SCICOS
a dynamic system builder and simulator

User's Guide - Version 0.1*

R. Nikoukhah S. Steer

Abstract

Scicos (Scilab Connected Object Simulator) is a Scilab package for modeling and simulation of hybrid dynamical systems. Scicos includes a graphical editor which can be used to build complex models by interconnecting blocks which represent either predefined basic functions defined in Scicos libraries (palettes), or user defined functions. A large class of hybrid systems can be modeled and simulated in Scicos.

An example of Scicos diagram

*Scicos Version 0.1 is a built-in library (toolbox) of Scilab Version 2.2. For information on Scicos and Scilab in general, send an email to `scilab@inria.fr`

Contents

1 Introduction

Even though it is possible to simulate mixed discrete continuous (hybrid) dynamics systems in Scilab using Scilab' s ordinary differential equation solver (the `ode` function), implementing the discrete recursions and the logic for interfacing the discrete and the continuous parts usually requires a great deal of programming. These programs are often complex, difficult to debug and slow. Scicos is a Scilab package that provides an easy to use graphical editor for building complex models of hybrid systems, a compiler which converts the graphical description into Scicos executable code, and a simulator for simulating Scicos executable codes.

The user-friendly GUI of Scicos (presented in Section 4), and specially the demos provided with the package, should allow new users to start building and simulating very quickly simple models. It is however highly recommended that new users start familiarizing themselves with basic concepts and elementary building blocks of Scicos by reading Section 3 first.

Scicos Version 0.1 is a pre-beta test version. It has not been fully tested and it has a number of limitations. Bug reports and suggestions should be sent to `scilab@inria.fr`.

2 Hybrid Model

There have been a number of models proposed in the literature for hybrid systems. A simple, yet powerful model is the following:

$$\dot{x} = f(x) \tag{2.1}$$

$$\text{if } h_i(x) = 0, \quad \text{then } x := g(i,x),\ i = 1,2,...,m, \tag{2.2}$$

where $x \in I\!R^n$ is the state of the system, f is a vector field on $I\!R^n$, g is a mapping from $I\!N \times I\!R^n \to I\!R^n$ and h_i's are continuous functions. (2.2) should be interpreted as: when $h_i(x)$ crosses zero, the state jumps from x to $g(i,x)$; and of course, between two state jumps, (2.1) describes the evolution of the system. The zero crossing of $h_i(x)$ is referred to as an event i. An event then causes a jump in the state of the system.

This model may seem very simple, yet it can model many interesting phenomena. Consider for example the dependence on time. It may seem that (2.1)-(2.2) cannot model time-dependent systems. This, however, can be done by state augmentation. For that it suffices to add

$$\dot{t} = 1 \tag{2.3}$$

to system equations and augment the state by adding t to it.

To model discrete-time systems, i.e., systems that evolve according to

$$\xi(k+1) = f(k, \xi(k)), \tag{2.4}$$

first, an event generator is needed. A simple event generator would be

$$\dot{e} = -1 \tag{2.5}$$

with $e := 1$ if $e = 0$. So e starts off as 1 and with constant slope of -1, it reaches zero one unit of time later. At this point, an event is generated and e goes back to 1 and the process starts over.

These events can then be used to update ξ. The complete system would then be:

$$\dot{k} = 0 \tag{2.6}$$
$$\dot{\xi} = 0 \tag{2.7}$$
$$\dot{e} = -1 \tag{2.8}$$
$$\text{if } e = 0, \quad \text{then } e := 1, \xi := f(k, \xi), k := k + 1. \tag{2.9}$$

State augmentation is a nice way of modeling hybrid systems in the form (2.1)-(2.2), however, it is in most cases not a good idea for the purpose of simulation. It is clearly too costly to integrate (2.5) for realizing an event generator, or integrate 1 to obtain t. Even for model construction, (2.1)-(2.2) does not provide a very useful formalism. To describe hybrid models in a reasonably simple way, a richer set of operators need to be used (even if they can all, at least in theory, be realized by state augmentation as (2.1)-(2.2)).

Scicos is a modeler and simulator which proposes a fairly rich set of operators for modeling hybrid systems from a modular description. The modular aspect (possibility of constructing the model by interconnection of other models) introduces additional complexity concerning, mainly, event scheduling and causality. In the current version of Scicos, basic operators are not as powerful as they could (and should) be, but they suffice to model many interesting problems in systems, control and signal processing applications.

3 Basic concepts

Models in Scicos are constructed by interconnection of Basic Blocks. There exist two types of Basic Blocks and two types of connecting paths (links) in Scicos. Basic Blocks can have two types of inputs and two types of output: regular inputs, event inputs, regular outputs and event outputs. Regular inputs and outputs are interconnected by regular paths, and event inputs and outputs, by event paths (regular input and output ports are placed on the sides of the blocks, event input ports are on top and event output ports are on the bottom; blocks can have an unlimited number of each type of input and output ports).

Regular paths carry piece-wise right-continuous functions of time whereas event paths transmit timing information concerning discrete events. One way to think about event signals as physical signal is to consider them as impulses, so in a sense event paths transmit impulses from output event ports to input even ports. To see how event signals (impulses) are generated and how they affect the blocks, a look at the behavior of Basic Blocks is necessary.

3.1 Basic Blocks

There are two types of Basic Blocks in Scicos.

3.1.1 Regular Basic Block

Regular Basic Blocks can have both regular and event input and output ports. Regular Basic Blocks can model continuous and discrete dynamics systems. They can have a continuous state x

and a "discrete" state z. Let u denote the regular inputs and y the regular outputs. Then a regular Basic Block imposes the following constraints

$$\dot{x} = f(t, x, z, u, p) \tag{3.1}$$
$$y = h(t, x, z, u, p) \tag{3.2}$$

where f and h are block specific functions, and p is a vector of constant parameters. Constraints (3.1)-(3.2) are imposed by the Basic Block as long as no impulses arrive on its event inputs. An event input can cause a jump in the states of the Basic Blocks. Let's say an impulse arrives on a Basic Block's event port at time t, then the states jump according to the following equations:

$$x^+ = g_c(t, x^-, z^-, u^-, p, n_{evi}) \tag{3.3}$$
$$z^+ = g_d(t, x^-, z^-, u^-, p, n_{evi}) \tag{3.4}$$

where g_c and g_d are block specific functions; n_{evi} is the port by which the event has arrived; u^-, x^- and z^- are the values of the regular inputs, the continuous state and the discrete state at the arrival of the event signal. x^+ and z^+ are the values of the states right after the arrival of the event. Needless to say that z remains constant between any two successive events.

Finally, regular Basic Blocks can generate event signals on event output ports. These events can only be scheduled at the arrival of an input event. The time of each output event impulse is generated according to

$$t_{evo} = k(t, x^-, z^-, u^-, p, n_{evi}) \tag{3.5}$$

for a block specific function k and where t_{evo} is a vector of time, each entry of which corresponds to one event output port. Normally all the elements of t_{evo} are larger than or equal to t. If an element is less than t, it simply means the absence of an output event signal on the corresponding event output port.

Event generations can also be pre-scheduled. In most cases, if no event is pre-scheduled, nothing would ever happen (at least as far as events are concerned). Pre-scheduling of events can be done by setting the "initial firing" variable of each block. Initial firing is a vector with as many elements as the block's output event ports. Initial firing can be considered as the initial condition for t_{evo}. By setting the i-th entry of the initial firing vector to t_i, an output event is scheduled on the i-th output event port of the block at time t_i if $t_i \geq 0$; no output event is scheduled if $t_i < 0$. This event is then fired when time reaches t_i.

Initially, only one output event can be scheduled on each output event port. In the course of the simulation also, usually, there should not be more than one event scheduled at any output event port (this means that by the time a new event is to be scheduled, the old one must have been fired). This is natural because the register that contains firing schedule of a block should be considered as part of the state, having dimension equal to the number of output event ports.

In "Simulation Mode 1", if the simulator encounters such a conflict, it stops and returns the error message *event conflict*. On the other hand, in "Simulation Mode 2", more than one event can be scheduled at any time. There exists however an upper limit on the total number of events that can be scheduled at any time in the whole system. The error message *event conflict*, in Simulation Mode 2, means that the system is trying to schedule more events than the total number allowed. This usually means the model is not correct and that the simulator is not doing what

the user thinks it is doing. In that case, the timing and synchronization of the system should be re-examined (sometimes this can be very tricky). The default simulation mode is 1 because in this mode, the simulator runs faster.

Most of the elementary blocks in Scicos are regular Basic Blocks: all the blocks in all the palettes except the Threshold palette are regular. The followings are a few examples.

Static blocks A static block is one where the (regular) output is a static function of the input. For example the block that "realizes" $y = \sin(u)$ is a static block. Static blocks have no input or output event ports, and they have no state. Clearly these blocks are special cases of regular Basic Blocks. The `Non linear` palette contains a number of examples of such blocks.

Discrete-time state-space systems A discrete-time system

$$\xi(k+1) = m(k, \xi(k), u(k)) \tag{3.6}$$
$$y(k) = n(k, \xi(k)) \tag{3.7}$$

can be implemented as a regular Basic Block if the block receives, on its event input port, event signals on a regular basis. In this case z is used to store ξ, x is empty, and there is no event output.

Clocks A clock is a generator of event signals on a periodic basis. A regular Basic Block (or any other type of Basic Block) cannot act as a clock. The reason is that, except for a possible pre-scheduled initial output event, a general Basic Block must receive an event signal on one of its event input ports to be able to generate an output event. The way to generate a clock in Scicos is by using an "event delay block". An event delay block is a general Basic Block which has no state, no regular input or output. It has one event input port and one event output port. When an event arrives, after a period of time, it generates an event on its event output port. By feeding back the output to the input (connecting the event output port to the event input port, see Figure 3.1.1), a clock can be constructed. For that, an output event should be pre-scheduled on the event output port of the event delay block. This is done by setting the block's initial firing vector to 0 (or any $t \geq 0$ if the clock is to start operating at time t).

Constructing an event clock using feedback on a delay block

This way of defining clocks may seem complicated, however it provides a lot of flexibility. For example systems with multiple asynchronous clocks driving various system components are very easy to model using such blocks.

3.1.2 Zero crossing Basic Block

Zero crossing Basic Blocks have regular inputs and event outputs but no regular outputs, or event inputs. A zero crossing block 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). The simplest example of a surface crossing Basic Block is the `ZCROSS` block in `Treshold` palette. This block generates an event if all the inputs cross simultaneously 0. Other examples are `+ to -` and `- to +` which generate an output event when the input crosses zero, respectively, with a negative and a positive slope. The most general form of this block is realized by the block `GENERAL` in the `Threshold` palette.

Zero crossing Basic Blocks cannot be modeled as regular Basic Blocks because in a regular Basic Block, no output event can be generated unless an input event has arrived beforehand.

Treshold palette

3.2 Paths (Links)

There are two different types of paths in Scicos. The function of regular paths is pretty clear but that of event paths is more subtle. An event signal is a timing information which specifies the time when the blocks connected to the output event port generating the event signal are updated according to (3.3)-(3.4) and (3.5). This timing information (event impulse) is transmitted by event paths. These paths specify which event output ports are connected to which event input ports, and thus specify which blocks should be updated when an output event is fired. If event paths are considered as links that transport event impulses from output event ports to input event ports, a split on an event path becomes an impulse doubler: when the split receives an impulse, it generates one on each of its two outputs. This of course is just a way of presenting the split; it is not the way the simulation is done.

Besides split, there exists another block which operates on event paths: the event addition. Just like the event split, event addition is not really a Scicos block because it is discarded during compilation. Adding two timing information corresponds simply to merging the two information. In terms of event impulses, this operation corresponds to superposition of the two signals. If two input event impulses on different inputs of the addition block have the same time, the output would still consist of two event impulses having the same time as that of the input event signals.

Event links: Split and addition

Even if two event signals have the same time, they are not necessarily synchronized, meaning one is fired just before or just after the other but not "at the same time". For example a regular block may generate output events with zero delay, but the output will always come after the input event (causality).[1] The only way two event signals can be synchronized is that they can be traced back, through event paths and event additions alone, to a common origin (a single output event port).

This means, in particular, that a block can never have two "synchronized" output event ports (Super Block's however can have synchronized output event ports; see for example the `2-freq clock` in the `Event` palette, this block is illustrated in Figure 3.3).

3.3 Super blocks

Not all blocks in Scicos' palettes are Basic Blocks; some are Superblocks. A Super block is obtained by interconnecting Basic Blocks and other Super blocks. The simplest example of such a block is the `CLOCK` which is obtained from one regular Basic Block and two event paths and one output event port. As far as the user is concerned, in most case, there is no real distinction between Basic Blocks and Super blocks.

Super blocks can also be defined by the user. For that, the `Super block` in the `Others` palette should be copied into the Scicos window and opened. This opens up a new Scicos window in which a block diagram can be edited. Super blocks can be used within Super blocks without any limit on the

[1] This restriction will be lifted in future versions.

number or the depth.

Super block defining a 2 frequency clock.

At compilation, all the Super blocks in the Scicos model are expanded and a diagram including only Basic Blocks is generated. This phase is completely transparent to the user.

4 Using the graphical user interface

This section describes various functions of the graphical user interface of Scicos. To invoke Scicos, user should type `scicos()` in a regular Scilab session. This opens up a Scicos window inside a Scilab graphics window. By default, this window is entitled `Untitled` and contains an empty model. An existing model can be loaded at this point using the `load` button in `File..` menu, or a new model can be constructed in the `Untitled` window. The model can be saved on file in various formats. All the operations in Scicos are done clicking on various buttons in the Scicos windows. A detailed description is given in the Appendix. Description can also be obtained on each button by clicking first on the `Help` button, then on the button in question. For help on blocks, user can click first on the `Help`, then on the block in question.

The most common way of constructing Scicos models is by using existing blocks in Scicos' palettes. Click on the `Palettes` button in the `edit..` menu to open the palettes. In various palettes, user can find elementary blocks such as addition, gain, multiplication, ..., linear system blocks (continuous, discrete both in transfer function and state-space forms), nonlinear blocks, input output devices (reading from and writing to file, scope, ...), event generating blocks (clock, event delay, ...) and more. Any number of palettes can be open and active at any time.

4.1 Blocks in palettes

Blocks in palettes can be copied into Scicos window by clicking on the `Copy` button in the `Edit..` menu, then on the block to be copied and finally where the block is to be placed in Scicos window. By clicking on a block in Scicos window, a dialog opens up and the user can set block parameters. Help on a block can be obtained by clicking on the `Help` button, then on the block (in the palette

or in the Scicos window).

Event input ports are always placed on top, and event output ports on the bottom. regular input and output ports however can be placed on either side. User can use the `Flip` button to change the places of input and output ports. The regular input and output ports are numbered from top to bottom, and the event input and output ports, from left to right (whether the block is flipped or not).

4.2 Connecting blocks

Connecting blocks can be accomplished by clicking on the `Link` button in the `Edit..` menu, then on the output port and then the input port. This makes a straight line connection. For more complex connections, before clicking on the input port, user can click on intermediary points on Scicos window, to guide the path. Whenever possible, Scicos draws perfectly horizontal or vertical paths. Obviously only output ports and input ports of the same type can be connected. The color of the path can be changed by clicking on the path.

For some blocks, the number of inputs and outputs may depend on block parameters (for example linear systems and scope). In this case, user should adjust block parameters before connecting its ports.

A path can originate from a path, i.e., a path can be split, by clicking on the `Link` button and then on an existing path, where split is to be placed, and finally on an input port (or intermediary points before that).

4.3 How to correct mistakes

If a block is not correctly placed it can be moved. This can be done by clicking on the `Move` button (in the `Edit..` menu) first, then by clicking on the block to be moved, dragging the block to the desired location where a last click fixes the position (pointer position corresponds to the lower left corner of the block box).

If a block or a path is not needed, it can be removed using the `Delete` Button. This can be done by clicking first on `Delete` and then the object to be removed. If a block is removed, all paths connected to this block are also removed.

If an incorrect editing operation is performed, it can be taken back. See the help on the `Undo` button.

4.4 Save model and simulate

Once the Scicos model is constructed, it can be saved, compiled and simulated. A finished model should not contain any unconnected ports. If some ports are left unconnected and are not needed, they should be connected to the `Trash` block. To simulate the model, user should click on the `Run` button in the `Simulation..` menu. Simulation parameters can be set using the `setup` button.

System parameters can be modified in the course of the simulation. Clicking on `Stop` button on top of the main Scicos window halts the simulation and activates the Scicos panel. The system can then be modified and the simulation continued or restarted by clicking on the `Run` button. The

compilation is done automatically if needed. If after a modification the simulation does not work properly, a manual compilation (`Compile` button) may be necessary. Such situations should be reported.

5 Block construction

In addition to the blocks available in Scicos' palettes, user can use custom blocks. There are various ways of constructing blocks: Super blocks, `Sci-bloc` and `Scifunc` blocks which allow block functionality to be defined by Scilab expressions, and, C or Fortran functions, dynamically linked or permanently interfaced with Scilab. The latter method gives of course the best results as far as simulation performance is concerned.

5.1 Super blocks

To construct a Super block, user should copy the `Super block` block from the `Others` palette into the Scicos window and click on it. This will open up a new Scicos window in which the Super block should be defined. The construction of the Super block model is done as usual: by copying and connecting blocks from various palettes. Input and output ports of the Super block should be specified by input and output block ports available in the `Inputs/outputs` palette. Super blocks can be used within Super blocks.

A Super block can be saved in various formats. If the Super block can be of interest in other constructions, it can be converted into a block and placed in a user palette. See the help on the `Newblk` button.

If the Super block is only used in the particular construction, then it need not be saved. A click on the `Exit` will close the Super block window and activate the main Scicos window (or another Super block window if the Super block was being defined within another Super block). Saving the block diagram saves automatically the content of all Super blocks used inside of it.

A Super block, like any other Block, can be duplicated using the `Copy` button.

5.2 Sci-block

Often, specially during in the early phases of the development, it is very useful to have a way of defining Scicos blocks in the Scilab language. The block `Sci-block` has no specific behavior; it simply acts as an interface to Scilab functions which specify the nature of the block(type, number of inputs and outputs, discrete and continuous behaviors, etc...). Any number of `Sci-blocks`, interfacing one or more Scilab functions can be used in the same Scicos model.

To use a Sci-block, user should place a copy of the `Sci-block` in the Scicos model, click on it and enter the name of the corresponding Scilab function. The Scilab function must already be present in the Scilab environment.

Scilab functions which can be interfaced in Scilab through a `Sci-block` must have a special calling sequence. Developing such functions requires some understanding of the way block properties are coded in Scicos environment and a basic understanding of the way the graphical editor and the compiler access and use these properties. Section 6 presents some useful information.

The Scilab function call should resemble the following

```
function [out1,out2]=myfun(t,x,z,u,nclock,flag,rpar,ipar)
```

Variables x and z represent the continuous and discrete states, u is the input. nclock is zero unless the function is being called because an input event has arrived and the new states or t_{evo} are required. rpar and ipar are parameter vectors. flag specifies what the function should compute and return, and t is the current time, except when flag< 0 in which case it is a list containing the model of the block.

The outputs out1 and out2 return different things depending on the value of flag and nclock.

- If flag=1, out1 is y and out2=[ ].
- If flag=2 and nclock=0, out1 is $\dot{x}$ and out2=[ ].
- If flag=2 and nclock>1, out1 is the new value of x and out2, the new value of z, to be updated because an event has arrived on the input event port nclock.
- If flag=3, out1 is t_{evo}.

When flag is negative, out1 is the new model and out2, block's label (a string which contains the name of the block, to be placed underneath the block in Scicos window). Negative flag means the graphical editor is calling the function, either for initializing the states and parameters or updating them. This information is stored in `model`. It is very important that the user be familiar with the way this information is coded in `model` (see 6.1).

The following is a simple example of a Sci-block function. The corresponding block is a sine wave generator.

```
function [out1,out2]=m_sin(t,x,z,u,nclock,flag,rpar,ipar)
out1=[];out2=[];
select flag
case 1 then //regular output
  out1=sin(rpar(1)*t+rpar(2))
case 2 then //state evolution
case 3 then //event outputs
case -1 then //initialization
  model=t
  label='Sin'
  state=[]
  dstate=[]
  rpar=[1;0]
  model=list(model(1),0,1,0,0,state,dstate,rpar,[],'d',-1,[%f %t])
  out1=list(model,label)
case -2 then //parameter update
  model=t
  label=x
```

```
  rpar=model(8);gain=rpar(1);phase=rpar(2)
  [ok,label1,gain,phase]=getvalue('Set Sin block parameters',..
      ['Block label';'Frequency';'Phase'],list('str',1,'vec',1,'vec',1),..
      [label;string(gain);string(phase)])
  if ok then
    model(8)=[gain;phase]
    label=label1
  end
  out1=list(model,label)
end
```

5.3 Scifunc

The `Scifunc` block is similar to `Sci-block` in that it allows using Scilab language for defining Scicos blocks. It is however a lot easier to use; in particular, it does not require any deep knowledge of the way the graphical editor and the compiler work. The block information, limited to the number of inputs and outputs, initial states, type of the block, the initial firing vector and Scilab expressions defining functions f, h, g_c, g_d and k, have to be entered by the user. This is done interactively by clicking on the `Scifunc` block, once copied in the Scicos window. The main disadvantage of `Scifunc` is that the dialogue for updating block parameters cannot be customized.

Note that the graphical editor is not well protected against syntax errors user may introduce in the definition of the `Scifunc`. So, it is highly recommended that the block diagram be saved before opening the `Scifunc`.

Diagram with a Scifunc block

5.4 Fortran and C programs

For best performance, Scicos blocks should be defined by Fortran or C programs. In that case, a Scilab function is used to interface the program which contains information on the dynamics behavior of the block. The Scilab function contains information on the geometry of the block and handles the dialog for defining and updating block parameters and states. The Scilab function is used by the Graphical editor, and the Fortran or C program, by the simulator.

Section 6 describes how the Scilab function and the underlying programs should be developed.

6 Inside Scicos

Each Scicos block is defined by two functions. The first one which is in Scilab language, handles graphical operations. It is this function that specifies what the geometry of the block should be, how many inputs and outputs it should have, what the block type is, etc.... It is also this block that handles user interface (updating block parameters and initializing the states). This function is referred to as the *interfacing* function.

The second function, which called the *computational* function, should normally be written in Fortran or C, but can also be a Scilab function (using, in particular, `scifunc` and `Sci-block`).

6.1 Interfacing function

The interfacing function is only used by the Scicos editor to initialize, draw, connect the block and to modify its parameters. What the interfacing function should return depends on an input flag `job`.

Syntax

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

Parameters

- job=='plot' : the function draws the block and its label.
  - `arg1` is the data structure of the block.
  - `arg2` is unused.
  - `x,y,typ` are unused.

  In general, can use `standard_draw` function which draws a rectangle with input and output ports.
- job=='getinputs': the function returns position and type of input ports (regular and event).
  - `arg1` is the data structure of the block.
  - `arg2` is unused.

  - `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.

  In general can use the `standard_input` function.

- job=='getoutputs' : returns position and type of output ports (regular and event).
  - `arg1` is the data structure of the block.
  - `arg2` is unused.
  - `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 can use the `standard_output` function.

- job=='getorigin' : returns coordinates of the lower left point of the rectangle containing the block's shape.
  - `arg1` is the data structure of the block.
  - `arg2` is unused.
  - `x` is the vector of x coordinates of output ports.
  - `y` is the vector of y coordinates of output ports.
  - `typ` is unused.

  In general can use the `standard_origin` function.

- job=='set' : function opens a dialog for block parameter acquisition.
  - `arg1` is the data structure of the block.
  - `arg2` is unused.
  - `x` is the new data structure of the block.
  - `y` is unused.
  - `typ` is unused.

- job=='define': initialization of block's data structure. Set the initial type, number of inputs, outputs,... of the block.
  - `arg1, arg2` are unused.
  - `x` is the data structure of the block.
  - `y` is unused.
  - `typ` is unused.

Example ABS block interfacing function.

```
function [x,y,typ]=ABSBLK_f(job,arg1,arg2)
x=[];y=[];typ=[];
select job
case 'plot' then
  standard_draw(arg1)
  graphics=arg1(2);[orig,sz]=graphics(1;2);
  xstringb(orig(1),orig(2),'Abs',sz(1),sz(2),'fill')
case 'getinputs' then
  [x,y,typ]=standard_inputs(arg1)
case 'getoutputs' then
  [x,y,typ]=standard_outputs(arg1)
case 'getorigin' then
  [x,y]=standard_origin(arg1)
case 'set' then
  x=arg1;
  graphics=arg1(2);label=graphics(4)
  model=arg1(3);  nin=model(2)
  [ok,label,nin1]=getvalue('Set Abs block parameters',..
                           ['Block label';
                            'Number of inputs (outputs)'],..
                            list('str',1,'vec',1),..
                            [label;sci2exp(nin)])
  if ok then
    if nin1 > 0 then
      nin=nin1
      // Check and set port numbers
      [model,graphics,ok]=check_io(model,graphics,nin,nin,0,0)
      // modify data structure
      graphics(4)=label
      model(2)=nin;model(3)=nin;
      x(2)=graphics;x(3)=model
    else
      x_message('Number of inputs must be positive')
    end
  end
case 'define' then
  model=list('absblk',1,1,0,0,[],[],[],[],'c',%f,[%t %f])
  x=standard_define([2 2],model)
end
```

The only hard part of defining an interfacing function is the ‘‘`set`’’ case.

Block data-structure definition Each Scicos block is defined by a Scilab data structure (list) as follows:

```
list('Block',graphics,model,init,'standard_block')
```

where `graphics` is the data structure of graphics data:

```
graphics=list([xo,yo],[l,h],orient,label,pin,pout,pcin,pcout)
```

- xo: x coordinate of block origin
- yo: y coordinate of block origin
- l: block width
- h: block height
- orient: boolean, specifies if block is flipped
- label: character string,the block label
- 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 event input port, or 0 if this port is not connected.
- pcout : vector, `pcout(i)` is the number of the link connected to `i`th event output port, or 0 if this port is not connected.

and `model` is the data structure relative to simulation data

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

- eqns : simulation function name if it is defined by a fortran or C routine, or Scilab function code
- #input : number of regular inputs
- #output : number of regular ouputs
- #clk_input : number of event inputs
- #clk_output : number of event outputs
- state : vector (column) of initial state
- dstate : vector (column) of initial discrete state
- rpar : vector (column) of real parameters passed to associated Computational function.

- ipar : vector (column) of integer parameters passed to associated Computational function.
- typ : string: 'z' if zero-crossing, else regular block
- firing : vector of initial firing times of size -number of clock outputs- which includes preprogrammed event firing times (<0 if no firing) or (for backward compatibility) boolean vector: i-th entry true if i-th output fired at zero.
- deps : [timedep udep]
  - timedep boolean, true if system has time varying output
  - udep boolean, true if system has direct feedthrough

6.2 Computational function

The Computational function is called in various ways by the simulator. In Fortran syntax, this function should look something like

```
myfun(t,x,nx,z,nz,u,nu,rpar,nrpar,ipar,nipar,nclock,out,nout,flag)
```

where the inputs are:

- t (double precision): time
- x (double precision vector): continuous state
- nx (integer): size of x
- z (double precision vector): discrete state
- nz (integer): size of z
- u (double precision vector): input (regular)
- nu (integer) : size of input
- rpar (double precision vector): parameter
- nrpar (integer): size of rpar
- ipar (integer vector) : parameter
- nipar (integer) : size of ipar
- nclock (integer): event port number if the an event has arrived; 0 otherwise.
- flag (integer): 1, 2, 3, 4 or 5 indicating what the function should do, see below for more precision.
- nout (integer): size of out.

and the outputs are:

- out (double precision vector) : output y if flag=1, a work space area of size $\max(nx, nz)$ if flag=2 and nclock>0, $\dot{x}$ if flag=2 and nclock=0, t_{evo} if flag=3.
- x (double precision vector): new continuous state if flag=2 and nclock>0. Initialization if flag=4. Ending if flag=5.
- z (double precision vector): new discrete state if flag=2 and nclock>0. Initialization if flag=4. Ending if flag=5.
- flag (integer) : negative value indicates an error has been encountered.

There are different ways this function can be called.

Initialization If the function is called with flag=4, then the continuous and discrete states can be initialized, or more specifically reinitialized (if necessary) because they are already initialized by the interfacing function. This option is not useful in most cases, it is used by blocks that read and write data from file to open the file or by `scope` for initializing the graphics window.

Output update When flag=1, the simulator is requesting the output y which means that the function should realize the f function defined in (3.2). y should be returned in out.

State update When the simulator calls the function with flag=2 and nclock>0, it means that an event has arrived on the event input port number nclock, and that the simulator wants to update the states x and z according to (3.3)-(3.4). The current values of x and z are provided in x and z, and should be updated at the same place (this avoids useless and time consuming copies, in particular when part, or all of x or z is not to be changed). In out, a workspace of size $\max(nx, nz)$ is provided. If more space is needed, par and ipar can be used, in that case, dummy parameters need to be defined by the interfacing function. Otherwise, dynamic memory allocation can be used; but this usually slows down the simulation and should be avoided.

Integrator calls During the integration, the solver calls the function (very often) for the value of $\dot{x}$. This is done with flag=2 and nclock=0. In this case the function should realize f in (3.2) and place it on out.

Event scheduler To update the event schedule, the simulator calls the function with flag=3. In that case, out should return the t_{evo} as defined in (3.5).

Ending Once the simulation is done or at user request (by responding `End` in the `Run` menu) the simulator calls the function with flag=5. This is useful for example for closing files which have been opened by the block at the beginning or during the simulation and/or to flush buffered data.

The best way to learn how to write this function is to examine the Fortran routines in Scilab directory "routines/scicos". There you will find the computational functions of all Scicos blocks available in various palettes.

Example The following is the computational function associated with the `Abs` block.

```
      subroutine absblk(t,x,nx,z,nz,u,nu,rpar,nrpar,ipar,nipar,nclock,
     &     out,nout,flag)
      double precision t,x(*),z(*),u(*),rpar(*),out(*)
      integer ipar(*),flag
c     Absolute Value
      do 15 i=1,nu
         out(i)=abs(u(i))
 15   continue
      end
```

This example is particularly simple because `absblk` is only called with `flag` equal to 1. That is becaue this block has no state and no output event port. For more complex computational functions, see Fortran programs in the Scilab directory `routines/scicos`.

6.3 Creating user palettes

Once a block is defined by defining the corresponding interfacing and computational functions, it should be placed in a user defined palette. Users can have any number of palettes. To create user palettes, the Scilab variable `user_pal_dir` should be defined as a column vector of strings containing the paths to user palette directories (one directory for each palette). Each of these directories should contain a file named `blocknames` containing the name of the blocks in the corresponding palette (one name on every line). The name of the block is the name of the interfacing function without the `.sci` extension. The interfacing functions should of course be placed in this directory (see also the `Newblk` button in the `Edit` menu of Super Block).

Before using these blocks in Scicos, user must make sure that 1- the interfacing functions exist in Scilab environment (lib or getf), and 2- the computational functions exist in Scilab environment (dynamic link or permanent link if in Fortran and C, lib or getf if in Scilab).

6.4 Compiler and simulator

When the diagram is completed, the graphical information describing the diagram is first converted into a compact data structure using the function `c_pass1.sci` in the `SCI/macros/scicos` directory. Only information useful to the compiler and simulator is preserved (in particular all graphical data is discarded). Then, this data structure is used by the compiler which is the function `c_pass2.sci` in the `SCI/macros/scicos` directory. The result of the compilation is then used by the simulator (`scicosim` in the `SCI/routines/scicos`. The result of the compilation contains the information concerning the blocks and the order in which these blocks should be called under different circumstances. For example, the result of the compilation may be: when event i is fired, then blocks j and k should be called with flag 3, then block j and l with flag 2 and finally blocks i, j, l with flag 1. The order only matters for the case of flag 1.

Note that, when a block is called with flag 2 which normally means that its states should be updated, other operations can also be done. For example in the `scope` block, the plotting is done in this situation. In the blocks that handle read and write operations, the reading and writing is

done when the block is called with flag 2. To quickly inspect the value of a signal, it is very easy to do a `disp` of the input within a `scifunc` for example. Same thing can be done for flag 1 and 3, the problem, at least with flag 1 is that user may not know when and how many times the simulator is calling the block with this flag.

7 Examples

Scicos comes with a number of examples which are part of the Scilab demos. These diagrams can be examined, modified, simulated and copied. They present a good source of information on how the blocks should be used. Figures 7 through 7 illustrate some of the examples provided in the demos.

A simple diagram: a sine wave is generated and visualized. Note that Scope need to be driven by a clock!

A ball trapped in a box bounces on the boundaries. The x and y dynamics of the ball are defined in Super blocks

The x position Super block of Figure 7

A thermostat controls a heater/cooler unit in face of random perturbation

A Scicos GUI reference guide

A.1 Main menu

Right side of the Scicos window contains a set of buttons. At the start, available buttons are

- Help: To get help on an object or a menu button, click first on this button and then on the selected object or button.
- Edit..: Click on this button to obtain the edition menu.
- Simulate..: Click on this button to obtain the execution menu.
- File..: Click on this button to obtain the file management menu (Save, Load,...).
- View: To shift the diagram left, right, up or down, click first on the `View` button then on a point in the diagram. This point will appear in the middle of the graphic window.
- Exit: Click on this button to leave Scicos.

A.2 Edition Menu

- Help: See `help button` above.
- Palettes: A click on this button opens up a selection window with the list of available palettes.
- Move: If a block is not correctly placed it can be moved. This can be done by clicking on the `Move` button first then by clicking on the block to be moved, dragging the block to the desired location where a last click fixes the position (pointer position corresponds to the lower left corner position of the block box).
- Copy: To copy a block in main Scicos window, click first on this `Copy button`, then click on the "to-be-copied" block (in Scicos window or in a palette), and finally click where you want the copy to be placed in the Scicos window.
- Align: To obtain a nice diagram, you can align ports of different blocks, vertically or horizontally. Click first on the `Align` button then on the port of the first block and finally on the port of second block. The second block is moved. Note that a connected blocks cannot be aligned.
- Link: To connect an output port to an input port, click first on the `Link` button, then on the output port and finally on the input port. To split a link, click first on the `Link` button, then on the link where the split should be placed and finally on a block input port. Only one link can go from and to a port.
- Delete: To delete a block or a link, click first on the `Delete` button, then on the object to delete. All links connected to it are deleted as well.

- Flip: To reverse the position of the regular inputs and outputs of a block, click on the `Flip` button first and then on the desired block. This does not affect the order nor the position of the input and output event ports. A connected block cannot be flipped.
- Save: See `Save` button in `File..` menu.
- Undo: Click on this button to undo the last edit operation.
- Replot: Scicos window stores the complete history of the editing session. Click on the `Replot` button to erase the history and replot the diagram. Replot diagram before printing or generating postscript figures.
- View: See the description of `View` in the main menu.
- Back: Click on this button to go back to main menu.

A.3 File Menu

- Help : See `help button` above.
- New : Click on this button to create a new empty diagram.
- Save: Click on the `Save` button to save the block diagram in a binary file already selected by a previous call to `Save As` or `Load` button. If you click on this button and you have never used `Save As` or `Load` button, diagram is saved in the current directory as <window_name>.cos.
- Save As: Click on the `Save As` button to save the block diagram in a binary file. A dialog box will pop up.
- FSave: Click on the `FSave` button to save the block diagram in a formatted ascii file. A dialog box allows choosing the file which must have a `.cosf` extension. Formatted save is much slower than binary save but allows diagram transfer to different computers.
- Newblk: Click on the Newblk button to save the Super Block as a new Scicos block. A Scilab function is generated and saved in a file <window_name>.sci in a user palette directory. <window_name> is the name of the Super Block appearing on top of the window. A dialog allows choosing the palette directory.

  To make Scicos recognize your palettes, you should define the Scilab variable `user_pal_dir` as a column vector of strings containing the paths to your palette directories. This can, for example, be done in your `.scilab` (user initialization) file. You should also make sure the functions included in your palettes exists in Scilab environment before using Scicos (lib or getf).
- Load: Click on the Load button to load an ascii or binary file containing a saved block diagram. A dialog box allows choosing the file.
- Back: Click on this button to go back to the main menu.

A.4 Simulate menu

- Help : See `help button` above.
- Setup: In the main Scicos window, clicking on the `Setup` button invokes a dialog box that allows you to change window dimensions, integration parameters: absolute and relative error tolerances and the time tolerance (the smallest time interval for which the ode solver is used to update continuous states), and the simulation mode (1 or 2).

  In a Super Block window, clicking on this button invokes a dialog box that allows changing the window name (Super Block name) and window dimensions.
- Compile: Click on the `Compile` button to compile the block diagram. This button need never be used since compilation is performed automatically, if necessary, before the beginning of every simulation (`Run` button). Normally, a new compilation is not needed if only system parameters and internal states are modified. In some cases however these modifications are not correctly updated and a manual compilation may be needed before a `Restart` or a `Continue`. Please report if you encounter such cases.
- Run: Click on the `Run` button to start the simulation. If the system has already been simulated, a dialog box appears where you can choose to `Continue`, `Restart` or `End` the simulation.

  You may interrupt the simulation by clicking on the `stop` button, change any of the block parameters and continue the simulation with the new values.
- Back: Click on this button to go back to the main menu.

List of Figures