Annex 3 A first start in 4DScript

4DScript is the programming language of Enterprise Dynamics. Everything that is executed in Enterprise Dynamics is or can be done via 4DScript.

This document provides for beginners the structure of 4DScript and a list of often used commands, illustrated with examples.

We start with the syntax rules together with the mathematical and logical operators. Second, the concept of referencing is illustrated with examples. The third part consists of the most common commands in ED.

1. The basics of 4DScript

The basic syntax of 4DScript is simple and is valid at any position where you can write 4DScript:

1 the language contains a number of words (commands);
2 these 4DScript commands can have (up to 255) parameters;
3 the parameters are placed between parenthesis ( );
4 parameters can be *values*, *strings* or *expressions* (other 4DScript words);
5 parameters will always be separated by commas;
6 if a parameter should be interpreted as a string, the string parameters are always placed between square brackets [ ]. If the parameter should be executed as 4DScript code, the parameter is written in a normal fashion;
7 comments can be placed between squiggly brackets { }.

These rules apply everywhere and are always valid. However, for some statements these rules do not result in very readable code. In order to make parts of the code easier to understand for others, the 4DScript syntax makes an exception in some cases. This applies in particular to mathematical and logical symbols.

Mathematical symbols

Consider the following valid statements:

A more natural way:
+(12,7) results in 19
12+7
+(/(100,10),6) results in 16 100/10+6 or 6+100/10

All the special mathematical operators are evaluated in the following order:

*	=	multiplication
/	=	division
+	=	addition
-	=	subtraction

The normal priority rules for these operators apply. In case you have doubts, use parenthesis ()!

Logical symbols

For logical operators (like >, < , =) it is also possible to disregard the rule that the 4DScript command needs to be written first and then the parameters

Consider the following valid statement:

>(10,6)	can also be written in the `normal' fashion as:	10>6
<=(23,12)	can also be written as:	23<=12

This example can be extended to more 4DScript logical symbols:

=	=	equal
>	=	larger than
<	=	smaller than
>=	=	larger than or equal to
<=	=	smaller than or equal to
<>	=	not equal

and
syntax: and(e1,e2)

Returns 1 if e1 and e2 are true (1), returns 0 if not. If e1 is not true, then e2 is not evaluated. Instead of writing and(e1,e2) the user can also write e1 **and** e2See also **or**

So, both commands are used as usual…

max
Syntax: max(e1,e2)

Returns the maximum of e1 and e2. See also **min**

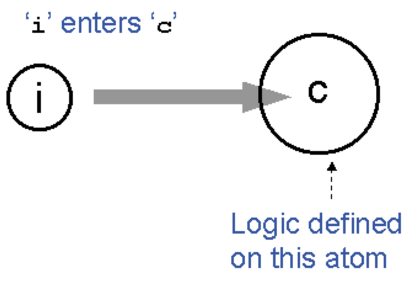
We end this chapter by a few commands on time and time conversion. Remember that ED `thinks' in seconds, but it is easy to convert time:

time	returns the current model time <u>in seconds</u>
mins(e1)	returns e1 (minutes) in seconds: multiplies e1 with 60.
hr(e1)	returns e1 (hours) in seconds: multiplies e1 with 3600.

2. Referencing

A very important subject in ED is referencing: if we are talking about the `next' atom, what is our viewpoint? It gets more complicated if there are atoms contained in other atoms like products in a queue.

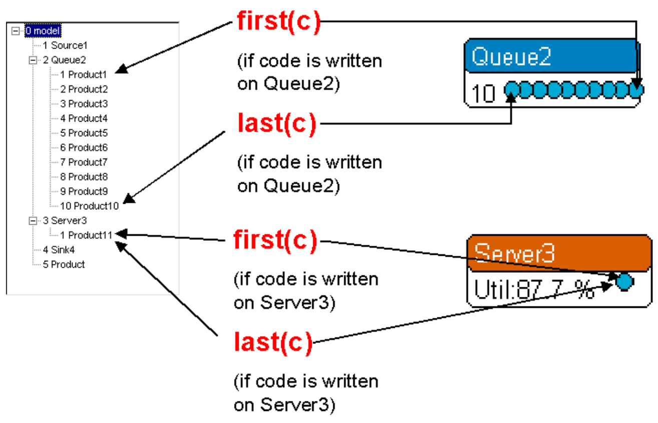
We have seen the letters c and i in different statements, now we will explain them:
`c' refers to the <u>c</u>urrent atom where the statement is written on
`i' refers to the <u>**i**nvolved</u> atom, the one entering or leaving the current atom



Picture 1: References i and c

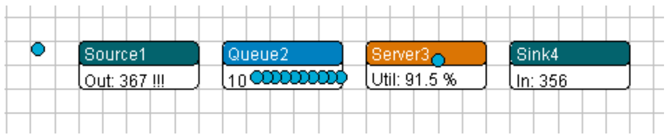
To keep it simple: the `i' is used only in statements written on Trigger on Entry or Trigger on Exit!

The key to understanding the concept of referencing is the next example:



Picture 2: Model Tree of a simple Queueing System. Explaining First and Last atom references

On the left of picture 2 you see the Model Tree of a simple queuing system, frozen at one moment in time:



Picture 3: Model of a simple queueing system

On the same level we find the Source, the Queue, the Server and the Sink and the Product atom placed before the Source (number 5 in the model tree, because it is mostly added after you reset your model the first after making the lay-out).

Queue2 contains 10 products, named Product1 to Product 10. They form a second level, compared to the Product, Source, Queue, Server and Sink. Of course Product1 and, lets say, Product5 are on the same level. Server3 contains 1 product, named Product11.

The highest level is the model itself. In picture 1 you can see this in the model tree! Can you predict how the model tree changes if you reset this model?

If you understand the hierarchy in this model, you will understand referencing…

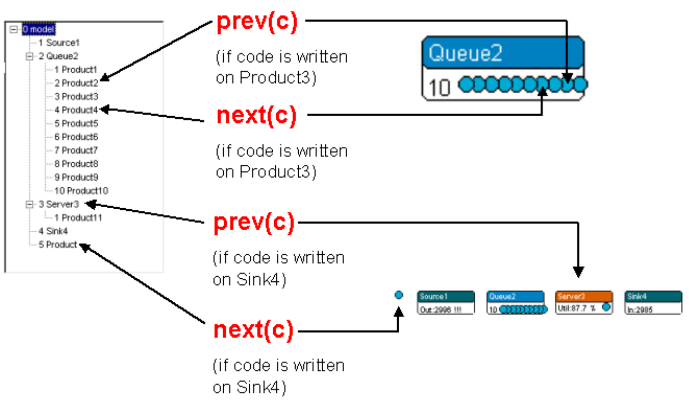
First a few commands which are often used for referencing:

first(e1)	refers to the first atom inside atom e1
last(e1)	refers to the last atom inside atom e1
next(e1)	refers to the next atom on the same level as atom e1
prev(e1)	refers to the atom in front on the same level atom e1

So first(c) refers to the first atom inside the current. If the current is a Queue it could be the first product in the queue. Now look at picture 1 again and the examples on *first* and *last.*

So, first and last are always looked at one level deeper from the atom where the statement is written on!

The commands next and previous are explained below:



Picture 4: Model Tree of a Queueing System. Explaining Prev and Next atom references

Now look at picture 3 and the examples on *prev* and *next.*

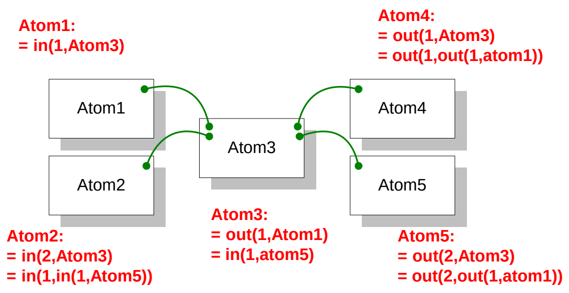
So, prev and next are always looked at the same level from the atom where the statement is written on!

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

out(e1,e2)	refers to the atom connected to output channel e1 of atom e2
in(e1,e2)	refers to the atom connected to input channel e1 of atom e2
OpenOutput(e1)	opens the general output of atom e1
CloseOutput(e1)	closes the general output of atom e1. If closed, the channels cannot be ready, regardless of individual channel settings.

In a similar way you can define **OpenInput** and **CloseInput.**

Take a look at picture 5 regarding the use of *in* and *out:*



Picture 5: Explaining in and out atom references

Example 1:
So refers *in(1,c)* (written on Atom3, the point of view) to Atom1, but the same statement written on Atom4 refers to Atom3!

Example 2:
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.

Example 3:
CloseOutput(in(2,c)) closes the output of the atom connected to the second input channel of the current

Notice the way commands are nested…

3. Important commands

In every programming language there are important commands, which perform as some kind of building blocks.

3.1 For conditional statements: **if**

Syntax: if(e1,e2 {,e3})
Executes e2 if e1 is true (1); otherwise executes e3 (if specified). Returns result of e2 or e3.

Example 1:
if(time>3600, msg[more than an hour], msg[less than an hour])

Translated: if time>3600 seconds then give message `more than an hour' else give message `less than an hour'.

Example 2:
if(content(c)>10, closeinput(in(1,c))

Translated: if the content of the current atom is more than 10 then stop the input of the atom connected with the first incoming channel else do nothing

Don't worry about the used commands like msg, content or closeinput. We'll explain them later!

3.2 For executing more than one statement: **do**

Syntax: do(e1,e2,...,e25)
Executes e1, e2, etc. in order of sequence. Returns result of last expression. You can use up to 25 parameters. If more parameters are needed, several do loops can be nested.

Example:

```
        do(      set(color(i), colorred),set(icon(i) ,2),
        setlabel([temperature], uniform[20,40], i)
)
```

Three things are done at the **i**nvolved atom (mostly a product): the color is set to red, the icon is changed to iconnumber two and a label with the name *temperature* is stamped on the product, its value is chosen randomly between 20 and 40.

3.3 Labels

The use of labels is the way to store local (temporary) variables by users. They are mostly attached to products and can represent a weight, a customernumber, a productiontime etcetera. They are defined with the command *setlabel* and reproduced with *label.*

a. setlabel(e1,e2,e3)

Defines a label on atom e3 where e1 is the name and e2 is the new content of the label. Labels do not have to be created, at the moment they are referenced they exist. The *sddb* command does the same.

Example 1:
setlabel([Weight],10.24,i)

First select an atom and create (and give a value to) a label with the command setlabel. Suppose the name of the label is v1 and the value is 100:

Example 2:
setlabel([v1],100,animatom)

label([v1],animatom) now returns the specified value (100) of the selected atom.

b. label(e1,e2,{e3})

Returns the contents of label named e1 defined on atom e2. Labels do not have to be created, at the moment they are referenced they exist. <u>The label name e1 is always a string and is case sensitive.</u>

The result is a string or a value, depending on the contents and on e3: if e3 is not specified or e3=0 then the result is a value if the contents 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. If you use long names and many labels you lose speed.