

SCICOS - A Dynamic System Builder and Simulator

User's Guide *

R. Nikoukhah

S. Steer

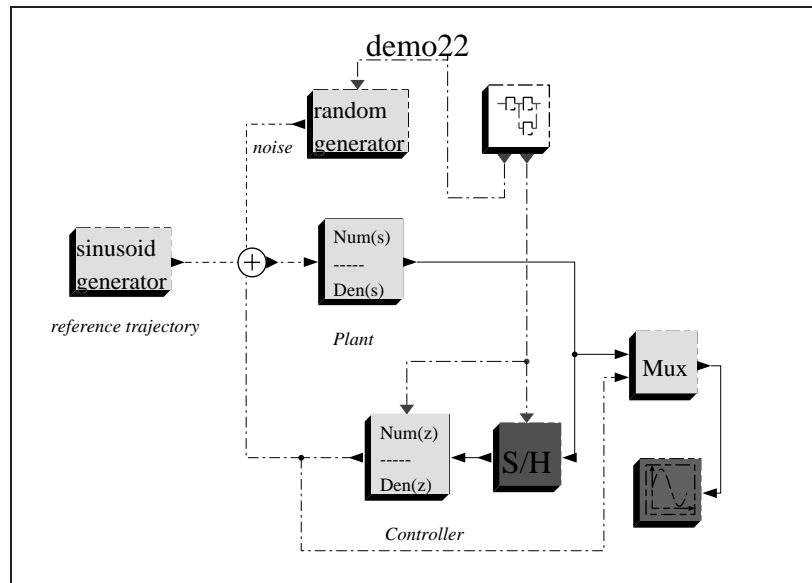


Figure 1: A typical Scicos diagram

1 Introduction

Scicos (Scilab Connected Object Simulator) is a Scilab package for modeling and simulation of dynamical systems including both continuous and discrete sub-systems. Scicos includes a graphical editor for constructing models by interconnecting blocks (representing predefined basic functions or user defined functions).

*Scicos is a Scilab toolbox. This version of Scicos is included in Scilab-2.4. For more information on Scilab see: <http://www-rocq.inria.fr/scilab/>

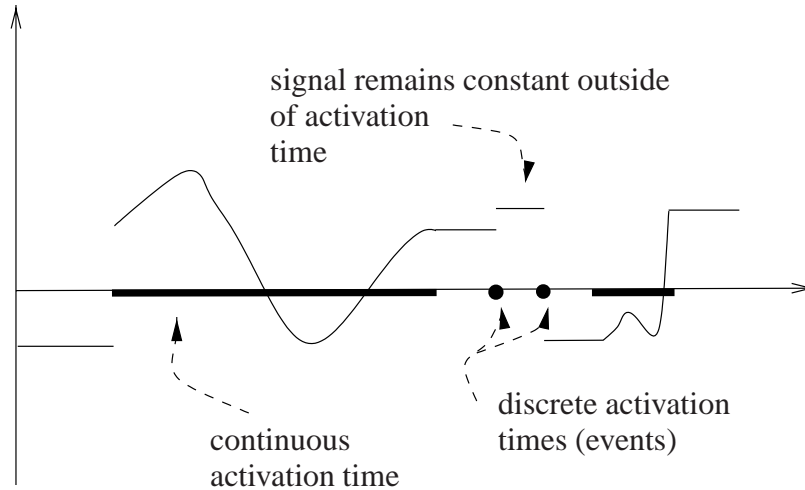


Figure 2: A signal in Scicos and its activation time set.

Associated with each signal, in Scicos, is a set of time indices, called activation times, on which the signal can evolve. Outside their activation times, Scicos signals remain constant (see Figure 2). The activation time set is a union of time intervals and isolated points called events.

Signals in Scicos are generated by blocks driven by activation signals. An activation signal causes the block to evaluate its output as a function of its input and internal state. The output signal, which inherits its activation time set from the generating block, can be used to drive other blocks.

Blocks are activated by activation signals which are received on activation input ports placed on top of blocks. A block with no input activation port is permanently active (called time dependent) otherwise it inherits its activation times from the union of activations times of its input signals.

Ports placed at the bottom of blocks are output activation ports. The outgoing signals are activation signals generated by the block. For example, the `Clock` block generates an activation signal composed of a train of regularly spaced events in time. If this output is connected to the input activation port of a scope block (such as the `MScope` block), it specifies at what times the value of the inputs of the scope must be displayed.

2 Scicos editor

In Scicos, systems are modeled by interconnecting blocks and subsystems (Super blocks); blocks can be found in various palettes or be defined by user. Scicos has an easy to use graphical user interface for editing diagrams. To start the editor, in Scilab, type `scicos()`. This opens up Scicos' main window.

Construction of Scicos model typically consists of

- opening one or more palettes (using `Palettes` button in the `Edit` menu),
- copying blocks from palettes into Scicos' main window; this can be done by selecting the `copy` button in the `Edit` menu, then clicking on the block to be copied, and finally in the Scicos' main window, where the block is to be placed.
- connecting the input and output ports of the blocks by selecting first the `link` button in the `Edit` menu, then clicking on the output port and then on the input port (or intermediary points before that).

Note that to make a link originate from another link (to split a link), user should first click on the `Link` button and then on an existing link, where split is to be placed, and finally on an input port (or intermediary points before that). The process of link creation can be stopped and current link deleted by clicking on the right mouse button.

Note also that at least one scope or a "write to file" block should be placed in any Scicos diagram to visualize or save the result of the simulation. See Scicos demos for examples.

2.1 Parameter adaptation

Block parameters can be modified by opening the block dialogs. This can be done using the `Open/set` button. Most blocks have dialog menus which can be used to set or modify block parameters. These parameters can be defined using valid Scilab expressions. Scilab variables can be used in the definition of these expressions if they are already defined in the context of diagram. These expressions are memorized symbolically, and then evaluated.

The context of the diagram can be edited by selecting the `Context` button. The context is evaluated by the `Eval` button. This is necessary only if the context modification includes a change in the value of a variable previously used in the definition of a block parameter.

2.2 Simulation

A completed diagram can be simulated using `Run` in the `Simulate` menu. Selecting this button results in a compilation of the diagram (if not already compiled) and simulation. The simulation can be stopped by clicking on the `stop` button on top of the Scicos main window.

A compiled Scicos diagram, saved as a `*.cos` file, does not need compilation the next time it is loaded; the saved file contains, the result of the compilation. It is also possible to extract just the data needed to do simulation, and do the simulation without having to enter the Scicos environment. This can be done using the `scicosim` function.

2.3 Other functionalities

The editor provides many other functionalities such as

- saving and loading diagrams in various formats
- zooming and changing the point of view
- changing block aspects and colors
- changing diagram's background and foreground colors
- placing text in the diagram
- printing and exporting Scicos diagrams
- and many other standard GUI functions.

The `Help` button can be used to obtain help on various aspects of Scicos. Selecting `Help` and then clicking on a block displays the manual page of the block. Selecting `Help` and then selecting another button, displays the manual page of the button.

Finally, an important feature in Scicos is the possibility of creating sub-systems (Super Blocks). Clearly, it would not be desirable to fit a complex system with hundreds of components in one Scicos diagram. For that, Scicos provides the possibility of grouping blocks together and defining sub-diagrams called Super Blocks. These blocks behave like any other block but can contain an unlimited number of blocks, and even other Super Blocks.

3 Basic Blocks

There are three types of Basic Blocks in Scicos: Regular Basic Blocks, Zero Crossing Basic Blocks and Synchro Basic Blocks. These blocks have can have two types of inputs and two types of outputs ports: regular inputs, activation inputs, regular outputs and activation outputs ports. Regular inputs and outputs are interconnected by regular links, and activation inputs and outputs, by activation links. Note that activation input ports are placed on top and activation output ports at the bottom of the blocks.

3.1 Regular Basic Block

Regular Basic Blocks (RBB) can have a *continuous state* x and a *discrete state* z . If it does have an x and if u denotes its regular input, then, when the block is active over an interval of time, x evolves continuously according to

$$\dot{x} = f(t, x, z, u, p, n_e) \quad (1)$$

where f is a vector function, p is a vector of constant parameters and n_e is the *activation code* which is an integer designating the port(s) through which the block is activated. In particular, if activating input ports are i_1, i_2, \dots, i_n , then

$$n_e = \sum_{j=1}^n 2^{i_j-1}.$$

On the other hand, activated by an event, the states x and z jump instantaneously according to the following equations:

$$x(t_e) = g_c(t_e, x(t_e^-), z(t_e^-), u(t_e), p, n_e) \quad (2)$$

$$z(t_e) = g_d(t_e, x(t_e^-), z(t_e^-), u(t_e), p, n_e) \quad (3)$$

where t_e denotes the event time. The discrete state z remains constant between any two successive events so $z(t_e^-)$ can be interpreted as the previous value of z .

During activation times, the regular output of the block is defined by

$$y(t) = h(t, x(t^-), z(t^-), u(t), p, n_e) \quad (4)$$

and is constant when the block is not active.

Finally, RBB's can generate activation signals of event type. If it is activated by an event at time t_e , the time of each output event is given by

$$t_{evo} = k(t_e, z(t_e), u(t_e), p, n_e) \quad (5)$$

where t_{evo} is a vector of time, each entry of which corresponds to one activation output port. The absence of event corresponds to a time smaller than the current time. Event generations can also be pre-scheduled. Pre-scheduling of events can be done by setting the "initial firing variables of blocks with event output ports.

3.2 Zero Crossing Basic Block

Zero Crossing Basic Block (ZBB) can generate event outputs only if at least one of the regular inputs crosses zero (changes sign). In such a case, the generation of the event, and its timing, can depend on the combination of the inputs which have crossed zero and the signs of the inputs just before the crossing occurs.

A few examples of ZBB's can be found in the `Threshold` palette.

3.3 Synchro Basic Block

Synchro Basic Blocks (SBB) generate output activation signals that are synchronized with their input activation signals. These blocks have a unique activation input port; they route their input activation signals to one of their activation outputs. The choice of this output depends on the value of the regular input. Examples are the `event select` block and the `If-then-else` block in the `Branching` palette.

4 Time dependence and inheritance

To avoid explicitly drawing all the activation signals in a Scicos diagram, a feature called inheritance is provided in Scicos. In particular, if a block has no activation input port, it inherits its activation signal from its regular input signals. And for blocks which are active permanently, they can be declared as such ("time dependent") and they do not need input activation ports. Note that time dependent blocks do not inherit.

5 Block construction

A new block can be constructed as a Super Block (by interconnection of basic blocks) and compiled. As for a new basic block, it can be defined by a pair of functions:

- an *Interfacing* function for handling the user-interface
- a *Computational* function for specifying its dynamic behavior.

The *Interfacing* function is always written as a Scilab function. See Scilab functions in `<SCIDIR>/macros/scicos_blocks` for examples. The *Computational* function can be written in C or Fortran. See `<SCIDIR>/routines/scicos` for examples. But it can also be written in Scilab language. C and Fortran routines dynamically linked or permanently interfaced with Scilab give the better results as far as simulation performance is concerned.

The `Scifunc`, `GENERIC`, `C_block` and `Fortran_block` blocks provide generic *Interfacing* functions, very useful for rapid prototyping and testing user-developed *Computational* functions.

5.1 Interfacing function

The *Interfacing* function determines the geometry, color, number of ports and their sizes, icon, etc..., in addition to the initial states, parameters. This function also handles the block's user dialog.

What the interfacing function should do and should return depends on an input flag `job`. The syntax is as follows:

5.1.1 Syntax

```
[x,y,typ]=block(job,arg1,arg2)
```

Parameters

- `job=='plot'`: the function draws the block.
 - `arg1` is the data structure of the block.
 - `arg2` is not used.
 - `x,y,typ` are not used.

In general, we can use `standard_draw` function which draws a rectangular block, and the input and output ports. It also handles the size, icon, and color aspects of the block.

- `job=='getinputs'`: the function returns position and type of input ports (regular or activation).
 - `arg1` is the data structure of the block.
 - `arg2` is not used.

- `x` is the vector of `x` coordinates of input ports.
- `y` is the vector of `y` coordinates of input ports.
- `typ` is the vector of input ports types (1 for regular and 2 for activation).

In general, we can use the `standard_input` function.

- `job=='getoutputs'`: returns position and type of output ports (regular and activation).

 - `arg1` is the data structure of the block.
 - `arg2` is not used.
 - `x` is the vector of `x` coordinates of output ports.
 - `y` is the vector of `y` coordinates of output ports.
 - `typ` is the vector of output ports types .

In general, we can use the `standard_output` function.

- `job=='getorigin'`: returns coordinates of the lower left point of the rectangle containing the block's silhouette.

 - `arg1` is the data structure of the block.
 - `arg2` is not used.
 - `x` is the `x` coordinate of the lower left point of the block.
 - `y` is the `y` coordinate of the lower left point of the block.
 - `typ` is not used.

In general, we can use the `standard_origin` function.

- `job=='set'`: opens up a dialogue for block parameter acquisition (if any).

 - `arg1` is the data structure of the block.
 - `arg2` is not used.
 - `x` is the new data structure of the block.
 - `y` is not used.
 - `typ` is not used.

- `job=='define'`: initialization of block's data structure (name of corresponding *Computational* function, type, number and sizes of inputs and outputs, etc...).

 - `arg1`, `arg2` are not used.
 - `x` is the data structure of the block.
 - `y` is not used.
 - `typ` is not used.

5.1.2 Block data-structure definition

Each Scicos block is defined by a Scilab data structure as follows:

```
list('Block', graphics, model, unused, GUI_function)
```

where `GUI_function` is a string containing the name of the corresponding *Interfacing* function and `graphics` is the structure containing the graphical data:

```
graphics = . .
```

```
list([xo, yo], [l, h], orient, dlg, pin, pout, pcin, pcout, gr_i)
```

- **xo**: x coordinate of block origin
- **yo**: y coordinate of block origin
- **l**: block's width
- **h**: block's height
- **orient**: boolean, specifies if block is flipped or not (regular inputs are on the left or right).
- **dlg**: vector of character strings, contains block's symbolic parameters.
- **pin**: vector, `pin(i)` is the number of the link connected to *i*th regular input port, or 0 if this port is not connected.
- **pout**: vector, `pout(i)` is the number of the link connected to *i*th regular output port, or 0 if this port is not connected.
- **pcin**: vector, `pcin(i)` is the number of the link connected to *i*th activation input port, or 0 if this port is not connected.
- **pcout**: vector, `pcout(i)` is the number of the link connected to *i*th activation output port, or 0 if this port is not connected.
- **gr_i**: character string vector, Scilab instructions used to draw the icon.

The data structure containing simulation information is `model`:

```
model = list(eqns, #input, #output, #clk_input, #clk_output, . .  
state, dstate, rpar, ipar, typ, firing, deps, label, unused)
```

- **eqns**: list containing two elements. First element is a string containing the name of the *Computational* function (fortran, C, or Scilab function). Second element is an integer specifying the type of the *Computational* function. The type of a *Computational* function specifies essentially its calling sequence; more on that later.

- **#input**: vector of size equal to the number of block's regular input ports. Each entry specifies the size of the corresponding input port. A negative integer stands for "to be determined by the compiler". Specifying the same negative integer on more than one input or output port tells the compiler that these ports have equal sizes.
- **#output**: vector of size equal to the number of block's regular output ports. Each entry specifies the size of the corresponding output port. Specifying the same negative integer on more than one input or output port tells the compiler that these ports have equal sizes.
- **#clk_input**: vector of size equal to the number of activation input ports. All entries must be equal to 1. Scicos does not support vectorized activation links.
- **#clk_output**: vector of size equal to the number of activation output ports. All entries must be equal to 1. Scicos does not support vectorized activation links.
- **state**: column vector of initial continuous state.
- **dstate**: column vector of initial discrete state.
- **rpar**: column vector of real parameters passed on to the corresponding *Computational* function.
- **ipar**: column vector of integer parameters passed on to the corresponding *Computational* function.
- **typ**: string. Basic block type: 'z' if ZBB, 'l' if SBB and anything else for except 's' for RBB.
- **firing**: column vector of initial firing times of size equal to the number of activation output ports of the block. It includes preprogrammed event firing times (<0 if no firing).
- **deps**: [udep timedep]
 - **udep**: boolean. True if system has direct feed-through, i.e., at least one of the outputs depends explicitly on one of the inputs.
 - **timedep**: boolean. True if block is time dependent.
- **label**: character string, used as block identifier. This field may be set by the `label` button in `Block` menu.

5.2 Computational function

The *Computational* function evaluates outputs, new states, continuous state derivative and the output events timing vector depending on the type of the block and the way it is called by the simulator.

5.2.1 Behavior

Simulator calls the *Computational* function for performing different tasks:

- **Initialization** The simulator calls the *Computational* function once at the start for state and output initialization (inputs are not available then). Other tasks such as file opening, graphical window initialization, etc..., can also be performed at this point.
- **Re-initialization** The simulator can call the block a number of times for re-initialization. This is another opportunity to initialize states and outputs. But this time, the inputs are available.
- **Outputs update** The simulator calls for the value of the outputs. Thus the *Computational* function should evaluate (4).
- **States update** One or more events have arrived and the simulator calls the *Computational* function to update the states x and z according to (2) and (3).
- **State derivative computation** The simulator is in a continuous phase; the solver requires \dot{x} . This means that the *Computational* function must evaluate (1).
- **Output events timing** The simulator calls the *Computational* function about the timing of its output events. The *Computational* function should evaluate (5).
- **Ending** The simulator calls the *Computational* function once at the end (useful for closing files, free allocated memory, etc...).

The simulator uses a flag to specify which task should be performed (see Table 1).

Flag	Task
0	State derivative computation
1	Outputs update
2	States update
3	Output events timing
4	Initialization
5	Ending
6	Re-initialization

Table 1: Tasks of *Computational* function and their corresponding flags

5.2.2 Types of *Computational* functions

In Scicos, *Computational* functions can be of different types and co-exist in the same diagram. Currently defined types are listed in Table 2. The type of the *Computational* function is stored in the second field of `eqns` (see Section 5.1.2).

Function type	Scilab	Fortran	C	Comments
0	yes	yes	yes	Fixed calling sequence
1	no	yes	yes	Varying calling sequence
2	no	no	yes	Fixed calling sequence
3	yes	no	no	Inputs/outputs are Scilab lists

Table 2: Different types of the *Computational* functions. Type 0 is obsolete.

Computational function: type 0 In blocks of type 0, the simulator constructs a unique input vector by stacking up all the input vectors, and expects the outputs, stacked up in a unique vector as well. This type is supported for backward only.

The calling sequence is identical to that of *Computational* functions of type 1 with one regular input and one regular output.

Computational function: type 1 The simplest way of illustrating this type is by considering an example: for a block with two regular input vectors and four regular output vectors, the *Computational* function has the following synopsis.

Fortran case

```

subroutine myfun(flag,nevpert,t,xd,x,nx,z,nz,tvec,
& ntvec,rpar,nrpar,ipar,nipar,u1,nul,u2,nu2,
& y1,ny1,y2,ny2,y3,ny3,y4,ny4)
c
double precision t,xd(*),x(*),z(*),tvec(*),rpar(*)
double precision u1(*),u2(*),y1(*),y2(*),y3(*),y4(*)
integer flag,nevpert,nx,nz,ntvec,nrpar,ipar(*)
integer nipar,nul,nu2,ny1,ny2,ny3,ny4

```

See Tables 3 for a description of the arguments.

C case Type 1 *Computational* functions can also be written in C language, the same way. Note that, arguments must be passed as pointers.

The best way to learn how to write these functions is to examine the routines in the Scilab directory SCIDIR/routines/scicos where *Computational* functions of all Scicos blocks are available. Most of them are fortran type 0 and 1.

Computational function type 2 This *Computational* function type is specific to programming in C. The synopsis is:

```

#include "<SCIDIR>/routines/machine.h"
void selector(flag,nevpert,t,xd,x,nx,z,nz,tvec,ntvec,
rpar,nrpar,ipar,nipar,inptr,insz,nin,outptr,outsz,nout)

integer *flag,*nevpert,*nx,*nz,*ntvec,*nrpar;

```

I/O	Args.	Description
I	flag	0,1,2,3,4,5 or 6, (see Table 1)
I	nevprt	activation code
I	t	time
O	xdot	derivative of the continuous state
I/O	x	continuous state
I	nx	size of x
I/O	z	discrete state
I	nz	size of z
O	tvec	times of output events (for flag=3)
I	ntvec	number of activation output ports
I	rpar	parameter
I	nrpar	size of rpar
I	ipar	parameter
I	nipar	size of ipar
I	ui	i th input (regular), i=1,2,...
I	nui	size of i th input
O	yj	j th output (regular), j=1,2,...
I	nyj	size of j th output

Table 3: Arguments of *Computational* functions of type 1. I: input, O: output.

```
integer ipar[],*nipar,insz[],*nin,outsz[],*nout;

double x[],xd[],z[],tvec[],rpar[];
double *inptr[],*outptr[],*t;
```

See Table 4 for a description of arguments.

Computational function type 3 This *Computational* function type is specific to programming in Scilab. The calling sequence is as follow:

```
[y,x,z,tvec,xd]=test(flag,nevprt,t,x,z,rpar,ipar,u)
```

See table 5 for a description of arguments.

Example The following is the *Computational* function associated with a block that displays in a Scilab window, every time it receives an event, the number of events it has received up to the current time, and the values of its two inputs.

```
function [y,x,z,tvec,xd]=test(flag,nevprt,t,x,z,rpar,ipar,u)
y=list();tvec=[];xd=[]
if flag==4 then
    z=0
elseif flag==2 then
```

I/O	Args.	description
I	*flag	0,1,2,3,4,5 or 6, (see Table 1)
I	*nevprt	activation code
I	*t	time
O	xd	derivative of the continuous state (flag= 0)
I/O	x	continuous state
I	*nx	size of x
I/O	z	discrete state
I	*nz	size of z
O	tvec	times of output events (flag=3)
I	*ntvec	number of activation output ports
I	rpar	parameter
I	*nrpar	size of rpar
I	ipar	parameter
I	*nipar	size of ipar
I	inptr	inptr[i] is pointer to beginning of ith input
I	insz	insz[i] is the size of the ith input
I	*nin	number of input ports
I	outptr	outptr[j] is pointer to beginning of jth output
I	outsz	outsz[j] is the size of the jth output
I	*nout	number of output ports

Table 4: Arguments of *Computational* functions of type 2. I: input, O: output.

```

z=z+1
write(%io(2),'Number of calls:'+string(z))
[u1,u2]=u(1:2)
write(%io(2),'first input');disp(u1)
write(%io(2),'second input');disp(u2)
end

```

Example The advantage of coding inputs and outputs as lists is that the number of inputs and outputs need not be specified explicitly. In this example, the output is the element-wise product of all the input vectors, regardless of the number of inputs.

```

function [y,x,z,tvec,xd]=elemprod(flag,nevprt,t,x,z,rpar,ipar,u)
tvec=[];xd=[]
y=u(1)
for i=2:length(u)
    y=y.*u(i)
end
y=list(y)

```

I/O	Args.	description
I	flag	0,1,2,3,4,5 or 6 (see Table 1)
I	nevprt	activation code (scalar)
I	t	time (scalar)
I	x	continuous state (vector)
I	z	discrete state (vector)
I	rpar	parameter (any type of scilabtt variable)
I	ipar	parameter (vector)
I	u	$u(i)$ is the vector of i th regular input (list)
O	y	$y(j)$ is the vector of j th regular output (list)
O	x	new x if flag=2, 4, 5 or 6
O	z	new z if flag=2, 4, 5 or 6
O	xd	derivative of x if flag= 0 (vector), [] otherwise
O	tvec	times of output events if flag=3 (vector), [] otherwise

Table 5: Arguments of *Computational* functions of type 3. I: input, O: output.

6 Conclusion

This document gives only a brief description of Scicos and its usage. More information can be found in the manual pages of Scicos functions (Scilab help under Scicos library). Scicos demos provided with Scilab constitute also an interesting source of information. Often, it is advantageous to start off from and edit a Scicos demo rather than starting with an empty diagram.

Contents

1	Introduction	1
2	Scicos editor	2
2.1	Parameter adaptation	3
2.2	Simulation	3
2.3	Other functionalities	4
3	Basic Blocks	4
3.1	Regular Basic Block	4
3.2	Zero Crossing Basic Block	5
3.3	Synchro Basic Block	5
4	Time dependence and inheritance	5
5	Block construction	6
5.1	<i>Interfacing</i> function	6
5.1.1	Syntax	6
5.1.2	Block data-structure definition	8
5.2	<i>Computational</i> function	9
5.2.1	Behavior	10
5.2.2	Types of <i>Computational</i> functions	10
6	Conclusion	14