Plant Simulation编程语言

SimTalk2.0官方说明

ReadMe

SimTalk

SimTalk

The programming language *SimTalk* extends the ways you can model and control your simulation. Each object has built-in properties providing many useful features. When your model requires more detailed or completely different properties, you will program these in the programming language *SimTalk*.

First, you will enter the statements, which the built-in Interpreter program executes, into the object Method . You can also combine the *Method* object with the material flow objects and the information flow objects to create models of great complexity. Then press the F7 and F5 keys to execute your source code.

When you then run the simulation, the Interpreter executes the source code, which you entered into your *Method* line-by-line and takes the actions you programmed.

SimTalk normally does not distinguish between upper- and lower-casing for the names of *methods* and *attributes* which you enter into the source code of your Method objects.

SimTalk 2.0 makes programming methods in *Plant Simulation* faster, easier, and less error-prone.

You can activate it by clicking New Syntax 

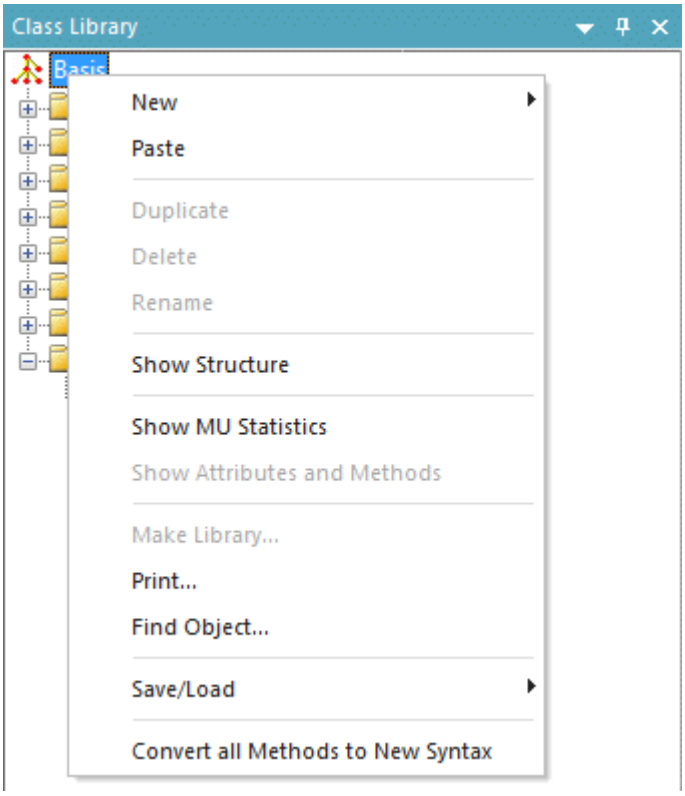
on the **Tools** ribbon tab of the *Method* you are programming. If you want to use *SimTalk* 2.0 syntax for all new *Methods* which you are going to program, activate **New Syntax** in the *Method class* in the *Class Library*. Changes caused by *SimTalk* 2.0 also affect how the dialog Show Attributes and Methods displays the signature of attributes and methods.

You can select if you want to use the**SimTalk 2.0 notation** or the **SimTalk 1.0 notation** for each and every of your *Methods*. You can also freely mix **SimTalk 2.0 notation** and **SimTalk 1.0 notation** in your simulation models. There is no need to reprogram your existing source code.

Clicking 

in an existing *Method* which you programmed in **SimTalk 1.0 notation** automatically converts the source code to the correct **SimTalk 2.0 notation**.

To convert the source code of all existing *Methods* in a simulation model to the new syntax, hold down the **Shift** key, click on the object **Basis** in the *Class Library* with the right mouse button and click Convert all Methods to New Syntax.



SimTalk 2.0 encompasses:

- A **line-controlled syntax**: You no longer need to type a semicolon (**;**) at the end of each statement. Instead, you can simply enter a single statement into a line. As you still might want to be able to split a statement and distribute it over several lines, the interpreter needs to automatically determine if the statement is incomplete and is being continued in the next line. In addition, you can enter a semicolon to add another statement in the same line.

SimTalk 2.0	SimTalk 1.0
`MySingleProc.setName("MyStation")`	`MySingleProc.setName("MyStation");`

- A **simplified body syntax**: In *SimTalk* 1.0 the source code requires the keywords **is do end**. *SimTalk* 2.0 does not need these keywords. In *SimTalk* 2.0 you will declare **parameters** using the keyword **param**, **local variables** using the keyword **var**, and the **return value** using the keyword **->** (hyphen plus right angle bracket).

SimTalk 2.0	SimTalk 1.0

`MySingleProc.setName("MyStation")`	`is`
	`do`
	`  MySingleProc.setName("MyStation");`
	`end`

- **Improved referencing of methods and global variables**: In *SimTalk* 1.0 *Methods* and *Variables* are referenced with the reference operator . In *SimTalk* 2.0 you reference *Methods* and *Variables* with a leading **& operator**, for example `var o:object := &Method`.

- **New div/mod operator**: In *SimTalk* 1.0 the *//* **operator** represents an integer division and the \\ **operator** represents an integer modulo operation. In *SimTalk* 2.0 the keyword **div** represents an **integer division** and the keyword **mod** represents an **integer modulo operation**.

- **Improved literals**: In *SimTalk* 2.0 a backslash (\) inside of a *string literal* will only quote double quotes and line breaks. The backslash will not quote another backslash any longer as in *SimTalk* 1.0.

- **Time literals**. A *time literal* starts with a digit and must contain one or more colons. A *time literal* can contain a decimal point and decimal places.

```
wait 1:30  -- wait for 1 minute and 30 seconds warten
&Methode.methCall(1:0:0:0.5) -- call the method in 1 day and half a second
```

- **Default arguments**: You can define default arguments for formal parameters by entering an assignment operator and the default value after the parameter declaration, for example: `param x := 0`.

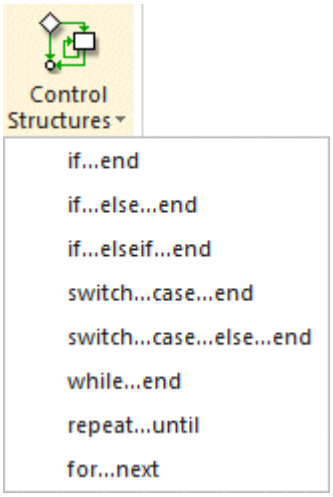
- **Simplified control flow statements**: The control structures which you can insert with the command Insert Control Structure into your source code, reflect this. You can now enter control flow statements using the simplified syntax:

- if-else-end instead of if-then-else-end

- for-next instead of for-to-loop-next

- switch-case-end instead of inspect-when-then-end

- while-end instead of while-loop-end



- The keyword **then** in **if-statements** is optionally allowed.

- The keyword **loop** in **loops** is optionally allowed.

- We added the keyword **continue**, which skips the rest of the loop iteration and continues with the next iteration. If the iteration was the last iteration, **continue** exits the loop.

- A **changed about equal operator**. You now have to use:
- `~=` instead of `==` , compare Tolerance for about equal (~=) comparison.
- `<~=` instead of `<==`
- `>~=` instead of `>==`
- **New operators for adding, subtracting, or multiplying** a value:
- `x += y` is short for x := x + y
- x `-= y` is short for x := x - y
- x `*= y` is short for x := x * y
- An **improved list and table syntax**: You now only enter list ranges using curly brackets `{}`. The optional 1.0 syntax using a grave accent `` `[] `` is no longer available.

You can read out an element of a **one-dimensional list** with the bracket operator `[]`. This works for *Stacks* and *Queues* as well. You can read and remove an element from a *CardFile* with the new built-in method remove.

When reading the contents of a cell of a *CardFile* with the **bracket operator [ ]** in *SimTalk* 1.0, the contents of the designated cell was read and removed. The remaining cells moved up by one position.

When assigning a value to the designated cell with the **bracket operator** to a *CardFile* in *SimTalk* 1.0, this value was inserted into the cell and the existing cells moved down by one position.

When reading the contents of a cell with the **bracket operator** in *SimTalk* 2.0, the contents of the designated cell will be read, but will remain in the *StackFile*, *QueueFile*, and the *CardFile*.

When assigning a value to the designated cell with the **bracket operator** to a *CardFile* in *SimTalk* 2.0, this value will just overwrite the contents of the existing cell. This way a *CardFile* will behave the same way as a *TableFile* with one column.

SimTalk 2.0 and SimTalk 1.0 Compared

The following table provides an overview of the changed language constructs between *SimTalk* 2.0 and *SimTalk* 1.0.

Feature	New Syntax (SimTalk 2.0)	Old Syntax (SimTalk 1.0)
Parameter Declaration	`param v1,v2: integer, name: string`	`(v1,v2:integer; name:string)`

Return Type Declaration	`->boolean`
	or
	`param v1,v2: integer, name: string -> boolean`
		`:boolean`
		or
		`(v1,v2:integer; name:string): boolean`

Empty Method	—	`is`
		`do`
		`end;`

Variable Declaration between is and do	`var v: integer`
	`var s: string`
		`is`
		`  v:integer;`
		`  s:string;`
		`do`
		`end;`

Variable Declaration	`var s: string`	`local s: string`

if condition	`if a > 3`
	`  print a`
	`end`
		`if a > 3 then`
		`  print a;`
		`end;`

for loop	`for var i := 1 to 10`
	`  print i`
	`next`
		`for local i := 1 to 10 loop`
		`  print i;`
		`next;`

while loop	`while a  10`
	`  a += 1`
	`  print a`
	`end`
		`while a  10 loop`
		`  a := a + 1;`
		`  print a;`
		`end;`

repeat loop	`repeat`
	`  a += 1`
	`  print a`
	`until a > 10`
		`repeat`
		`  a := a + 1;`
		`  print a;`
		`until a > 10;`

switch-/inspect-statement	`switch a`
	`  case 1`
	`  print 1`
	`  case 2`
	`  print 2`
	`end`
		`inspect a`
		`  when 1 then`
		`  print 1;`
		`  when 2 then`
		`  print 2;`
		`end;`

waituntil-statement	`waituntil name = "Test"`
	or
	`waituntil name = "Test" prio 1`
		`waituntil name = "Test" prio 1;`

CardFile operator [ ]	`Cardfile.remove(2)`	`Cardfile[2]`

The following table provides an overview of the changed operators between *SimTalk* 2.0 and *SimTalk* 1.0.

Feature	New Syntax (SimTalk 2.0)	Old Syntax (SimTalk 1.0)

About equal operator	`~=`
	`<~=`
	`>~=`
		`==`
		`<==`
		`>==`

Add a value
Subtract a value
Multiply a value
	x += y is short for `x := x + y`
	`x -= y` is short for `x := x - y`
	`x *= y` is short for `x := x * y`
		—

Modulo operator	`mod`	`\\`
Division operator	`div`	`//`

reference operator	`&`
	e.g. `.Models.Frame.&Method`
		`.ref()`
		e.g.
		`.ref(.Models.Frame.Method)`

You can activate **SimTalk 2.0 notation** by clicking New Syntax 

on the **Tools** ribbon tab of the *Method* you are programming. If you want to use if for all new *Methods* which you are going to program, activate **New Syntax** in the *Method class* in the *Class Library*. Instead, you can also use the attribute UsingNewSyntax.

Clicking 

in an existing *Method* which you programmed in **SimTalk 1.0 notation** automatically converts the source code to the correct **SimTalk 2.0 notation**.

A Quick Tour Through SimTalk 2.0

A Quick Tour Through SimTalk 2.0

Typically the first example for a programming language prints the words "Hello, World!". In *SimTalk*, you can accomplish this in a single line by entering the following into a Method:

```
print "Hello, World!"
```

This is a complete source code which you can run. You don't need to type in semicolons at the end of each statement as you need to do in C, in C++, or in JavaScript.

This tour provides you with enough information to start writing code in *SimTalk* by showing you how to approach a variety of programming tasks. Do not worry if you don't understand something, everything that we are going to introduce in this tour is explained in detail in the rest of the *Online Help*.

Below we describe:

- Simple Values
- Control Flow
- Default Arguments

Simple Values

Use **var** to create a variable:

```
var myVariable := 42
```

Note that a variable must have the same data type as the value you want to assign to it.

If the initial value does not provide enough information, or if there is no initial value, specify the data type by entering it in after the variable, separated by a colon:

```
var myDouble: real := 3.1415
```

You can create *arrays* using brackets [ ] and access their elements by entering the index within brackets.

```
var myIntegerArray: integer[]
myIntegerArray.append(3)
print myIntegerArray[1]
```

Control Flow

Use **if** and **switch** to create *conditionals*. Use **for**, **while**, and **repeat** to create **loops**. Parentheses around the *condition* or the *loop variable* are optional, meaning that you can enter them, but you do not have to do so.

```
var ages := MakeArray(34,42,18,44,53,12,63)
var countBelow30: integer
var countAbove29: integer
for var i := 1 to ages.dim
   if ages[i]  30
       countBelow30 := countBelow30 + 1
   else
       countAbove29 := countAbove29 + 1
   end
next

print "Below 30: ", countBelow30
print "Above 29: ", countAbove29
```

Use **switch** if you have to check for a big amount of different values:

```
var currentDay := 3
switch currentDay
case 1
   print "Monday"
case 2
   print "Tuesday"
case 3
   print "Wednesday"
```

```
case 4
   print "Thursday"
case 5
   print "Friday"
case 6
   print "Saturday"
case 7
   print "Sunday"
end
```

Use **while** to repeat a block of code until a condition changes.

```
var n := 2
while n  10
   n := n * 2
end
```

The condition of a *loop* can also be located at the end instead, ensuring that the loop is executed at least once using the *repeat loop*.

```
var n := 2
repeat
   n := n * 2
until n  10
```

Default Arguments

You can define default arguments in *SimTalk* 2.0. If the *Method* is then called without the argument, the associated parameter will be set to the default value, which is defined in the *Method*.

Example:
```
param x: integer := 123
```

The *Method* can be called with or without argument. If the *Method* is called without argument, the parameter `x` has the value 123 in the example above.

It is also possible to only define default arguments for a part of the parameters. When a parameter has a default argument, all following parameters also have to have a default argument.

Example:
```
param a: string,
b, c: string := "",
d: boolean := true
```

The *Method* in this example can be called with one to four arguments, but not without any argument at all, as no default argument was defined for the parameter `a`.

Possible calls are:

```
Method("A")
Method("A", "B")
Method("A", "B", "C")
Method("A", "B", "C", false)
```

Only constant values are allowed as default argument. You cannot define numbers, strings (string constants), true, false, void, and pi.

Parameters of data type *object*, *table*, *list*, *stack*, *queue*, or *any* can only have `void` as default argument. Parameters of data type *date* or *datetime* cannot have a default argument at all. The same is true for parameters of data type a *array*, they cannot have a default argument.

In addition reference parameters, i.e., parameters declared by the keyword byref, cannot have a default argument.