STGraph how-tos

vers. 8.8.10

Table of Contents

Note: the references to the STGraph UI in these how-to*s* are to menu items; the possible correspondence to contextual menus (right click) and toolbars are left to the reader.

Note: in all examples of these how-to*s* the index origin of arrays is assumed to be 0 (it can be set to 1 in the model definition dialog (`[Edit | Edit Model Definition...]`).

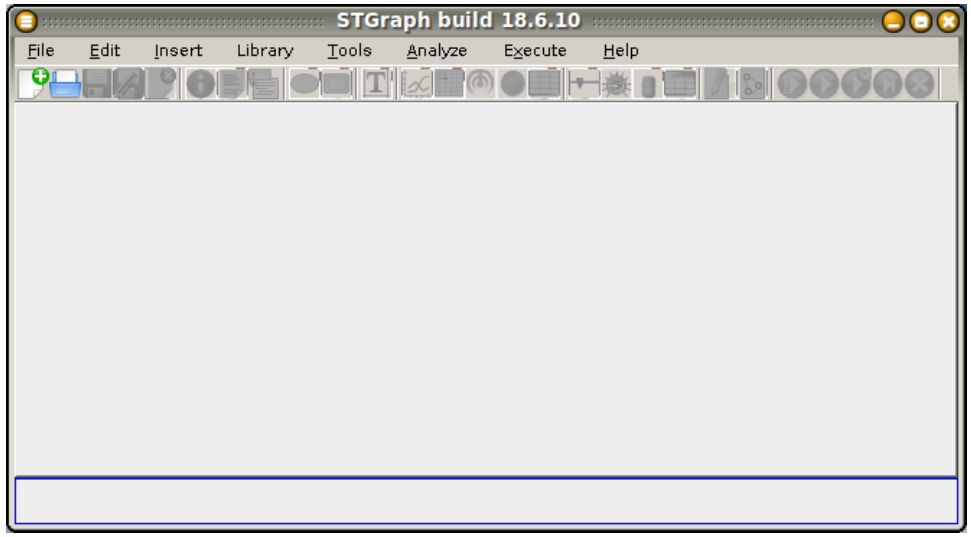
How to create a basic model

Introduction

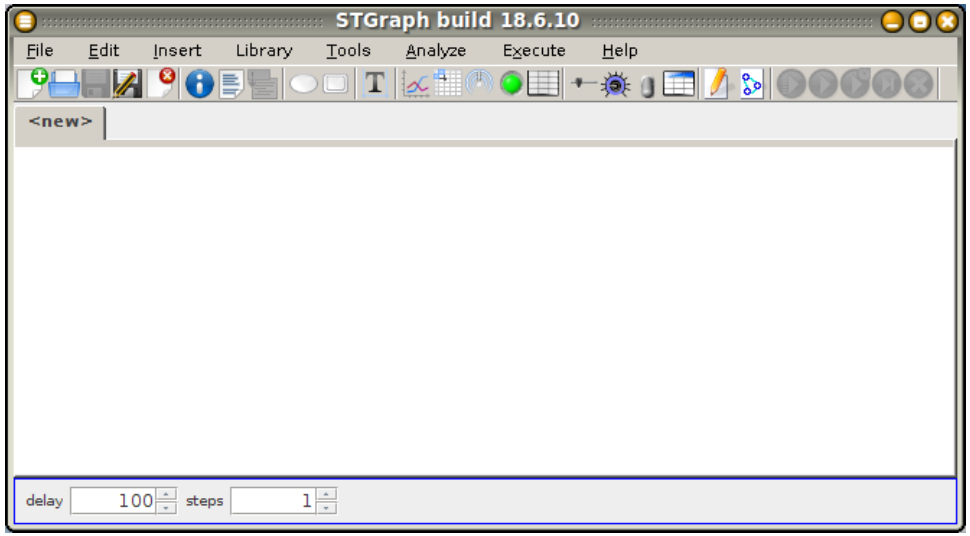
The main front-end of STGraph is a GUI interactive editor, which allows the user to create graphs with usual drag-and-drop techniques. As usual the same result can be usually obtained with different techniques (menu item selection, toolbar icon selection, contextual right-click, keyboard shortcut).

A step-by-step procedure

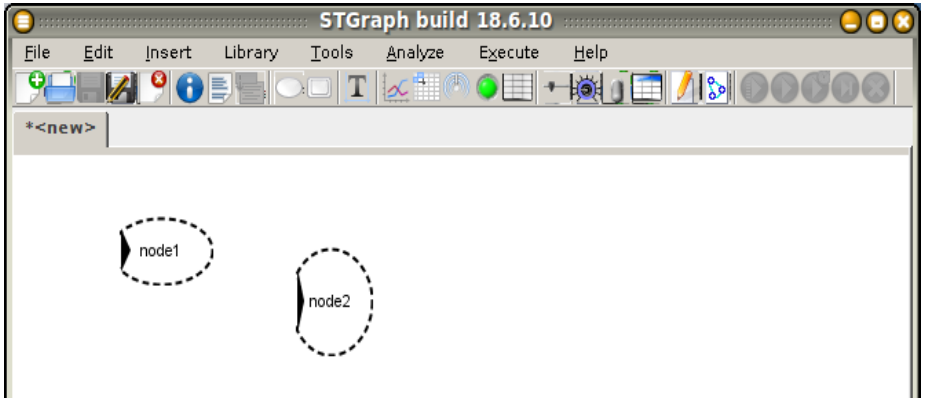
When just started, STGraph presents the following window:



To create a new model select `[File | Create New Model]` (or click on the first icon of the toolbar, or ctrl-N: in the following such lists of alternatives will be avoided). This is the result:

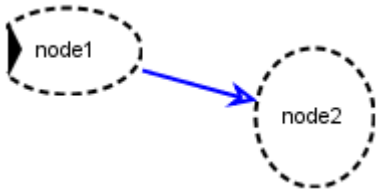


A blank, still unnamed, sheet has been created. Let us introduce a couple of nodes with `[Insert | Insert Variable Node]` and move / resize them with the usual techniques.

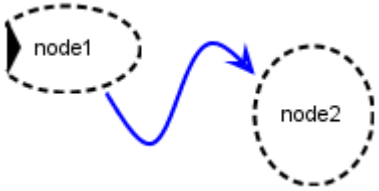


By moving the mouse pointer to the center of the node `node1`, and drag-and-drop to `node2`, we create an arrow from `node1` to `node2`:

How to create a basic model



Control points can be added and removed to the arrow by shift-left clicking:



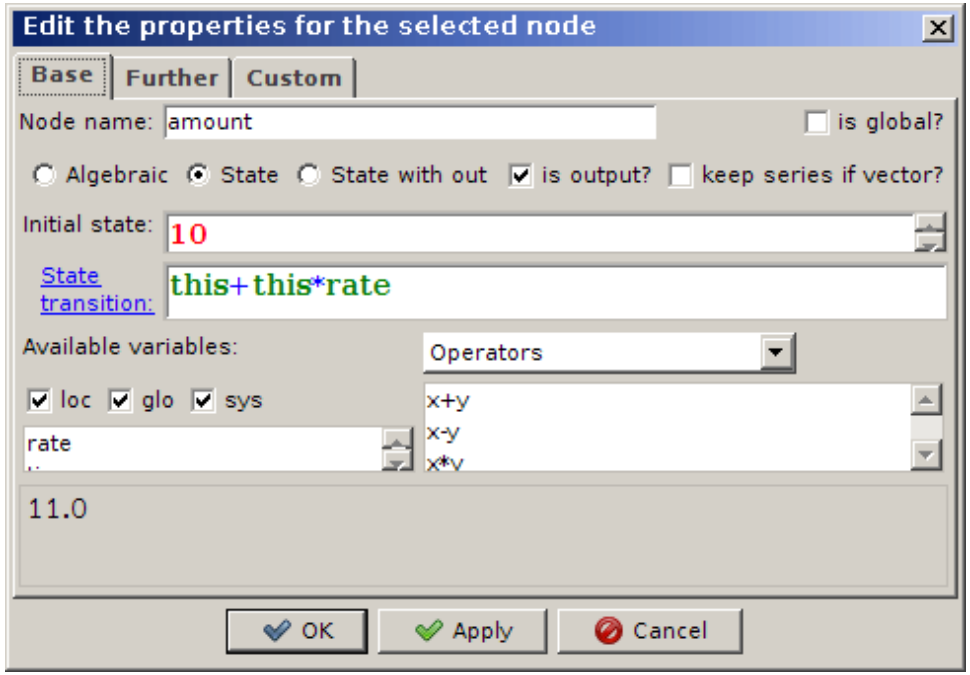
The graphical editor always checks that the graph is syntactically correct, and in particular prevents introducing parallel and loop arrows.

Node shapes are automatically set by STGraph: in particular ellipses represent algebraic variables (the default type), and the black arrow entering a node indicates that it is an input node, i.e., it has no incoming arrows.

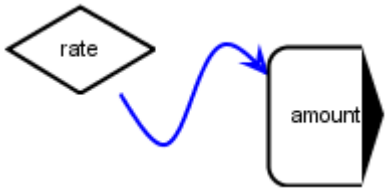
By double clicking on the two nodes, their property window opens and some of their properties can be changed:

- change the default name `node1` to `rate` (note that STGraph is case sensitive); set `0.1` as output function;
- change the default name `node2` to `amount`; change its type to 'state'; set `10` as initial state and `this+this*rate` as state transition; finally, set the 'is output?' switch on.

This is the property window for the second node:



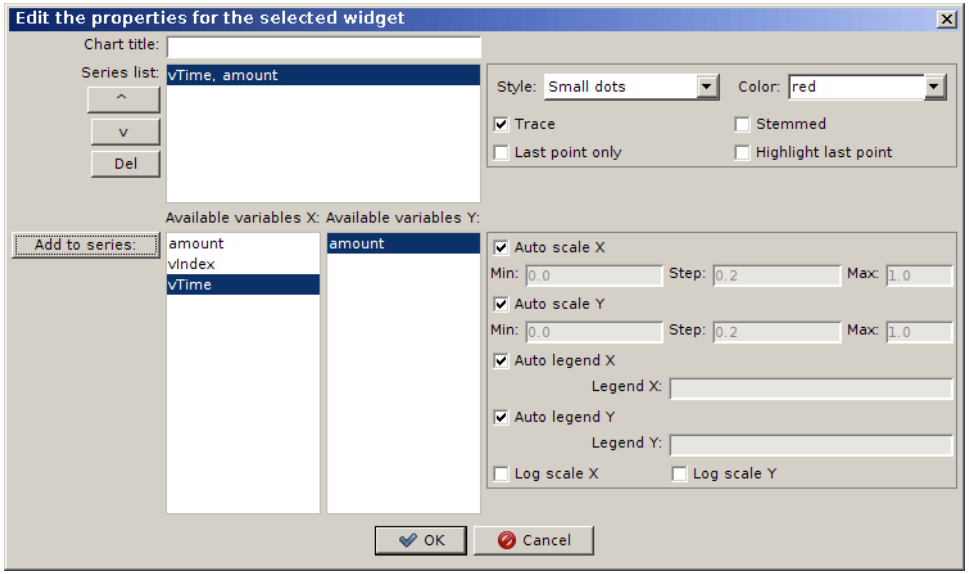
and this is the resulting graph:



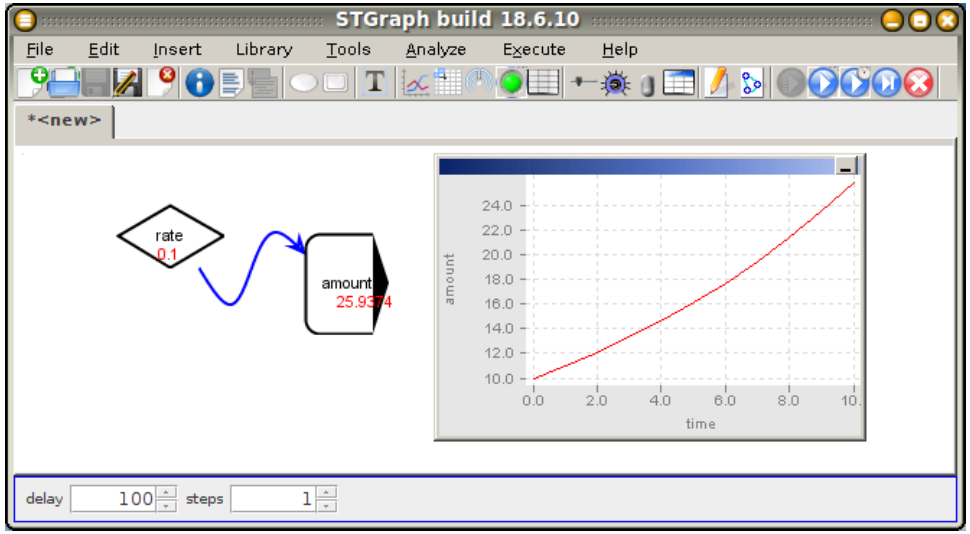
The shape contour of the nodes is now continuous, to represent the fact that their quantitative component has been correctly defined. Furthermore, note the new shapes of `rate` (a constant) and `amount` (a state). The black arrow leaving a node indicates that it is an output node, i.e., a node whose values can be observed by some external device or model.

How to create a basic model

Let us introduce one of such devices: an output widget, for example a chart, by selecting `[Insert | Insert Chart as Output Widget]`, and modify its properties by double clicking in its (still empty) image. Let us introduce a new series, having `vTime` (the system defined vector of simulation time instants) as X variable and `amount` as Y variable (note that `amount` is listed among the observable variables because it was set as an output variable).



The model can be now simulated, by selecting `[Execute | Run the Simulation Process]`, or in interactive, step-by-step, execution, by the related menu items (but take a look at the toolbar icons...). This is the result:



The chart X axis shows that the simulation has been run from time `0` to `10`. These are global properties of the model, that can be changed by selecting `[Edit | Edit Model Definition]` (double clicking the sheet background).

How to document a model

Introduction

A good model is a *documented* model, i.e., a model whose components can be easily and readily understood.

STGraph offers several means for introducing documentation items directly and contextually in the model, and then for possibly exporting them in a structured way to external files.

All documentation items are optional.

Model documentation

`[Edit | Edit definition...]` (while neither nodes nor widgets are selected) opens a dialog allowing to set a name and a general description for the model, together with a name for the time unit. This data is included in the `Definition data of the model` window (`[Analyze | Show Definition...]` while neither nodes nor widgets are selected) and in the `Complete information of the model` window (`[Analyze | Show Complete Info on Model...]`).

`[Insert | Insert Text Comment]` inserts a text comment that can be edited and formatted with a point-and-click by a WYSIWYG editor. The whole graph image, including the inserted texts, can be exported to external programs via the clipboard by `[Tools | Copy Graph Image to Clipboard]`.

Node documentation

The dialog for editing node properties (`[Edit | Edit Definition...]` while a node is selected) allows to insert various documentation items related to the currently selected node:

- tab Further: a Documentation text can be set;
- tab Custom: several custom properties can be set (see How to operate with custom properties).

This data is also included in the `Complete information of the model` window (`[Analyze | Show Complete Info on Model...]`).

How to operate with custom properties

Introduction

With each graph node a list of optional custom properties is associated, each of them expressed as a couple `name=value`. While such properties can be freely assigned for documentation, some of them, with given `name`s, are predefined together with their behavior, as follows (note that `name`s are case-sensitive).

General properties

- `Name`: descriptive name of the node (it can include spaces and other special characters), displayed alternatively to the variable name in the node tooltip
- `Unit`: evaluation / measurement unit, displayed in the node tooltip

Properties related to interrupt conditions

(these properties are used in computation only if in the model definition dialog (`[Edit | Edit definition...]` while neither nodes nor widgets are selected) the `Handle interrupts?` switch is activated)

- `Min`: minimum allowed value, displayed in the node tooltip (also used in STGraphWeb)
- `Max`: maximum allowed value, displayed in the node tooltip (also used in STGraphWeb)

All the following `On...` properties execute the macro specified by `value` and operate on the current value only for scalar nodes:

- `OnBelowMin`: execute if less than `Min`
- `OnAboveMax`: execute if greater than `Max`
- `OnZero`: execute if equal to zero
- `OnTrue`: execute if greater than zero
- `OnFalse`: execute if equal or less than zero

The macro can be:

- `pause("`*`message`*`")`: pause and display *`message`* in a dialog to continue or stop the simulation
- `end("`*`message`*`")`: display *`message`* in a dialog and stop the simulation

Properties used in STGraphWeb

(see How to generate a model for the web).

How to operate with arrays

How to operate with arrays

Introduction

STEL has a single data type: *array*.

Arrays are `n` dimensional structures (`n` being a non-negative integer), characterized by their order, i.e., the number `n` of their dimensions, and their size, i.e., the number of elements for each dimension. Hence:

- `0`-order arrays are scalars
  (e.g., `-4`, `1.2`, and `2.3E4` are scalars);
  the size of scalars is `0`;
- `1`-order arrays are vectors of scalars, and their size is a non-negative scalar
  (e.g., `[-4, 1.2, 2.3E4]` is a `[3]`vector, i.e., a vector of size `3` because with `3` scalar elements)
  (e.g., `[]` is a `[0]`vector, i.e., a vector of size `0`);
- `2`-order arrays are matrices of scalars, in fact (column) vectors of (row) vectors
  (e.g., `[[-4,1.2,2.3E4],[2,3,4]]` is a `[2,3]`matrix, i.e., of size `2`x`3` because with `2` rows and `3` columns, and therefore with globally `6` scalar elements)
  (e.g., `[[]]` is a `[0,0]`matrix);
- higher order arrays can also be generated
  (e.g., `[[[1,2],[3,4]],[[5,6],[7,8]]]` is a `[2,2,2]`tensor).

Any `n`>`0`-order array can be thought of as a sequence of `n`-`1`-order subarrays
(e.g., a `[2,3]`matrix as a sequence of `2 [3]`vectors)
(e.g., a `[4,3,2]`tensor as a sequence of `4 [3,2]`matrices, and therefore, recursively each of them as `3 [2]`vectors).

Arrays have always a normal form, i.e., all their subarrays have the same size.

Arrays are dynamic structures, since the number of elements of any of their dimensions can be increased by adding new elements and decreased by removing existing elements.

Each element of an `n`>`0`-order array is identified by a `[n]`vector of indexes.

Generating an array

While a `0`-order array is directly defined by its scalar value, an `n`>`0`-order array can be generated in several ways, basically:

- for each dimension, by explicitly listing the array elements
  (e.g., the `[3]`vector `[10,20,30]`)
  (e.g., the `[2,3]`matrix `[[1,2,3],[4,5,6]]`, whose element in `0`-th row and `0`-th column is the scalar `1`, the element in `0`-th row and `1`st column is the scalar `2`, and so on)
  or by defining the sequence of the array elements by means of the operator `:`
  (e.g., `[3:5]` is the `[3]`vector `[3,4,5]`)
  (e.g., `[6:4]` is the `[3]`vector `[6,5,4]`)
  (e.g., `[2:3:8]` is the `[3]`vector `[2,5,8]`)
  (e.g., `[[1:3]]` is the `[1,3]`matrix `[[1,2,3]]`)
  (e.g., `[[1:3],[2, 3, 4]]` and `[[1:3],[2:4]]` are both the `[2,3]`matrix `[[1,2,3],[2,3,4]]`);

How to operate with arrays

- by applying the function `array(s,x)`, where *s* is the *n*-order vector $[s_0,...,s_{n-1}]$, that generates a *n*-order $[s_0,...,s_{n-1}]$ array by evaluating the expression *x* for all the array elements
  (e.g., `array([2,3],0)` generates the `[2,3]` matrix `[[0,0,0],[0,0,0]]`);
  the expression *x* can contain the system variables `$i0`, `$i1`, …, each of them spanning over the corresponding index range, i.e., from `0` to s_0-1 for the `0`-th dimension, and so on
  (e.g., `array([3],$i0^2)` (or equivalently `array(3,$i0^2)`) generates the `[3]` vector `[0,1,4]`)
  (e.g., `array([2,2],$i0*$i1)` generates the `[2,2]` matrix `[[0,0],[0,1]]`).

It is worth noting some distinctions:

- `123` and `[123]` are different, as the first is a scalar and the second is a `[1]` vector;
- `[1,2,3]`, `[[1,2,3]]`, and `[[1],[2],[3]]` are different, as the first is a `[3]` vector, the second is a `[1,3]` matrix, and the third is a `[3,1]` matrix.

The empty vector is `[]`, the empty matrix is `[[]]`, and so on, while by convention the empty scalar is `0.0`.

By means of the function `array` such empty arrays can be generated by `array([0],0)`, `array([0,0],0)`, and so on. This shows that an empty *n*-order array can be parametrically generated by `array(array(n,0),0)`.

Getting the size and the order of an array

The function `size(a)` (shortcut: `@a`) returns the size of the array *a* (e.g., `@[[1,2,3],[4,5,6]]` returns `[2,3]`).

The function `order(a)` returns the order of an array (e.g., `order([[1,2,3],[4,5,6]])` returns `2`).

Modifying an array

An *n*>`0`-order array *a* can be modified in several ways, basically:

- by resizing *a* by means of the function `resize(a,v)`, where *v* is the vector of the new size
  (e.g., `resize([0,1,2,3,4,5],[2,3])` generates the `[2,3]` matrix `[[0,1,2],[3,4,5]]`);
  the resizing operation is performed by possibly trimming the exceeding elements or padding with `0` the missing elements
  (e.g., `resize([[1,2],[3,4]],[3])` generates the `[3]` vector `[1,2,3]`) and `resize([[1,2],[3,4]],[5])` generates the `[5]` vector `[1,2,3,4,0]`);
- by transposing *a* by means of the function `transpose(a)`, that generates an array of the same order than *a*, and whose first dimension is the last one of *a* and so on
  (e.g., `transpose([[[1.0,2.0,3.0],[4.0,5.0,6.0]]])`, that is a `[1,2,3]` array, generates the `[3,1,2]` array `[[[1.0,4.0]],[[2.0,5.0]],[[3.0,6.0]]]`);
- by concatenating *a* with an array *b* by means of the function `conc(a,b)` (shortcut: `a#b`).
  As a general rule, *a* and *b* must be of the same order and if *n* is such order then the size of the first $n-1$ dimensions must

How to operate with arrays

be the same, so that, e.g, from the concatenation of a `[4,3,2]` array with a `[4,3,1]` array a `[4,3,3]` array is obtained
(e.g., `[[1,2],[3,4],[5,6]]#[[7],[8],[9]]` generates the `[3,3]` matrix `[[1,2,7],[3,4,8],[5,6,9]]`).
A simplification is allowed to this rule, as the order of *b* can be *n*`-1` and the size of the first *n*`-1` dimensions of *a* and the dimensions of *b* must be the same (the same holds by reversing *a* and *b*); in this way, in particular, scalars and vectors can be freely concatenated with each others, and a vector is generated
(e.g., `1#2`, `[1]#2` and `[1]#[2]` all generate the `[2]` vector `[1,2]`)
and $[i_0,i_1]$ matrices can be concatenated with $[i_0]$ vectors
(e.g., `[[1,2],[3,4],[5,6]]#[7,8,9]` generates the `[3,3]` matrix `[[1,2,7],[3,4,8],[5,6,9]]`);

- by removing (i.e., "decatenating") *b* elements from the last dimension of *a* by means of the function `dec(a,b)` (shortcut: *a*`##`*b*) (the same holds by reversing *a* and *b*).
(e.g., `[[1.0,2.0,3.0],[4.0,5.0,6.0]]##1` generates the `[2,2]` matrix `[[1,2],[4,5]]`)
(e.g., `[[1.0,2.0,3.0],[4.0,5.0,6.0]]##2` generates the `[2,1]` matrix `[[1],[4]]`)
(e.g., `2##[[1.0,2.0,3.0],[4.0,5.0,6.0]]` generates the `[2,1]` matrix `[[3],[6]]`);

Extracting elements / subarrays from an array

Both single elements and subarrays can be extracted from an *n*`>0`-order array *a* by means of the function `get(`*a*`,`i_0`,`i_1`,...)` (shortcut: *a*`[`i_0`,`i_1`,...]`), where $i_0,i_1,...$ are indexes, respectively for the dimension `0`, `1`, …

Each index can be either the null vector (i.e., `[]`), or a scalar, or a non-null vector:

- if it is the null vector (or the index is not specified), then all components for the related dimension are selected;
- if it is a scalar, then the component whose index is the scalar for the related dimension is selected;
- if it is a non-null vector, then all components whose indexes are in the vector for the related dimension are selected

(e.g., if *a* is a `[3,3]` matrix then:
- *a*`[0,1]`, and *a*`[[0],[1]]` are equivalent and extract the element at the `0`-th row and `1`-th column of *a*;
- *a*`[0]`, *a*`[[0]]`, *a*`[0,[]]`, and *a*`[[0],[]]` are equivalent and extract the `0`-th row of *a*;
- *a*`[[],1]` and *a*`[[],[1]]` are equivalent and extract the `1`-th column of *a*;
more advanced usages are:
- *a*`[[0,1]]` extracts the `[2,3]` matrix of the `0`-th and the `1`-th rows of *a*;
- *a*`[[2,1,0]]` extracts the `[3,3]` matrix of the `2`-th, the `1`-th, and the `0`-th rows, in this order, of *a*;
- *a*`[[0,2],[0,2]]` extracts the `[2,2]` matrix of the elements of *a* whose indexes are both even numbers).

Setting elements / subarrays of an array

Complementary to the function `get(`*a*`,`i_0`,`i_1`,...)`, the function `set(`a_1`,`i_0`,`i_1`,...,`a_2`)` substitutes the subarray `get(`*a*`,`i_0`,`i_1`,...)`

with the array a_2.

Applying functions to an array

STEL is equipped with a fair amount of functions and operators, many of them behaving in a polymorphic way on arrays:

- monadic functions are computed element by element (e.g., `-[0:2])` is equivalent to `[-0,-1,-2]`, and `sin([0:2])` is equivalent to `[sin(0),sin(1),sin(2)]`);
- dyadic functions are computed element by element in two cases:
  - when the two operands have the same size (and therefore the same order)
  (e.g., `[[0,1],[2,3]]+[[10,20],[30,40]]` equals to `[[10,21],[32,43]]`);
  - when one operand is scalar
  (e.g., `[[0,1],[2,3]]^2` equals to `[[0,1],[4,9]]`);
  possible, i.e., in the cases (x stands for a vector or a matrix and s for a scalar; commutativity is left to the reader).

Finally, the available iterator meta-functions (reduction $f/$, scan $f\backslash$, pairscan $f|$, and `iter`) where f is a dyadic function, can be applied to any $n>0$-order array
(see How to operate with iterator meta-functions).

How to operate with the conditional function if()

Introduction

Being a purely functional language, STEL does not include any statement, and control structures are expressed by means of functions in their turn. This applies in particular to the conditional structure.

The conditional function

The `if()` function operates as a chain of if ... then ... else if ... then ... Its simplest form:

`if(`c`,`v_1`,`v_2`)`

is equivalent to the structure if c then v_1 else v_2.

The general form of the conditional function is:

`if(`c_1`,`v_1`,...,`c_n`,`v_n`,`v_{n+1}`)`

with an odd number ≥3 of arguments, and such that the arguments in odd position but the last one are conditions c_i and the other ones are values v_j.

The first true condition selects the value that follows in the argument list, whereas the last value is selected if all conditions are false

(e.g., `if(1,2,3)` equals to `2`)
(e.g., `if(0,2,0,3,4)` equals to `4`).

The conditional function behaves “as polymorphically as possible”: in particular, if c_1, v_1, and v_2 arrays of the same dimension, it produces an array of that dimension, such that (in the case of vectors for the sake of simplicity) `if(`c`,`v_1`,`v_2`)` is a vector v_3 such that for each i, v_3`[`i`]` equals to v_1`[`i`]` if c_1`[`i`]` is true and to v_2`[`i`]` otherwise

(e.g., `if([1,0],2,3)` equals to `[2,3]`).

Accordingly, the rule for the general form is: the conditions c_i must have the same dimension; if the conditions are scalars, the values v_j are not constrained in their dimension; otherwise they must have the same dimension of the conditions or they must be scalars.

How to operate with iterator meta-functions

How to operate with iterator meta-functions

Introduction

By default, in STEL any dyadic function is computed element by element
(e.g., `max([2,4],[3,1]` equals to `[3,4]`)
(e.g., `[[0,1],[2,3]]+[[10,20],[30,40]]` equals to `[[10,21],[32,43]]`).

On the other hand, it is sometimes useful for a dyadic function to be applied in a different way, such as in the case of summation, where the same operator, `+`, is computed on the first two elements of the given vector, then on the result and the third element, and so on.

This behavior can be generalized by admitting that any dyadic operator can be iteratively computed on the elements of an array. To this goal, STEL includes some iterator meta-functions: *reduction, scan*, and *pairscan*, that have a similar simple syntax and behavior, and that can be applied only to the STEL predefined operators, and the more general and complex *iter*

(note that several references are made here to the contents of How to operate with arrays).

The reduction meta-function

For any dyadic operator f and any n>0-order array a, the expression f/a can be computed, where `/` is called the *reduction* meta-function.

In the case a is a vector, the result is the scalar
$f(f(a[0],a[1]),a[2])...$
(e.g., `+/[1:4]` equals to `((1+2)+3)+4`, i.e., summation)
(e.g., `*/[1:4]` equals to `((1*2)*3)*4`, i.e., factorial product).

In the case of a higher order array, the operator is iteratively computed along its last dimension, so that in the case of a matrix the result is
$f(f(a[],0],a[],1]),a[],2])...$
(e.g., `+/[[1,2],[3,4],[5,6]]` equals to `[1+2,3+4,5+6]`, i.e. `[3,7,11]`)
(e.g., `-/[[1,2,3],[4,5,6]]` equals to `[(1-2)-3,(4-5)-6]`, i.e. `[-4,-7]`).

For the same reason, if a is a matrix then `+/+/`a is the sum of all its elements.

Hence, if a is n-order $[s_0,...,s_{n-2},s_{n-1}]$ array, then f/a is a n-1-order $[s_0,...,s_{n-2}]$ array.

The reduction meta-function:

- if applied to an empty n-order array, a scalar or an array whose last dimension has size `1`, returns an empty n-1-order array.

The scan meta-function

For any dyadic operator f and any n>0-order array a, the expression $f\backslash a$ can be computed, where `\` is called the *scan* meta-function.

In the case a is a vector, the result is the vector
$[a[0],f(a[0],a[1]),f(f(a[0],a[1]),a[2]),...]$
(e.g., `+\[1:4]` equals to `[1,1+2,(1+2)+3,((1+2)+3)+4]`).

How to operate with iterator meta-functions

Hence, if a is n-order $[s_0,\ldots,s_{n-2},s_{n-1}]$ array, then $f\backslash a$ is a n-order $[s_0,\ldots,s_{n-2},s_{n-1}]$ array.

The scan meta-function:

- if applied to an empty n-order array, returns an empty n-order array;
- if applied to a scalar or an array whose last dimension has size `1`, returns the array itself.

The pairscan meta-function

For any dyadic operator f and any n>`0`-order array a, the expression $f|a$ can be computed, where `|` is called the *pairscan* meta-function.

In the case a is a vector, the result is the vector `[f(a[0],a[1]),f(a[1],a[2]),f(a[2],a[3]),...]` (e.g., `+|[1:4]` equals to `[1+2,2+3,3+4]`).

Hence, if a is n-order $[s_0,\ldots,s_{n-2},s_{n-1}]$ array, then $f|a$ is a n-order $[s_0,\ldots,s_{n-2},s_{n-1}-1]$ array.

The pairscan meta-function:

- if applied to an empty n-order array, a scalar or an array whose last dimension has size `1`, returns an empty n-order array.

The iter meta-function

Some computation can be iteratively performed on any n>`0`-order array a by means of the meta-function `iter(a,e,z)`, where e is an expression to be iteratively computed and z is the “zero” element for such expression. The computation of e is repeated `@a` times. e can contain some system-defined variables, whose value in each iteration is set as follows:

- in each step the variable `$i` is set to the step index;
- the variable `$0` is initially set to z and in each step is set to the result obtained in computing e;
- in each step the variable `$1` is set to the element of the array a whose index is `$i`,

so that, in particular, if a is a vector, `iter(a,$0,z)==z` (“null iteration”) and `iter(a,$1,z)==a[@a-1]` (“memoryless iteration”). Moreover, `iter(a,$0#$1,[])==a`.

Together with the dyadic function `max`, the meta-function `iter` can be used, for example, to find the maximum in the elements of a vector v as `iter(v,max($0,$1),0)`, under the assumption that all elements are non-negative, so that `0` is a proper “zero” element for the expression `max($0,$1)`.

To see how `iter` operates, let us assume that v is the `[3]`vector `[1,4,2]`. The execution trace of `iter(v,max($0,$1),0)` is then as follows:

- step 0: `result ← $0 ← 0` (the “zero” element);
- step 1: `$0 ← result; $i ← 0; $1 ← v[$i]; result ← max($0,$1)` (i.e., `result ← 1 ← max(0,1)`);
- step 2: `$0 ← result; $i ← 1; $1 ← v[$i]; result ← max($0,$1)` (i.e., `result ← 4 ← max(1,4)`);
- step 2: `$0 ← result; $i ← 2; $1 ← v[$i]; result ← max($0,$1)` (i.e., `result ← 4 ← max(4,2)`).

How to operate with iterator meta-functions

Hence, `iter(v,max($0,$1),0)` actually computes `max(max(max(0,1),4),2)`.

Because of the polymorphism of the function `max`, the same logic can be applied to higher order arrays, so that, for example, `iter([[3,5],[4,2]],max($0,$1),[0,0])` returns `[5,4]`.

It can be demonstrated that `iter` generalizes the reduction, scan, and pairscan meta-functions; under the simplifying hypothesis that by means of `iter` a function *f* is computed on a vector *v*, being *z* the "zero" element of *f*:

- *f*`/`*v* `== iter(`*v*`,`*f*`($0,$1),`*z*`)`;
- *f*`\`*v* `== array(@`*v*`,if($i0==0,`*v*`[0],iter(`*v*`[[0:$i0]],` *f*`($0,$1),`*z*`)))`;
- *f*`|`*v* `== iter(`*v*`[[0:@`*v*`-2]],$0#(`*v*`[$i]+`*v*`[$i+1]),[])`.

Note that also:

- *f*`\`*v* `== iter(x,$0#if($i==0,`*v*`[0],iter(`*v*`[[0:$i]],` *f*`($0,$1),`*z*`)),[])`

showing that `iter` calls can be nested.

Some examples

For any vector *a*, `(+/`*a*`)/@a` returns the average value of the elements of *a*.

For any vector *a* whose average value is *m*, `sqrt((+/(`*a*`-`*m*`)^2)/(@a-1)` returns the sample standard deviation of the elements of *a*.

For any *n*>0-order array *a*, `resize(`*x*`,[*/@`*a*`])` generates a linearized, vector version of *a*.

For any pair of vectors *a* and *b* with the same size, `+/(`*a*`*`*b*`)` returns the inner (scalar) product of *a* and *b*.

For any vector *a* and any scalar *b*, `+/(`*a*`==`*b*`)` returns the number of elements of *a* that are equals to *b*.

For any vector *a*, `iter(`*a*`,$1#$0,[])` returns the reversed version of *a*.

For any vector *a*, `iter(`*a*`,if($1!=0,$0#$1,$0),[])` returns a version of *a* in which all `0`s are removed.

For any vector *a*, `iter(`*a*`,$0#$1#0,[])` returns a version of *a* whose elements are interleaved with `0`s.

For any pair of matrices *a* and *b* that can be multiplied with each other, `array([get(@`*a*`,0),get(@`*b*`,1)],+/(`*a*`[$i0]*`*b*`[[],$i1]))` is their vector product.

How to operate with submodels

Introduction

Whenever a model grows in complexity, the number of nodes and arrows of its graph can become hard to manage. Each model can be included in a graph as a single node, being thus dealt with as a submodel of the including supermodel. As several instances of the same model can be included in the supermodel, and each of them can be independently characterized in its behavior, this option introduces a simple object-oriented management of models.

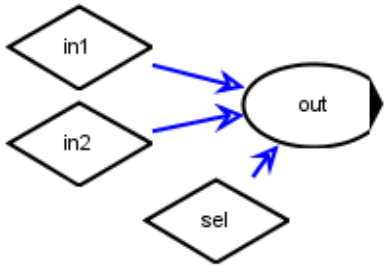
Creating and using a submodel

Each model can be included as a model node in a graph, so that the resulting model can be structured in a hierarchy of a supermodel, that can contain some submodels, that can contain sub-submodels, and so on.

The only constraint for a model to be used as a submodel is that the *.stg files of both the contained model and the container model must be stored in the same directory: in this way, the files can be freely moved to a different directory and possibly a different computer.

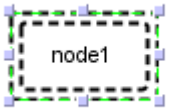
Let us suppose that the model operations.stg has been created as follows:

```
in1: value = 0
in2: value = 0
sel: value = 0
out: value = if(sel,in1+in2,in1-in2)
```

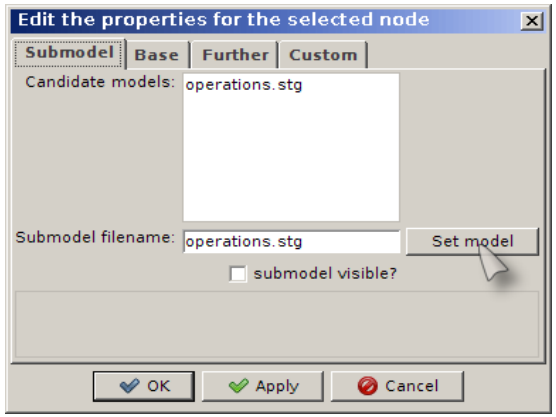


This is going to be used as a submodel for a model created and then saved in the same directory as operations.stg.

In the empty supermodel, let us introduce a model node (`[Insert | Insert Model Node]`):



and set its associated submodel by double clicking on the node:



How to operate with submodels

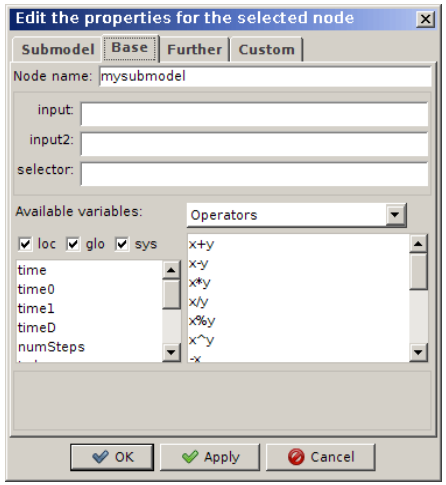
(note that the list of candidate models is simply that of all models stored in the directory of the current model).

After having renamed the node, this is the result:

showing that STGraph has recognized operations.stg as a correct (continuous contour) algebraic (elliptical shape instead of roundangle, as it were have been in the case operations.stg contains one or more state nodes) submodel.

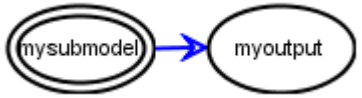
Now the node `mysubmodel` wraps the whole submodel stored in operations.stg: the inputs to `mysubmodel` are the submodel inputs, and the outputs from `mysubmodel` are the submodel outputs:

- submodel inputs can be set in the property window of the model node, that is in fact updated dynamically with the list of the submodel inputs:

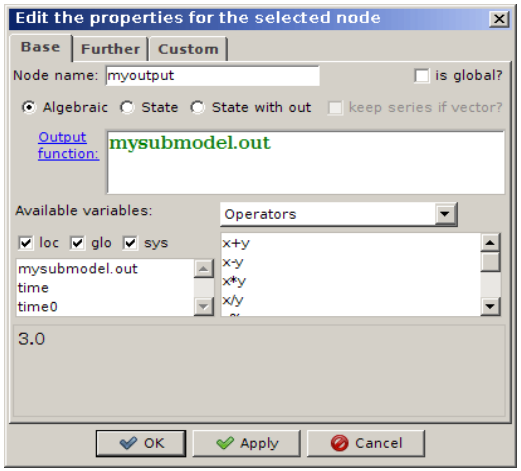


each input can be set, as usual, by specifying an expression that possibly references the variables that are visible to the model node (because an arrow has been drawn from them to the model node or they have been set as global), but it can also be left blank: in this case, the value specified in the submodel is maintained as a default;

- submodel outputs can be obtained by linking one or more nodes from the model node:



because of the arrow linking them from the model node, such nodes can access the submodel output variables with the usual object-oriented syntax *`model_node.output_var`*:



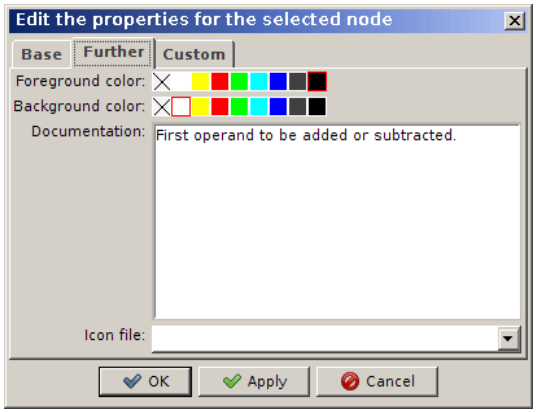
How to operate with submodels

Documenting a submodel

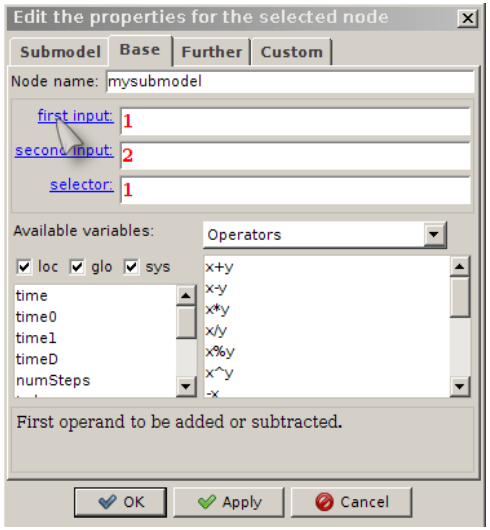
To make a model more suitably usable as a submodel, it can be properly documented.

First of all, the name of each submodel input variable the model node property window displays the value of the Name custom property (see How to operate with custom properties), that can be then specified so to make it better understandable the variable role in the model. Furthermore, a Documentation text can be set for its input and output variables, as explained in How to document a model.

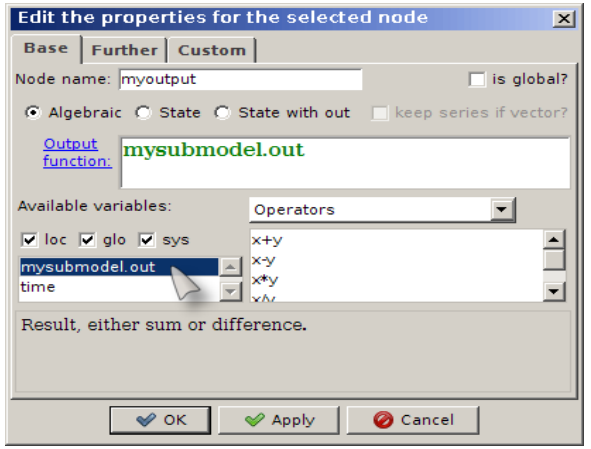
If, for example, the following description is set for the variable `in1`:



then when the model node property window is opened and the mouse pointer is moved on the variable name the corresponding text is displayed in the help area of the window:



Analogously, if a submodel variable is selected in the property window of the nodes linked from the model node, the help area of the window will display its documentation text:



How to define a new function

How to define a new function

Introduction

STEL enables the user to define new functions by means of two complementary techniques: in graph nodes, so that the function scope is the model, or in external XML files, so that functions are visible to all models.

Functions defined in graph nodes

Each node whose names starts with an '_' (underscore) is assumed being of algebraic type and having an expression:

`function(e)`

where `e` is the expression defining the function, whose name is the node name itself.

The arguments of the defined function are not declared, and are directly referred in the expression as `$a0`, `$a1`, … (currently up to `$a3`).

For example, if the expression of the node `_x` is:

`function(($a0+$a1)/2)`

then the evaluation of `_x(2,3)` produces the result `(2+3)/2)`.

Furthermore, the expression `e` can contain the system variable `$numArgs`, that is set to the number of arguments currently passed to the function.

The meta-function `function(e)` allows recursive definitions. For example, the factorial product function can be defined in a node named `_fact` as follows:

`function(if($a0==0,1,$a0*_fact($a0-1)))`

so that calling, e.g., `_fact(4)` produces as result `24`.

Functions defined in XML files

The directory ./fun_lib, where '.' is the STGraph installation directory, can contain one or more XML files with extension 'stf'. Each of them is aimed at containing the definition of one or more new functions.

Each *.stf file has the following syntax:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM
"http://java.sun.com/dtd/properties.dtd">
<properties>
<entry key="...">...</entry>
<entry key="...">...</entry>
...
</properties>
```

Each item `<entry>` contains the definition of a new function according to the syntax:

```
<entry key="menu_and_function_name">
function_definition
// <![CDATA[
function_documentation
]]>
</entry>
```

where:

- *menu_and_function_name* contains the multilingual

How to define a new function

definition of the menu name in which the function will be included and the function name

(e.g.,

```
<entry key="en:Mathematical functions__
it:Funzioni matematiche|abs">
```

)

where '`|`' is the separator menu names – function name, '`__`' (double underscore) is the separator between language-dependent versions of menu name, in which '`:`' is the separator language ISO code – menu name;
hence the previous example is related to the definition of the function `abs`, that will be included in the English menu "Mathematical functions" and in the Italian menu "Funzioni matematiche";

- *`function_definition`* contains the expression defining the new function

(e.g.,

```
if($numArgs!=1,
"en:One argument is required__
it:E' richiesto un argomento",
if($a0>=0,$a0,-$a0)
)
```

)

everything has been stated in the previous Section "Functions defined in graph nodes", applies also in this case, with the exceptions listed below;

- *`function_documentation`* contains the multilingual documentation for the defined function

(e.g.,

```
en:<code>abs(x)</code>: absolute value of
<code>x</code>__
it:<code>abs(x)</code>: valore assoluto di
<code>x</code>
```

)

written with the same syntax as the menu names and possibly including HTML tags.

With respect to the functions defined in graph nodes, the functions defined in *.stf files:

- are not identified by the meta-function `function()`;
- can contain multilingual error messages as strings written with the same syntax as the menu names;
- can include multiple expressions, separated by '*`;`*' (semicolon); if *n*>1 expressions are present, the first *n*-1 expressions are evaluated in order, and the value of the *i*th expression is stored in the system variable `$vi`; finally, the last expression, that can contain the reference to the variables `$vi`, is evaluated and its result is returned as the function value; note that the scope of the variables `$vi` is local to the function.

The following is an example showing all these features (note that the indentation is optional).

```
<entry key="en:Array functions__
it:Funzioni array|lastDim">
order($a0);
if($numArgs!=1,
  "en:One argument is required__
  it:E' richiesto un argomento",
```

How to define a new function

```
$v0==0,
  "en:The argument cannot be a scalar__
  it:L'argomento non puo' essere uno scalare",
$v0==1,
  @$a0,
  get(@$a0,$v0-1)
) // <![CDATA[
en:<code>lastDim(x)</code>: number of elements in the
last (fastest) dimension of the array <code>x</code>__

it:<code>lastDim(x)</code>: numero degli elementi
nell'ultima dimensione dell'array <code>x</code>
]]>
</entry>
```

How to operate with Property Changer

How to operate with Property Changer

Introduction

As the number of nodes in a model increases, their efficient management becomes a critical requirement. Some tasks can be automated by means of the Property Changer tool (`[Tools | Property Changer]`), that takes statements of the general form:

`select` *`condition`* `do` *`action`*

(note that single spaces are used as token separators, and item are case sensitive) and executes them on the current model, by selecting the nodes that satisfy *`condition`* and performing *`action`* on them.

Specifications for conditions

The recognized conditions are:

- `all`: select all nodes;
- `current`: maintain the current selection;
- `name=`*`expr`*: select all nodes whose name matches *`expr`*, which can start or end with the wild char '`*`';
- `forecol=`*`expr`*: select all nodes whose foreground color is *`expr`*;
- `backcol=`*`expr`*: select all nodes whose foreground color is *`expr`*;
- *`cprop`*`=`*`expr`*: select all nodes whose custom property of name *`cprop`* has value *`expr`*.

Specifications for actions

The recognized action are:

- `nothing`: do nothing;
- `forecol=`*`expr`*: set foreground color to *`expr`*;
- `backcol=`*`expr`*: set background color to *`expr`*;
- *`cprop`*`=`*`expr`*: set custom property of name *`cprop`* to value *`expr`*.

Examples

`select all do nothing`: select all nodes;

`select name=out* do OutputType=2`: select all nodes whose name starts with "`out`" and set their custom property named "`OutputType`" to `2`;

`select forecol=green do backcol=yellow`: select all nodes whose foreground color is green and set their background color to yellow.

How to generate models for the web

How to generate models for the web

Introduction

STGraph is the tool to generate models that can be played on the web by means of the STGraphWeb server system. To this aim, not only a model must be correct, but some further constraints must be forced and some further information must be added to document the semantics of the model itself.

The following constraints are checked by `[Analyze | Check Model for Web...]`.

General rules

- time steps must range from `0` to the last round with `timeD = 1`;
- the `interrupts` and `save simulation data` model options must be switched off;
- general / market data are in scalar nodes, under the hypothesis that they are the same for all the teams;
- player data (both inputs / decisions and outputs / results) are in vector nodes: each player is associated to an element of the vector;
- each player is assumed to be identified by the corresponding vector index (from `0`);
- to make them observable, both input and output nodes must have the switches `output` and `keep series if vector` turned on;
- each initial value of a state node must be set via an input node;
- a constant (input) node called `Group` stores information on application-specific groups, through its custom properties `1 = description for group 1`, `2 = description for group 2`, and so on; both input and output nodes must have a custom property `Group`, with value either `1`, or `2`, ... to express that that node belongs to group `1`, `2`, ...

Node type identification

- only input nodes and output nodes are exposed to the web app;
- input nodes must have a custom property `InputType`, whose possible values are described below;
- input nodes with `InputType < 6` must have the custom properties `DefaultValue`, `Decimals`, and `Group`;
- output nodes must have a custom property `OutputType`, whose possible values are described below;
- output nodes must have the custom properties `Decimals` and `Group`;
- data dictionary includes input nodes with `InputType = 3` ("Decisions");
- data dictionary includes output nodes with `OutputType = 2` or `3` ("Results"), and `= 4` or `5` ("Market data"), and `= 6` ("Target variable").

How to generate models for the web

Input types

- `InputType = 0`: initial value of a state node: it must be vector;
- `InputType = 1`: exogenous, team number independent, input: it must be scalar;
- `InputType = 2`: exogenous, team number dependent: it must be scalar (i.e., value to be multiplied by the number of teams);
- `InputType = 3`: player decision: it must be vector;
- `InputType = 6`: constant parameters to be displayed to the players;
- `InputType = 7`: constant parameters not to be displayed to the players (typically for utility nodes, such as `Group` and custom functions).

Output types

- `OutputType = 0`: variable visible to administrators only (not to players);
- `OutputType = 1`: player decision (corresponding to `InputType = 3`);
- `OutputType = 2`: player result;
- `OutputType = 3`: player result that should not visualized if zero;
- `OutputType = 4`: general / market data;
- `OutputType = 5`: general / market data that should not visualized if zero;
- `OutputType = 6`: overall ("goal") variable for ranking players.