` which converts minutes to seconds. For example, to have a cycletime of exactly 10 minutes for each product you should enter the following:

A first start in 4DScript

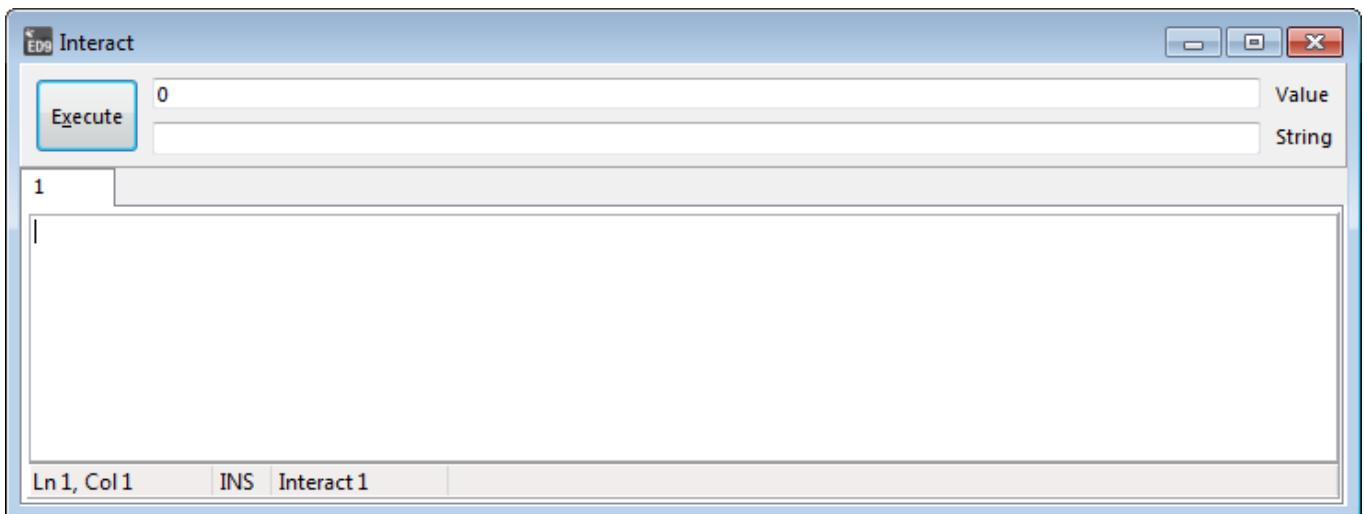
Cycletime: deterministic (minutes)

```
mins(10)
```

Until now you have seen two types of fields in which you can enter 4DScript code: the triggers and the cycletime. Using the predefined codes we have changed the visualization of the products during the run and we have changed the cycletime in the Server. By altering the predefined code and eventually writing your own 4DScript code you will have a lot of control on your model. For example, you will be able to change the routing of the products or write data to a table.

3.2 Experimenting with the Interact window

The 4DScript interact window can be used to run 4DScript code outside of the model. It can be opened from the tool tab of the main menu. Click 4DScript Interact on the *windows* group of the Tools tab, the interact window appears, see Picture 3.2.



Picture 3.2: 4DScript Interact window

It allows you to type code and run it immediately to see the result. This can be very helpful when you are learning the 4DScript language. Using the 4DScript Interact window you are able to get or set information during modeling or a simulation run.

The bottom side of the window contains a 4DScript editor similar to the editor that appears when you edit a property field marked with 4DS in your model. The button Execute (top left) is used to execute the 4DScript expression that is entered in the editor. The return value of the 4DScript expression is displayed in the value or string box.

Note that you can use the interact window as a calculator. Open the 4DScript interact. Begin with a simple calculation. Enter the following 4DScript statement and press Execute:

Arithmetic

```
3 + 4
```

Note that the result of the sum appears on the left side of the value box.

You can also run code that has a side effect besides displaying the return value in the value and or string box. An example is to open a message box.

Open a message box

```
Msg([Hello world!])
```

Replace the `3 + 4` with the above code and press execute. A message box will appear containing the text 'Hello World!'. Click OK to close the message box.

Besides using the interact window as a calculator or to execute code to open a message box you can also use the interact to get information during a simulation run.

Build or use the simple single server system that we used in the beginning of this chapter, see Picture 3.1. Start the simulation and make sure that it runs slowly. Remove the message box code and type in the following code; do not click Execute yet:

Getting information from the model via the 4DScript interact window 1

```
Output (AnimAtom)
```

Before you run this code make sure that you select the Queue atom in your model. While the model is running slowly press Execute several times. Note that the value in the *value box* changes each time that you press Execute. Because the model is running each time you press execute more products have left the Queue atom. Select the Source and repeat the procedure. Note that the return value is equal to the value displayed by the Source.


Do the same with the following code

Getting information from the model via the 4DScript interact window 2

```
Input (AnimAtom)
```

If you now press execute the value in the value box will change and return how many products at that specific moment have entered the selected atom.


`AnimAtom` is an example of an atom reference. An atom reference is used to indicate to ED which atom should be addressed. In a later chapter referencing will be explained in more detail.


 Use AnimAtom to refer to the selected atom in the active model layout. You will typically use this code in the interact window and not in your model.


3.3 Getting help

In the previous two sections you have seen some examples of 4DScript code, either used in the model or run from the 4DScript Interact window.

Many of these 4DScript words are functions that have parameters that you can edit. When you are editing predefined code or if you are writing your own code you can easily get information about the parameters of 4DScript functions. Short descriptions of the parameters and the return value of a function can be found in the 4DScript function list or in the Help file.

 To open the 4DScript function list either:

- In the 4DScript editor place the cursor anywhere in the 4DScript word and press F2. If the list is behind the editor window just shrink it or move it aside. Note that the 4DScript list is opened and displays the information about the selected 4DScript word.
- Press the 4DScript list button in the group window of the Tools tab of the main menu 

 More detailed information can be found in the Help file. To open the Help:

- In the 4DScript editor place the cursor anywhere in the 4DScript word and press F1.
- Press Help on the Help tab of the main menu.

4 Basics of 4DScript

4.1 Basic 4DScript

4DScript is the programming language of Enterprise Dynamics. It can be used in many of the property fields in the model. Recall that these fields are marked with the letters 4DS or contain predefined logics. To be able to understand and be able to alter the predefined codes that come with these fields and eventually write your own code it is useful to have some knowledge of the 4DScript programming language. In this section we will start with the syntax rules and explain the basic 4DScript words illustrated with examples.

Basic structure of 4DScript

The basic syntax of 4DScript is simple and is valid at any position where you can write 4DScript. The language contains a number of words. Examples that we have seen are `mins`, `uniform`, `icon` and `input`. These 4DScript words can have parameters. The parameters are placed between parenthesis () and separated by commas. For example, in the previous section we have seen that `mins` has one parameter, `mins(3)` converts the value three into seconds and that `uniform` generates random values from a uniform distribution has two parameters `uniform(2, 10)` to indicate the minimum and maximum value of the distribution.

Arithmetic operators

The most common arithmetic operations that you will use in the 4DScript language are:

| | |
|-----|----------------|
| + | addition |
| - | subtraction |
| * | multiplication |
| / | division |
| Min | minimum |
| Max | maximum |
| Mod | modulo |

Operations of addition, subtraction, multiplication and division correspond precisely with their respective mathematical operators. The other three are the 4DScript word `Min` that needs at least two parameters and returns the lowest one, the 4DScript word `Max` that needs at least two parameters and returns the highest value and the 4DScript word `Mod` that is the modulo. Modulo is the operation that gives the remainder of a division of two values.

For example, write the following code in the interact ('4DScript Interact' in the on-line documentation) and press Execute:

```
minimum  
Min(8, 3)
```

The minimum function will return the value 3, since 3 is the maximum value of 8 and 3.

For example, write the following code in the interact and press Execute:

```
modulo  
Mod(11, 3)
```


The modulo function will return the value 2, since 2 is the remainder from dividing 11 by 3.

Logical operators

We may want to compare two expressions, for example, to know if they are equal or if one is greater than the other. Here is a list of the relational and equality operators that can be used in 4DScript:

| | |
|----|--------------|
| = | equal to |
| <> | not equal to |
| > | greater than |

| | |
|----|--------------------------|
| < | less than |
| <= | greater than or equal to |
| >= | less than or equal to |

 The above operators can be applied to values. If you need to compare text you cannot use these. To check if two strings are equal use the 4DScript word CompareText, see the section about Strings.

Here are some examples that you can run from the 4DScript interact:

For example, write the following code in the interact and press Execute:

```
equal to 1
2 = 3
```

The test will return the value 0 (false), since 2 and 3 are not equal.

```
equal to 2
3 = 3
```

The test will return the value 1 (true), since 3 and 3 are equal.

For example, write the following code in the interact and press Execute:

```
greater or equal
5 >= 5
```

The test will return the value 1 (true), since 5 is greater than or equal to 5.

Other useful logical operators in the 4DScript language are:

| | |
|-----------------|---|
| And(e1, e2, ..) | test if all parameters are true i.e., equal to 1 |
| Or(e1, e2, ..) | test if at least one of the parameters is true i.e., equal to 1 |

These functions return 0 (false) if the test fails otherwise the result is 1 (true).

Here are some examples that you can try from the 4DScript interact window. Write the following code in the interact and press Execute:

```
And
And(5 = 5, 3 > 6)
```

The test will return the value 0 (false), since 3 is not greater than 6.

```
Or
Or(5 = 5, 3 > 6)
```

The test will return the value 1 (true), since the first parameter is true; 5 and 5 are equal.

Time

There are three common used 4DScript functions related to time. These are

| | |
|----------|---|
| hr(e1) | converts the specified number of hours into seconds (multiplies by 3600 = 60 x 60). |
| mins(e1) | converts the specified number of minutes into seconds (multiplies by 60) |
| Time | returns the current time (in seconds) in the simulation run |

Here are some examples that you can try from the 4DScript interact window. For example, write the following code in the interact and press Execute:

```
convert minutes to seconds
mins(2)
```

A first start in 4DScript

The `mins` function will return the value 120, since 2 minutes are equal to 120 seconds.

convert hours to seconds

```
hr(1)
```

The `hr` function will return the value 3600 since 1 hour is equal to 3600 seconds.

For the next example make sure that you have a simple Enterprise Dynamics model running slowly. Write the following code in the interact and press Execute several times while the model is running:

Get the current time in the simulation

```
Time
```

Note that the 4DScript word `time` will return the current time in the simulation in seconds. You can stop the simulation run and execute the code one more time to easily compare the current time in the simulation displayed by the clock with the return value in the value box of the 4DScript interact window. To make it easier to compare the return value with the clock time you can also replace the statement `Time` with `Time/60` or `Time/3600`.

Conditional statements

The 4DScript word `if` is used to execute a piece of code only if a condition is fulfilled. Its form is `if(condition, statement)`, where `condition` is the expression that is being evaluated. If this condition is true, the statement is executed. If it is false, the statement is ignored (not executed). The `if(condition, statement)` as a whole returns whatever the statement returned, or 0 if the condition was false.

Here is an example

if statement

```
if(Time > 3600, 1)
```

The above code will execute parameter two of the `if` function and return the value 1 if the current time in the simulation is greater than 3600 seconds. Otherwise it will return 0.

We can additionally specify what we want to happen if the condition is not fulfilled by using the optional third parameter of the `if` function. Its form is `if(condition, statement1, statement2)`, where `statement 1` is executed if the condition is true and `statement2` is executed if the condition is false.

Here is an example

if else statement

```
if(Time > 3600, 1, 2)
```

The above code will execute parameter two of the `if` function and return the value 1 if the current time in the simulation is greater than 3600 seconds. If it is false then the third parameter is executed and the above code will return the value 2.

Assignment (:=)

To assign a value to a property in most cases you can use the assignment operator `:=`. An example that we have seen is the predefined logic to change the icon of a product.

Build a simple model. Run it and stop the model such that you can select a product atom in your model. From the interact you can now run the following code:

Assignment

```
Icon(AnimAtom) := 24
```

Multiple statements

If we want more than a single statement to be executed from the 4DScript Interact, an edit field or from within an `if`-statement we need to put these statements as parameters of the 4DScript word `Do`. The `do` statement will return the result of the last expression.

Here are some examples that can be run from the 4DScript Interact window.

do statement

```
Do (msg ([A], msg ([B]))
```

if statement

```
if (
  2 > 1,
  Do (
    msg ([A]),
    msg ([B])
  )
)
```

Comments

Comments are parts of the code that simply do nothing. Their purpose is only to allow the programmer to insert notes or descriptions within the code. In the 4DScript language a comment is placed between curly brackets {}.

4.2 Control structures

Iteration

Loops have as purpose to repeat a statement a certain number of times or while a condition is fulfilled. Some common loop function in 4DScript are

- Repeat(e1,e2)** Repeats statement e2 for the by e1 specified number of times.
The 4DScript word **Count** is a counter that is incremented each time e2 is executed. During the first time that the statement is executed **Count** will have the value 1.
- While(e1, e2)** The while loop repeats statement e2 while the condition e1 is true.
- LoopUntil(e1,e2)** Executes the specified 4DScript expression e2 until expression e1 is true.

Below follow some examples how to use these loop functions.

The **Repeat** function repeats a statement for a specified number of times. Note that the code that is passed as second parameter is run twice if you run the following example from the interact window.

Repeat loop

```
Repeat ( 2, Msg (String (Count)) )
```

Note the use of the 4DScript word **Count**. This is a counter that can be used in combination with several loop functions. During the first execution of the statement, **Count** will be equal to one. Before the second execution of statement e2, **Count** is incremented and has a value 2.

If multiple statements have to be evaluated in your loop then the statements can be passed using the 4DScript function **Do** as we have seen earlier with the if-statement.

Repeat loop with multiple statements

```
Repeat (
  5,
  Do (
    Trace ([Message], Count ),
    Msg (String (Count))
  )
)
```

A first start in 4DScript

The while loop repeats statement e2 while the condition is true. Run the following script from the interact window.

The While-loop is run 4 times

```
Do (
  var([valCounter], vbValue, 1),

  While(valCounter < 4, Inc(valCounter)),

  valCounter
)
```

Because the variable is repeated as a last statement of the Do function the return value of this code will be the value of the variable `valCounter`. In the above example the return value will be 4.

The `LoopUntil` function executes the specified expression until the first expression is true. Run the following example from the interact.

Nine times display the string Test with the value of count in the tracer

```
LoopUntil(Count = 10, Trace([Test], Count))
```

The results will be written to the tracer window. Note that again the loop function can be used in combination with the 4DScript counter named `Count`.

4.3 Strings

Parameters of 4DScript functions can be values but they can also be text. Examples are `Msg` to display a message box containing the text that was passed as a first parameter, or the function `Name` that can be used to set the name of an atom where the name itself is passed as a text. These text type parameters are called strings and in code a text has to be placed between square brackets `[]`.

String

```
[This is a string]
```

The function to display a messages box requires one string parameter.


Display a message box containing the text "Hello World!"

```
Msg([Hello World!])
```

The text that should be displayed is between square brackets `[]`.

Here follows a list of some useful 4DScript functions when dealing with strings:

| | |
|-----------------------------|---|
| <code>CompareText</code> | This function compares two strings. The comparison is not case sensitive. |
| <code>StringLength</code> | This function determines the length of a string. |
| <code>Concat</code> | Concatenates two or more strings into one. |
| <code>String</code> | This function converts a value to a string. |
| <code>StringPos</code> | This function allows you to search for a specified sub-string within another string. It returns the position where the sub-string starts, starting at position 1. |
| <code>SubStringCount</code> | This function searches for the number of times a certain sub-string is present within a given string. |

 Note that you cannot compare strings with the = operator but that you need the 4DScript function `CompareText`.

If you need to compare a string to several strings the 4DScript word `Inlist` can also be useful. This function compares the first parameter with each of the other parameters and returns the index with respect to the list of other parameters if it finds a match.

Comparing several strings

```
inlist([d], [a],[b],[c],[d],[e])
```

The above code will return 4.

5 Enterprise Dynamics scripting

5.1 Probability distribution

Probability distributions form the engine of every stochastic model. Here we assume you are familiar with (the concept of) discrete and continuous probability distributions and a few commonly used probability distributions. You can find them in every textbook on simulation or statistics.

Discrete probability distributions

Some common discrete probability distributions in 4DScript are

`Bernoulli(e1,e2,e3)` Generates a random value from a Bernoulli distribution, e1% probability of e2 else e3.

`DUniform(e1,e2)` Generates a random value from a discrete uniform distribution with minimum value e1 and maximum value e2.

Run the following examples several times from the interact window:

Draw a value from a Bernoulli distribution

```
Bernoulli(70, 5, 20)
```

In most cases (70%) the above code will return the value 5, otherwise it will return 20.

The following example models a throw of a dice.

Draw a value from a discrete uniform distribution

```
DUniform(1,6)
```

The above code will return the value 1, 2, 3, 4, 5 or 6. If you repeat this experiment a hundred times the average will be around 3.5.

Continuous probability distributions

Some common continuous probability distributions in 4DScript are

`NegExp(e1)` Generates a random value from a negative exponential distribution with mean e1.

`Uniform(e1,e2)` Generates a random value uniformly distributed between lower bound e1 and upper bound e2.

`Normal(e1,e2)` Generates a random value from a normal distribution with mean e1 and standard deviation e2

Run the following examples several times from the interact window.

Example 1 generates a random value uniformly distributed between 10 and 20 minutes. If you repeat this experiment a hundred times you will find 100 values everywhere between lower bound 10 minutes and upper bound 20 minutes without clustering.

Example 1

```
Uniform(Mins(10), Mins(20))
```

Example 2 generates a random value from the negative exponential value with a mean of half an hour. If you repeat this experiment a hundred times you will find 100 positive values. The mean will be around 30 minutes with most values below 30 minutes and a few large values. This distribution is often used to generate random arrivals.


Example 2

```
NegExp(hr(0.5))
```

Example 3 generates a random value from the negative exponential value with a mean of 120 seconds and a standard deviation of 10. If you repeat this experiment a hundred times you will find 100 values clustered around 120 with about 50 values under 120 and 50 values above 120. This distribution is often used to model manual labor times.

Example 3

```
Normal(120,10)
```

 There are more than 20 standard distributions in Enterprise Dynamics.

5.2 Atom statistics

A very important part of modeling is getting results from your model. When you are defining performance metrics in an experiment or setting the variable for a result atom then you can make use of the atom statistics functions. These can be found in the 4DScript **category Atom Status** ('4DScript Category Atom Status' in the **on-line documentation**). Besides using these function in performance metrics they can also be very useful in a *send to* logic, a trigger or any other edit field in your model. Here you will find some examples of the use of these functions.

Atom status functions

The following functions are common examples used to get the status of atoms:

| | |
|----------------|--|
| Input(e1) | Returns the input of atom e1. The input is defined as the number of atoms that have entered atom e1 until now. The opposite is the output of the atom. |
| Output(e1) | Returns the output of atom e1. The output is defined as the number of atoms that have exited atom e1 until now. The opposite is the input of the atom. |
| Content(e1) | Returns the contents (the number of atoms contained at the highest level) of atom e1. |
| AvgContent(e1) | Returns the average contents (the average number of atoms contained) of atom e1 where e1 is an atom reference. |
| MaxContent(e1) | Returns the maximum contents (the maximum number of atoms ever contained) of atom e1. |
| AvgStay(e1) | Returns the average staying time of atoms in atom e1 over the total runtime. |
| Age(e1) | Returns the age of atom e1. The age is the difference between current time and creation time. When a run is reset, the creation time of all atoms becomes 0. |

Example 1: Use 4DScript to write the `Age(i)` to a table on the *Trigger on Entry* of a Sink atom thus collecting the lead-time of your products through the model. The make sure that each entry is in a new row we can make use of another atom status function.

Trigger on entry of the Sink: write the lead-time to a table with alias MyTable

```
SetMyTable(Input(c), 1, Age(i))
```

Example 2: Monitoring the number of products waiting in queue. If you select the option *Content* in the list of option for the *variable to monitor* of a Generic Monitor atom then the 4DScript code behind it would be

Variable to monitor: content

```
Content(In(1, c))
```

The atom that is monitored should be connected to the first input channel of the Generic Monitor atom and `in(1,c)` returns a reference to that atom. For a further explanation see the section about referencing.

5.3 Atom references

A very important subject when scripting in Enterprise Dynamics is referencing. If you need to update or query a parameter from an atom in your model you need to reference that atom. For example, if you want to change the icon of a product in your model, you need to refer to the product of which you would like to change the

A first start in 4DScript

icon. The *cycletime* of a Server could depend on the type of product, again you need to refer to the product that is in the Server to check its type.

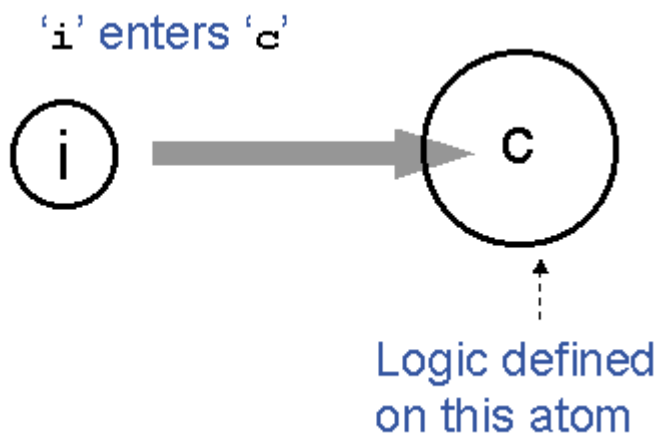
For example, in the `icon(i) := 23` code that we used in the exit trigger of a Queue atom, `i` returns an atom reference to the product that just left the Queue. Another example that we have seen is `AnimAtom`, which returns an atom reference to the atom selected in the active animation window. Atom references are a fast way to refer to atoms in your model.

Some examples of 4DScript words that take an atom reference as a parameter are

| | |
|-------------|--|
| Content(e1) | Returns the number of atoms contained in atom e1. |
| Input(e1) | Returns the number of atoms that entered atom e1. |
| Age(e1) | Returns the time (in seconds) that atom e1 existed in the model. Useful when you are interested in lead times. |

Getting a reference to an atom in your model

The two most common atom references in Enterprise Dynamics are the letters `i` and `c`, which are part of the 4DScript language. '`c`' refers to the *current* atom, this is the atom on which the statement is written. '`i`' refers to the *involved* atom, the one entering or leaving the current atom, see Picture 5.1.



Picture 5.1: The difference between the current and the involved atom.

For example, the *Trigger on entry* is a property of a Server atom, code written in this field is executed each time a product enters the Server. In this trigger we can use `i` to refer to the involved product; the product that just entered the Server and use `c` to refer to the Server itself. If we want to change the icon of all products that enter the Server then we can write code on the entry trigger of that server.

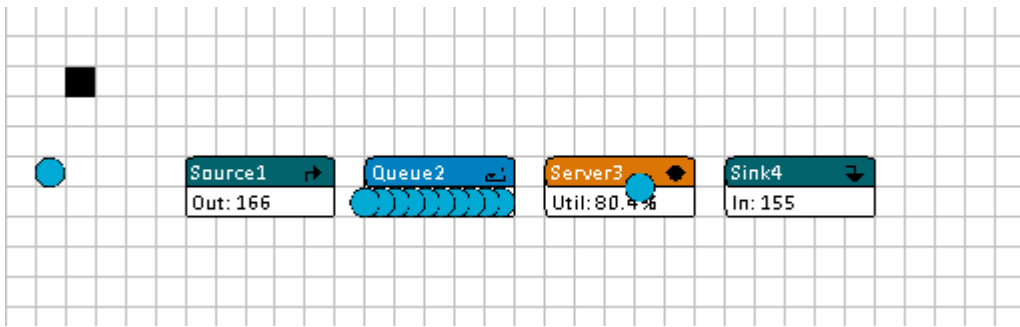
Trigger on Entry: Changing the icon of the product

```
Icon(i) := 23
```

Each time a product enters the Server, the entry trigger code is executed and `i` refers to the involved product. The involved product is the product that just caused the entry trigger to be executed.

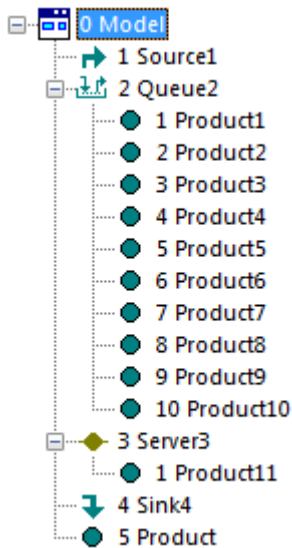
Understanding atom referencing

To understand the concept of referencing we use a simple single server system, see Picture 5.2.



Picture 5.2: Single Server system


In Picture 5.3 you see the model tree of this single server system, frozen at the same moment in time.



Picture 5.3: Model tree for the Single Server system

On the same level in the model tree we find Source1, Queue2, Server3, Sink4 and the Product atom. The product on this level is the product atom connected to the input channel of the Source. Queue2 contains 10 products, named Product1 to Product10. They form a second level, compared to the Product, Source1, Queue2, Server3 and Sink4. Of course Product1 and, let's say, Product5 are on the same level. Server3 contains 1 product, named Product11. The highest level is the model itself. In Picture 5.3 you can see this in the model tree!

Can you predict how the model tree changes if you reset this model?

 The model tree is not updated automatically. Click the model tree to make this window active and press F5 to refresh the window.

If you understand the hierarchy in this model tree, you will understand referencing...

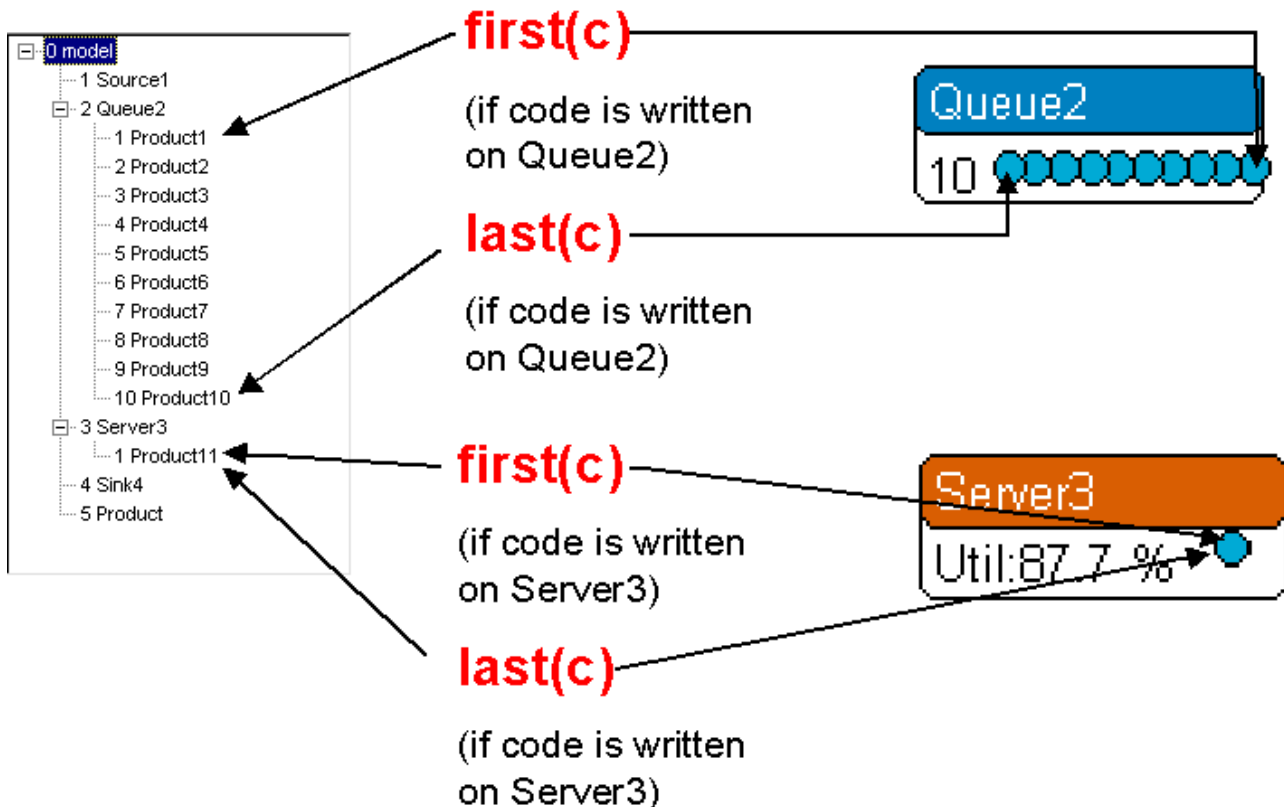
The following list of relative atom references are available to refer to an atom one level up or one level down in the hierarchy:

- First(e1) Returns a reference to the first atom contained in atom e2.
- Last(e1) Returns a reference to the last atom contained in atom e2.
- Rank(e1,e2) Returns a reference to the atom at position e1 within atom e2.
- Up(e1) Returns a reference to the atom in which atom e1 is situated.

To get a better understanding how to use these relative atom references take a look at Picture 5.4. For example, if code is written on the *Send to*, the *Queue discipline*, a trigger or any other fields of Queue2 in which 4DScript can be entered the statement `First(c)` returns a reference to Product1. While `Last(c)` refers to Product10.

Do you understand that on Server3 `First(c)` and `Last(c)` refer to the same product?

A first start in 4DScript



Picture 5.4: Referencing using the relative references First and Last.

The following global references can be used:

- Model** Returns a reference to the model atom. The model atom contains all the atoms in your simulation model.
- Library** Returns a reference to the Library atom. The Library atom is the atom that contains the pre-defined atoms with which you create a simulation model.

You can run the following 4DScript statement from the interact window:

Returns a reference to the third atom in your model

```
Rank(3, Model)
```

If we would run the above code when we have the single server system open in ED then this will return a reference to Server3. We could pass this reference to the function `Name`.

Returns the name of third atom in your model

```
Name(Rank(3, Model))
```

Running the above script from the interact would return the string Server3. We can also change the name if we run the following script from the interact:

Changes the name of third atom in your model

```
Name(Rank(3, Model)) := [The new name of the Server]
```

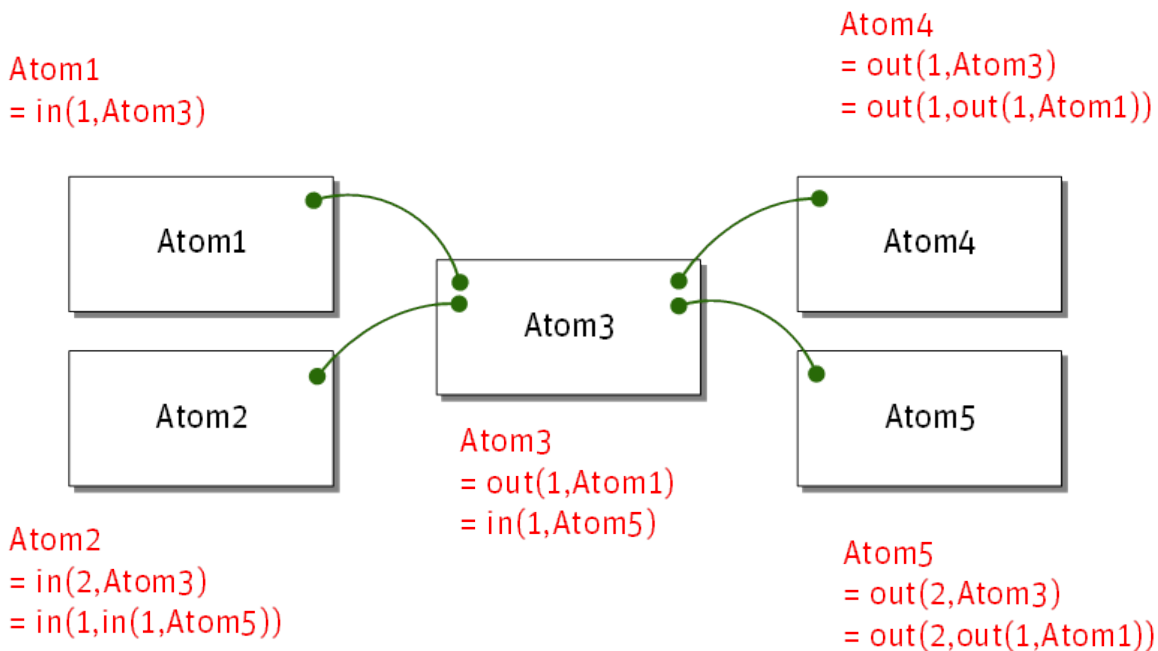
The name of the third atom in your model would change to "The new name of the Server".

There are a few commands that are often used for flow control and related to channels:

- `in(e1,e2)` Refers to the atom connected to input channel e1 of atom e2.
- `out(e1,e2)` Refers to the atom connected to output channel e1 of atom e2.

Take a look at Picture 5.5 regarding the use of `in` and `out`. For example if `in(1,c)` is written in a field of

Atom3 then `c` refers to Atom3 itself and `in(1,c)` refers to Atom1, but the same statement written on Atom4 refers to Atom3!



Picture 5.5: Referencing with in and out.

Using these relative references via the channels you can create all kind of models. For example, a system in which a second Server only processes products if there are more than four products in the Queue before it. To build such a model you need to open and close the input of the second Server connected to the Queue. You would need a statement in the triggers of the Queue such as `CloseInput(out(2,c))`. This will close the input of the atom connected to second output channel of the current atom.

Another example, the statement `Content(in(3,c))` returns the content (number of atoms) contained in the atom which is connected to input channel 3 of the current atom.

You can also refer to atoms by their name. To do that you can use the 4DScript function `AtomByName`:

`AtomByName(e1,e2)` Returns a reference to the atom named `e1` contained in atom `e2`.

This function should not be used in trigger fields or other fields that are executed often during a simulation run because it will slow down the simulation speed. However it can be an easy way to refer to atoms in for example an Initialize atom which is only executed ones in a simulation run.

Here follows an example

Refer to an atom by its name

```
AtomByName([Source1], Model)
```

The above code returns the reference to Source1.

5.4 Labels

The use of labels is a way to store information on atoms. Labels are similar to local (temporary) variables. Differences are that variables need to be declared and labels do not and that labels are attached to an atom and variables are not.

Labels are mostly attached to product atoms and can represent for example a weight, a customer number, a production time, etcetera. They can be queried using the 4DScript word `Label` and updated with the `:=` operator. Labels do not have to be declared, at the moment they are referenced they exist. You can assign as many labels as you want to an atom and it is possible to store values and strings in a label.


A first start in 4DScript

Query a label

Because you can assign multiple labels to one atom you need both the name of the label and a reference to the atom to which it is assigned to be able to query its value.

`Label(e1, e2 {, e3})` Returns the contents of label named `e1` defined on atom `e2`. If `e3` is not specified or 0 then the result is a value if the content can be converted to a value otherwise the result is a string. If `e3=1`, the result is always a value. If the contents cannot be converted to a value the value is 0. If `e3=2`, the result is always a string.

The Label name `e1` is always a string and is case insensitive.

 If you use long names and many labels you loose speed.

The following code returns the value of the label named `type` on the involved atom `i`:

Query a label

```
Label([type], i)
```

Update a label

We can easily assign labels to products using a trigger field. Typically you can assign a label in the *Trigger on creation* of a Source atom. Every time a product is created by the Source this field is executed. At that moment a label can be assigned.

Trigger on creation Source: Assign a label named Weight containing the value 9

```
Label([Weight], i) := 9
```

In the above example each product is assigned a label named `Weight` containing the value 9. If we replace the 9 with the expression `Uniform(8,11)`, i.e., on the creation trigger of the Source write:

Trigger on creation Source: Set the value of the label named Weight

```
Label([Weight], i) := Uniform(8,11)
```

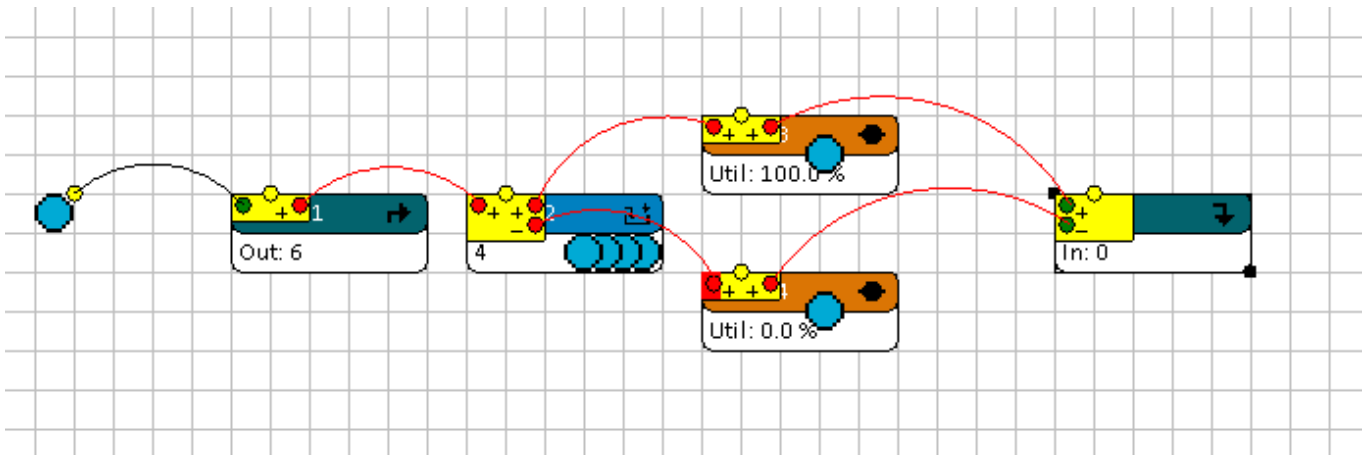
All product created by this Source will get a label named `Weight` but the value of the label will differ for each product.

5.5 Flow control

Besides certain atoms that can help with flow control there are also several 4DScript functions that can be very useful these are

- `CloseInput(e1)` Closes the general input of atom `e1`. Products are not allowed to enter atom `e1`.
- `CloseOutput(e1)` Closes the general output of atom `e1`. Products are not allowed to exit atom `e1`.
- `OpenInput(e1)` Opens the general input of atom `e1`. Products are allowed to enter atom `e1`.
- `OpenOutput(e1)` Opens the general output of atom `e1`. Products are allowed to exit atom `e1`.

If you open and close the general input and output a green or red bar will be shown over the channels to indicate this. In Picture 5.6, the general input of the bottom Server is closed. From the moment the general input is closed products are not allowed to enter the Server. If a product is already in the Server then this products will be served and is allowed to exit.



Picture 5.6: The general input of the bottom Server is closed.

You can use the flow control functions in trigger fields to open and close the general input and output of atoms in your model. For example, you can use this to model a pull system or model a situation such as in picture 5.6 where the second Server is only used if there is a Queue longer than four products.

To close the general input of Server3 you could execute the following code:

Close the general input of Server3

```
CloseInput (AtomByName ([Server3], Model))
```

If you write the above code in a field of the Server itself it is recommended using the reference `c`, i.e., `CloseInput (c)`. If the code is written on the Queue of which the second output channel is connected to the first input channel of the Server then use `CloseInput (out (2,c))`.

5.6 Examples code in standard fields

There are several fields that you can find on many of the atoms. Examples are the *send to* field, the *input strategy* and the *trigger* fields. Here follow some example codes that you can use in these fields. Of course you can also examine the code of the predefined logics of these fields.

Trigger fields

There are several types of trigger fields. For example the Source has a *Trigger on creation* and a *Trigger on exit*. The Queue and the Server both have *Trigger on entry* and *Trigger on exit* fields. The Server also has a *Trigger on end of setup*, *Trigger on breakdown* and a *Trigger on repair*. By default no code is executed in these fields but these triggers allow you to make something happen at the moment the trigger is executed. For example, when a product is created by the Source you can assign a label to it, or when a product enters a Server you can call a Human Resource to assist with the process or when a product exits a Queue you can write the waiting time to a table. These are just a few examples of all the possibilities.

Below follow several examples codes that can be written in trigger fields.

Trigger on entry or exit: change three things of the involved atom

```
Do (
  Color(i) := ColorRed,
  Icon(i) := 24,
  Label([temperature], i) := Uniform(20,40)
)
```

Three things are done with the involved atom (mostly a product): the color is set to red, the icon is changed to icon number twenty four and a label with the name *temperature* is stamped on the product, it's value is chosen randomly between 20 and 40.

The next example shows how you can do flow control, this code can for example be placed in the Trigger on entry of a Queue.

A first start in 4DScript

Trigger on Entry: flow control

```
if(Content(c) > 10, CloseInput(in(1,c))
```

If the content of the current atom is more than 10 then close the atom connected to the first input channel.

The following example shows how to assign a label to a product. You can use this code in the *Trigger on Creation* but also in a *Trigger on entry* or *Trigger on exit*.

Trigger on Creation: Assign labels

```
Label([type], i) := Bernoulli(80,1,2)
```

If the above code is written on the creation trigger of the Source, then each product that is created by this Source will get a label named *type*. For 80% of the products the value stored in this label will be 1 the others 2. We can use this to distinguish the two product types and give them different cycle times or a different routing through the system.

Send to field

If you write code in a *Send to* field then it is important that the return value of the code is the output channel number through which you want to send the product. The simplest code that you can write in this field is the statement:

Send to: channel number

```
2
```


If you write the above code in a *Send to* field then all product will be send to the atom connected to the second output channel of this atom.

The routing of products through the model can also be based on a label value assigned to the product. You can write the following code in the *Send to* field:

Send to: By label value

```
Label([dest], First(c))
```

If you have assigned a label to the products in your model (for instance on the trigger on creation or exit of the Source). You can use the values stored in the label to decide through which channel the product should leave the atom. To read the label of the product that is ready to leave the atom we use the atom reference `First(c)`.

 Typically you will use the reference `First(c)` to refer to the product on a *send to* field.


Cycletime field

The return value of the code in a *cycletime* field should be the cycletime in seconds. Here an example where the cycle time is based on the value of a label assigned to the product.

Cycletime: by label value

```
Label([cycletime], Last(c))
```

If you have assigned a Label named *cycletime* to the products in your model (for instance on the trigger on creation or exit of the Source). You can use the values stored in the label as cycle time. To read the label of the product that just entered and is ready to determine its cycle time we use the atom reference `Last(c)`. This models a situation in which you know beforehand how long the processing will take. You can also use the values in these labels to apply a *queue discipline* in a Queue atom to give short jobs a priority.

 Typically you will use the reference `Last(c)` to refer to the product on a *cycletime* field.

The cycle time is most of the time based on a probability distribution. If the cycletime should be based on a normal distribution you could write:

Cycletime: A normal distribution

```
Max(0, Normal(120,10))
```

The normal distribution can also be negative. To make sure this does not occur in a cycletime for a server we could use the above code taking the maximum of the value 0 and the result of a draw from the normal distribution.

A *cycletime* can also depend on the time in the simulation.


Cycletime: depends on the time in the simulation run

```
if (
  Time > 3600,
  NegExp(10),
  NegExp(15)
)
```

If time > 3600 seconds then the cycle time is `NegExp(10)` otherwise `NegExp(15)`.

5.7 Debugging

The **Tracer window** ([on-line documentation](#)) allows you to display messages. This can be very useful when you are debugging your model. Using the 4DScript function `Trace` you can write messages to the tracer window.

 The Tracer window does not need to be visible. To display the Tracer window use **DisplayTracer** ([on-line documentation](#)) or click the tracer button that can be found on the Tools tab of the main menu.

Useful 4DScript functions for debugging are:

| | |
|---------------------------------|--|
| <code>Trace(e1, {e2,..})</code> | This function adds a message to the Tracer window. |
| <code>Msg(e1)</code> | Displays a message box with the text e1. |