The Book of SumoSlang

"Assembly of Japanese bicycle require great peace of mind!"
–Robert M. Pirsig: *Zen and the Art of Motorcycle Maintenance*

Blank page

1 Introduction

Assembly of a processing plant model in Sumo require great peace of mind, to extend Mr. Pirsig's observation. This documentation contains the elements of the Sumo modeling language. Although it is not simple, once mastered, wonderful models can be created in a well-organized and documented fashion. From now on the modeling language will be referenced as SumoSlang.

1.1 What is SumoSlang?

SumoSlang is the modeling language used in the Sumo process modeling software. It was developed with modeling very large systems in mind: containing process units and process models. It was developed also with engineers in mind: write your equations in a familiar environment namely Microsoft Excel.

The naming, organization of tables and their content on various Excel worksheets and the keywords used is known as SumoSlang. Some of the main features of the language are:

- assignment oriented
    - assignment: only one variable is allowed at the left-hand side of the equation
    - formatted equation: symbols with arbitrary formatting (subscript, superscript, Greek characters) are allowed on both sides of the equation
- easy to comment and add notes
- documentation and code are at the same place
- algebraic calculations
- dynamic calculations

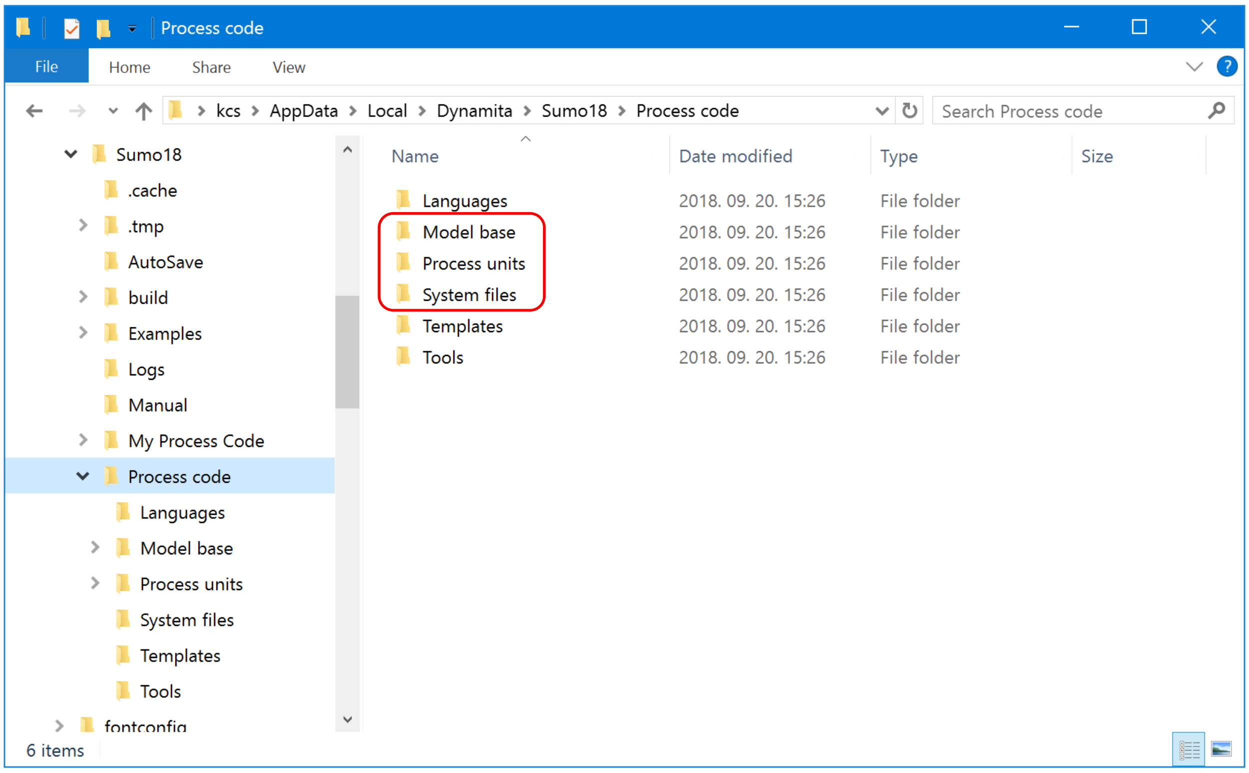
2 Structure

The Sumo process modeling software contains a standard library of process code. The elements of this library are process code files in the above-mentioned Excel document format which represents the source code of the library. There are three process code types:

- **general settings**—with the fixed file name `systemcode.xlsx`
- **process model**—with arbitrary file name, e.g. `Sumo1.xlsm` (the model files usually contain macros for validation, hence the `.xlsm` extension)
- **process unit**—with arbitrary file name, e.g. `CSTR with diffused aeration and input DO.xlsx`

SumoSlang is processed by the so-called Sumo Model Translator (SMT) first, which understands SumoSlang and converts it to an intermediate XML format for further processing. This documentation will discuss the SumoSlang features and rules from the SMT point of view. The graphical user interface of the Sumo modeling software may impose other requirements on the process code e.g. various worksheets containing irrelevant data for the SMT.

The standard library source code is placed in a specific folder structure shown on the following picture, where the Sumo install folder is open in Windows Explorer.



Name	Date modified	Type	Size
Languages	2018. 09. 20. 15:26	File folder	
Model base	2018. 09. 20. 15:26	File folder	
Process units	2018. 09. 20. 15:26	File folder	
System files	2018. 09. 20. 15:26	File folder	
Templates	2018. 09. 20. 15:26	File folder	
Tools	2018. 09. 20. 15:26	File folder	

Figure 1 Sumo process code structure

The red highlighted folders are used by the SMT to lookup for process code files mentioned at the beginning of the chapter (settings, models, process units).

It is considered a good practice to not modify the standard library, but place user code in the *My Process Code* folder which has the same inner structure as *Process Code*.

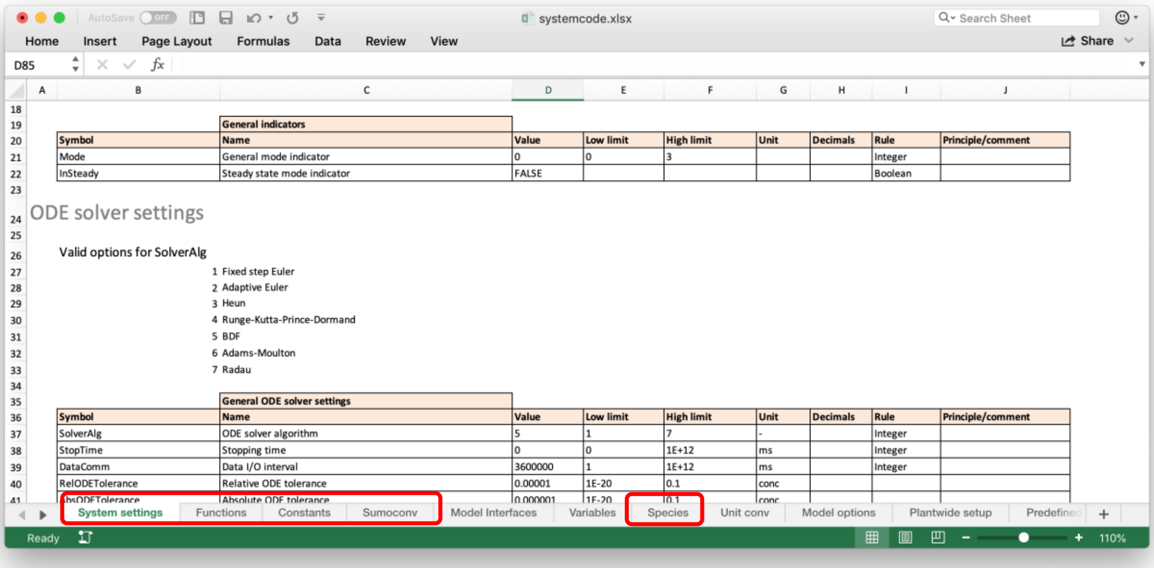
The following sections will explain the structure of the different process code files, what are their mandatory elements and what can be freely chosen by process code authors.

2.1 General settings

Sumo settings are stored in a special Excel file named `systemcode.xlsx`. The name of this file is mandatory.

Figure 2 shows the first worksheet of the system code with other worksheets expected by the SMT highlighted in red:

- **System settings**—various system wide settings and constants
- **Functions**—function declarations recognized by the SMT
- **Constants**—scientific constants used in calculations
- **Sumoconv**—dictionary of special (Greek) character conversions to C++ code
- **Species**—dictionary of chemical species



	General indicators							
Symbol	Name	Value	Low limit	High limit	Unit	Decimals	Rule	Principle/comment
Mode	General mode indicator	0	0	3			Integer	
InSteady	Steady state mode indicator	FALSE					Boolean	

ODE solver settings

Valid options for SolverAlg

1	Fixed step Euler
2	Adaptive Euler
3	Heun
4	Runge-Kutta-Prince-Dormand
5	BDF
6	Adams-Moulton
7	Radau

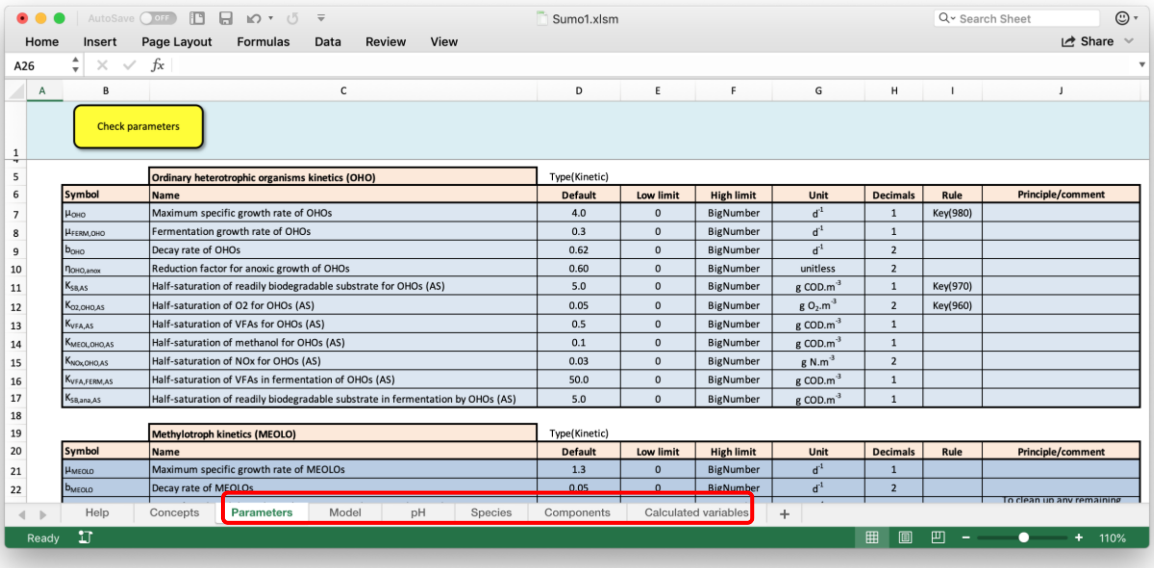
	General ODE solver settings							
Symbol	Name	Value	Low limit	High limit	Unit	Decimals	Rule	Principle/comment
SolverAlg	ODE solver algorithm	5	1	7	-		Integer	
StopTime	Stopping time	0	0	1E+12	ms		Integer	
DataComm	Data I/O interval	3600000	1	1E+12	ms		Integer	
RelODETolerance	Relative ODE tolerance	0.00001	1E-20	0.1	conc			
bsODETolerance	Absolute ODE tolerance	0.000001	1E-20	0.1	conc			

Figure 2 The `systemcode.xlsx` *file containing systemwide settings. The worksheets highlighted in red are expected by the SMT.*

2.2 Process model

The model files contain various scientific equations, matrices and parameter values used in simulations. Process units may reference one or more models thus, the variables and equations of the included model(s) are available in process unit calculations.

Figure 3 shows a model file example with frequently used worksheets highlighted.



Ordinary heterotrophic organisms kinetics (OHO) — Type(Kinetic)

Symbol	Name	Default	Low limit	High limit	Unit	Decimals	Rule	Principle/comment
μ_{OHO}	Maximum specific growth rate of OHOs	4.0	0	BigNumber	d^{-1}	1	Key(980)	
$\mu_{FERM,OHO}$	Fermentation growth rate of OHOs	0.3	0	BigNumber	d^{-1}	1		
b_{OHO}	Decay rate of OHOs	0.62	0	BigNumber	d^{-1}	2		
$\eta_{OHO,anox}$	Reduction factor for anoxic growth of OHOs	0.60	0	BigNumber	unitless	2		
$K_{SB,AS}$	Half-saturation of readily biodegradable substrate for OHOs (AS)	5.0	0	BigNumber	g $COD.m^{-3}$	1	Key(970)	
$K_{O2,OHO,AS}$	Half-saturation of O2 for OHOs (AS)	0.05	0	BigNumber	g $O_2.m^{-3}$	2	Key(960)	
$K_{VFA,AS}$	Half-saturation of VFAs for OHOs (AS)	0.5	0	BigNumber	g $COD.m^{-3}$	1		
$K_{MEOL,OHO,AS}$	Half-saturation of methanol for OHOs (AS)	0.1	0	BigNumber	g $COD.m^{-3}$	1		
$K_{NOx,OHO,AS}$	Half-saturation of NOx for OHOs (AS)	0.03	0	BigNumber	g $N.m^{-3}$	2		
$K_{VFA,FERM,AS}$	Half-saturation of VFAs in fermentation of OHOs (AS)	50.0	0	BigNumber	g $COD.m^{-3}$	1		
$K_{SB,ana,AS}$	Half-saturation of readily biodegradable substrate in fermentation by OHOs (AS)	5.0	0	BigNumber	g $COD.m^{-3}$	1		

Methylotroph kinetics (MEOLO) — Type(Kinetic)

Symbol	Name	Default	Low limit	High limit	Unit	Decimals	Rule	Principle/comment
μ_{MEOLO}	Maximum specific growth rate of MEOLOs	1.3	0	BigNumber	d^{-1}	1		
b_{MEOLO}	Decay rate of MEOLOs	0.05	0	BigNumber	d^{-1}	2		

Figure 3 Excel file containing model code. The worksheets highlighted in red are usually used in process unit calculations.

Some of the worksheet names are mandatory others are arbitrary. The mandatory names are expected by the SMT when a shorthand symbol is evaluated (see subsections of 5.1 Expandable symbols).

The fixed names are:

- **Parameters**—is a list of parameters having fixed numerical values used in calculations
- **Species**—a dictionary of equilibrium species used in pH calculations
- **Components**—contains state variables in dynamic systems, or model components in algebraic systems (e.g. chemical species that determine the chemical composition of the investigated solution: Ca2+, Mg2+, etc.)
- **Calculated variables**—contains equations calculating variables dependent on other variables

Other worksheets with arbitrary names are:

- **Model**—contains the so-called Gujer or kinetic matrix, which is a table used by process units to calculate biokinetic reactions.
- **pH**—contains all the information to calculate the equilibrium species in reactions that are considered during pH calculations

The model may contain other, user defined worksheets which can be referenced in process unit code. The referencing methods are discussed in section 5.1 Expandable symbols.

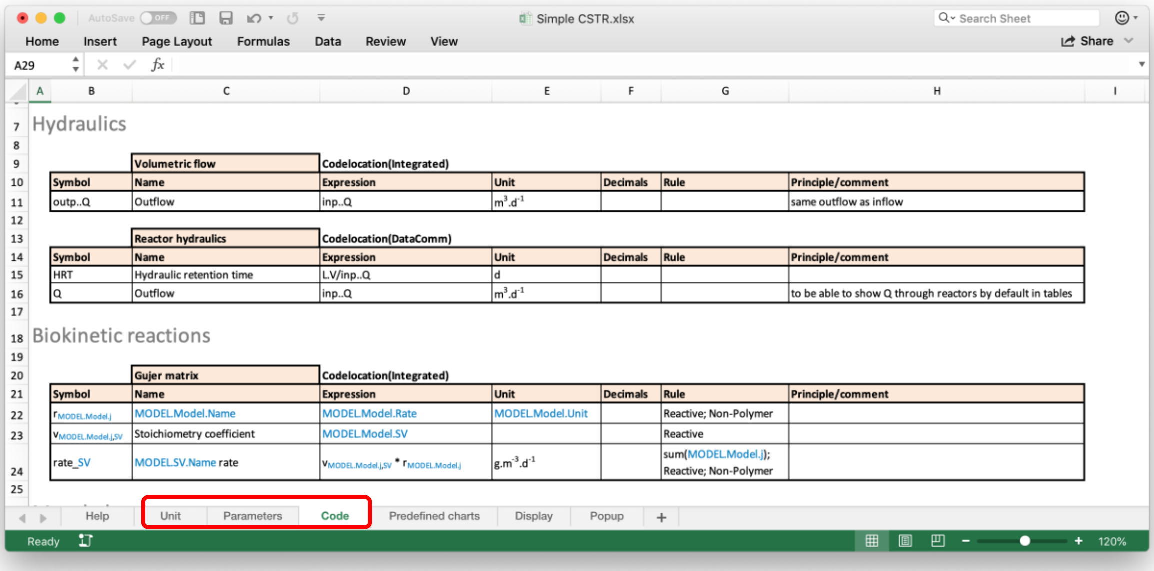
2.3 Process units

The process code contained in process units describe the behavior of bioreactors, flow elements separators and other modeled elements. As mentioned above the process units may "pull in" variables and equations from one or more model files and use them in their calculations.

The process units should be able to work with different models which means that they cannot reference models by name, but some other mechanism. See again section 5.1 Expandable symbols for more information.

Figure 4 shows the code of a process unit with the commonly used worksheets highlighted. The names of these worksheets are mandatory.

- **Unit**—a dictionary of attributes and component handlings valid in the process unit (see sections 5.2.1 Attributes and 5.2.2 Handlings)
- **Parameters**—contains the simulation parameters modifiable by the user in the Sumo process modeling software. These are valid only in the process unit, but model parameters can be "pulled in".
- **Code**—contains the process code of the process unit grouped in so-called code locations (see section 3.2.1 Table)



Hydraulics

	Volumetric flow	Codelocation(Integrated)				
Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
outp..Q	Outflow	inp..Q	$m^3.d^{-1}$			same outflow as inflow

	Reactor hydraulics	Codelocation(DataComm)				
Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
HRT	Hydraulic retention time	L.V/inp..Q	d			
Q	Outflow	inp..Q	$m^3.d^{-1}$			to be able to show Q through reactors by default in tables

Biokinetic reactions

	Gujer matrix	Codelocation(Integrated)				
Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
$r_{MODEL.Model.j}$	MODEL.Model.Name	MODEL.Model.Rate	MODEL.Model.Unit		Reactive; Non-Polymer	
$v_{MODEL.Model.j,SV}$	Stoichiometry coefficient	MODEL.Model.SV			Reactive	
rate_SV	MODEL.SV.Name rate	$v_{MODEL.Model.j,SV} * r_{MODEL.Model.j}$	$g.m^{-3}.d^{-1}$		sum(MODEL.Model.j); Reactive; Non-Polymer	

Figure 4 Process unit file with the commonly-used worksheets highlighted in red.

Other, optional worksheets usable by the SMT can be added to the process unit. These worksheets also have mandatory names and they are the following:

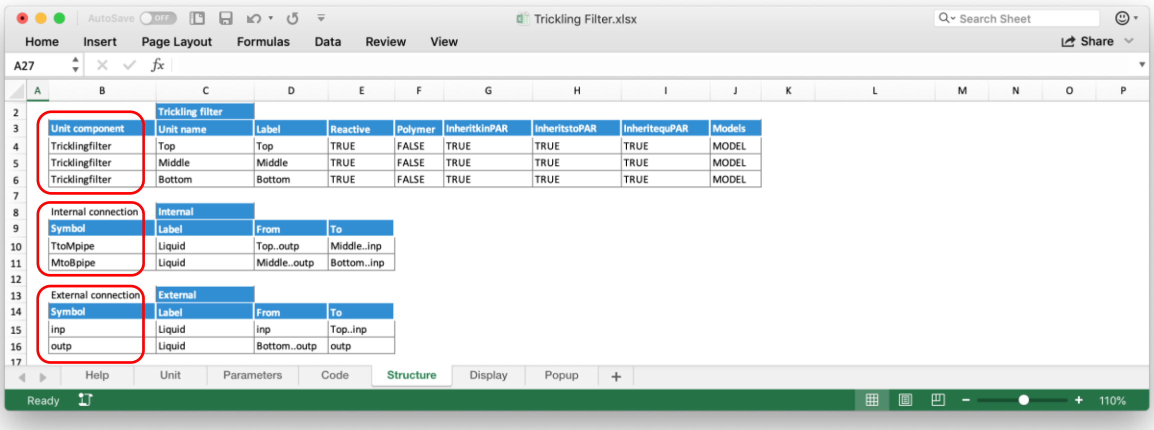
- **Components**—it is like the *Components* in the model but valid only in the process unit. They may overwrite model components.
- **Functions**—it is like the *Functions* in the `systemcode.xlsx` but valid only in the process unit.
- **Structure**—it is present in composite process units. Contains the list of components and connections between them, as well as connections to the outside world.

2.3.1 Composite process units

A process unit may consist of several subunits. The components are separate process units with their own process code files. Grouping them in a single process unit hides the internal complexity, the group behaving like a black box with few connections to the outside world.

The *Structure* worksheet introduced in the previous section contains this grouping information. Figure 5 shows an example of the *Structure* sheet of a composite unit.

Columns of the component table play an essential role in how the composite unit controls its components (see section 2.4 Process unit hierarchy).



	Trickling filter							
Unit component	Unit name	Label	Reactive	Polymer	InheritkinPAR	InheritstoPAR	InheritequPAR	Models
Tricklingfilter	Top	Top	TRUE	FALSE	TRUE	TRUE	TRUE	MODEL
Tricklingfilter	Middle	Middle	TRUE	FALSE	TRUE	TRUE	TRUE	MODEL
Tricklingfilter	Bottom	Bottom	TRUE	FALSE	TRUE	TRUE	TRUE	MODEL

Internal connection	Internal		
Symbol	Label	From	To
TtoMpipe	Liquid	Top..outp	Middle..inp
MtoBpipe	Liquid	Middle..outp	Bottom..inp

External connection	External		
Symbol	Label	From	To
inp	Liquid	inp	Top..inp
outp	Liquid	Bottom..outp	outp

Figure 5 Structure worksheet of a composite process unit showing three components, their internal connections and the external connections of the whole composite unit.

2.4 Process unit hierarchy

In the Sumo software the modeled plant is organized in a process unit hierarchy. The root element is (an artificial) composite process unit which lists the participating process units as components. These process units may be simple or composite too.

In the previous section it was mentioned that composite process units (parents) controls its components (children). This control is represented by passing so-called attributes, handlings and parameters from the parent to the children (see sections 5.2.1 Attributes and 5.2.2 Handlings).

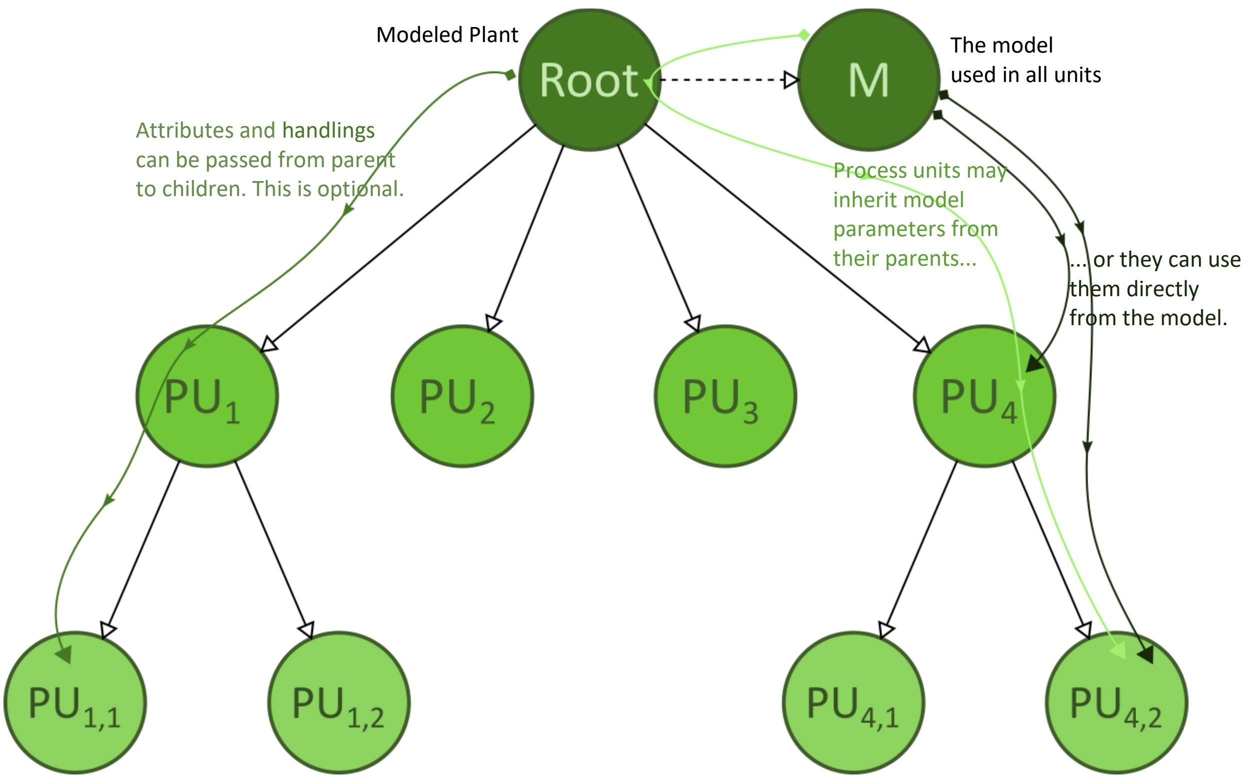
The artificial root element is provided for the user by the Sumo software to be able to set plantwide simulation properties and parameters. The *Unit* and *Parameters* worksheets of a process unit contains attributes, handlings and parameters valid in that process unit, but these values can be overwritten by the root element or by a parent composite unit (if the process unit is part of one).

The intention of overwriting can be declared in the columns of the component table present on the *Structure* sheet of the parent process unit. On Figure 5 the *Unit component* table has columns between *Label* and *Models*. Those are attribute names and the values in the different rows are passed to the corresponding process unit.

In a similar manner, columns after the *Models* column represent handling names and the values in the different rows are passed to the corresponding component (the example does not contain such columns).

Where a cell in a given column is empty no value is passed to the component unit thus the value defined in the component unit prevails.

Figure 6 shows the hierarchy of four modeled process units (PU_1, PU_2, PU_3, PU_4) where PU_1 and PU_4 are composite units consisting of two process units: $PU_{1,1}$, $PU_{1,2}$ and $PU_{4,1}$, $PU_{4,2}$ respectively. The example shows on the left side of the hierarchy how attributes and handlings can be passed from parent to children, while the right side shows two methods for passing model parameters from parent to children.



Modeled Plant
Root
M
The model used in all units
Attributes and handlings can be passed from parent to children. This is optional.
Process units may inherit model parameters from their parents...
... or they can use them directly from the model.
PU_1
PU_2
PU_3
PU_4
$PU_{1,1}$
$PU_{1,2}$
$PU_{4,1}$
$PU_{4,2}$

Figure 6 Simple process unit hierarchy. The dashed line between the Root and the Model means that the Root is not the parent of the Model, only uses it.

Parameter passing has a special name in SumoSlang: parameter inheritance. It is explained in section 5.2.1 Attributes in more detail. Here it is enough that when inheritance is ON the parameters come from the parent of the process unit, otherwise from the model. The root process unit always takes its parameters from the model.

2.4.1 Plantwide code

Global settings and properties can be given in the so-called plantwide code file. This is provided by the Sumo software per modeled plant and it is not part of the process unit library accompanying the software.

One can think of the plantwide code as if the software would give access to some elements of the artificial root process unit. The plantwide code file looks like a normal process unit, it has *Unit*, *Parameters* and *Code* worksheets and the data given on these worksheets will be incorporated in the root element. However, the user cannot reach the *Structure* sheet of the root element through plantwide code.

Composite process units can reach parameters and other variables of their **direct** children by prefixing the variables with the child unit name like this:

```
ChildName..variable
```

The plantwide code is an exception at the moment of this reference rule. When a variable from a child unit (direct or deep in the hierarchy) is needed in plantwide calculations it should be qualified with its full namespace prefix (see chapter 7 Namespaces).

3 Process code layout

Figure 2 shows a worksheet containing typical process code. The process code rows are placed in neat tables which gives an organized look and feel to the code. The tables are separated by at least one blank row. Note that the pretty formatting, coloring of the tables does not influence how they are perceived by the SMT. For example, if the user would insert some text in cell B10, the SMT would include it in the first table. The tables contain the process code and anything outside of them is considered as comment and ignored.

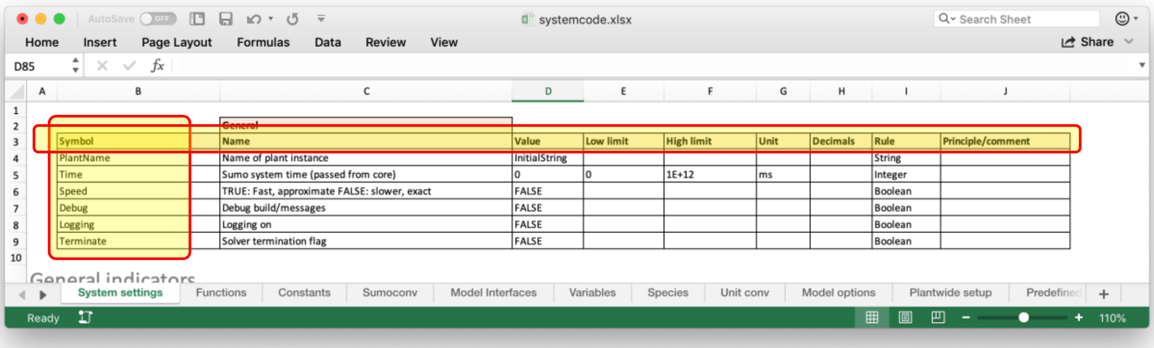
3.1 Table structure

Every table should have a

- **descriptor**—a 3-segment cell range `B2:D2` in the example mentioned above
    - **table type**—cell `B2`, it can be empty or may contain one of the following keywords
        - Array
        - if block
        - C++ code
        - Newton-Raphson—used in model code on the *Calculated variables* worksheet
        - Port, Attribute, Model, Handling—used on the *Unit* worksheet of process units
        - SolverConfiguration—meant to replace `Newton-Raphson` table types in tandem with `Equilibrium` code locations (see 9 Advanced topics)
    - **table name**—cell `C2`, contains a meaningful grouping name of the table content
    - **table tag**—cell `D2`, like table type it can be empty or may contain one of the following keywords
        - Pure—used only on the Functions worksheet of system settings
        - Type(arg)—used in model code (*Parameters*, *pH*, *Calculated variables*) for further grouping tables
        - Codelocation(arg)—used on *Code* worksheet of process units. The argument may be one or more of the following keywords
            - ZeroTime
            - DataComm
            - Integrated
            - Equilibrium
        - Scope(arg)—used on *Unit* worksheet of process units
- **header**—the range `B3:J3` in the example. The number of columns is arbitrary. Some of the column header names are fixed (e.g. *Symbol*, *Name*, *Value*, *Rule*) others are arbitrary.
- **body**—the code lines grouped in the table (the body can be empty). Its range is `B4:J9` in the first table of the example.

3.1.1 Simple table

Usually the range of a table is determined by the content of the *Symbol* column and the header row as shown on the following picture.



	A	B	C	D	E	F	G	H	I	J
1										
2			General							
3		Symbol	Name	Value	Low limit	High limit	Unit	Decimals	Rule	Principle/comment
4		PlantName	Name of plant instance	InitialString					String	
5		Time	Sumo system time (passed from core)	0	0	1E+12	ms		Integer	
6		Speed	TRUE: Fast, approximate FALSE: slower, exact	FALSE					Boolean	
7		Debug	Debug build/messages	FALSE					Boolean	
8		Logging	Logging on	FALSE					Boolean	
9		Terminate	Solver termination flag	FALSE					Boolean	
10										

General indicators

Figure 7 Table boundaries.

The key elements like descriptor, header and body described in the previous section are determined relative to the *Symbol* cell. The table extends to the right and down while the *Symbol* row and column contains non-empty cells.

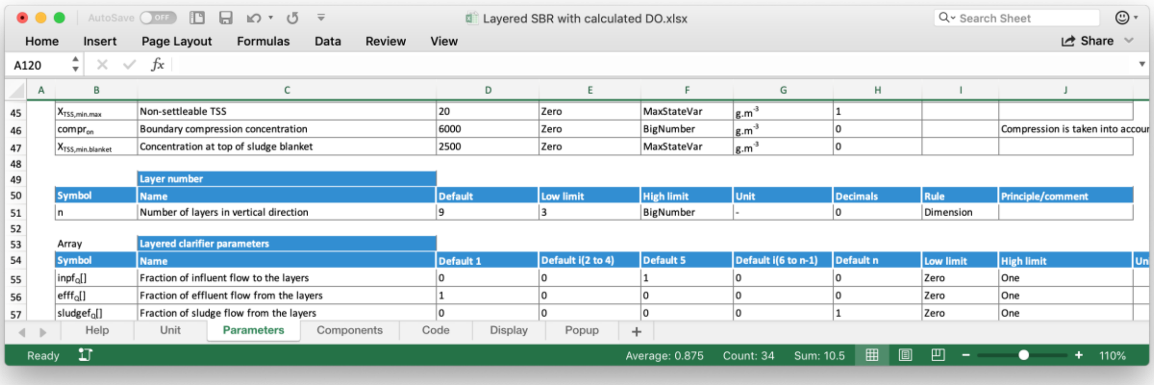
Some table types do not follow this structure but should comply to some different structure nevertheless. Such table types are the *Array*, *if block* and *C++ code*.

The rows of a code table represent assignments in the form of `a = b`, or more precisely `a ← b` (b goes to a). The left-hand side of the assignment is in the *Symbol* column of the table, while the right-hand side is in one of the *Value* or *Default* or *Expression* columns. In some cases, the *Expression* column may break in several parts (see 3.1.2 Array).

3.1.2 Arrays

Arrays are fundamental in every programming language. In SumoSlang they can be specified in three ways. The first is a simple table containing a row with its symbol in array notation e.g. `var[]`. SumoSlang require a so-called array rule in the *Rule* column of the array row, like `[n]`, where `n` is the size of the array and it is declared on the *Parameters* sheet of a process unit file. The examples were taken from the `Layered SBR with calculated DO.xlsx`.

Figure 8 shows an example of array size definition. On the *Parameters* worksheet the right-hand side of the assignments is contained in the *Default* or *Value* column.

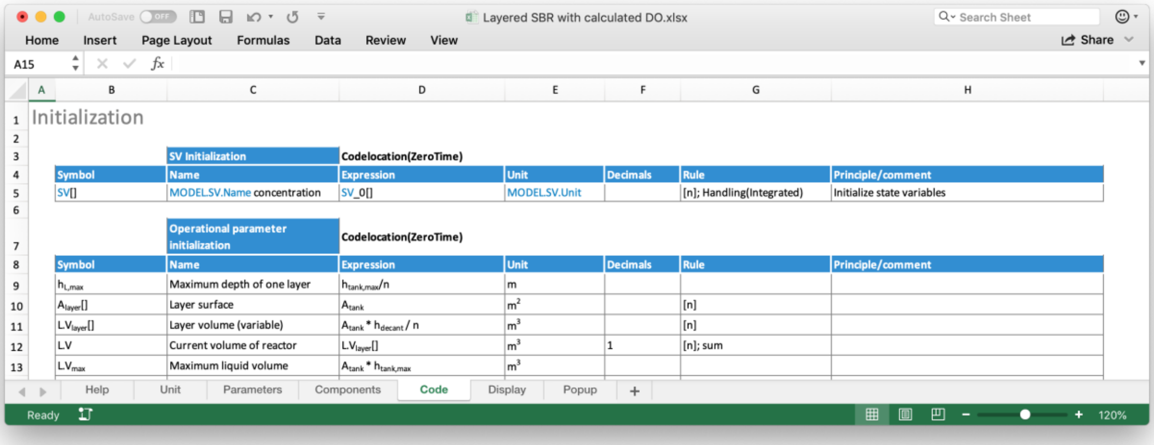


	A	B	C	D	E	F	G	H	I	J
45		$X_{TSS,min.max}$	Non-settleable TSS	20	Zero	MaxStateVar	$g.m^{-3}$	1		
46		$compr_{on}$	Boundary compression concentration	6000	Zero	BigNumber	$g.m^{-3}$	0		Compression is taken into accour
47		$X_{TSS,min.blanket}$	Concentration at top of sludge blanket	2500	Zero	MaxStateVar	$g.m^{-3}$	0		
48										
49			Layer number							
50		Symbol	Name	Default	Low limit	High limit	Unit	Decimals	Rule	Principle/comment
51		n	Number of layers in vertical direction	9	3	BigNumber	-	0	Dimension	
52										
53		Array	Layered clarifier parameters							
54		Symbol	Name	Default 1	Default i(2 to 4)	Default 5	Default i(6 to n-1)	Default n	Low limit	High limit
55		$inpf_Q[]$	Fraction of influent flow to the layers	0	0	1	0	0	Zero	One
56		$efff_Q[]$	Fraction of effluent flow from the layers	1	0	0	0	0	Zero	One
57		$sludgef_Q[]$	Fraction of sludge flow from the layers	0	0	0	0	1	Zero	One

Figure 8 Dimension type variable definition in row 51.

An array size definition requires a `Dimension` rule specifying that it is an array dimension type variable. It is worth mentioning that SumoSlang handles only one-dimensional, real number arrays for now.

Figure 9 shows an example of simple arrays in various code tables.



Initialization

Symbol	SV Initialization Name	Codelocation(ZeroTime) Expression	Unit	Decimals	Rule	Principle/comment
SV[]	MODEL.SV.Name concentration	SV_0[]	MODEL.SV.Unit		[n]; Handling(Integrated)	Initialize state variables

Symbol	Operational parameter initialization Name	Codelocation(ZeroTime) Expression	Unit	Decimals	Rule	Principle/comment
$h_{L,max}$	Maximum depth of one layer	$h_{tank,max}/n$	m			
$A_{layer}[]$	Layer surface	A_{tank}	m^2		[n]	
$LV_{layer}[]$	Layer volume (variable)	$A_{tank} * h_{decant} / n$	m^3		[n]	
LV	Current volume of reactor	$LV_{layer}[]$	m^3	1	[n]; sum	
LV_{max}	Maximum liquid volume	$A_{tank} * h_{tank,max}$	m^3			

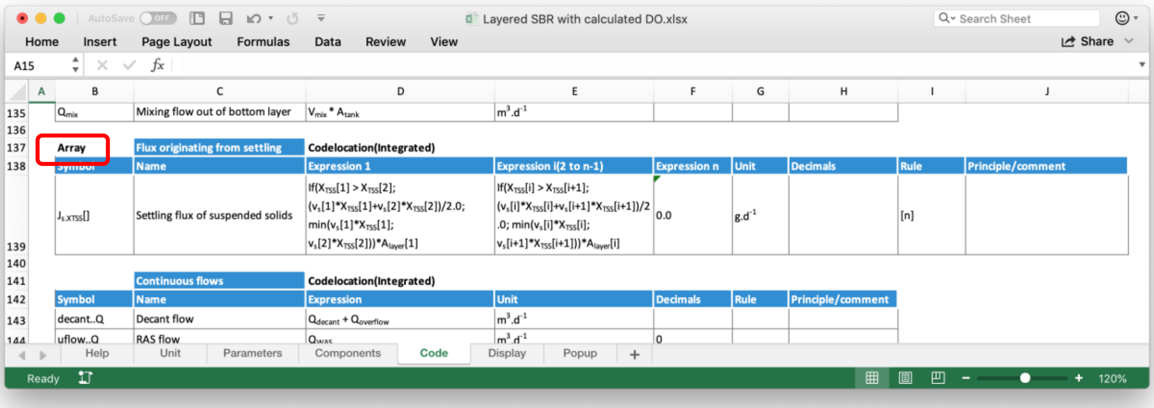
Figure 9 Array rows in two tables: rows 5, 10, 11 contain arrays with their symbol ending in brackets and with `[n]` *in their Rule column.*

The SMT will unwind an array row to a loop where the elements of the array will be assigned to the right-hand side contained in the *Expression* column. In our example `n` is defined as 9 meaning that all arrays with rule `[n]` have 9 elements. Row 5 for example will be unwound as follows:

```
SV[1] = SV_0[1]
SV[2] = SV_0[2]
...
SV[9] = SV_0[9]
```

or more precisely to an equivalent loop. Note that the expression is also in array notation, i.e. ending in brackets. Row 10 contains an example where every element of the array is assigned the same variable (no array notation in the expression). As an exercise try to unwind the array in row 11.

The second option to define an array is to set the table type to `Array` in the table descriptor. The SMT is prepared that the *Expression* column may be broken in several parts. Figure 10 shows an example of this array type.



Layered SBR with calculated DO.xlsx

A15

	A	B	C	D	E	F	G	H	I	J
135		Q_{mix}	Mixing flow out of bottom layer	V_{mix} * A_{tank}	$m^3.d^{-1}$					
136										
137		Array	Flux originating from settling	Codelocation(Integrated)						
138		Symbol	Name	Expression 1	Expression i(2 to n-1)	Expression n	Unit	Decimals	Rule	Principle/comment
139		$J_{s,XTSS}$[]	Settling flux of suspended solids	If(X_{TSS}[1] > X_{TSS}[2]; (v_s[1]*X_{TSS}[1]+v_s[2]*X_{TSS}[2])/2.0; min(v_s[1]*X_{TSS}[1]; v_s[2]*X_{TSS}[2]))*A_{layer}[1]	If(X_{TSS}[i] > X_{TSS}[i+1]; (v_s[i]*X_{TSS}[i]+v_s[i+1]*X_{TSS}[i+1])/2.0; min(v_s[i]*X_{TSS}[i]; v_s[i+1]*X_{TSS}[i+1]))*A_{layer}[i]	0.0	$g.d^{-1}$		[n]	
140										
141			Continuous flows	Codelocation(Integrated)						
142		Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment		
143		decant..Q	Decant flow	Q_{decant} + $Q_{overflow}$	$m^3.d^{-1}$					
144		uflow..Q	RAS flow	Q_{WAS}	$m^3.d^{-1}$	0				

Help | Unit | Parameters | Components | Code | Display | Popup

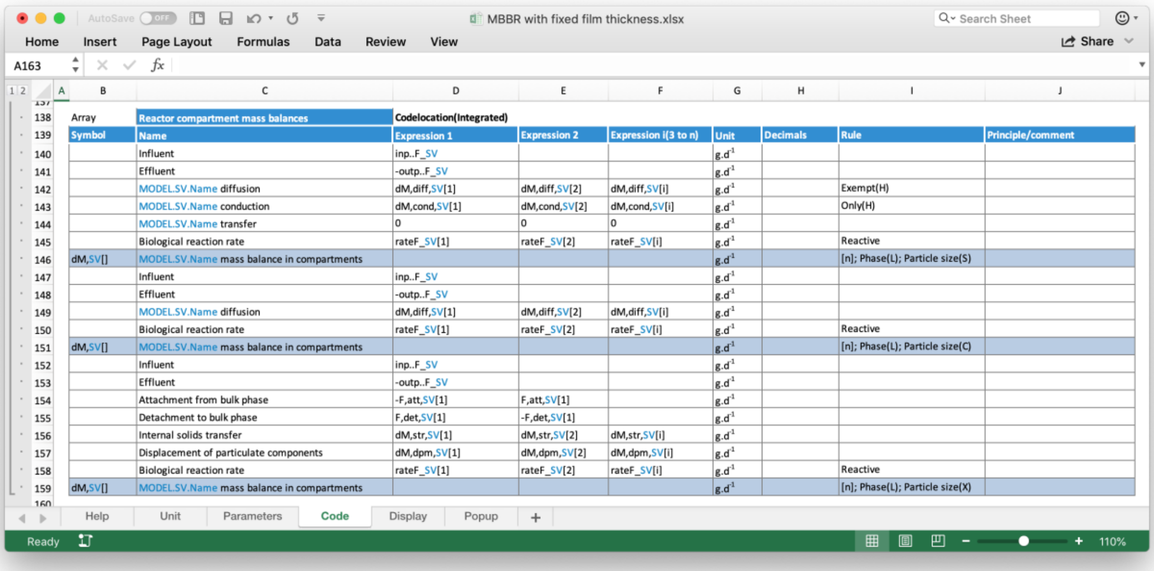
Figure 10 Array table type beginning in row 137 ending in 139.

In this array type a more fine-grained unwinding process can be specified; for example, the first and the last element will be assigned different values compared to the rest of the array:

```
Js.XTSS[1] = content of the Expression 1 column
Js.XTSS[2] = content of the Expression i(2 to n-1) column where i is 2
...
Js.XTSS[8] = content of the Expression i(2 to n-1) column where i is 8
Js.XTSS[9] = 0.0
```

Please note that indexer variables (`i` in the second *Expression* column) can be used only in their column, constant indexers can be used in all *Expression* columns (for example the first *Expression* column contains X_{TSS}`[1]` and X_{TSS}`[2]` but X_{TSS}`[n]` could be used as well).

The third option is the so-called composite array. It is like the previous one, but the unwinding process is even more sophisticated. Figure 11 shows an example of a composite array table taken from the `MBBR with fixed film thicknes.xlsx`.



MBBR with fixed film thickness.xlsx

A163

	A	B	C	D	E	F	G	H	I	J
138		Array	Reactor compartment mass balances	Codelocation(Integrated)						
139		Symbol	Name	Expression 1	Expression 2	Expression i(3 to n)	Unit	Decimals	Rule	Principle/comment
140			Influent	inp..F_SV			$g.d^{-1}$			
141			Effluent	-outp..F_SV			$g.d^{-1}$			
142			MODEL.SV.Name diffusion	dM,diff,SV[1]	dM,diff,SV[2]	dM,diff,SV[i]	$g.d^{-1}$		Exempt(H)	
143			MODEL.SV.Name conduction	dM,cond,SV[1]	dM,cond,SV[2]	dM,cond,SV[i]	$g.d^{-1}$		Only(H)	
144			MODEL.SV.Name transfer	0	0	0	$g.d^{-1}$			
145			Biological reaction rate	rateF_SV[1]	rateF_SV[2]	rateF_SV[i]	$g.d^{-1}$		Reactive	
146		dM,SV[]	MODEL.SV.Name mass balance in compartments				$g.d^{-1}$		[n]; Phase(L); Particle size(S)	
147			Influent	inp..F_SV			$g.d^{-1}$			
148			Effluent	-outp..F_SV			$g.d^{-1}$			
149			MODEL.SV.Name diffusion	dM,diff,SV[1]	dM,diff,SV[2]	dM,diff,SV[i]	$g.d^{-1}$			
150			Biological reaction rate	rateF_SV[1]	rateF_SV[2]	rateF_SV[i]	$g.d^{-1}$		Reactive	
151		dM,SV[]	MODEL.SV.Name mass balance in compartments				$g.d^{-1}$		[n]; Phase(L); Particle size(C)	
152			Influent	inp..F_SV			$g.d^{-1}$			
153			Effluent	-outp..F_SV			$g.d^{-1}$			
154			Attachment from bulk phase	-F,att,SV[1]	F,att,SV[1]		$g.d^{-1}$			
155			Detachment to bulk phase	F,det,SV[1]	-F,det,SV[1]		$g.d^{-1}$			
156			Internal solids transfer	dM,str,SV[1]	dM,str,SV[2]	dM,str,SV[i]	$g.d^{-1}$			
157			Displacement of particulate components	dM,dpm,SV[1]	dM,dpm,SV[2]	dM,dpm,SV[i]	$g.d^{-1}$			
158			Biological reaction rate	rateF_SV[1]	rateF_SV[2]	rateF_SV[i]	$g.d^{-1}$		Reactive	
159		dM,SV[]	MODEL.SV.Name mass balance in compartments				$g.d^{-1}$		[n]; Phase(L); Particle size(X)	

Help | Unit | Parameters | Code | Display | Popup

Figure 11 Composite array, the terms in the rows without symbol are summed by Expression columns.

The rows with empty symbol define summation terms for all elements of the array. The array variables are identified by a non-empty symbol and a `[n]` rule in rows 146, 151, 159 respectively. The result of unwinding the first array is as follows:

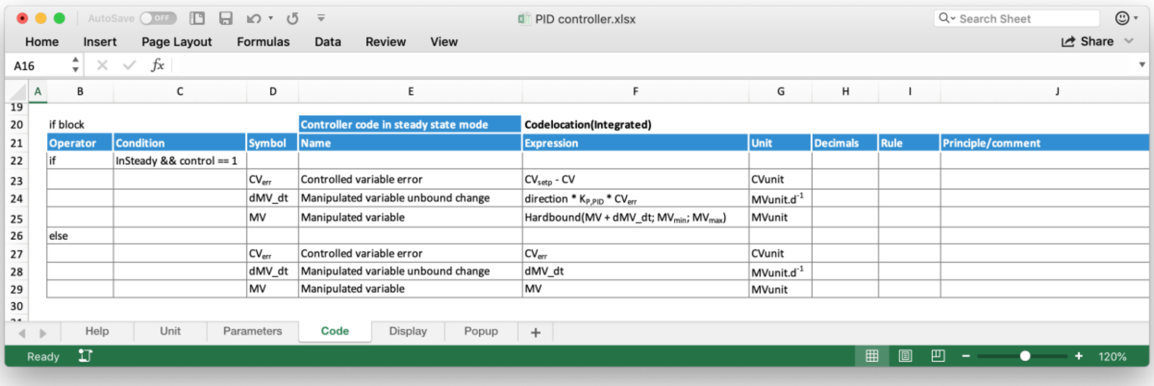
```
dM,SV[1] = (inp..F_SV + (-outp..F_SV) + ... + rateF_SV[1])
dM,SV[2] = (dM,diff,SV[2] + 0 + rateF_SV[2])
...
dM,SV[4] = (dM,diff,SV[4] + 0 + rateF_SV[4])
```

if the array size `n` is 4 in the example and the summation terms of the first array are taken from rows 140 to 145. As the example shows, multiple composite arrays can be defined in a single table. Sometimes it is worth to split the arrays in separate tables, for example the last array could be extracted in its own table, resulting in an opportunity to simplify the original one. It is obvious that *Expression 2* and *Expression i(3 to n)* columns have the same terms for the first two arrays, so they could be combined in a single *Expression i(2 to n)*.

Every summation term row may contain filtering rules influencing the resulting sum (see section 5.2.1 Attributes to learn about attribute filtering). In the example the last term, `rateF_L.SV[i]` is included only in case of *Reactive* process units.

3.1.3 if block

Conditional execution of program code is another important element of a programming language. In SumoSlang condition execution is declared in tables with type *if block*. Figure 12 shows an example of if block taken from `PID controller.xlsx`.



Row	Operator	Condition	Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
20	if block			Controller code in steady state mode	Codelocation(Integrated)				
22	if	InSteady && control == 1							
23			CV_{err}	Controlled variable error	CV_{setp} - CV	CVunit			
24			dMV_dt	Manipulated variable unbound change	direction * $K_{P,PID}$ * CV_{err}	MVunit.d^{-1}			
25			MV	Manipulated variable	Hardbound(MV + dMV_dt; MV_{min}; MV_{max})	MVunit			
26	else								
27			CV_{err}	Controlled variable error	CV_{err}	CVunit			
28			dMV_dt	Manipulated variable unbound change	dMV_dt	MVunit.d^{-1}			
29			MV	Manipulated variable	MV	MVunit			

Figure 12 Example for if block table.

This table type has some additional columns before *Symbol*, namely *Operator* and *Condition*. The *Operator* column may contain only the keywords `if`, `else if`, or `else`. An `if` operator always should have a matching `else` or `else if` pair and the symbols in their code rows should be the same (see the symbols in rows 23-25 and rows 27-29).

The *Condition* column contains relational operators similar to the C language with one exception: the assignment symbol `=` is recognized as the equality operator `==`, meaning that in the example `control = 1` would be the same as `control == 1`.

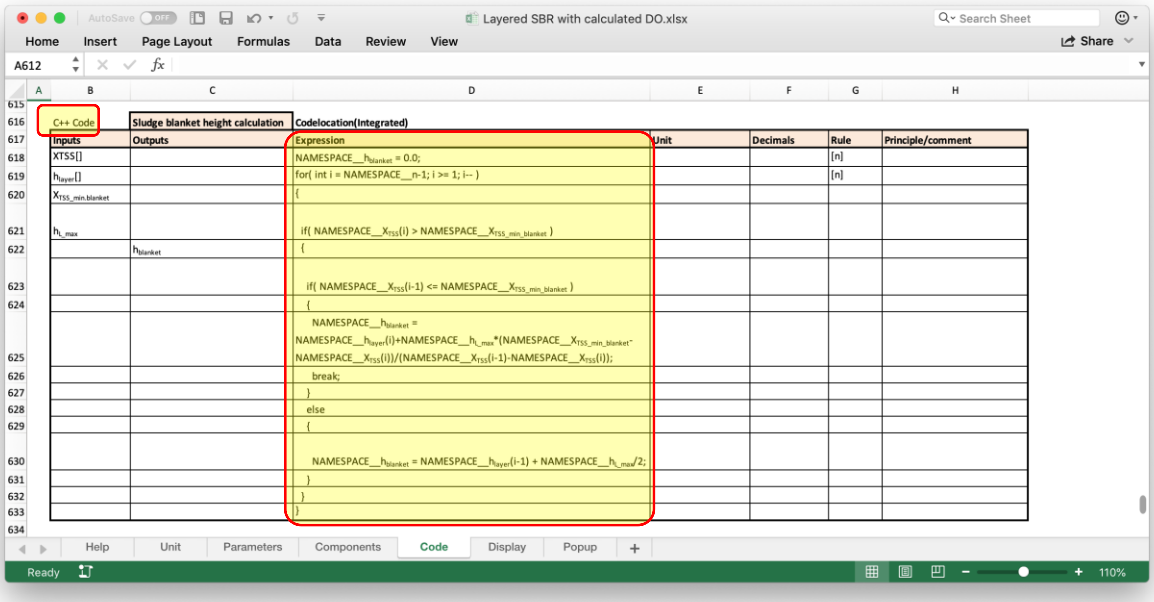
Nested if blocks are supported too, in that case there are many numbered *Operator* columns before the *Condition* column.

3.1.4 C++ code

Process code may include blocks of raw C++ code to perform calculations that are clumsy or impossible in SumoSlang. The Sumo variables used as input arguments by the C++ code are listed in the *Inputs* column, while the generated output variables are listed in the *Outputs* column.

The range of a C++ code table type is determined with help of the *Inputs* keyword (used like *Symbol* in simple tables) and the *Expression* keyword. The table ends at the row of the last non-empty cell in the *Expression* column.

The code in the *Expression* column may contain variables with a `NAMESPACE__` prefix. This prefix will be replaced by the SMT with the namespace of the process unit (see chapter 7 Namespaces).



C++ Code	Sludge blanket height calculation	Codelocation(Integrated)				
Inputs	Outputs	Expression	Unit	Decimals	Rule	Principle/comment
XTSS[]		NAMESPACE__$h_{blanket}$ = 0.0;			[n]	
h_{layer}[]		for(int i = NAMESPACE__n-1; i >= 1; i--)			[n]	
$X_{TSS\_min.blanket}$		{				
h_{L_max}		if(NAMESPACE__X_{TSS}(i) > NAMESPACE__$X_{TSS_min_blanket}$)				
	$h_{blanket}$	{				
		if(NAMESPACE__X_{TSS}(i-1) <= NAMESPACE__$X_{TSS_min_blanket}$)				
		{				
		NAMESPACE__$h_{blanket}$ = NAMESPACE__h_{layer}(i)+NAMESPACE__h_{L_max}*(NAMESPACE__$X_{TSS_min_blanket}$-NAMESPACE__X_{TSS}(i))/(NAMESPACE__X_{TSS}(i-1)-NAMESPACE__X_{TSS}(i));				
		break;				
		}				
		else				
		{				
		NAMESPACE__$h_{blanket}$ = NAMESPACE__h_{layer}(i-1) + NAMESPACE__h_{L_max}/2;				
		}				
		}				
		}				

Figure 13 C++ code example. The code lines in the Expression column are included in the generated XML after proper namespacing.

3.2 Table descriptor

Table names are part of the table descriptor range and on the *Parameters* worksheet (model or process unit) they should be unique. This is also true for tables with the same table type (if block, Array etc.) on the *Code* worksheet.

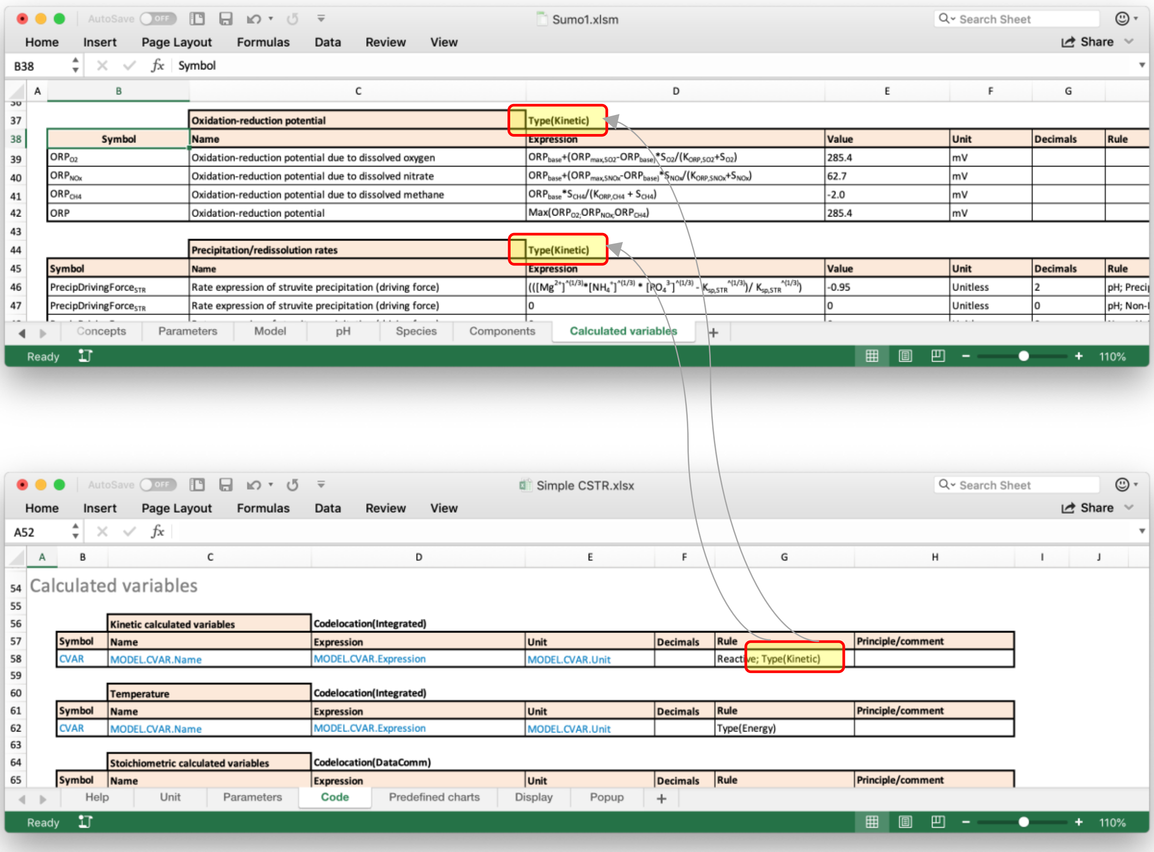
3.2.1 Table tag

This is the third section of the table descriptor right after the table name. It may contain table grouping information like `Codelocation` (which is a keyword) or code filtering information. The latter is an arbitrary text that can be used in the *Rule* column of an expandable code line to restrict the expansion from tables with that text in their table tag (see 5.3 Mechanism of expansion).

In the process code library provided by the Sumo software the most used table tag is `Codelocation` and `Type(...)`.

Type is used in model files for example to group different equation types (kinetic, stoichiometric, equilibrium, energy) on the *Calculated variables* worksheet. `Type` and its arguments are not keywords; the process code author can define arbitrary grouping texts which can be used in process units.

Figure 14 shows the *Calculated variables* worksheet of a model file with `Type(Kinetic)` groups. This group identifier is then referenced in the *Rule* column of a code line in a process unit meaning that the equations from all `Type(Kinetic)` tables will be included from the model (see also 5.2.3 Non-keyword rules).


Sumo1.xlsm

Oxidation-reduction potential		Type(Kinetic)				
Symbol	Name	Expression	Value	Unit	Decimals	Rule
ORP_{O2}	Oxidation-reduction potential due to dissolved oxygen	$ORP_{base}+(ORP_{max,SO2}-ORP_{base})*S_{O2}/(K_{ORP,SO2}+S_{O2})$	285.4	mV		
ORP_{NOx}	Oxidation-reduction potential due to dissolved nitrate	$ORP_{base}+(ORP_{max,SNOx}-ORP_{base})*S_{NOx}/(K_{ORP,SNOx}+S_{NOx})$	62.7	mV		
ORP_{CH4}	Oxidation-reduction potential due to dissolved methane	$ORP_{base}*S_{CH4}/(K_{ORP,CH4} + S_{CH4})$	-2.0	mV		
ORP	Oxidation-reduction potential	$Max(ORP_{O2;}ORP_{NOx;}ORP_{CH4})$	285.4	mV		

Precipitation/redissolution rates		Type(Kinetic)				
Symbol	Name	Expression	Value	Unit	Decimals	Rule
$PrecipDrivingForce_{STR}$	Rate expression of struvite precipitation (driving force)	$(([Mg^{2+}]^{\wedge(1/3)}*[NH_4^+]^{\wedge(1/3)} * [PO_4^{3-}]^{\wedge(1/3)} - K_{sp,STR}{}^{\wedge(1/3)})/ K_{sp,STR}{}^{\wedge(1/3)})$	-0.95	Unitless	2	pH; Preci
$PrecipDrivingForce_{STR}$	Rate expression of struvite precipitation (driving force)	0	0	Unitless	0	pH; Non-I

Simple CSTR.xlsx

Calculated variables

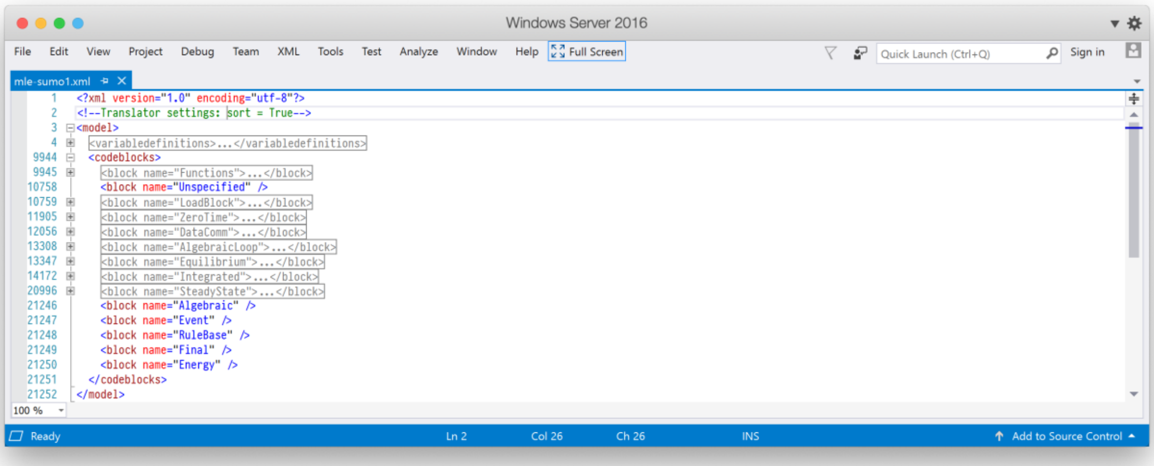
	Kinetic calculated variables	Codelocation(Integrated)				
Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
CVAR	MODEL.CVAR.Name	MODEL.CVAR.Expression	MODEL.CVAR.Unit		Reactive; Type(Kinetic)	

	Temperature	Codelocation(Integrated)				
Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
CVAR	MODEL.CVAR.Name	MODEL.CVAR.Expression	MODEL.CVAR.Unit		Type(Energy)	

	Stoichiometric calculated variables	Codelocation(DataComm)				
Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment

Figure 14 Type(...) table groups in a model file and referencing them in the Rule column of a process unit

Codelocation is used for grouping process unit code in groups (`ZeroTime`, `DataComm` etc.) described in section 3.1 Table structure. The previous example shows two Integrated and one `DataComm` code location in column D of the visible tables. The grouping, which becomes important in simulations, will be reflected in the generated XML file.


mle-sumo1.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!--Translator settings: sort = True-->
<model>
  <variabledefinitions>...</variabledefinitions>
  <codeblocks>
    <block name="Functions">...</block>
    <block name="Unspecified" />
    <block name="LoadBlock">...</block>
    <block name="ZeroTime">...</block>
    <block name="DataComm">...</block>
    <block name="AlgebraicLoop">...</block>
    <block name="Equilibrium">...</block>
    <block name="Integrated">...</block>
    <block name="SteadyState">...</block>
    <block name="Algebraic" />
    <block name="Event" />
    <block name="RuleBase" />
    <block name="Final" />
    <block name="Energy" />
  </codeblocks>
</model>
```

Figure 15 The generated XML file with collapsed nodes for better overview.

Figure 15 shows an overview of the generated XML file with collapsed nodes except the main `<model>` node and `<codeblocks>` where the code locations start in row 10759. Code locations with content inside are collapsed in the example.

The `Codelocation` may specify multiple code blocks together with code sections. The syntax is as follows:

```
Codelocation(block[,section] … [;block[,section]])
```

where

- block—is the code block name
- section—is the code section name under the block node
- , —is the section separator
- ; —is the block separator

Elements in brackets are optional. If `section` is not specified, the default "1" to "n" is used. If multiple distinct blocks are listed as arguments, the code lines in that table will be repeated in the XML within those blocks.

3.3 Generating the intermediate XML file

The project assembled in the Sumo modeling software contains the model, the process units and other settings. The Excel file format representing the process code is just a convenient, user-friendly format to store everything needed by the software.

The equations contained in the Excel files are translated by the SMT to an intermediate XML format which contains the instructions for the Sumo numerical engine to perform the simulation. The XML file then is transformed to an actual executable file usable by the Sumo numerical engine.

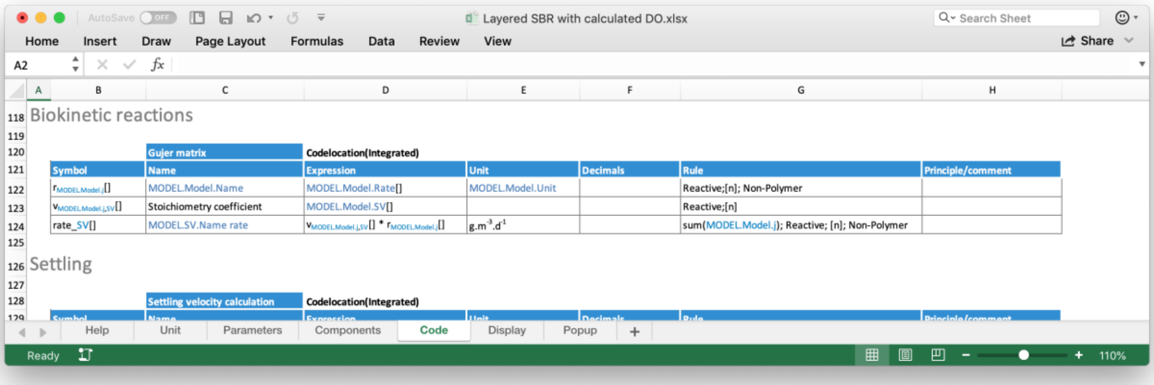
The SMT is used by Sumo automatically, but it can also be used manually. In the latter case the user should provide the *Root* element, discussed in section 2.4 Process unit hierarchy, of the project which contains the structure of the modeled plant.

4 Basic language elements

The following subsections contain the various SumoSlang elements in greater detail with examples.

4.1 Assignments

The most fundamental element of the SumoSlang is the assignment. Section 3.1.1 Simple table presented how an assignment is specified in a simple table. Figure 16 shows the table *Gujer matrix* on the *Code* sheet of a process unit.


Layered SBR with calculated DO.xlsx

Biokinetic reactions

	Gujer matrix	Codelocation(Integrated)				
Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
$r_{MODEL.Model.j}$[]	MODEL.Model.Name	MODEL.Model.Rate[]	MODEL.Model.Unit		Reactive;[n]; Non-Polymer	
$v_{MODEL.Model.j,SV}$[]	Stoichiometry coefficient	MODEL.Model.SV[]			Reactive;[n]	
rate_SV[]	MODEL.SV.Name rate	$v_{MODEL.Model.j,SV}$[] * $r_{MODEL.Model.j}$[]	$g.m^{-3}.d^{-1}$		sum(MODEL.Model.j); Reactive; [n]; Non-Polymer	

Settling

Settling velocity calculation Codelocation(Integrated)

Figure 16 Table containing assignments on the Code worksheet.

From the example the following information can be extracted:

- it is a simple table (no table type)
- the table represents the group of assignments named Gujer matrix
- the assignments will end up in the *Integrated* code location in the generated XML file
- the elements of the assignments are contained in the *Symbol* and the *Expression* columns. The three assignments in the example are:
    - $r_{MODEL.Model.j}$`[] = MODEL.Model.Rate[]`
        - the *Unit* of the left-hand side variable is coming from `MODEL.Model.Unit` (see 5.1 Expandable symbols)
        - it has two rules: `Reactive` and `[n]` (see 5.2 Rules)
    - $v_{MODEL.Model.j,SV}$`[] = MODEL.Model.SV[]`
        - it has two rules: `Reactive` and `[n]`
    - `rate_SV[] = `$v_{MODEL.Model.j,SV}$`[] * `$r_{MODEL.Model.j}$`[]`
        - the *Unit* of the left-hand side variable is explicitly given: g.m-3.d-1
        - it has three rules: `sum(MODEL.Model.j)`, `Reactive` and `[n]`

The most important columns of the table from the point of view of an assignment are *Symbol* and *Expression*. They contain the left-hand side and the right-hand side of the assignment; the other columns contain auxiliary information (e.g. the unit or the different filtering rules) used by the Sumo software.

The symbol on the left-hand side of the assignment is a variable while the right-hand side may contain variables, numerical values or constants, string constants (literals), functions and operators. Variable names should start with a letter, may contain numbers, comma characters (`,`), single dot characters (`.`) or single underscore characters (`_`) and may have subscript or superscript parts. The following variable name contains all the mentioned features:

$T_{local,max}\_P^{12.V1.0}\_x$

Double dot (`..`) and double underscore (`__`) are not allowed in variable names as they are namespacing keywords (see 7 Namespaces).

4.1.1 Operators

The assignments may use the following arithmetical, logical and relational operators:

Symbol	**Meaning**
+	addition
-	subtraction
*	multiplication
/	division
^	exponentiation
&&	logical AND
||	logical OR
!	logical NOT
==	relational Equal
!=	relational Not Equal
<	relational Less than
>	relational Greater than
<=	relational Less than or Equal
>=	relational Greater than or Equal

Table 1 Operators of SumoSlang

It is worth noting that the SMT automatically provides protection against division by zero errors by adding a very small number to every quotient.

4.2 Data types

The following data types are used in SumoSlang:

- Integer
- Real
- Boolean
- String
- Dimension—array size by a given dimension (only one at the moment)

The data type of a variable by default is Real. There is no type inference from the right-hand side (i.e. the SMT cannot determine the type of the left-hand side variable). The correct type should be specified by the process code author in the *Rule* column of a given code line, using one of the type names from the previous list.

4.3 Symbol roles

SumoSlang fits the symbols it finds in one of the following roles:

- **Constant**—various numerical constants (physical, chemical) used in calculations contained on the *Constants* worksheet of the `systemcode.xlsx`
- **Parameter**—simulation parameters modifiable by the user of the Sumo software; everything on *Parameters* worksheets except dimensions.
- **State variable (SV)**—dynamic simulation state variables; every variable matching a symbol found on the *Components* worksheet of either a process unit or a model. The following conditions should be met also:
    - the matching symbol on the *Components* worksheet of the model should have `Integrated` handling (see 5.2.2 Handlings)
    - a corresponding derivative of the symbol should be present in the process unit code
- **Derivative**—derivatives of state variables; every variable with the symbol equal to `d<state variable>_dt` where `<state variable>` is the place holder for a symbol with SV role
- **SystemState**—every other variable has this role (a better role name should be found).

The role of the various symbols is carried on to the XML file and is used by the simulation core of the Sumo software.

4.4 Functions

The *Functions* worksheet in `systemcode.xlsx` contains the function declarations used in process code. The usual mathematical functions (more precisely their C++ name) are listed in a separate table tagged as *Pure*. Functions in other tables are Sumo functions expressed in C++ code.

The function declaration (name and arguments) is in the *Symbol* column, its body is in the *Expression* column. The syntax is the following:

```
[type][array_sign] name(
          [type][array_sign] [name][;]
          [type][array_sign] [name]…)
```

where

- type is one of the following keywords
    - REAL—this is the default if type is omitted
    - INT—Integer type
    - BOOL—Boolean type
    - STRING—String type
- array_sign is the `[]` character pair
- name—any name starting with a letter and containing letters, numbers or underscores
- ; —argument delimiter

The elements in brackets are optional. Where type is missing REAL will be used. The following example shows the Average function declaration:

```
Average([] x; INT start; INT end)
```

where `Average` is the function name. The return type and array sign are missing which means that the function will return a REAL number. The missing type specification and the given array sign of the first argument `x` means that it is an array of REAL numbers. Both `start` and `end` are integer numbers.

5 Advanced language elements

The following sections describe how process code can be written in a very terse form. Terseness is achieved by the introduction of so-called expandables and filtering rules. Without these the process code would be much larger and more verbose. To see the difference, one can compare the *Code* worksheet of a process unit with the generated XML file which contains the fully expanded version of the process code.

5.1 Expandable symbols

The SMT recognizes variables written in a special, abbreviated or shorthand notation. During translation, these are replaced with a set of symbols contained in the process model. This means that instead of one code line there will be *n* code lines where *n* is the number of symbols included by the process called expansion.

The process code shipped with the Sumo software provides syntax highlighting of expandables as Figure 16 shows SV and the triplet of symbols starting with MODEL are colored blue.

5.1.1 SV

SV stands for state variable. A symbol containing this shorthand will be replaced with a set of symbols from the *Components* worksheet of the model. This set may contain all symbols from the model or a subset of them. Various filtering methods are available to specify which state variables are needed (see 5.2 Rules).

For state variables, besides rules, an alternative filtering method is available to write terser process code. These shorthand notations are also recognized by the SMT:

- `L.SV`—selects liquid phase state variables
- `G.SV`—selects gas phase state variables
- `S.SV`—selects solid phase state variables
- `sSV`—selects dissolved components (or small particle size state variables)
- `cSV`—selects colloidal components
- `xSV`—selects particulate components

The phase and particle size shorthand are composable (in this order). `L.sSV` for example means liquid dissolved components. These are keywords and the L, G, S, s, c, x prefixes are **not** taken from the content of the *Phase* and *Particle size* columns (please note the case difference of s, c, x).

The `SV` expandable is a shorthand of the `MODEL.SV.Symbol` notation (see 5.1.5 Triplet notation). Note that `L.SV` can be replaced with the rule `Phase(L)` and `xSV` with the rule `Particle size(X)`, (see section 5.2.3 Non-keyword rules) while the *Symbol* column would simply contain `SV`. In case of rule filtering, the argument of the rule should match the content of the *Phase* and *Particle size* columns (L, G, S in case of phase and S, C, X in case of particle size). If the column

Phase would contain `solid`, the rule would be `Phase(solid)` while the shorthand filtering method still would be `S.SV`.

It is important to mention that this alternative SV filtering method (`L.SV`, `xSV` etc.) works only with default models i.e., the first model attached to the process unit (see 8.1 Default model). If more than one models are attached to the process unit and the process code author wants to reference state variables from the second (third etc.) model, the full triplet syntax should be used, and the filtering should be implemented with rules instead of a shorthand. For example, `L.SV` in `MODEL_2` should be written as `MODEL_2.SV.Symbol` with `Phase(L)` rule.

5.1.2 PAR

`PAR` stands for parameters. This shorthand is replaced by symbols found on the *Parameters* worksheet of a model file like in the case of state variables. These are simulation parameters available to the user of the Sumo software for modification.

Filtering is possible by specifying a *Type*(...) selector in the *Rule* column in the process unit because the tables on the *Parameters* worksheet are grouped in *Type* groups.

`PAR` is a shorthand of the `MODEL.PAR.Symbol` triplet notation.

5.1.3 CVAR

`CVAR` stands for calculated variables. This shorthand is replaced by symbols found on the *Calculated variables* worksheet of a model file. Filtering is the same as in case of `PAR` shorthand: use a *Type*(...) selector in the Rule column in the process unit.

`CVAR` is a shorthand of the `MODEL.CVAR.Symbol` triplet notation.

5.1.4 SPC

`SPC` stands for species. This shorthand is replaced by symbols found on the *Species* worksheet of a model file. As *Species* contains a simple table without any grouping, the whole list of symbols is "pulled in" during expansion.
`SPC` is a shorthand of the `MODEL.SPC.Symbol` triplet notation.

5.1.5 Triplet notation

The triplet syntax allows a **general** way of referencing model parts in process code. This notation always has three elements:

- model identifier—represents the model ID or model variable name used on the *Unit* sheet of a process unit. This allows a name-independent reference to the model used by a process unit. Note that `MODEL` is not a keyword, it can be `PANCAKE` for example, if that ID is used on the *Unit* worksheet of the process unit to identify a model.
- worksheet name—selects a worksheet in the model file. Shorthand notation (`SV`, `PAR`, `CVAR`, `SPC`) or explicit names are allowed. The worksheet name may contain whitespaces.
- column name—selects a column in a code table. Shorthand notation or explicit names are allowed. If the column name contains white spaces the whole triplet should be placed in quotation marks (e.g. "`MODEL.PAR.Low limit`").

Figure 16 contains various examples of the triplet syntax usage. These are:

- $r_{\texttt{MODEL.Model.j}}$`[]`—where the subscript index of the variable `r` will be replaced with symbols taken from the default model file, worksheet *Model* and column *j*.
- `MODEL.Model.Rate[]`—where the expression will be replaced with symbols taken from the model file, worksheet *Model*, column *Rate*.

- `MODEL.SV.Name`—where the expression will be replaced by symbols taken from the default model file, worksheet *Components*, column *Name*. Note that the context of the `SV` shorthand is recognized, it represents a worksheet name, and it is replaced accordingly.
- `MODEL.Model.SV[]`—where the expression will be replaced with symbols taken from the default model file, worksheet *Model*, column name equal to symbols from the *Components* worksheet. This is a tricky expansion (see more in 5.3 Mechanism of expansion).

As the introduction of section 5.1 explained: the expansion means replacing one code line with many code lines.

5.2 Rules

The predominant usage of rules in process code lines is filtering equations "pulled in" from the model or other process units. Data types may also be specified in rules (see 4.2 Data types).

Filtering capabilities are related to attributes and handlings specified in process units (model files do not contain a *Unit* worksheet with attribute and handling specifications).

The evaluation of the *Rule* column results in a single Boolean value. If true, the code line (or its expansion) will be passed otherwise not.

The *Rule* column may contain several distinct rule elements separated by a semicolon. The rule result will be the composition of the partial results with the AND relational operator.

5.2.1 Attributes

Attributes are user defined symbols on the *Unit* worksheet of process units, or columns of the *Unit component* table on the *Structure* worksheet of a composite process unit. The attributes can be used in the *Rule* column of code lines (process unit or model) to skip some of them during expansion.

Attributes have an implicit Boolean data type. The SMT takes their value, or their negated value if they have a `Non-` prefix, to evaluate the rule in a given code line.

Section 2.4 Process unit hierarchy described parameter inheritance in a nutshell. Figure 17 shows how it is implemented with the help of attributes. Parameter inheritance means model parameter inheritance (there are other parameters defined in process units). Model parameters can be altered by process units and their children (if any) may want to specify which model parameters they need, hence the need of the two different inheritance methods.

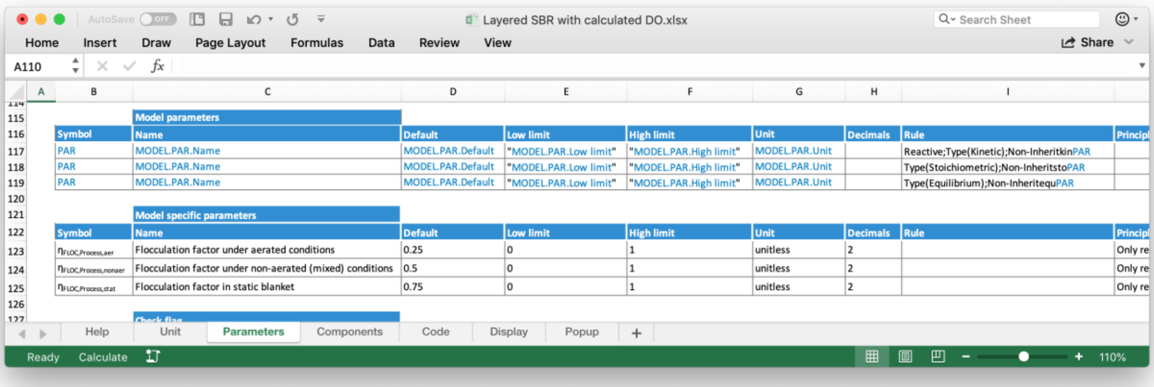
The *Rule* column of row 117 on the *Parameters* worksheet contains a `Non-InheritkinPAR` attribute while row 468 on the *Code* worksheet contains its counterpart, `InheritkinPAR`. The *Unit* worksheet of the process unit contains the definition of the `InheritkinPAR` attribute defaulting to `TRUE`.

Please note that the *Default* column of the inheritance table on the *Parameters* worksheet contains a reference to the model: `MODEL.PAR.Default` which means that model parameters in the process unit will be equal to values coming from the model. (This is shown with the dark green line on Figure 6.)

On the other hand, the *Expression* column on the *Code* worksheet contains a reference to the parent process unit: `Parent..PAR` which means that model parameters in the process unit will be equal to values coming from the parent. (This is shown with the light green line on Figure 6.)

When `InheritkinPAR` is `TRUE` (inheritance ON) the code lines from the *Code* sheet prevail, otherwise the code lines from the *Parameters* sheet (the `Non-` prefix means negation as described before). Quite simple.

It is worth noting how with a few lines of code the process code author can refer to a large number of code lines from the model, tailoring them as he or she sees fit.



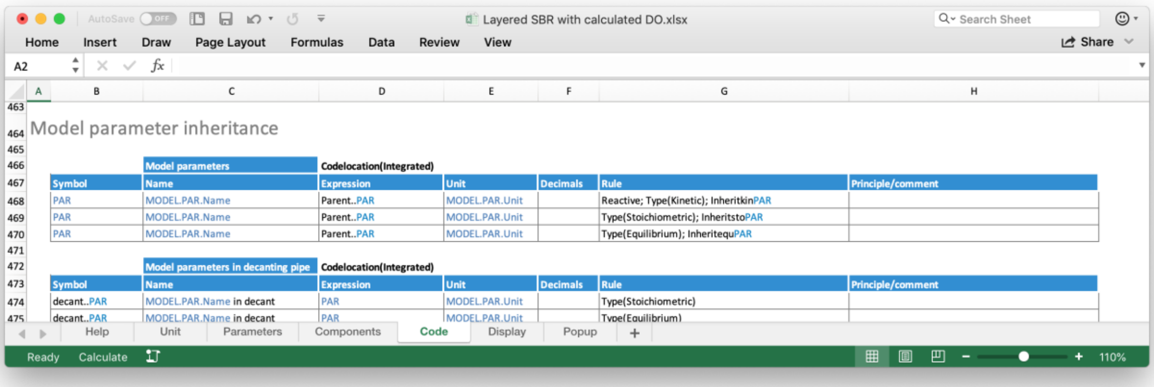
Layered SBR with calculated DO.xlsx

Model parameters

Symbol	Name	Default	Low limit	High limit	Unit	Decimals	Rule	Principl
PAR	MODEL.PAR.Name	MODEL.PAR.Default	"MODEL.PAR.Low limit"	"MODEL.PAR.High limit"	MODEL.PAR.Unit		Reactive;Type(Kinetic);Non-InheritkinPAR	
PAR	MODEL.PAR.Name	MODEL.PAR.Default	"MODEL.PAR.Low limit"	"MODEL.PAR.High limit"	MODEL.PAR.Unit		Type(Stoichiometric);Non-InheritstoPAR	
PAR	MODEL.PAR.Name	MODEL.PAR.Default	"MODEL.PAR.Low limit"	"MODEL.PAR.High limit"	MODEL.PAR.Unit		Type(Equilibrium);Non-InheritequPAR	

Model specific parameters

Symbol	Name	Default	Low limit	High limit	Unit	Decimals	Rule	Principl
$\eta_{FLOC,Process,aer}$	Flocculation factor under aerated conditions	0.25	0	1	unitless	2		Only re
$\eta_{FLOC,Process,nonaer}$	Flocculation factor under non-aerated (mixed) conditions	0.5	0	1	unitless	2		Only re
$\eta_{FLOC,Process,stat}$	Flocculation factor in static blanket	0.75	0	1	unitless	2		Only re



Layered SBR with calculated DO.xlsx

Model parameter inheritance

Model parameters — Codelocation(Integrated)

Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
PAR	MODEL.PAR.Name	Parent..PAR	MODEL.PAR.Unit		Reactive; Type(Kinetic); InheritkinPAR	
PAR	MODEL.PAR.Name	Parent..PAR	MODEL.PAR.Unit		Type(Stoichiometric); InheritstoPAR	
PAR	MODEL.PAR.Name	Parent..PAR	MODEL.PAR.Unit		Type(Equilibrium); InheritequPAR	

Model parameters in decanting pipe — Codelocation(Integrated)

Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
decant..PAR	MODEL.PAR.Name in decant	PAR	MODEL.PAR.Unit		Type(Stoichiometric)	
decant..PAR	MODEL.PAR.Name in decant	PAR	MODEL.PAR.Unit		Type(Equilibrium)	

Figure 17 Parameter inheritance implemented with attributes on the Parameters *and* Code *worksheets.*

5.2.2 Handlings

Handlings specify how state variables should be treated during dynamic simulations. This is a filtering method of code lines in process units. They are defined in the *Handling* column on the *Components* worksheet of models.

Handlings may have one of the following values:

- Integrated
  - The Component marked as "Integrated" will be declared as a true State Variable and will have a derivative. This is the default for most components in a dynamic model.
- Set
  - The Component marked as "Set" will become a parameter, will not have a derivative. An example is DO – when just an input value is desired (e.g. 2 mg/L), it is not necessary to design and tune a controller, but change this Component to Set and assign a value.
- Algebraic
  - The total rate (ν_{ij}*r_j) will be calculated (e.g. total CO2 production rate) without integrating the variable itself, so it is not present in the code.
- Balancing

- State variables with this handling are not taken in consideration during expansion, they are simply used to mass balance the model components in the model Excel sheet. An example is N2 – if dissolved and gaseous nitrogen is not important from the process standpoint but it is important to see that the N balance closes, Balancing rule can be used. The variable will not be present in the final compiled code.

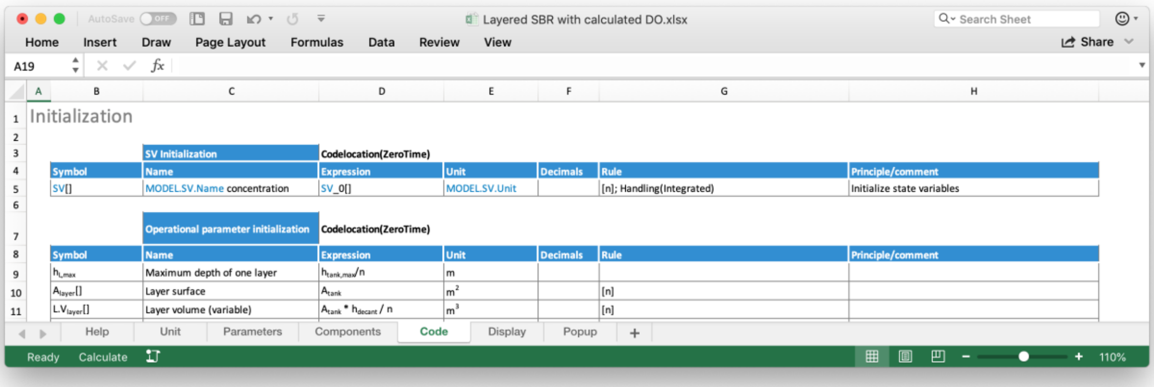
These are the default handling values of state variables. The default values can be changed on the *Unit* and *Structure* worksheets of process units. The handling symbols specified in the process unit refer to state variables present in the model.

5.2.3 Non-keyword rules

Section 3.2.1 Table explains the usage of *Type*(...) table grouping info in tandem with a `Type(…)` rule. The rule is an example of the more general table descriptor rules, namely the `Type(arg`$_1$`; … arg`$_n$`)` construct. This kind of rule allows code lines to be included from all tables matching the arguments. The source of the inclusion is usually the model, but it can be the parent or other direct child process units.

The `Handling(…)` rule is an example of the more general table header rules. A table header rule consists of a column name and an argument list representing distinct values from that column. The context, i.e. the worksheet where the column should be present, is determined by the SMT from the *Symbol* column of the process unit code line.

Figure 18 shows an example of table header rule in row 5 of the *Code* worksheet of a process unit.



Initialization

	SV Initialization	Codelocation(ZeroTime)				
Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
SV[]	MODEL.SV.Name concentration	SV_0[]	MODEL.SV.Unit		[n]; Handling(Integrated)	Initialize state variables

	Operational parameter initialization	Codelocation(ZeroTime)				
Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
$h_{L,max}$	Maximum depth of one layer	$h_{tank,max}/n$	m			
$A_{layer}[]$	Layer surface	A_{tank}	m^2		[n]	
$LV_{layer}[]$	Layer volume (variable)	$A_{tank} * h_{decant} / n$	m^3		[n]	

Figure 18 Handling(...) rule in row 5 of the Code *worksheet.*

The worksheet referred is *Components* of the model file because the *Symbol* column contains an `SV` (see 5.1.1 SV). The column name in the table header should be *Handling* and its content `Integrated`. The SMT will include only the rows where *Handling* contains the value `Integrated`.

It should be mentioned that in case of the worksheet contains multiple tables, all tables would be parsed by the rule, but only rows from matching tables (having *Handling* column) will be allowed.

If the `Handling(...)` rule contains more than one argument, all model rows with a matching *Handling* value would be allowed in the expansion. This means an OR relationship between the rule arguments.

The SumoSlang is flexible enough to let the user define arbitrary table descriptor (mostly table tags) or table header rules. It is important to mention that *Type* and *Handling* are not keywords in SumoSlang (note, however, that *Handling* appears as table type keyword on the *Unit* worksheet of process units). Any rule written in a function syntax, i.e. `name(arg₁; … argₙ)`, is tried to be interpreted as a table descriptor rule then a table header rule. This allows the user to introduce other grouping and filtering names than those supplied with the Sumo process code.

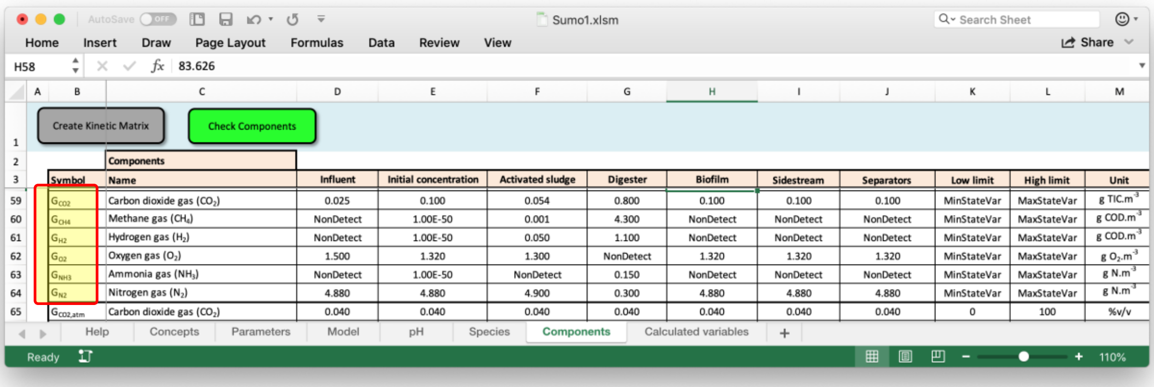
5.2.4 Exempt, Only

These two keywords are explicit symbol filters during expansion. They take a list of symbols as arguments and may contain expandables.

Exempt excludes the assignments resulting after expansion which have a left-hand side symbol listed in the arguments.

Only includes the assignments resulting after expansion which have left-hand side symbol listed in the arguments.

The following example shows a model which has some gas phase state variables and a hypothetical process unit using `Exempt(...)` and `Only(...)` rules to skip or include some of these state variables.

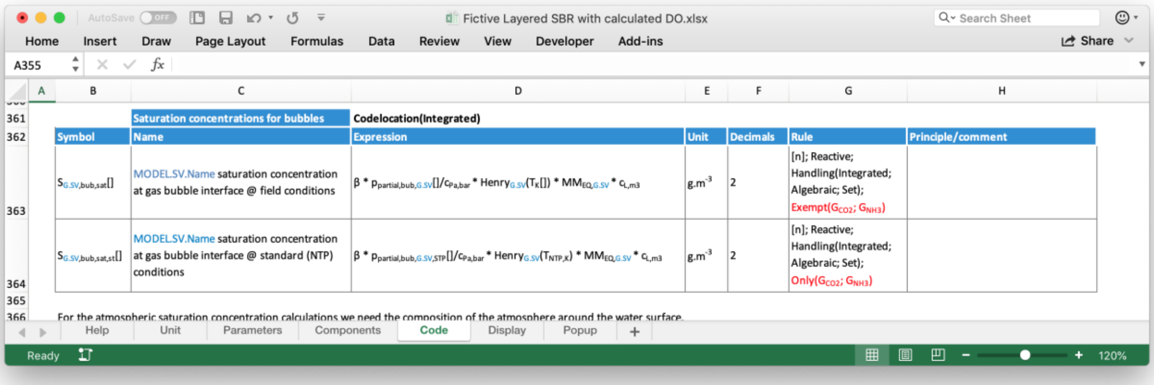


Sumo1.xlsm

H58 83.626

Create Kinetic Matrix | Check Components

Row	Symbol	Name	Influent	Initial concentration	Activated sludge	Digester	Biofilm	Sidestream	Separators	Low limit	High limit	Unit
2		Components										
59	G_{CO2}	Carbon dioxide gas (CO_2)	0.025	0.100	0.054	0.800	0.100	0.100	0.100	MinStateVar	MaxStateVar	g TIC.m^{-3}
60	G_{CH4}	Methane gas (CH_4)	NonDetect	1.00E-50	0.001	4.300	NonDetect	NonDetect	NonDetect	MinStateVar	MaxStateVar	g COD.m^{-3}
61	G_{H2}	Hydrogen gas (H_2)	NonDetect	1.00E-50	0.050	1.100	NonDetect	NonDetect	NonDetect	MinStateVar	MaxStateVar	g COD.m^{-3}
62	G_{O2}	Oxygen gas (O_2)	1.500	1.320	1.300	NonDetect	1.320	1.320	1.320	MinStateVar	MaxStateVar	g O_2.m^{-3}
63	G_{NH3}	Ammonia gas (NH_3)	NonDetect	1.00E-50	NonDetect	0.150	NonDetect	NonDetect	NonDetect	MinStateVar	MaxStateVar	g N.m^{-3}
64	G_{N2}	Nitrogen gas (N_2)	4.880	4.880	4.900	0.300	4.880	4.880	4.880	MinStateVar	MaxStateVar	g N.m^{-3}
65	$G_{CO2,atm}$	Carbon dioxide gas (CO_2)	0.040	0.040	0.040	0.040	0.040	0.040	0.040	0	100	%v/v



Fictive Layered SBR with calculated DO.xlsx

A355

Row	Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
361		Saturation concentrations for bubbles	Codelocation(Integrated)				
363	$S_{G.SV,bub,sat}[]$	MODEL.SV.Name saturation concentration at gas bubble interface @ field conditions	$\beta * p_{partial,bub,G.SV}[]/c_{Pa,bar} * Henry_{G.SV}(T_K[]) * MM_{EQ,G.SV} * c_{L,m3}$	g.m^{-3}	2	[n]; Reactive; Handling(Integrated; Algebraic; Set); Exempt(G_{CO2}; G_{NH3})	
364	$S_{G.SV,bub,sat,st}[]$	MODEL.SV.Name saturation concentration at gas bubble interface @ standard (NTP) conditions	$\beta * p_{partial,bub,G.SV,STP}[]/c_{Pa,bar} * Henry_{G.SV}(T_{NTP,K}) * MM_{EQ,G.SV} * c_{L,m3}$	g.m^{-3}	2	[n]; Reactive; Handling(Integrated; Algebraic; Set); Only(G_{CO2}; G_{NH3})	

366 For the atmospheric saturation concentration calculations we need the composition of the atmosphere around the water surface.

Figure 19 Exempt and Only rules in action.

The model in the example contains 6 gas phase state variables (in rows 59-64). The `Exempt(GCO2; GNH3)` rule in row 363 results in the following expansion of the symbol:

$S_{GCH4,bub,sat}[\,]$
$S_{GH2,bub,sat}[\,]$
$S_{GO2,bub,sat}[\,]$
$S_{GN2,bub,sat}[\,]$

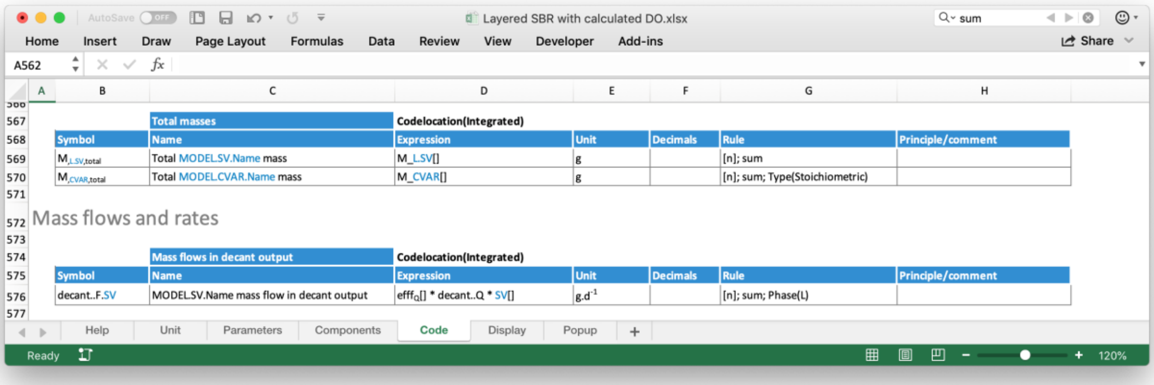
while the symbol in row 364 will be expanded in the following variables:

```
S_GCO2,bub,sat[]
S_GNH3,bub,sat[]
```

5.2.5 Other rules

Other less commonly used rules are:

- `Step(arg)`—in case of array assignments it specifies the incrementation step of the loop variable (as in section 3.1.2 Arrays was described, the elements of the array are assigned in a loop). The step can be positive or negative.
- `Call`—used in event handling (see 6.4 Event handling)
- `sum`—array summation. The following example shows how to calculate the sum of array elements given in the *Expression* column into scalar variables given in the *Symbol* column.



Layered SBR with calculated DO.xlsx

	Total masses	Codelocation(Integrated)				
Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
$M_{L.SV,total}$	Total MODEL.SV.Name mass	M_L.SV[]	g		[n]; sum	
$M_{CVAR,total}$	Total MODEL.CVAR.Name mass	M_CVAR[]	g		[n]; sum; Type(Stoichiometric)	

Mass flows and rates

	Mass flows in decant output	Codelocation(Integrated)				
Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
decant..F.SV	MODEL.SV.Name mass flow in decant output	$\mathrm{efff_Q[]}$ * decant..Q * SV[]	$g.d^{-1}$		[n]; sum; Phase(L)	

Figure 20 Code lines with sum rule stating that all elements of the arrays given in the Expression *column should be added. The symbol in these cases represents a scalar variable.*

- `sum(arg)`, `mul(arg)`—summation, multiplication of expansion terms (see section 6.1 Summation, multiplication)
- `sum(<number>)`—array initialization rule with additional distribution handlings. This rule distributes the values of the array so that their sum will be equal to the specified `<number>`. This rule is meant for the simulation engine and instructs it how to set the initial values of array elements. There are four auxiliary keywords to this `sum(n)` rule known as handlings:
    - `Free`—specifies that the elements of the array can be distributed freely to obtain the desired sum
    - `Head`—specifies that the first element should be equal to the sum (the remaining elements will be 0)
    - `Tail`—specifies that the last element should be equal to the sum (the remaining elements will be 0)
    - `Equal`—specifies that the elements should be distributed evenly to give the desired sum
- For example an array may have the following rules: `[n]; sum(1); Equal`. The `sum(1)` and the `Equal` parts instructs the simulation engine to distribute the array elements evenly so that their sum will be 1. If `n` is 4 the elements of the array will be `[.25, .25, .25, .25]`. The array initialization and distribution rule are handy shortcuts.

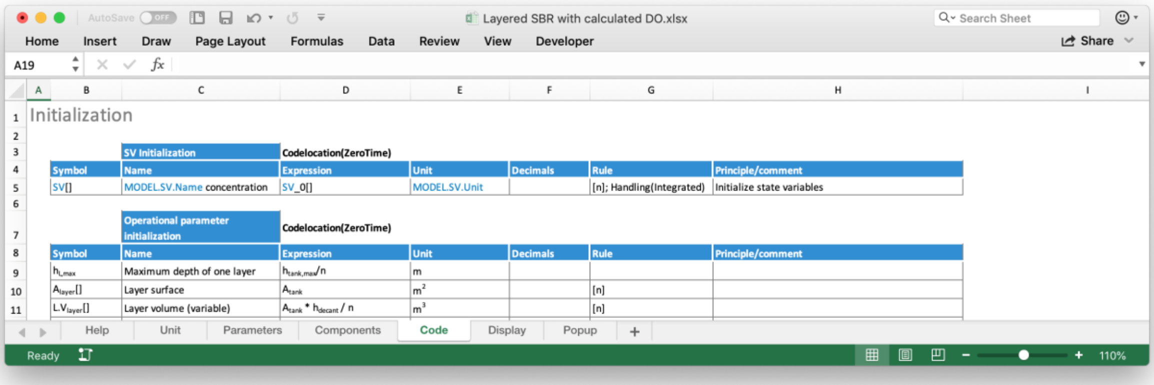
5.3 Mechanism of expansion

Section 5.1 Expandable symbols listed the available expandable shorthand and triplet notations and discussed how with the help of these symbols shorter and terser process code can be written. This section describes how the shorthand symbol replacement works, the rules governing the process, giving detailed examples. The transformation of expandable symbols to the final variable names needed in calculations is called expansion.

5.3.1 Simple expansion

The expansion is always driven by the content of the *Symbol* column of code tables, meaning that it selects the set of symbols used in expansion from a model worksheet. These symbols then will be the base of expansion for other columns which effectively means that expandables in other columns should match the *Symbol* column.

Figure 21 shows a simple case, where row 5 contains various state variable shorthand elements.



Initialization

	SV Initialization	Codelocation(ZeroTime)				
Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
SV[]	MODEL.SV.Name concentration	SV_0[]	MODEL.SV.Unit		[n]; Handling(Integrated)	Initialize state variables

	Operational parameter initialization	Codelocation(ZeroTime)				
Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
$h_{L,max}$	Maximum depth of one layer	$h_{tank,max}/n$	m			
A_{layer}[]	Layer surface	A_{tank}	m^2		[n]	
LV_{layer}[]	Layer volume (variable)	$A_{tank} * h_{decant} / n$	m^3		[n]	

Figure 21 SV shorthand example in row 5.

Taken Sumo1 as model, SV[] will be expanded by replacing it with all symbols taken from the *Components* worksheet of Sumo1, transforming that single row into the following table:

Nr.	Symbol	Name	Expression	Unit
1	S_{VFA}[]	Volatile fatty acids (VFA) concentration	S_{VFA}_0[]	g COD.m-3
…				
55	H[]	Enthalpy concentration	H_0[]	MJ.m-3

Table 2 State variables after simple expansion.

As mentioned before, the expandables in columns other than *Symbol* should match, because values are taken from the same worksheet and the same row as *Symbol* specifies. In the example this condition is met because:

- the *Name* column contains a triplet expandable which instructs the SMT to take a value from the model file, worksheet *Components*, column *Name*, and the row specified by the current symbol (S_{VFA} through H)
- the *Expression* column contains an SV expandable which instructs the SMT to take a value from the model file, worksheet *Components*, column *Symbol*
- the *Unit* column is like *Name*, but using the *Unit* column from the model

This simple case is spiced a bit with a `Handling(Integrated)` rule which tells the SMT to include only those rows from the model that have `Integrated` in their *Handling* column. These are the state variables from S_{VFA} to `H`. The replacement content is shown in blue.

The expansion would be impossible if a non-Symbol column would contain a non-matching expandable, for example if *Name* would contain `MODEL.CVAR.Name` where `CVAR` would instruct the SMT to take values from the *Calculated variables* worksheet and the row specified by the current symbol. But the symbols S_{VFA} through `H` are not present among the symbols of the *Calculated variables* worksheet.

Using expandables in the *Name* and *Unit* columns has the important benefit of easy maintainability. If something changes in the model nothing has to be changed in process units.

5.3.2 Advanced expansion

The *Symbol* column contains more than one expandable. In this case the other columns should contain matching expandables to those in the *Symbol* column.
The expandables in the *Symbol* column are expanded one by one producing a set of assignments that has the cardinality equal to the Cartesian product of the symbol count on model worksheets participating in the expansion. Too many expandables would produce a huge number of assignments (the process code shipped with Sumo has no more than two expandables per symbol). See example in section 6.2 Gujer matrix calculations and 6.3pH calculation without summation.

5.3.3 Even more advanced expansion

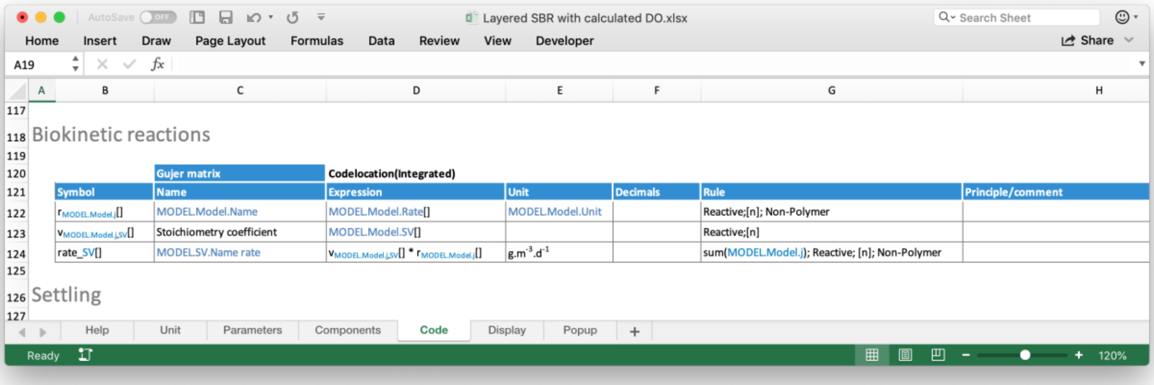
The *Symbol* column contains one or more expandables but *Expression* contains matching and non-matching expandables. In this case the non-matching expandables should be preprocessed with rules before expansion, eliminating the non-matching expandables. After that the procedure described in the previous subsection can be started. See the example in section 6.1 Summation, multiplication.

6 Special calculations

This chapter contains examples of advanced expansion. The examples are taken from a real process unit, `Layered SBR with calculated DO.xlsx`, and `Sumo1.xlsm` is used as model.

6.1 Summation, multiplication

Row 124 of the following process code table shows an example of 5.3.3 Even more advanced expansion. *Symbol* contains an expandable (`SV`) and *Expression* contains a matching `SV` and a non-matching `MODEL.Model.j` expandable.



Biokinetic reactions

	Gujer matrix	Codelocation(Integrated)				
Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
$r_{MODEL.Model.j}$[]	MODEL.Model.Name	MODEL.Model.Rate[]	MODEL.Model.Unit		Reactive;[n]; Non-Polymer	
$v_{MODEL.Model.j,SV}$[]	Stoichiometry coefficient	MODEL.Model.SV[]			Reactive;[n]	
rate_SV[]	MODEL.SV.Name rate	$v_{MODEL.Model.j,SV}$[] * $r_{MODEL.Model.j}$[]	$g.m^{-3}.d^{-1}$		sum(MODEL.Model.j); Reactive; [n]; Non-Polymer	

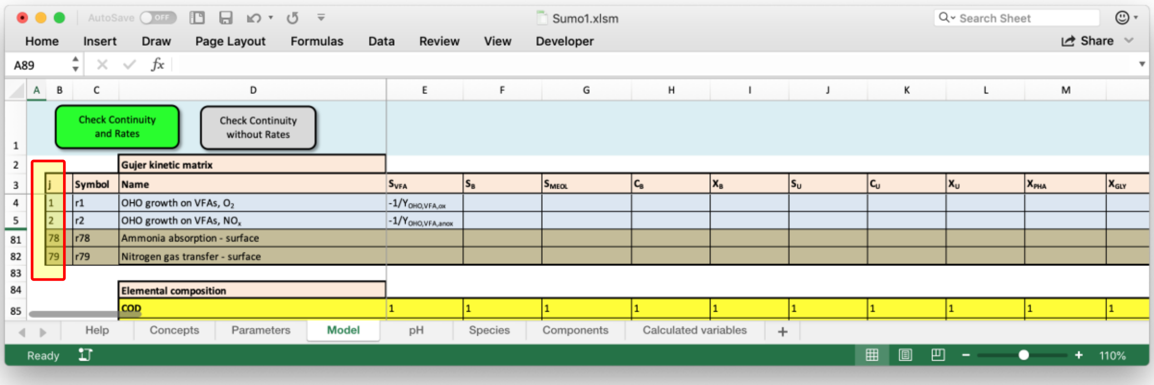
Settling

Figure 22 Advanced expansion examples in the Gujer matrix table.

In the example the non-matching expandable should be eliminated before expansion. This is done by the `sum(MODEL.Model.j)` rule, which instructs the SMT to unwind the expression in a sum by the `MODEL.Model.j` index as follows:

```
rate_SV[] = v₁,sv[]*r₁[] + v₂,sv[]*r₂[] + … + v₇₉,sv[]*r₇₉[]
```

The index `MODEL.Model.j` is taken from the *Model* worksheet, column *j* of the default model containing numbers from 1 to 79 which are substituted in the summation.



		Gujer kinetic matrix										
j	Symbol	Name	S_{VFA}	S_B	S_{MEOL}	C_B	X_B	S_U	C_U	X_U	X_{PHA}	X_{GLY}
1	r1	OHO growth on VFAs, O_2	$-1/Y_{OHO,VFA,ox}$									
2	r2	OHO growth on VFAs, NO_x	$-1/Y_{OHO,VFA,anox}$									
78	r78	Ammonia absorption - surface										
79	r79	Nitrogen gas transfer - surface										

Elemental composition										
COD	1	1	1	1	1	1	1	1	1	1

Figure 23 The indexer column j on the Model sheet of Sumo1.xlsx.

After this step a simple expansion can be executed because the expression contains only matching `SV` expandables (same is true for the *Name*) column. The result of the expansion is shown in the following table:

Nr.	Symbol	Name	Expression	Unit
1	rate_S_{VFA}[]	VFAs concentration	$v_{1,SVFA}$[]*r_1[] + … + $v_{79,SVFA}$[]*r_{79}[]	g.m-3.d-1
…				
64	rate_G_{N2}[]	Nitrogen gas concentration	$v_{1,GN2}$[]*r_1[] + … + $v_{79,GN2}$[]*r_{79}[]	g.m-3.d-1

Table 3 Result of expansion of row 124.

The *Unit* sheet does not contain an expandable, it's content will be preserved. Please note, that only the blue colored symbols were the result of the expansion.

6.2 Gujer matrix calculations

In the previous example the expandable summation was taken out of context. If the whole code table is taken in consideration it reveals calculation involving the so-called Gujer matrix.

The result of expanding rows 122-124 is the following set of assignments:

- in row 122 the columns contain one matching expandable represented by the `MODEL.Model.<column name>` notation, meaning that the SMT can take values from columns of the *Model* worksheet. In this case *Symbol* takes values from the *j* column while *Expression* from the *Rate* column of the worksheet *Model*.

Nr.	Symbol	Name	Expression	Unit
1	r_1[]	OHO growth on VFAs, O2	{$\mu_{OHO,T}$ * X_{OHO} * $Msat_{SVFA,KVFA}$ * $Msat_{SO2,KO2,OHO}$ * $Msat_{SNHx,KNHx,BIO}$ * $Msat_{SPO4,KPO4,BIO}$ * $Msat_{SCAT,KCAT}$ * $Msat_{SAN,KAN}$ * $Bellinh_{pH}$}[]	g.m-3.d-1
…				
79	r_{79}[]	Nitrogen gas transfer - surface	{$k_La_{GN2,sur}$ * ($S_{GN2sur,sat}$ - S_{N2})}[]	g.m-3.d-1

*Table 4 Result of expanding row 122. The curly braces in **Expression** indicate that the array sign is applied to the whole expression where it is the case.*

- in row 123 *Symbol* contains two expandables. *Expression* contains two expandables as well but in a tricky way: there is the matching triplet syntax expandable and, embedded in it, the `SV` column expandable. This is OK because *Symbol* specifies the same two worksheets: *Model* and *Components* to work with. The column part of the triplet syntax will be expanded according to its matching *Symbol* expandable (i.e. to `MODEL.Model.SVFA` when the *Symbol* expansion is $v_{1,SVFA}$`[]` and so on). The result of the expansion is (taking only the columns with expandables):

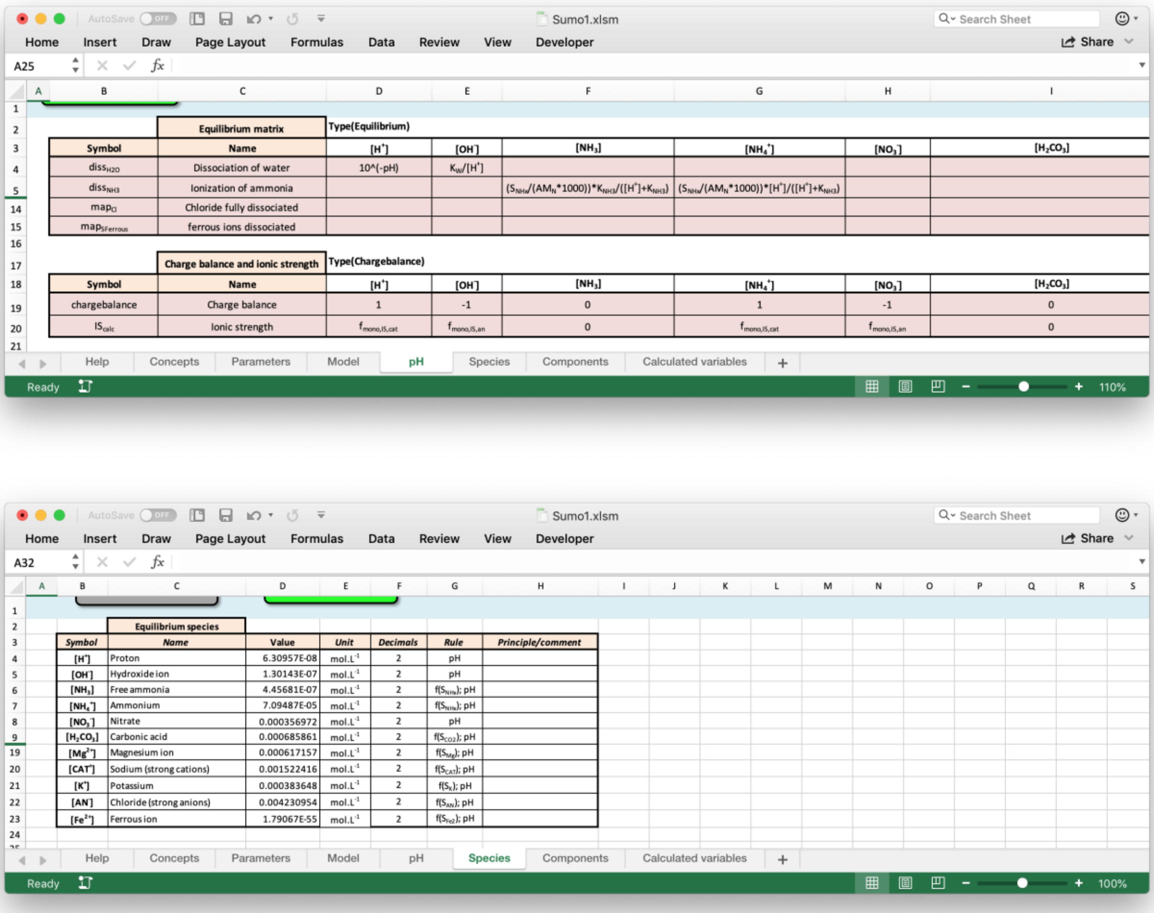
Nr.	Symbol	Expression	Unit
1	$v_{1,SVFA}[]$	$-1/Y_{OHO,VFA,ox}$	g.m-3.d-1
…			
79	$v_{79,SVFA}[]$		g.m-3.d-1
80	$v_{1,SB}[]$		g.m-3.d-1
…			
158	$v_{79,SB}[]$		g.m-3.d-1
…			
5056	$v_{79,GN2}[]$		g.m-3.d-1

Table 5 The result of expanding row 123.

- for row 124 see section 6.1 Summation, multiplication.

6.3 pH calculation

In pH calculations the relevant worksheets of the model are *pH* and *Species* shown on the next picture:



Equilibrium matrix — Type(Equilibrium)

Symbol	Name	$[H^+]$	$[OH^-]$	$[NH_3]$	$[NH_4^+]$	$[NO_3^-]$	$[H_2CO_3]$
$diss_{H2O}$	Dissociation of water	10^(-pH)	$K_W/[H^+]$				
$diss_{NH3}$	Ionization of ammonia			$(S_{NHx}/(AM_N*1000))*K_{NH3}/([H^+]+K_{NH3})$	$(S_{NHx}/(AM_N*1000))*[H^+]/([H^+]+K_{NH3})$		
map_{Cl}	Chloride fully dissociated						
$map_{SFerrous}$	ferrous ions dissociated						

Charge balance and ionic strength — Type(Chargebalance)

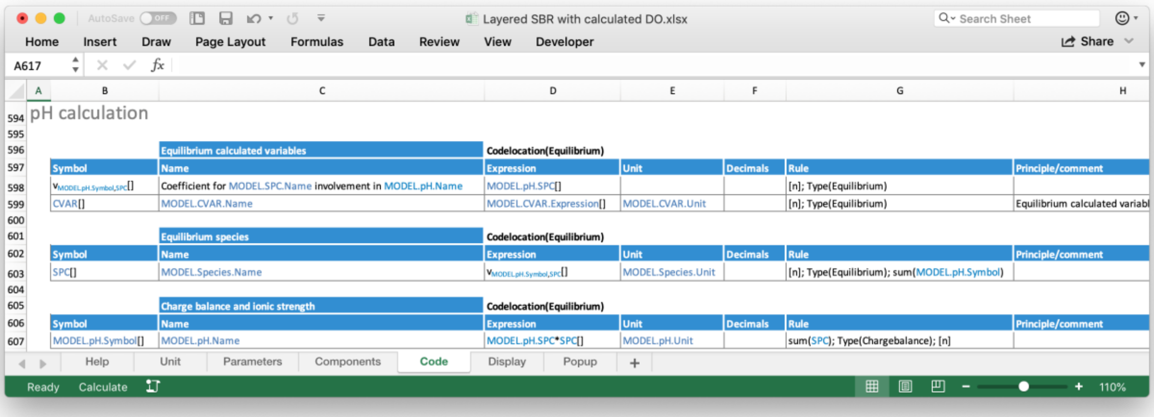
Symbol	Name	$[H^+]$	$[OH^-]$	$[NH_3]$	$[NH_4^+]$	$[NO_3^-]$	$[H_2CO_3]$
chargebalance	Charge balance	1	-1	0	1	-1	0
IS_{calc}	Ionic strength	$f_{mono,IS,cat}$	$f_{mono,IS,an}$	0	$f_{mono,IS,cat}$	$f_{mono,IS,an}$	0

Equilibrium species

Symbol	Name	Value	Unit	Decimals	Rule	Principle/comment
$[H^+]$	Proton	6.30957E-08	$mol.L^{-1}$	2	pH	
$[OH^-]$	Hydroxide ion	1.30143E-07	$mol.L^{-1}$	2	pH	
$[NH_3]$	Free ammonia	4.45681E-07	$mol.L^{-1}$	2	$f(S_{NHx})$; pH	
$[NH_4^+]$	Ammonium	7.09487E-05	$mol.L^{-1}$	2	$f(S_{NHx})$; pH	
$[NO_3^-]$	Nitrate	0.000356972	$mol.L^{-1}$	2	pH	
$[H_2CO_3]$	Carbonic acid	0.000685861	$mol.L^{-1}$	2	$f(S_{CO2})$; pH	
$[Mg^{2+}]$	Magnesium ion	0.000617157	$mol.L^{-1}$	2	$f(S_{Mg})$; pH	
$[CAT^+]$	Sodium (strong cations)	0.001522416	$mol.L^{-1}$	2	$f(S_{CAT})$; pH	
$[K^+]$	Potassium	0.000383648	$mol.L^{-1}$	2	$f(S_K)$; pH	
$[AN^-]$	Chloride (strong anions)	0.004230954	$mol.L^{-1}$	2	$f(S_{AN})$; pH	
$[Fe^{2+}]$	Ferrous ion	1.79067E-55	$mol.L^{-1}$	2	$f(S_{Fe2})$; pH	

Figure 24 The **pH** and **Species** worksheets of the Sumo1 model. Please note that some rows are hidden.

Process units can reference the content of these worksheets with shortcut symbols and the triplet notation. Figure 25 show the pH calculation section of a process unit:


Layered SBR with calculated DO.xlsx

pH calculation

	Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
596		Equilibrium calculated variables	Codelocation(Equilibrium)				
597	Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
598	$v_{MODEL.pH.Symbol,SPC}$[]	Coefficient for MODEL.SPC.Name involvement in MODEL.pH.Name	MODEL.pH.SPC[]			[n]; Type(Equilibrium)	
599	CVAR[]	MODEL.CVAR.Name	MODEL.CVAR.Expression[]	MODEL.CVAR.Unit		[n]; Type(Equilibrium)	Equilibrium calculated variabl
601		Equilibrium species	Codelocation(Equilibrium)				
602	Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
603	SPC[]	MODEL.Species.Name	$v_{MODEL.pH.Symbol,SPC}$[]	MODEL.Species.Unit		[n]; Type(Equilibrium); sum(MODEL.pH.Symbol)	
605		Charge balance and ionic strength	Codelocation(Equilibrium)				
606	Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
607	MODEL.pH.Symbol[]	MODEL.pH.Name	MODEL.pH.SPC*SPC[]	MODEL.pH.Unit		sum(SPC); Type(Chargebalance); [n]	

Figure 25 The pH calculation section on the ***Code*** *sheet of a process unit.*

The three tables contain the following advanced expansions:

- in row 598 *Symbol* contains two expandables: `MODEL.pH.Symbol` and `SPC`. The *Name* column contains also two matching expandables which will not cause any problem. Expression has two expandables in the tricky way, explained in the previous section. This indicates that the result will be the same bloated table of assignments as in the previous example:

Nr.	Symbol	Name	Expression
1	$v_{dissH2O,[H+]}$	Coefficient for Proton involvement in Dissociation of water	10^(-pH)
…			
12	$v_{mapSFerrous,[H+]}$	Coefficient for Proton involvement in ferrous ions dissociated	
13	$v_{dissH2O,[OH-]}$	Coefficient for Hydroxide ion involvement in Dissociation of water	KW/[H+]
…			
24	$v_{mapSFerrous,[OH-]}$	Coefficient for Hydroxide ion involvement in ferrous ions dissociated	
…			
240	$v_{mapSFerrous,[Fe2+]}$	Coefficient for Ferrous ion involvement in ferrous ions dissociated	S_{Fe2}/(AM_{Fe}*1000)

Table 6 Result of expanding row 598.

- in row 599 a simple expansion can be performed with a simple result set like in the Table 2, but instead of state variables the result will contain calculated variables of type `Equilibrium`.
- in row 603 *Symbol* contains one expandable, but *Expression* contains a matching and a non-matching expandable. The non-matching should be eliminated by a rule, in the case by the rule `sum(MODEL.pH.Symbol)`. First, the summation result will be:

```
SPC[] = v_dissH2O,SPC[] + ... + v_mapSFerrous,SPC[]
```

and after expansion the result is the following list:

Nr.	Symbol	Name	Expression	Unit
1	[H+][]	Proton	$v_{dissH2O,[H+]}$[] + … + $v_{mapsFerrous,[H+]}$[]	mol.L-1
…				
20	[Fe2+][]	Ferrous ion	$v_{dissH2O,[Fe2+]}$[] + … + $v_{mapsFerrous,[Fe2+]}$[]	mol.L-1

Table 7 Result of expanding row 603.

- in row 607 *Symbol* contains one expandable, but *Expression*, again, contains a matching and a non-matching expandable. The elimination of the non-matching expandable is performed by the `sum(SPC)` rule. First, the summation result will be:

```
MODEL.pH.Symbol[] = MODEL.pH.[H+] * [H+][] + … + MODEL.pH.[Fe2+] *
[Fe2+][]
```

  The `Type(Chargebalance)` rule selects the second table on the *pH* worksheet and the resulting list of assignments will be:

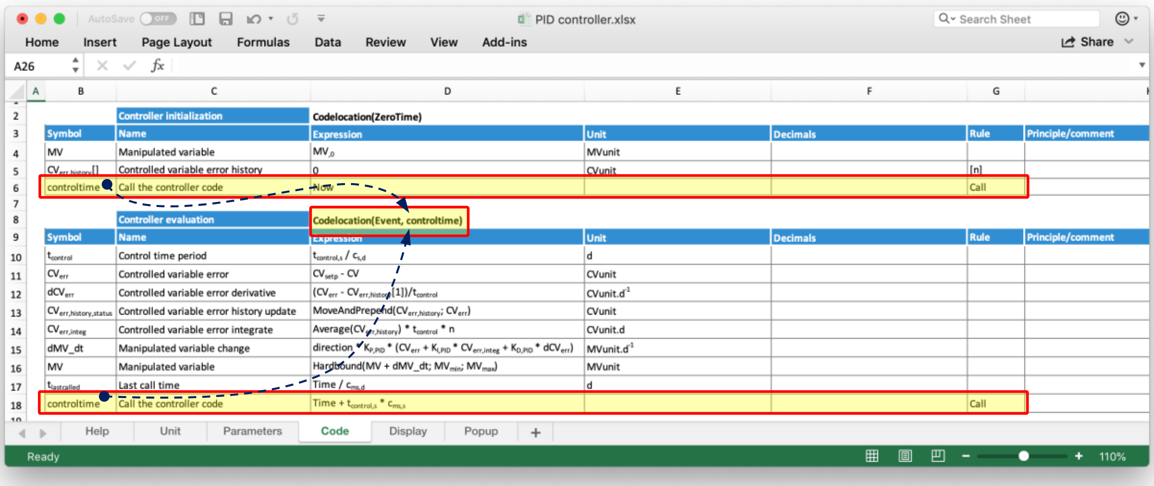
Nr.	Symbol	Name	Expression	Unit
1	chargebalance[]	Charge balance	1 * [H+][] + … + 2 * [Fe2+][]	mol.L-1
2	IScalc[]	Ionic strength	$f_{mono,IS,cat}$ * [H+][] + … + $f_{di,IS,cat}$ * [Fe2+][]	mol.L-1

Table 8 Result of expanding row 607.

6.4 Event handling

Events in SumoSlang are represented by a chain of function calls. An event function is defined in the `Event` code location. The name of the function is the section part of the `Codelocation(…)` argument. The event function has one argument usually to pass timing information.

The event function can be called in process code by setting *Symbol* to the function name, *Expression* to the function argument and specifying a `Call` rule, as the next example shows:



	Controller initialization	Codelocation(ZeroTime)				
Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
MV	Manipulated variable	$MV_{,0}$	MVunit			
$CV_{err,history}$[]	Controlled variable error history	0	CVunit		[n]	
controltime	Call the controller code	Now			Call	

	Controller evaluation	Codelocation(Event, controltime)				
Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
$t_{control}$	Control time period	$t_{control,s}$ / $c_{s,d}$	d			
CV_{err}	Controlled variable error	CV_{setp} - CV	CVunit			
dCV_{err}	Controlled variable error derivative	(CV_{err} - $CV_{err,history}$[1])/$t_{control}$	CVunit.d^{-1}			
$CV_{err,history,status}$	Controlled variable error history update	MoveAndPrepend($CV_{err,history}$; CV_{err})	CVunit			
$CV_{err,integ}$	Controlled variable error integrate	Average($CV_{err,history}$) * $t_{control}$ * n	CVunit.d			
dMV_dt	Manipulated variable change	direction * $K_{P,PID}$ * (CV_{err} + $K_{I,PID}$ * $CV_{err,integ}$ + $K_{D,PID}$ * dCV_{err})	MVunit.d^{-1}			
MV	Manipulated variable	Hardbound(MV + dMV_dt; MV_{min}; MV_{max})	MVunit			
$t_{lastcalled}$	Last call time	Time / $c_{ms,d}$	d			
controltime	Call the controller code	Time + $t_{control,s}$ * $c_{ms,s}$			Call	

Figure 26 Event call example.

In this example the event function is defined in the table named Controller evaluation, and it is called in two places: in row 6 of the Controller initialization table with the argument Now, and in row 18 of the function definition table with the timing argument. The latter means that the event calls form an infinite loop. `Now` and `Time` are keywords representing the time at the start of the simulation and the current time at every call.

7 Namespaces

Namespaces are qualifiers or prefixes of symbols used in assignments. These prefixes usually are not visible in the process code, only in the generated XML file. One exception is the plantwide code file, where fully qualified symbols could be used (see 2.4.1 Plantwide code). There are two types of namespacing

- implicit—performed automatically by the SMT behind the scenes
- explicit—declared in process code by the author of the process code to alter the automatic namsespacing

7.1 Implicit namespaces

The modeled plant may contain equations where symbols are coming from different process unit instances of the same kind. The generated XML contains equations from all process units. To make the symbols unique and to place them in the context of their containing process unit, they are qualified or decorated with the path from the plant root to their process unit (see Figure 6).

The path elements are delimited with double underscore "`__`". For example, the following assignment in $PU_{1,2}$:

```
SV = SV_0
```

will be generated as

```
Sumo__Plant__PU1__PU1_2__SV = Sumo__Plant__PU1__PU1_2__SV_0
```

in the XML file. The first segment, Sumo, is fixed. The root of the hierarchy is represented by `Plant` which is the second segment and is provided by the Sumo software. The other segments are the names of process units which lead to $PU_{1,2}$, finally comes the variable name.

The process unit names are defined on the *Structure* worksheet of the root. Please remember that the root is a virtual (invisible) process unit provided by the Sumo software, which provides the plant name as well.

There is one more segment automatically included in case of model parameters: the model name. When parameter inheritance is ON, process units will contain the following equation:

```
PAR = Parent..PAR
```

which in process unit $PU_{1,2}$ will expand as the following (only the first model parameter of `Type(Kinetic)` is shown, see *Parameters* worksheet in `Sumo1.xlsm` file and the `Parent..` construct in the next section.):

```
Sumo__Plant__PU1__PU1_2__Sumo1__muOHO = Sumo__Plant__PU1__Sumo1__muOHO
```

where the model parameters are prefixed with the model name Sumo1 (`Sumo1__muOHO`).

7.2 Explicit namespaces

Explicit namespacing is visible in the process code. The namespace should be placed as a prefix to a symbol delimited with double dot ".." characters. The namespace can be one of the following elements:

- **Parent**—keyword representing the direct parent process unit. At the moment, only the direct parent unit can be referenced.
- **Root**—keyword representing the root of the hierarchy. With this prefix process units can reference variables directly from the root.
- **port name**—one of the port names listed on the *Unit* worksheet of the process unit. This allows calculations in pipes connecting process units.
- **model ID**—the ID specified on the *Unit* worksheet of the process unit. This can be useful for process units using multiple models (see 8 Using multiple models).
    - **model name**—if the process unit is restricted to a specific model, the model name can be used in namespacing. The restriction is made on the *Unit* worksheet of the process unit, in the table with type `Model` and in column *Valid*. The model names listed in the *Valid* column can be used in namespacing.
- **process unit name**—this should be a process unit name defined on the Structure worksheet of a process unit. This means that this construct can reference symbols only from child process units (a process unit knows the names of its children but does not know the name of its parent).

The namespacing elements listed above are used as relative paths, but they will be extended to absolute paths in the XML file. For example, on Figure 6 the process unit PU_1 may contain a symbol $PU_{1,2}$..Q (referencing a symbol from one of its child process units). It will end up as `Sumo__Plant__PU1__PU1_2__Q` in the XML file. To access a symbol from the direct parent process unit the `Parent` keyword should be used, because a process unit does not know the name of its parent.

8 Using multiple models

The SMT can handle scenarios where more than one process models are used in a plant. The following picture shows the process unit hierarchy using two models:

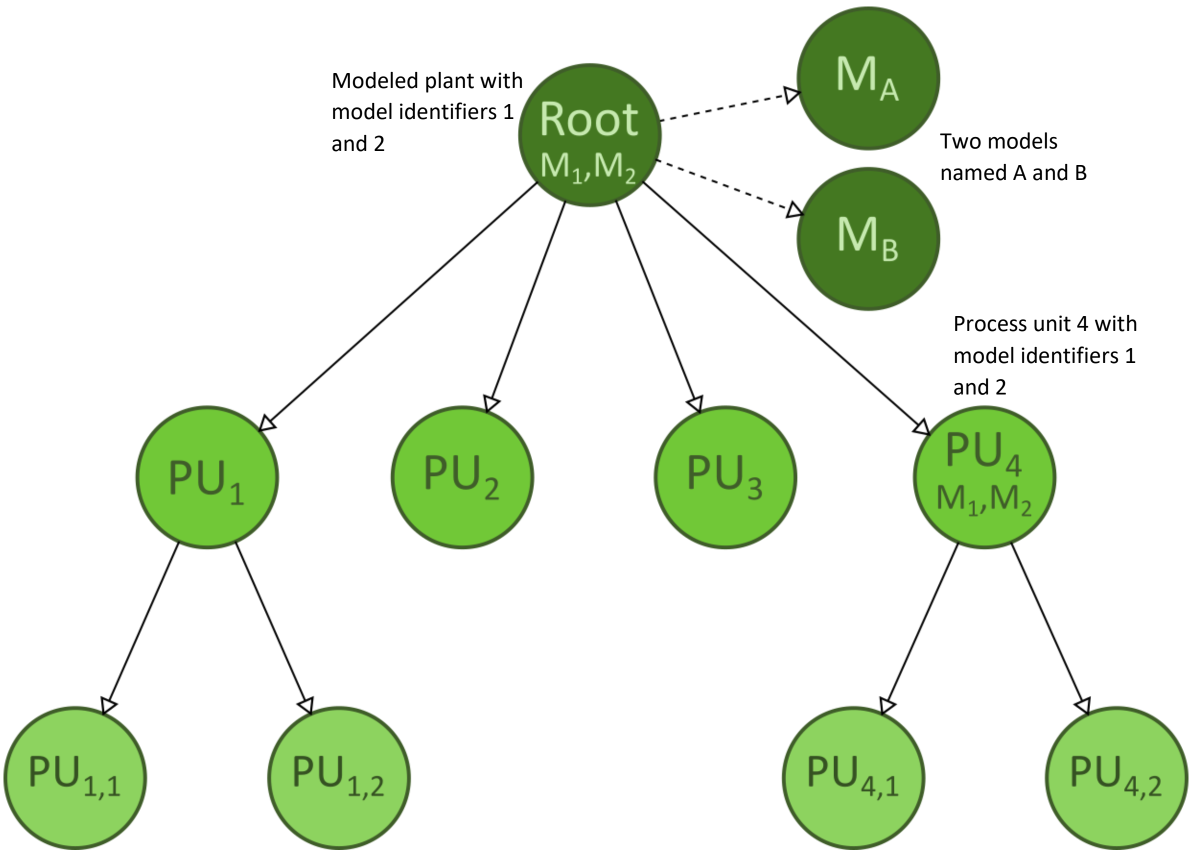


Figure 27 Process unit hierarchy with 2 models.

The graphical user interface does not handle yet multi-model plants. To understand how the SMT handles multi-model projects, see 9.1 SMT command line usage.

Process units contain model identifiers on their *Unit* sheet. These are not model names—it would mean that the units would be tied to one model. The identifiers are like local variables in a structured programming language; they can be used in the whole process unit to refer to a given model. *Given* here means that actual model names are coming somehow from the outside and they are mapped to the model identifiers during a simulation (see this too in section 9.1).

8.1 Default model

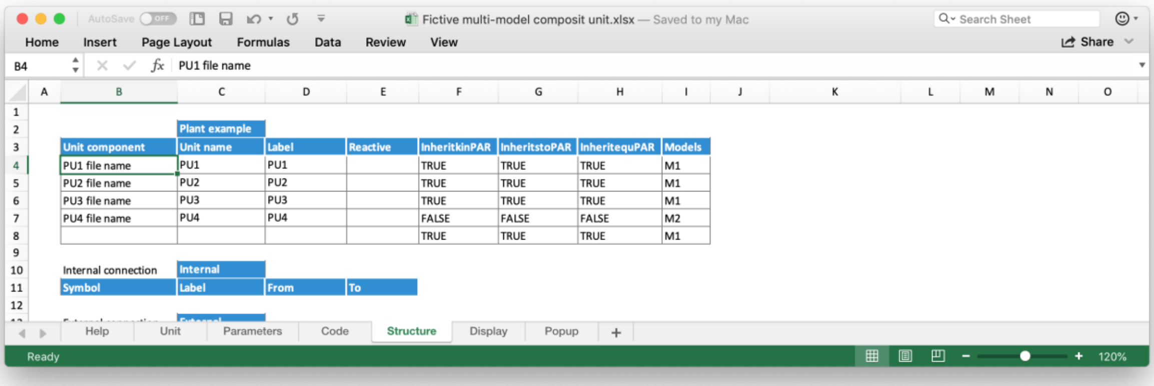
The model ID specified on the *Unit* sheet represents, or in other words is mapped to, the model used in the plant. It means that every expandable will be expanded from that model. If the plant uses one model, there is no confusion. When more than one models are specified there is need for clarification: it should be obvious which model is mapped to which ID.

The first model ID specified on the *Unit* worksheet is the default, and it will be used in shorthand expansions. Further model IDs can be specified if the user would like to switch to other models during expansion.

In this case the symbols should be namespaced explicitly with the other model ID (see 7.2 Explicit namespaces). A missing namespace prefix, again, means default model usage in expansions.

The model names coming from outside are mapped to the plant class in the order of the model identifiers on the *Unit* worksheet, i.e. M_1 in the root will be mapped to `A`, and M_2 to `B`. This affects the expansion and the namespacing processes: expandables will be expanded and model parameters will be namespaced from the correct model.

The order of model mapping from a composite process unit to its components is specified on the *Structure* worksheet. The following picture shows the root process units from Figure 27.


Fictive multi-model composit unit.xlsx — Saved to my Mac

B4 PU1 file name

		Plant example						
	Unit component	Unit name	Label	Reactive	InheritkinPAR	InheritstoPAR	InheritequPAR	Models
4	PU1 file name	PU1	PU1		TRUE	TRUE	TRUE	M1
5	PU2 file name	PU2	PU2		TRUE	TRUE	TRUE	M1
6	PU3 file name	PU3	PU3		TRUE	TRUE	TRUE	M1
7	PU4 file name	PU4	PU4		FALSE	FALSE	FALSE	M2
8					TRUE	TRUE	TRUE	M1

Internal connection	Internal		
Symbol	Label	From	To

Figure 28 Root of the hierarchy shown as a process unit.

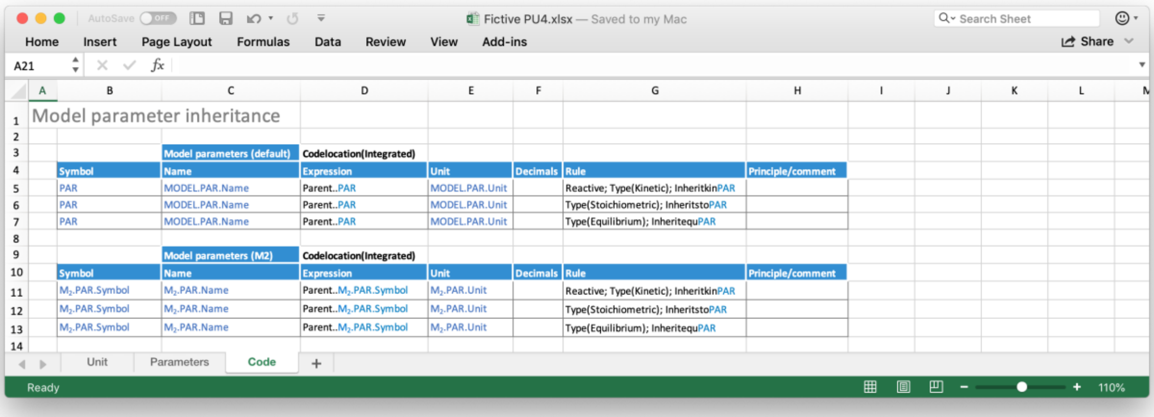
Row 7 contains a trick, it "pushes down" M_2 first then M_1 in row 8, meaning that the first model ID in PU_4 will be mapped to the content of M_2 from the root (i.e. `B`), and the second model ID in PU_4 will be mapped to `A`. Thus, `B` will be the default model in PU_4.

The model IDs are used to write model independent process code. Not just that, but it allows shuffling the models in child process units which would not be possible if model names were used directly to identify the models.

8.2 Using non-default models

Section 5.1 Expandable symbols explains how shorthand keywords like `SV`, `PAR`, `CVAR`, `SPC` could be substituted by a corresponding triplet notation, for example `SV` is a shorthand of `MODEL.SV.Symbol`, where `MODEL` is the default identifier specified on the *Unit* worksheet of the process unit. The shorthand notation cannot be used if non-default models are needed in the process code. In that case the full triplet syntax should be used.

Figure 29 shows an example of multiple model usage for parameter inheritance in the fictive plant depicted in Figure 27.



Model parameter inheritance

	Model parameters (default)	Codelocation(Integrated)				
Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
PAR	MODEL.PAR.Name	Parent..PAR	MODEL.PAR.Unit		Reactive; Type(Kinetic); InheritkinPAR	
PAR	MODEL.PAR.Name	Parent..PAR	MODEL.PAR.Unit		Type(Stoichiometric); InheritstoPAR	
PAR	MODEL.PAR.Name	Parent..PAR	MODEL.PAR.Unit		Type(Equilibrium); InheritequPAR	

	Model parameters (M2)	Codelocation(Integrated)				
Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
M_2.PAR.Symbol	M_2.PAR.Name	Parent..M_2.PAR.Symbol	M_2.PAR.Unit		Reactive; Type(Kinetic); InheritkinPAR	
M_2.PAR.Symbol	M_2.PAR.Name	Parent..M_2.PAR.Symbol	M_2.PAR.Unit		Type(Stoichiometric); InheritstoPAR	
M_2.PAR.Symbol	M_2.PAR.Name	Parent..M_2.PAR.Symbol	M_2.PAR.Unit		Type(Equilibrium); InheritequPAR	

Figure 29 Parameter inheritance with two models.

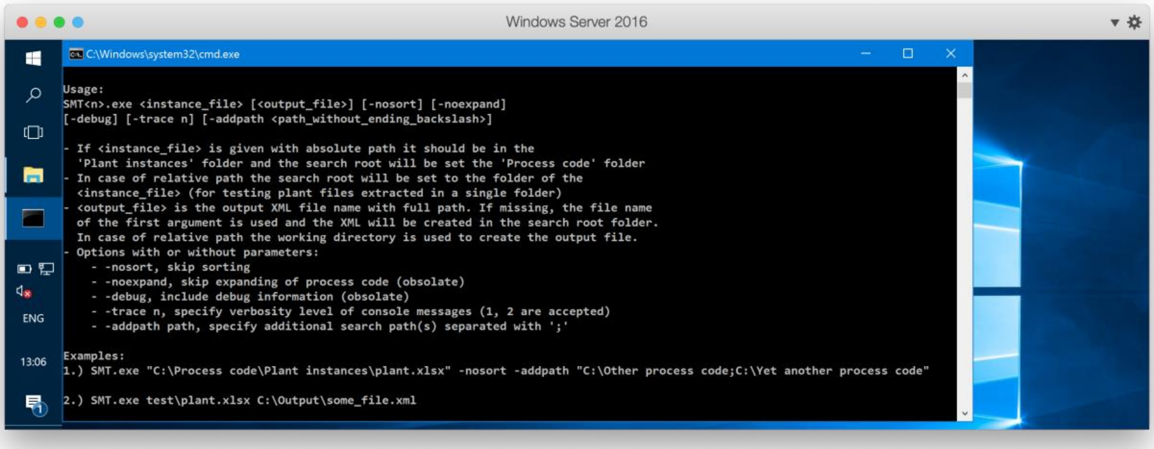
The first PAR triplet shows the shorthand notation representing the default model, while the second PAR triplet shows the usage of the full triplet syntax (please remember that the shorthand can be used only for *Symbol*, *Name* and *Unit* must use the full syntax in both cases).

9 Advanced topics

This chapter contains descriptions of some deeper usage and translation mechanics in relation to the SMT. This knowledge is not required in everyday modeling and the Sumo software, but it is good to know nonetheless.

9.1 SMT command line usage

The SMT can be used as a standalone executable program. It is useful to quickly test the correctness of the process code representing one or more process units. Figure 30 shows the help information when the SMT is started in a command line window.



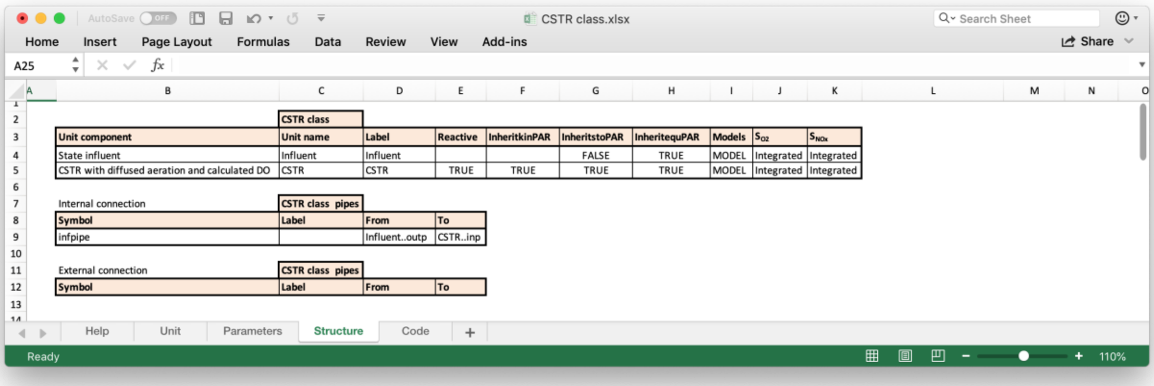
```
Usage:
SMT<n>.exe <instance_file> [<output_file>] [-nosort] [-noexpand]
[-debug] [-trace n] [-addpath <path_without_ending_backslash>]

- If <instance_file> is given with absolute path it should be in the
  'Plant instances' folder and the search root will be set the 'Process code' folder
- In case of relative path the search root will be set to the folder of the
  <instance_file> (for testing plant files extracted in a single folder)
- <output_file> is the output XML file name with full path. If missing, the file name
  of the first argument is used and the XML will be created in the search root folder.
  In case of relative path the working directory is used to create the output file.
- Options with or without parameters:
    - -nosort, skip sorting
    - -noexpand, skip expanding of process code (obsolate)
    - -debug, include debug information (obsolate)
    - -trace n, specify verbosity level of console messages (1, 2 are accepted)
    - -addpath path, specify additional search path(s) separated with ';'

Examples:
1.) SMT.exe "C:\Process code\Plant instances\plant.xlsx" -nosort -addpath "C:\Other process code;C:\Yet another process code"

2.) SMT.exe test\plant.xlsx C:\Output\some_file.xml
```

Figure 30 The SMT used from the command line.

In case of command line usage, the user should reproduce the root element of the process unit hierarchy (provided automatically when the SMT is used through the Sumo software). Moreover, some initialization data is required to reproduce the Sumo software functionality where user settings are forwarded to the SMT, also automatically.

In addition to the folder structure shown on Figure 1 the command line user should add a new folder named `Plant classes`. This should contain the process code files representing the root element of a plant hierarchy. The root element is a composite process unit like the example shown on Figure 5, i.e., it should list its components on a *Structure* worksheet. Figure 31 shows an example with two process units linked together where the link is represented in the `Internal connection` table. The *Unit component* column contains the process unit file names without extension.

The *Models* column contains a single symbol, `MODEL`, which is the model identifier containing the actual model name and it is declared on the *Unit* worksheet. The model identifier is initialized with the correct model name by a separate file explained below. This separation allows the reuse of the same plant root with different initialization data.



CSTR class.xlsx

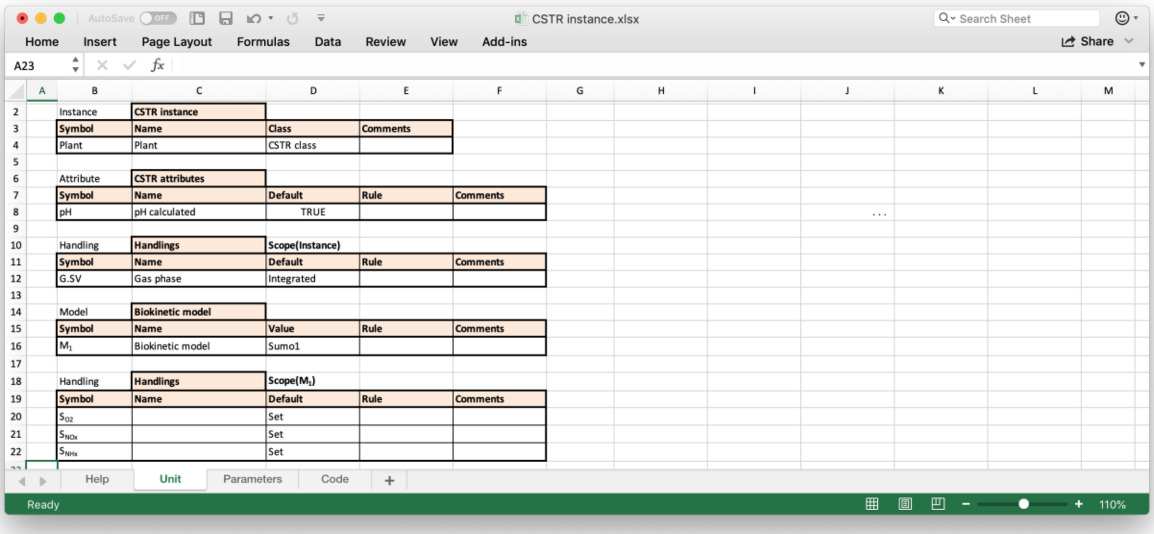
	CSTR class								
Unit component	**Unit name**	**Label**	**Reactive**	**InheritkinPAR**	**InheritstoPAR**	**InheritequPAR**	**Models**	**S_{O2}**	**S_{NOx}**
State influent	Influent	Influent			FALSE	TRUE	MODEL	Integrated	Integrated
CSTR with diffused aeration and calculated DO	CSTR	CSTR	TRUE	TRUE	TRUE	TRUE	MODEL	Integrated	Integrated

Internal connection	**CSTR class pipes**		
Symbol	**Label**	**From**	**To**
infpipe		Influent..outp	CSTR..inp

External connection	**CSTR class pipes**		
Symbol	**Label**	**From**	**To**

Figure 31 The root process unit representing a simple CSTR test plant.

The other initialization file contains global settings available to the whole plant. This file should be placed in a subfolder with arbitrary name under the `Process code` folder. It is called a plant instance file and its main purpose is to specify the model, some attributes and state variable handlings globally used in the plant. Figure 32 shows the initialization file of the previous CSTR test plant.



CSTR instance.xlsx

Instance	**CSTR instance**		
Symbol	**Name**	**Class**	**Comments**
Plant	Plant	CSTR class	

Attribute	**CSTR attributes**			
Symbol	**Name**	**Default**	**Rule**	**Comments**
pH	pH calculated	TRUE		

Handling	**Handlings**	**Scope(Instance)**		
Symbol	**Name**	**Default**	**Rule**	**Comments**
G.SV	Gas phase	Integrated		

Model	**Biokinetic model**			
Symbol	**Name**	**Value**	**Rule**	**Comments**
M_1	Biokinetic model	Sumo1		

Handling	**Handlings**	**Scope(M_1)**		
Symbol	**Name**	**Default**	**Rule**	**Comments**
S_{O2}		Set		
S_{NOx}		Set		
S_{NHx}		Set		

Figure 32 Initialization information for the simple CSTR example.

The table types and names speak for themselves, the only new element is the *Scope* tag in handlings. The *Scope(Instance)* tag means that the listed state variable handlings are enforced in all models while the handlings listed in *Scope(M_1)* are enforced only in the model identified by the argument (row 16 contains the model ID used in scoping). In this example there is only one model identifier, but more than one can be specified if needed (see 8 Using multiple models).

Please note that the `Process code` folder is not required to be the one installed by Sumo. In fact, for testing purposes, it is a good practice to prepare a separate process code folder somewhere else then copy the content of the installed `Process code` folder in it. Sumo provides such a test folder named `My Process Code` which has the benefit of being taken in consideration by the Sumo software, but for simple testing with the SMT the test folder can be anywhere.

9.2 Attribute and handling propagation

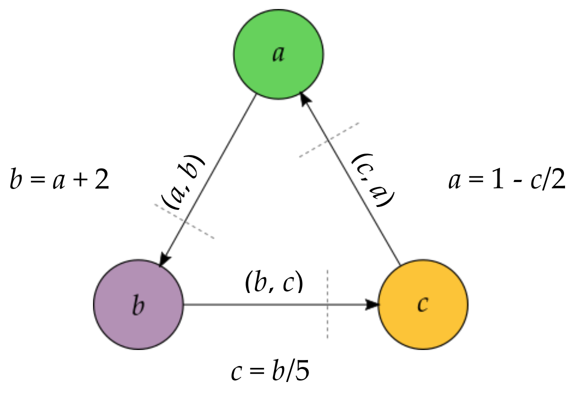
In the simple CSTR example the root *Structure* worksheet contains state variable handlings after the *Models* column which will be enforced in all component process units. Root handlings, on the other hand, will be overridden by handlings from the initialization file with *Scope(Instance)* or *Scope(M_1)*. The latter overrides the root handlings only if a component uses the model specified by the scope argument.

Attributes propagate like handlings, but without the complication of scoping. As a general rule the root will override the attributes and handlings of its components while the initializer overrides the attributes and handlings in the whole plant. Otherwise the attributes and handlings specified on the *Unit* sheet of the individual process units will prevail.

9.3 Algebraic loops

When the plant contains feedback loops some equations in the participating process units contain circular dependencies called algebraic loops. To solve such an equation system the Sumo numeric engine uses an iterative process described later in this section.

The next picture illustrates a simple algebraic loop:



the equation system $\begin{cases} a = 1 - c/2 \\ b = a + 2 \\ c = b/5 \end{cases}$

Definitions

- an equation system is represented by a directed graph
- nodes—represent variables
- vertices—represent the equations dependencies as a variable pair (v_{from}, v_{to})
- the direction of the vertices is from the right-hand side to the left-hand side. In case of the equation $b = a + 2$ this means $b \leftarrow a$ or a goes to b.
- an algebraic loop is equivalent to a directed loop in the graph

Default sorter behavior

- break the loop at an arbitrary node (represented by the dashed lines)
- define that node as a loop breaker variable

Figure 33 Simple algebraic loop.

The SMT contains an equation sorter module which recognizes algebraic loops and tries to flatten (break) them at one of the vertices and declare the left-hand side variable as loop breaker which will be calculated with the iterative process.

Calculating the arbitrarily chosen loop breaker may or may not produce a satisfactory result (since the iterative process should be converging). The process code author should be able to intervene and define a better loop breaker manually. This is a trial and error process: if the automatic loop breaking does not give a result or it is calculated slowly, the author determines which algebraic loop is the culprit and intervenes by giving a better loop breaker.

The SumoSlang allows this intervention; the author declares a loop breaker variable and gives its properties (initial value, minima, maxima) with the help of a `SolverConfiguration` table (see section 9.3.3). Figure 34 shows the abstract representation of this:

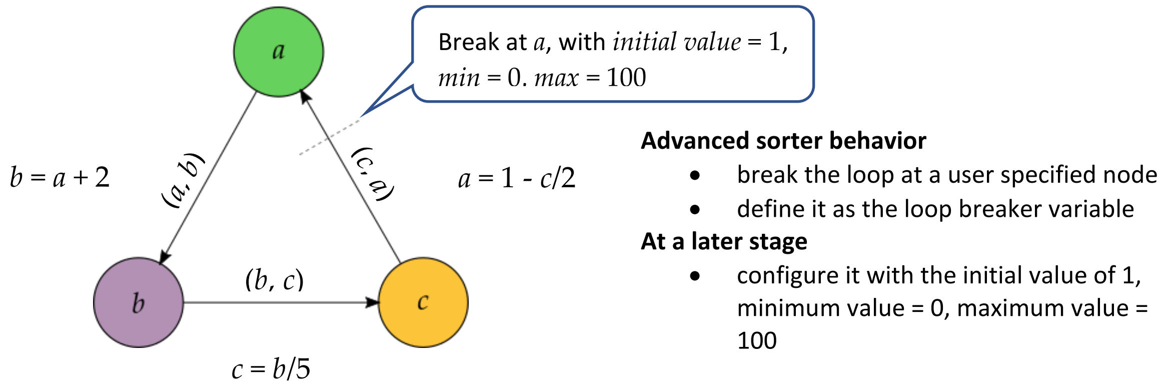

Break at a, with $initial\ value$ = 1, min = 0. max = 100
a
$b = a + 2$
(a, b)
(c, a)
$a = 1 - c/2$
b
(b, c)
c
$c = b/5$
Advanced sorter behavior
break the loop at a user specified node
define it as the loop breaker variable
At a later stage
configure it with the initial value of 1, minimum value = 0, maximum value = 100

Figure 34 The same loop with a loop breaker given from outside.

9.3.1 1 Solving the equation system

Solving an algebraic loop involves calculating its loop breaker with an iterative process. To do this, the original equation is transformed to an error function and the loop breaker is calculated by minimizing the error through the iterative process. This requires a convergent error function and the value of the loop breaker will be the one where the error is the closest to 0.

Breaking the loop, then defining the error function, then applying the iterative process to minimize it, is done by the SMT behind the scenes or more precisely it prepares the instructions for the Sumo numerical engine to do the iteration properly.

Figure 35 shows the abstract representation of this process:

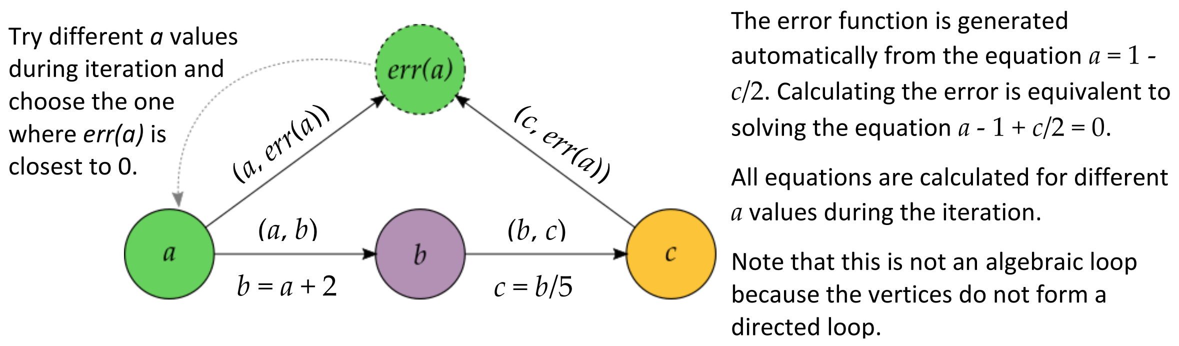
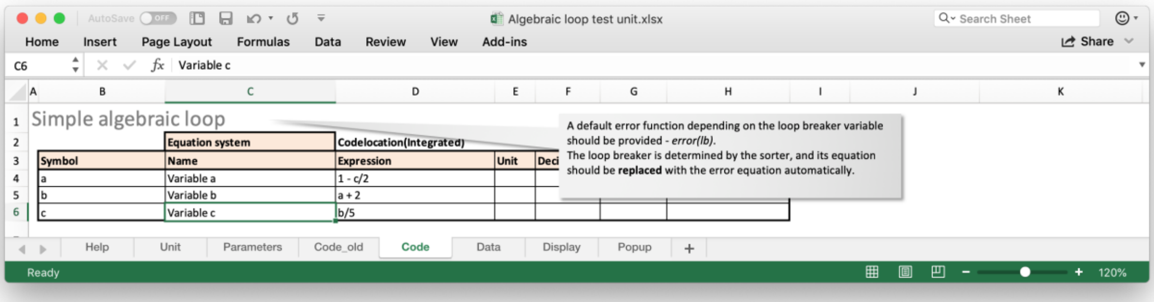

Try different a values during iteration and choose the one where $err(a)$ is closest to 0.
$err(a)$
$(a, err(a))$
$(c, err(a))$
a
(a, b)
b
$b = a + 2$
(b, c)
c
$c = b/5$
The error function is generated automatically from the equation $a = 1 - c/2$. Calculating the error is equivalent to solving the equation $a - 1 + c/2 = 0$.
All equations are calculated for different a values during the iteration.
Note that this is not an algebraic loop because the vertices do not form a directed loop.

Figure 35 Solving the loop by introducing an error function depending on the loop breaker variable.

9.3.2 Process code example for automatic loop breaking

The equations from Figure 33 can be written in SumoSlang as follows:



Simple algebraic loop

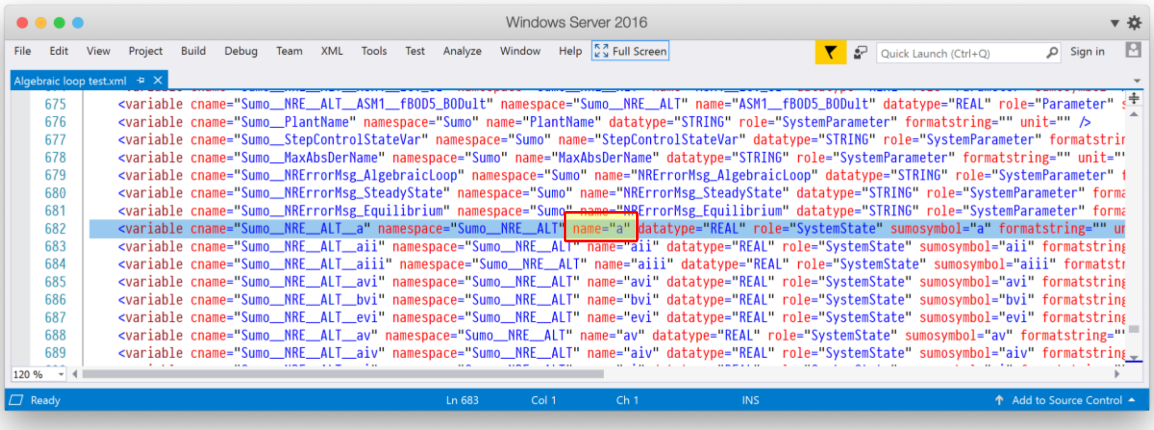
	Equation system	Codelocation(Integrated)		
Symbol	Name	Expression	Unit	Deci
a	Variable a	1 - c/2		
b	Variable b	a + 2		
c	Variable c	b/5		

A default error function depending on the loop breaker variable should be provided - *error(lb)*.
The loop breaker is determined by the sorter, and its equation should be **replaced** with the error equation automatically.

Figure 36 Process code representation of the simple equation system.

Expected XML output

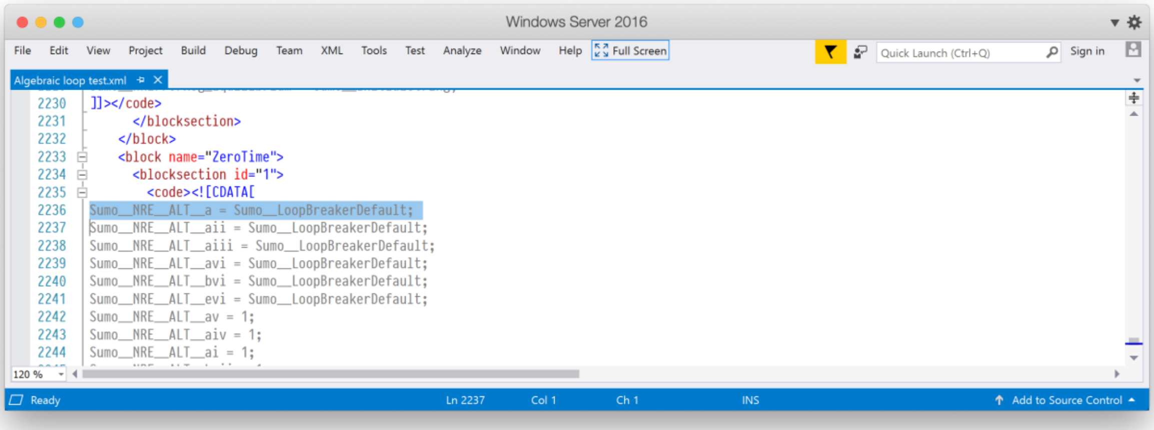
The `<variabledefinitions>…</variabledefinitions>` block of the XML file should contain the highlighted entry:



```
<variable cname="Sumo__NRE__ALT__ASM1__fBOD5_BODult" namespace="Sumo__NRE__ALT" name="ASM1__fBOD5_BODult" datatype="REAL" role="Parameter" s
<variable cname="Sumo__PlantName" namespace="Sumo" name="PlantName" datatype="STRING" role="SystemParameter" formatstring="" unit="" />
<variable cname="Sumo__StepControlStateVar" namespace="Sumo" name="StepControlStateVar" datatype="STRING" role="SystemParameter" formatstrir
<variable cname="Sumo__MaxAbsDerName" namespace="Sumo" name="MaxAbsDerName" datatype="STRING" role="SystemParameter" formatstring="" unit=""
<variable cname="Sumo__NRErrorMsg_AlgebraicLoop" namespace="Sumo" name="NRErrorMsg_AlgebraicLoop" datatype="STRING" role="SystemParameter" f
<variable cname="Sumo__NRErrorMsg_SteadyState" namespace="Sumo" name="NRErrorMsg_SteadyState" datatype="STRING" role="SystemParameter" forma
<variable cname="Sumo__NRErrorMsg_Equilibrium" namespace="Sumo" name="NRErrorMsg_Equilibrium" datatype="STRING" role="SystemParameter" forma
<variable cname="Sumo__NRE__ALT__a" namespace="Sumo__NRE__ALT" name="a" datatype="REAL" role="SystemState" sumosymbol="a" formatstring="" ur
<variable cname="Sumo__NRE__ALT__aii" namespace="Sumo__NRE__ALT" name="aii" datatype="REAL" role="SystemState" sumosymbol="aii" formatstring
<variable cname="Sumo__NRE__ALT__aiii" namespace="Sumo__NRE__ALT" name="aiii" datatype="REAL" role="SystemState" sumosymbol="aiii" formatstr
<variable cname="Sumo__NRE__ALT__avi" namespace="Sumo__NRE__ALT" name="avi" datatype="REAL" role="SystemState" sumosymbol="avi" formatstring
<variable cname="Sumo__NRE__ALT__bvi" namespace="Sumo__NRE__ALT" name="bvi" datatype="REAL" role="SystemState" sumosymbol="bvi" formatstring
<variable cname="Sumo__NRE__ALT__evi" namespace="Sumo__NRE__ALT" name="evi" datatype="REAL" role="SystemState" sumosymbol="evi" formatstring
<variable cname="Sumo__NRE__ALT__av" namespace="Sumo__NRE__ALT" name="av" datatype="REAL" role="SystemState" sumosymbol="av" formatstring=""
<variable cname="Sumo__NRE__ALT__aiv" namespace="Sumo__NRE__ALT" name="aiv" datatype="REAL" role="SystemState" sumosymbol="aiv" formatstring
```

Figure 37 Variable 'a' in the `variabledefinitions` *block*

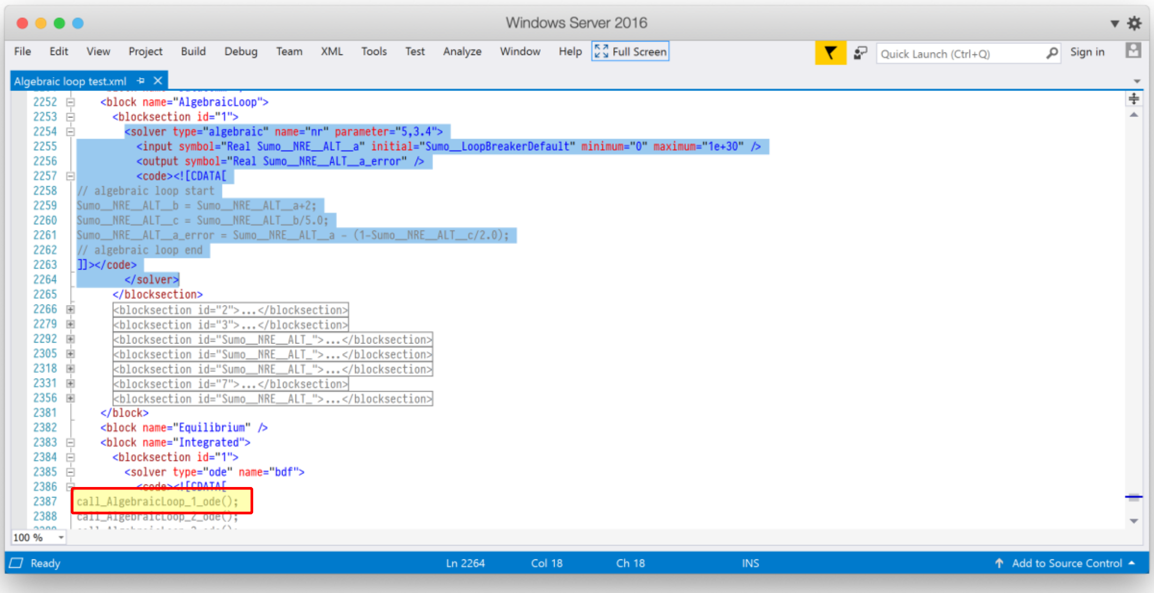
The `ZeroTime/1` section should contain the highlighted entry to keep the SMT sorter happy (avoid the variable `Sumo__NRE__ALT__a` was not found in the error equation). This is inserted automatically by the SMT.



```
]]></code>
      </blocksection>
    </block>
    <block name="ZeroTime">
      <blocksection id="1">
        <code><![CDATA[
Sumo__NRE__ALT__a = Sumo__LoopBreakerDefault;
Sumo__NRE__ALT__aii = Sumo__LoopBreakerDefault;
Sumo__NRE__ALT__aiii = Sumo__LoopBreakerDefault;
Sumo__NRE__ALT__avi = Sumo__LoopBreakerDefault;
Sumo__NRE__ALT__bvi = Sumo__LoopBreakerDefault;
Sumo__NRE__ALT__evi = Sumo__LoopBreakerDefault;
Sumo__NRE__ALT__av = 1;
Sumo__NRE__ALT__aiv = 1;
Sumo__NRE__ALT__ai = 1;
```

Figure 38 The same variable in the `ZeroTime` *block.*

The `AlgebraicLoop/1` section should contain the blue highlighted entry and the `Integrated/1` section the yellow highlighted entry where the function call name is derived from the block name and section name of the algebraic loop. The blue highlighted entry contains information for the solver where its input is the loop breaker variable *a* and its output is the error value. The `code` section inside `solver` contains the flattened equation system and the error calculation equation inserted by the SMT.

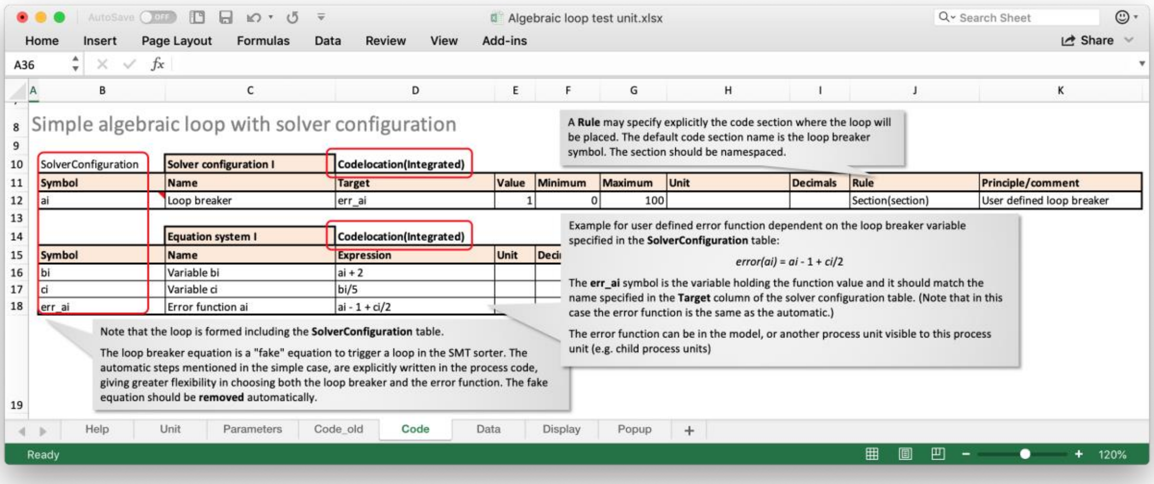


```
    <block name="AlgebraicLoop">
      <blocksection id="1">
        <solver type="algebraic" name="nr" parameter="5,3.4">
          <input symbol="Real Sumo__NRE__ALT__a" initial="Sumo__LoopBreakerDefault" minimum="0" maximum="1e+30" />
          <output symbol="Real Sumo__NRE__ALT__a_error" />
          <code><![CDATA[
// algebraic loop start
Sumo__NRE__ALT__b = Sumo__NRE__ALT__a+2;
Sumo__NRE__ALT__c = Sumo__NRE__ALT__b/5.0;
Sumo__NRE__ALT__a_error = Sumo__NRE__ALT__a - (1-Sumo__NRE__ALT__c/2.0);
// algebraic loop end
]]></code>
        </solver>
      </blocksection>
      <blocksection id="2">...</blocksection>
      <blocksection id="3">...</blocksection>
      <blocksection id="Sumo__NRE__ALT_">...</blocksection>
      <blocksection id="Sumo__NRE__ALT_">...</blocksection>
      <blocksection id="Sumo__NRE__ALT_">...</blocksection>
      <blocksection id="7">...</blocksection>
      <blocksection id="Sumo__NRE__ALT_">...</blocksection>
    </block>
    <block name="Equilibrium" />
    <block name="Integrated">
      <blocksection id="1">
        <solver type="ode" name="bdf">
          <code><![CDATA[
call_AlgebraicLoop_1_ode();
call_AlgebraicLoop_2_ode();
```

Figure 39 XML overview showing the `AlgeraicLoop` and `Integrated` blocks to solve the equation system in the example.

9.3.3 Process code example for manual loop breaking

Figure 40 shows how the process code author can manually declare a loop breaker for the simple equation system used in the previous sections as example (the worksheet contains many examples with the same variables which requires a distinction between them. In the different examples the variables are suffixed with roman numerals.)



Simple algebraic loop with solver configuration

SolverConfiguration	Solver configuration I	Codelocation(Integrated)							
Symbol	**Name**	**Target**	**Value**	**Minimum**	**Maximum**	**Unit**	**Decimals**	**Rule**	**Principle/comment**
ai	Loop breaker	err_ai	1	0	100			Section(section)	User defined loop breaker

	Equation system I	Codelocation(Integrated)		
Symbol	**Name**	**Expression**	**Unit**	**Deci**
bi	Variable bi	ai + 2		
ci	Variable ci	bi/5		
err_ai	Error function ai	ai - 1 + ci/2		

A **Rule** may specify explicitly the code section where the loop will be placed. The default code section name is the loop breaker symbol. The section should be namespaced.

Example for user defined error function dependent on the loop breaker variable specified in the **SolverConfiguration** table:

error(ai) = ai - 1 + ci/2

The **err_ai** symbol is the variable holding the function value and it should match the name specified in the **Target** column of the solver configuration table. (Note that in this case the error function is the same as the automatic.)

The error function can be in the model, or another process unit visible to this process unit (e.g. child process units)

Note that the loop is formed including the **SolverConfiguration** table.

The loop breaker equation is a "fake" equation to trigger a loop in the SMT sorter. The automatic steps mentioned in the simple case, are explicitly written in the process code, giving greater flexibility in choosing both the loop breaker and the error function. The fake equation should be **removed** automatically.

Figure 40 Solver configuration used to specify a loop breaker.

Please note that the loop breaker equation

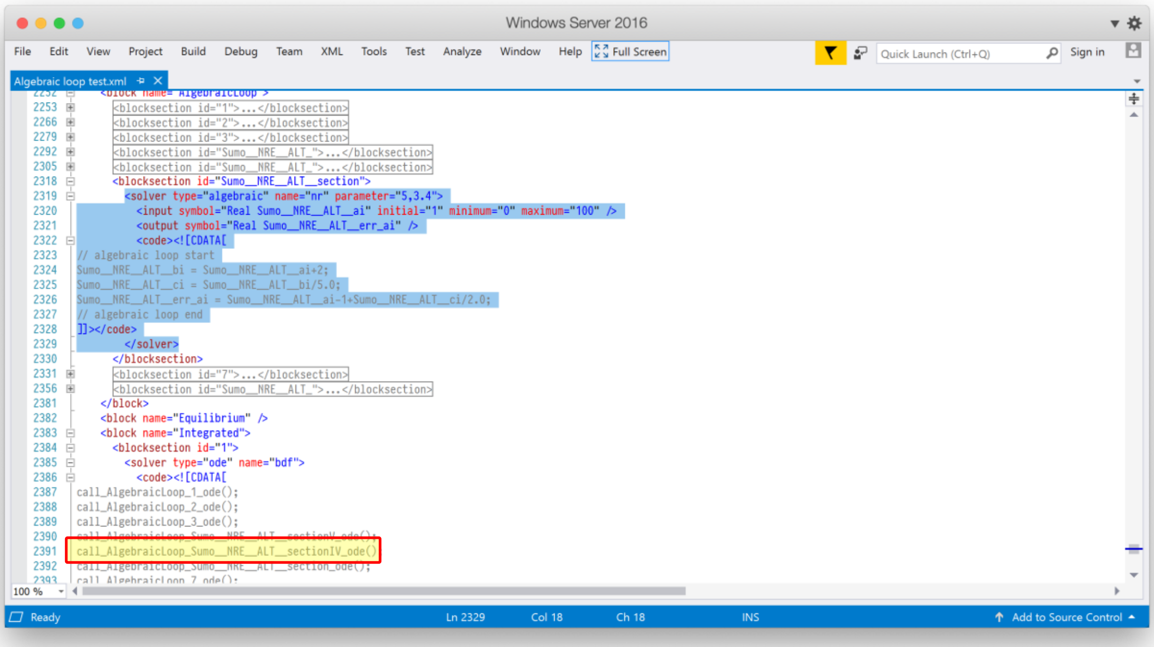
```
a = 1 - c/2
```

is not present in the *Equation system I* table, it is replaced by the error equation. This error equation is automatically provided by the SMT in the default mode discussed previously.

The sorter module recognizes the loop breaker variable and would not complain when it is used on the right-hand side of other equations. The SMT will use the information in the `SolverConfiguration` table at a later stage to provide input and output symbols to the solver used to calculate the error (see Figure 39).

Expected XML output

The `AlegbraicLoop` block now contains a custom section name `Sumo__NRE__ALT__section` specified in the *Rule* column of the `SolverConfiguration` table and namespaced according to the container process unit.



```
    <block name="AlgebraicLoop">
      <blocksection id="1">...</blocksection>
      <blocksection id="2">...</blocksection>
      <blocksection id="3">...</blocksection>
      <blocksection id="Sumo__NRE__ALT_">...</blocksection>
      <blocksection id="Sumo__NRE__ALT_">...</blocksection>
      <blocksection id="Sumo__NRE__ALT__section">
        <solver type="algebraic" name="nr" parameter="5,3.4">
          <input symbol="Real Sumo__NRE__ALT__ai" initial="1" minimum="0" maximum="100" />
          <output symbol="Real Sumo__NRE__ALT__err_ai" />
          <code><![CDATA[
// algebraic loop start
Sumo__NRE__ALT__bi = Sumo__NRE__ALT__ai+2;
Sumo__NRE__ALT__ci = Sumo__NRE__ALT__bi/5.0;
Sumo__NRE__ALT__err_ai = Sumo__NRE__ALT__ai-1+Sumo__NRE__ALT__ci/2.0;
// algebraic loop end
]]></code>
        </solver>
      </blocksection>
      <blocksection id="7">...</blocksection>
      <blocksection id="Sumo__NRE__ALT_">...</blocksection>
    </block>
    <block name="Equilibrium" />
    <block name="Integrated">
      <blocksection id="1">
        <solver type="ode" name="bdf">
          <code><![CDATA[
call_AlgebraicLoop_1_ode();
call_AlgebraicLoop_2_ode();
call_AlgebraicLoop_3_ode();
call_AlgebraicLoop_Sumo__NRE__ALT__sectionV_ode();
call_AlgebraicLoop_Sumo__NRE__ALT__sectionIV_ode()
call_AlgebraicLoop_Sumo__NRE__ALT__section_ode();
call_AlgebraicLoop_7_ode();
```

Figure 41 XML overview showing the AlgeraicLoop and Integrated blocks in case of manual loop breaker definition. The outcome is the same.

9.3.4 4 Complex equation system

Figure 42 shows the graph representation of a complex equation system containing multiple loops:

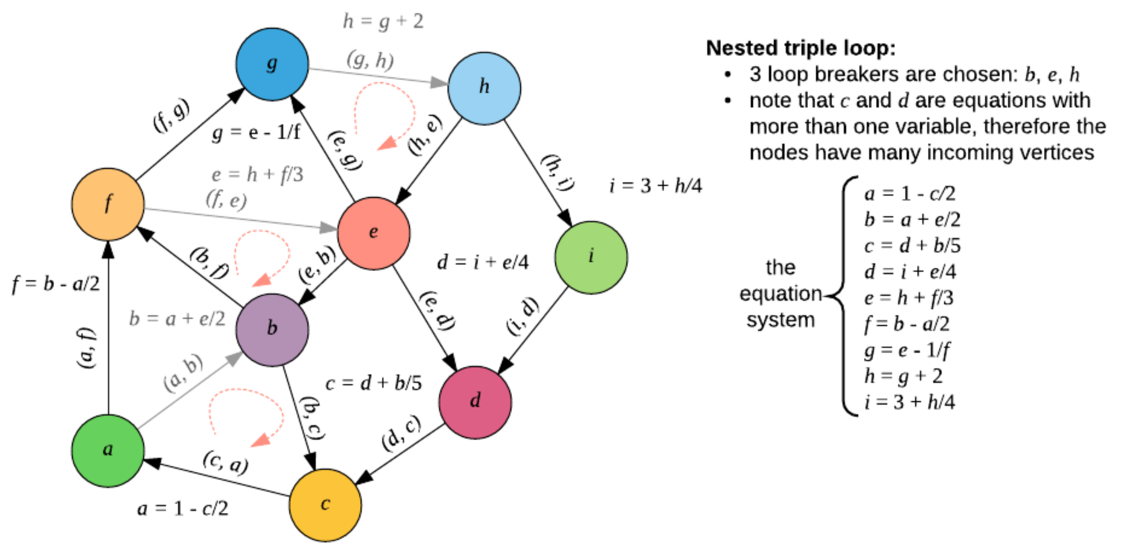
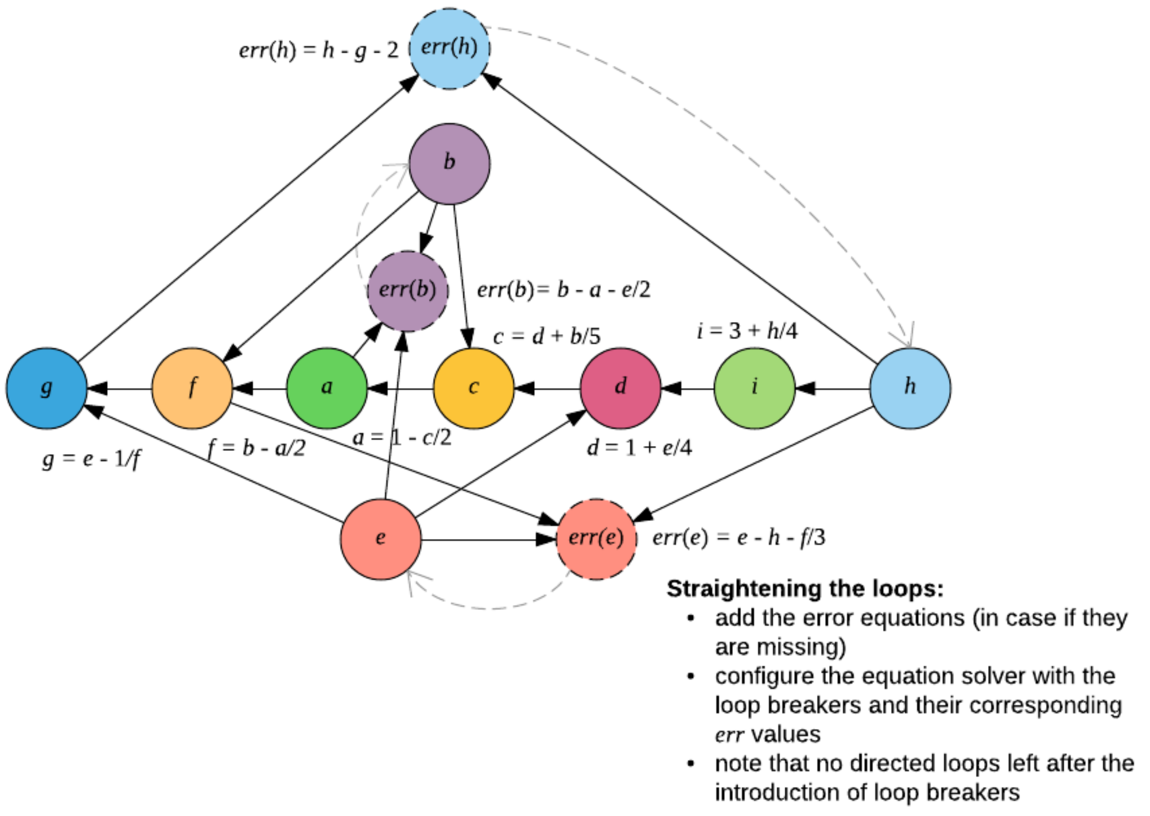


Figure 42 Complex equation system with many algebraic loops and "manually" chosen loop breakers.

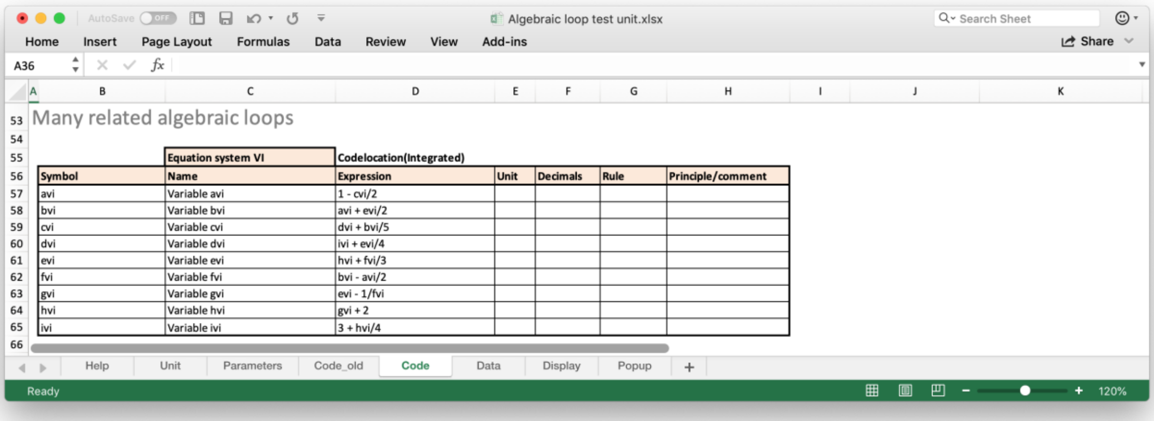
The lighter colored vertices represent the manually chosen loop breakers. Solving this equation system require introducing 3 error functions (in default mode the SMT will choose the loop breakers and the error equations) as shown in the following picture:



err(h) = h - g - 2
err(h)
b
err(b)
err(b)= b - a - e/2
c = d + b/5
i = 3 + h/4
g
f
a
c
d
i
h
g = e - 1/f
f = b - a/2
a = 1 - c/2
d = 1 + e/4
e
err(e)
err(e) = e - h - f/3
Straightening the loops:
add the error equations (in case if they are missing)
configure the equation solver with the loop breakers and their corresponding err values
note that no directed loops left after the introduction of loop breakers

Figure 43 Solving the complex loop.

Process code example for the default method:

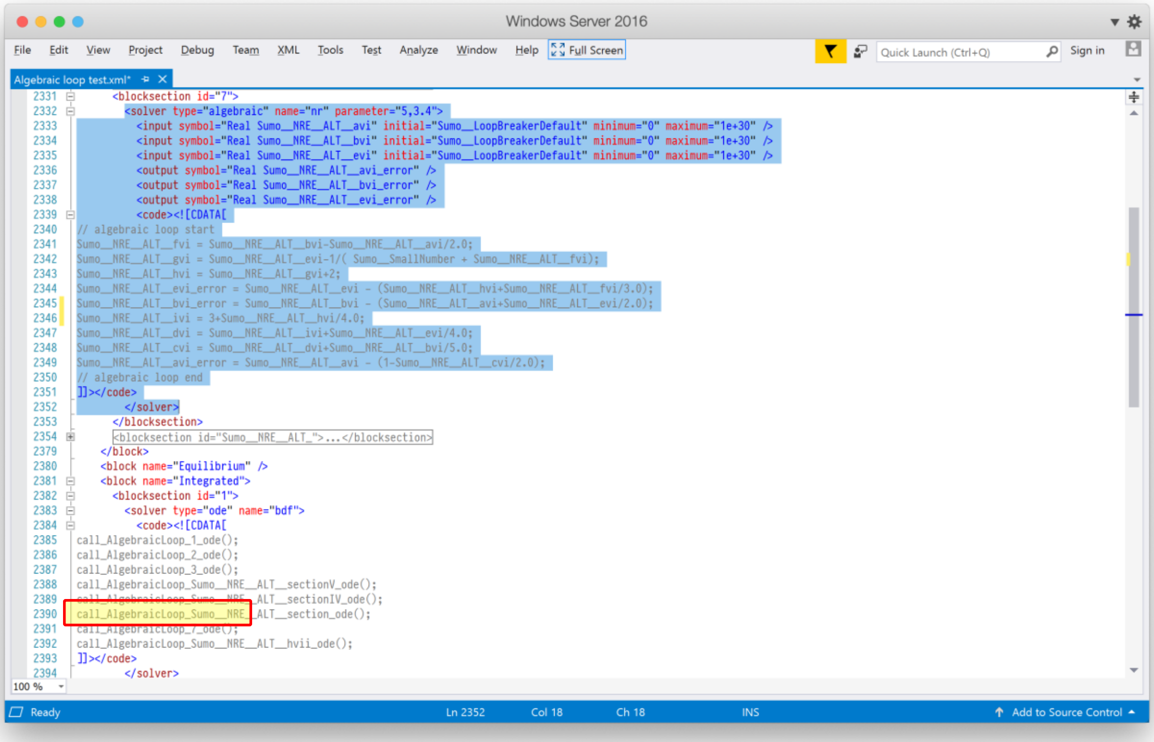


Many related algebraic loops

	Equation system VI	Codelocation(Integrated)				
Symbol	**Name**	**Expression**	**Unit**	**Decimals**	**Rule**	**Principle/comment**
avi	Variable avi	1 - cvi/2				
bvi	Variable bvi	avi + evi/2				
cvi	Variable cvi	dvi + bvi/5				
dvi	Variable dvi	ivi + evi/4				
evi	Variable evi	hvi + fvi/3				
fvi	Variable fvi	bvi - avi/2				
gvi	Variable gvi	evi - 1/fvi				
hvi	Variable hvi	gvi + 2				
ivi	Variable ivi	3 + hvi/4				

Figure 44 Process code example of the complex equation system.

The process code author relies on the SMT to solve the algebraic loop. If it is not satisfactory, a manual intervention is required. Figure 45 shows the XML representation of the default method, where the loop breakers chosen by the SMT are *a*, *b*, *e*. In this example the section name 7 is given automatically and it is a plant wide unique sequence number.

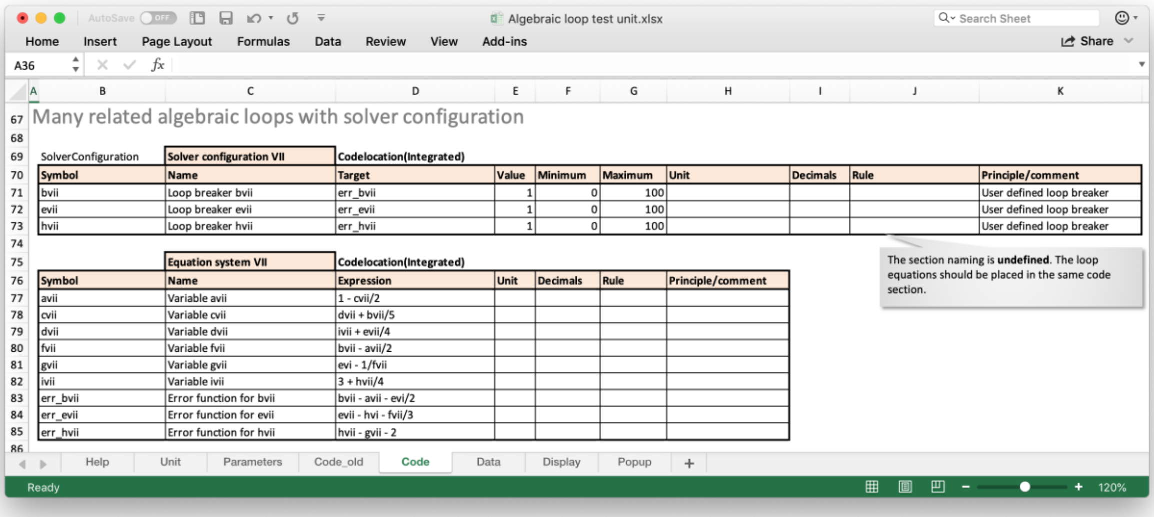


```
        <blocksection id="7">
          <solver type="algebraic" name="nr" parameter="5,3.4">
            <input symbol="Real Sumo__NRE__ALT__avi" initial="Sumo__LoopBreakerDefault" minimum="0" maximum="1e+30" />
            <input symbol="Real Sumo__NRE__ALT__bvi" initial="Sumo__LoopBreakerDefault" minimum="0" maximum="1e+30" />
            <input symbol="Real Sumo__NRE__ALT__evi" initial="Sumo__LoopBreakerDefault" minimum="0" maximum="1e+30" />
            <output symbol="Real Sumo__NRE__ALT__avi_error" />
            <output symbol="Real Sumo__NRE__ALT__bvi_error" />
            <output symbol="Real Sumo__NRE__ALT__evi_error" />
            <code><![CDATA[
// algebraic loop start
Sumo__NRE__ALT__fvi = Sumo__NRE__ALT__bvi-Sumo__NRE__ALT__avi/2.0;
Sumo__NRE__ALT__gvi = Sumo__NRE__ALT__evi-1/( Sumo__SmallNumber + Sumo__NRE__ALT__fvi);
Sumo__NRE__ALT__hvi = Sumo__NRE__ALT__gvi+2;
Sumo__NRE__ALT__evi_error = Sumo__NRE__ALT__evi - (Sumo__NRE__ALT__hvi+Sumo__NRE__ALT__fvi/3.0);
Sumo__NRE__ALT__bvi_error = Sumo__NRE__ALT__bvi - (Sumo__NRE__ALT__avi+Sumo__NRE__ALT__evi/2.0);
Sumo__NRE__ALT__ivi = 3+Sumo__NRE__ALT__hvi/4.0;
Sumo__NRE__ALT__dvi = Sumo__NRE__ALT__ivi+Sumo__NRE__ALT__evi/4.0;
Sumo__NRE__ALT__cvi = Sumo__NRE__ALT__dvi+Sumo__NRE__ALT__bvi/5.0;
Sumo__NRE__ALT__avi_error = Sumo__NRE__ALT__avi - (1-Sumo__NRE__ALT__cvi/2.0);
// algebraic loop end
]]></code>
          </solver>
        </blocksection>
        <blocksection id="Sumo__NRE__ALT_">...</blocksection>
      </block>
      <block name="Equilibrium" />
      <block name="Integrated">
        <blocksection id="1">
          <solver type="ode" name="bdf">
            <code><![CDATA[
call_AlgebraicLoop_1_ode();
call_AlgebraicLoop_2_ode();
call_AlgebraicLoop_3_ode();
call_AlgebraicLoop_Sumo__NRE__ALT__sectionV_ode();
call_AlgebraicLoop_Sumo__NRE__ALT__sectionIV_ode();
call_AlgebraicLoop_Sumo__NRE__ALT__section_ode();
call_AlgebraicLoop_7_ode();
call_AlgebraicLoop_Sumo__NRE__ALT__hvii_ode();
]]></code>
          </solver>
```

Figure 45 XML representation of the default algebraic loop solution.

Process code example for the manual method:

The process code author lists the manually chosen loop breakers in the `SolverConfiguration` table and the loop breaker equations are replaced manually with the error equations.



Many related algebraic loops with solver configuration

SolverConfiguration	Solver configuration VII	Codelocation(Integrated)							
Symbol	Name	Target	Value	Minimum	Maximum	Unit	Decimals	Rule	Principle/comment
bvii	Loop breaker bvii	err_bvii	1	0	100				User defined loop breaker
evii	Loop breaker evii	err_evii	1	0	100				User defined loop breaker
hvii	Loop breaker hvii	err_hvii	1	0	100				User defined loop breaker

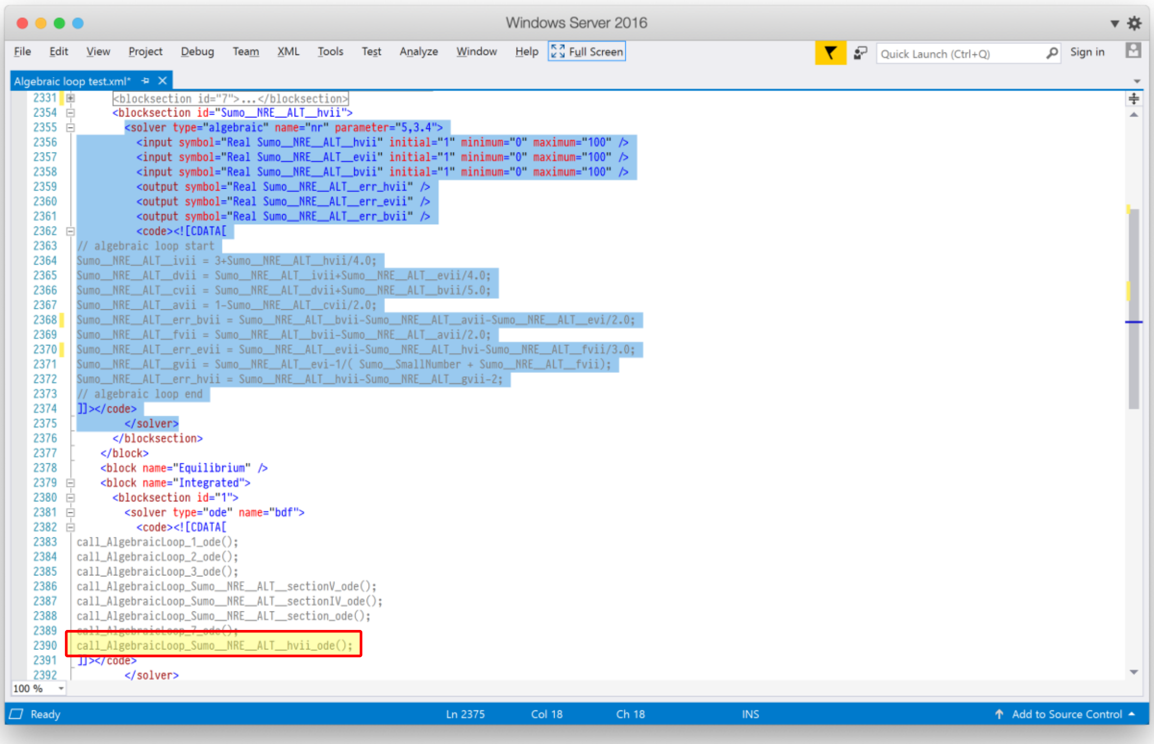
	Equation system VII	Codelocation(Integrated)				
Symbol	Name	Expression	Unit	Decimals	Rule	Principle/comment
avii	Variable avii	1 - cvii/2				
cvii	Variable cvii	dvii + bvii/5				
dvii	Variable dvii	ivii + evii/4				
fvii	Variable fvii	bvii - avii/2				
gvii	Variable gvii	evi - 1/fvii				
ivii	Variable ivii	3 + hvii/4				
err_bvii	Error function for bvii	bvii - avii - evi/2				
err_evii	Error function for evii	evii - hvi - fvii/3				
err_hvii	Error function for hvii	hvii - gvii - 2				

The section naming is **undefined**. The loop equations should be placed in the same code section.

Figure 46 Complex equation system with user defined algebraic loop handling.

The section name in the `AlgebraicLoop` block is undefined in this case but it should be different from the default mode. One solution is to use one of the loop breaker variable names as section name. In the following example the last variable name is chosen: *hvii*. Being a variable name (i.e. not a plant wide unique name), it should be namespaced according to the container process unit.

Figure 47 shows the XML representation of the manual loop breaker selection:



```
        <blocksection id="7">...</blocksection>
        <blocksection id="Sumo__NRE__ALT__hvii">
          <solver type="algebraic" name="nr" parameter="5,3.4">
            <input symbol="Real Sumo__NRE__ALT__hvii" initial="1" minimum="0" maximum="100" />
            <input symbol="Real Sumo__NRE__ALT__evii" initial="1" minimum="0" maximum="100" />
            <input symbol="Real Sumo__NRE__ALT__bvii" initial="1" minimum="0" maximum="100" />
            <output symbol="Real Sumo__NRE__ALT__err_hvii" />
            <output symbol="Real Sumo__NRE__ALT__err_evii" />
            <output symbol="Real Sumo__NRE__ALT__err_bvii" />
            <code><![CDATA[
// algebraic loop start
Sumo__NRE__ALT__ivii = 3+Sumo__NRE__ALT__hvii/4.0;
Sumo__NRE__ALT__dvii = Sumo__NRE__ALT__ivii+Sumo__NRE__ALT__evii/4.0;
Sumo__NRE__ALT__cvii = Sumo__NRE__ALT__dvii+Sumo__NRE__ALT__bvii/5.0;
Sumo__NRE__ALT__avii = 1-Sumo__NRE__ALT__cvii/2.0;
Sumo__NRE__ALT__err_bvii = Sumo__NRE__ALT__bvii-Sumo__NRE__ALT__avii-Sumo__NRE__ALT__evi/2.0;
Sumo__NRE__ALT__fvii = Sumo__NRE__ALT__bvii-Sumo__NRE__ALT__avii/2.0;
Sumo__NRE__ALT__err_evii = Sumo__NRE__ALT__evii-Sumo__NRE__ALT__hvi-Sumo__NRE__ALT__fvii/3.0;
Sumo__NRE__ALT__gvii = Sumo__NRE__ALT__evi-1/( Sumo__SmallNumber + Sumo__NRE__ALT__fvii);
Sumo__NRE__ALT__err_hvii = Sumo__NRE__ALT__hvii-Sumo__NRE__ALT__gvii-2;
// algebraic loop end
]]></code>
          </solver>
        </blocksection>
      </block>
      <block name="Equilibrium" />
      <block name="Integrated">
        <blocksection id="1">
          <solver type="ode" name="bdf">
            <code><![CDATA[
call_AlgebraicLoop_1_ode();
call_AlgebraicLoop_2_ode();
call_AlgebraicLoop_3_ode();
call_AlgebraicLoop_Sumo__NRE__ALT__sectionV_ode();
call_AlgebraicLoop_Sumo__NRE__ALT__sectionIV_ode();
call_AlgebraicLoop_Sumo__NRE__ALT__section_ode();
call_AlgebraicLoop_7_ode();
call_AlgebraicLoop_Sumo__NRE__ALT__hvii_ode();
]]></code>
          </solver>
```

Figure 47 XML representation of the manual algebraic loop solution.

The examples can be found in the `references` subfolder of the documentation installed with the Sumo software.