SCICOS A Dynamic System Builder and Simulator User's Guide - Version 1.0

Ramine Nikoukhah, Serge Steer

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

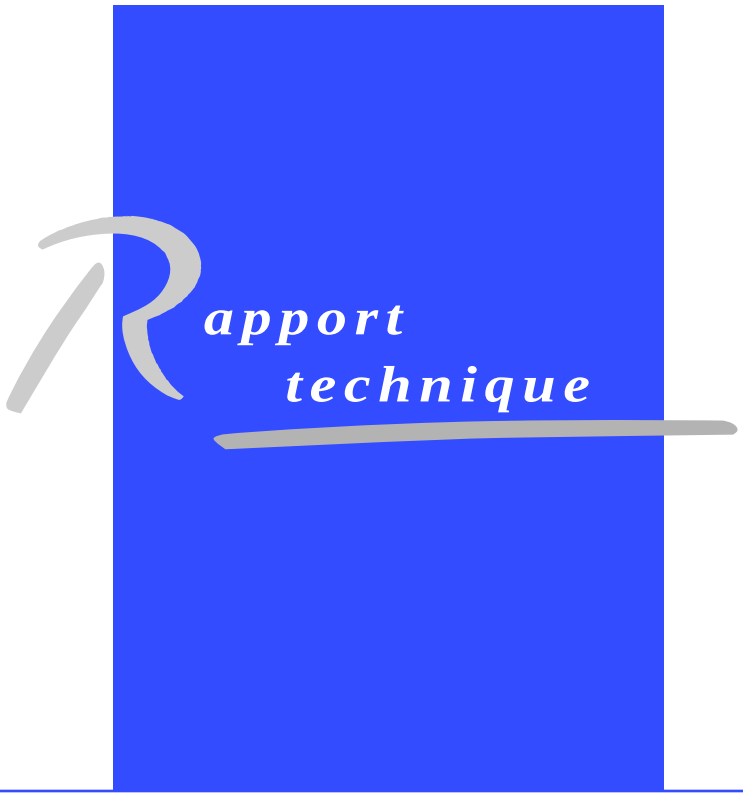
SCICOS
A Dynamic System Builder and Simulator
User's Guide - Version 1.0

Ramine Nikoukhah , Serge Steer

N ° 0207

Juin 1997

THÈME 4



ISSN 0249-0803

SCICOS
A Dynamic System Builder and Simulator
User's Guide - Version 1.0

Ramine Nikoukhah , Serge Steer

Thème 4 — Simulation et optimisation
de systèmes complexes
Projet META2

Rapport technique n ° 0207 — Juin 1997 — 80 pages

Abstract: Scicos (Scilab Connected Object Simulator) is a Scilab package for modeling and simulation of hybrid dynamical systems. More specifically, Scicos is intended to be a simulation environment in which both continuous systems and discrete systems co-exist. Unlike many other existing hybrid system simulation softwares, Scicos has not been constructed by extension of a continuous simulator or of a discrete simulator; Scicos has been developed based on a formalism that considers both aspects from the beginning. 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 this way.

Key-words: Nonlinear simulation, hybrid systems, block-diagram modeling, dynamic systems.

(Résumé : tsvp)

The development of Scicos is part of the ongoing project "Scilab" at INRIA. All the programs developed in this project are distributed free, with all the sources, through Internet and other media.

The objective of the development of Scicos is along the lines of that of Scilab, that is to provide the scientific community with a a completely open and free environment for scientific computing. Just like Scilab, Scicos is more than just a research tool, it has already been used in a number of industrial projects. Even though its applications so far have been mostly limited to the areas of control and signal processing, Scicos, and specially this new version of it, should find other applications in other areas.

This version of Scicos is a built-in library (toolbox) of Scilab 2.3. For information on Scicos and Scilab in general, consult `http://www-rocq.inria.fr/scilab` and newsgroup `comp.soft-sys.math.scilab`. The latest version of Scilab can be obtained by anonymous ftp from `ftp.inria.fr`. The source code and binary versions for various platforms can be found in the directory `/INRIA/Scilab`.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

SCICOS
Un éditeur bloc-diagramme et un simulateur de systèmes dynamiques

Résumé : Scicos (Scilab connected object simulator) est une boite à outils Scilab dédiée à la description et à la simulation des systèmes dynamiques hybrides. Plus précisément, Scicos et un environnement de simulation de systèmes, incluant des parties “continues” et “événementielles”. Contraitrement à d’autres simulateurs de systèmes hybrides, Scicos n’a pas été construit par extension d’un simulateur de systèmes continus ou de systèmes discrets. Scicos est basé sur un formalisme qui prend en compte les deux aspects. Scicos comprend un éditeur graphique de schéma-blocs qui peut être utilisé pour décrire des modèles complexes en connectant des blocs qui représentent des fonctions de base prédéfinies, disponibles dans des “palettes”, ou des fonctions utilisateur. Une large classe de systèmes hybrides peuvent être modélisés.

Mots-clé : Simulation non linéaire, systèmes hybrides, description bloc-diagramme, systèmes dynamiques.

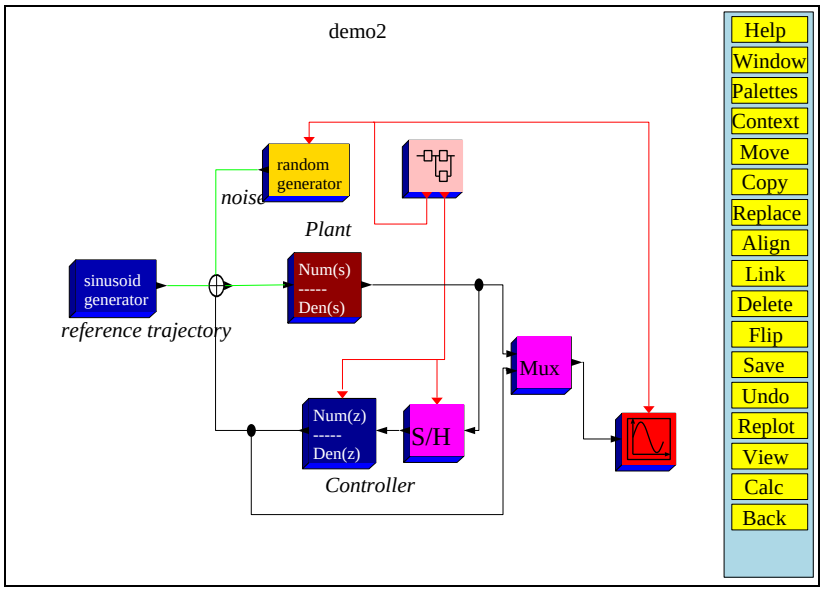
Contents

List of Figures

List of Tables

1 Introduction

Even though it is possible to simulate mixed discrete and 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.



There have been a number of models proposed in the literature for hybrid dynamical systems (see for example [3, 4]). A simple, yet powerful model is the following:

$$\dot{x} = f(x) \tag{1}$$

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

where $x \in \mathbb{R}^n$ is the state of the system, f is a vector field on $\mathbb{R}^n$, g is a mapping from $\mathbb{N} \times \mathbb{R}^n \to \mathbb{R}^n$ and h_i's are continuous functions. (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, (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 (1)-(2) cannot model time-dependent systems. This, however, can be done by state augmentation. For that it suffices to add $\dot{t} = 1$ to system equations and augment the state by adding t to it.

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

$$\xi(k+1) = f(k, \xi(k)), \tag{3}$$

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

$$\dot{e} = -1 \tag{4}$$

$$\text{if } e = 0, \quad \text{then } e := 1. \tag{5}$$

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:

$$\dot{k} = 0 \tag{6}$$

$$\dot{\xi} = 0 \tag{7}$$

$$\dot{e} = -1 \tag{8}$$

$$\text{if } e = 0, \quad \text{then } e := 1, \xi := f(k, \xi), k := k + 1. \tag{9}$$

State augmentation is a nice way of modeling hybrid systems in the form (1)-(2), however, it is in most cases not useful for the purpose of simulation. It is clearly too costly to integrate (5) for realizing an event generator, or integrate 1 to obtain

t. Even for model construction, (1)-(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 (1)-(2)). Scicos proposes a fairly rich set of operators for modeling hybrid systems from a modular description. The modular aspect (possibility of constructing a model by interconnection of other models) introduces additional complexity concerning, mainly, event scheduling and causality.

Scicos basic operators suffice to model many interesting problems in systems, control and signal processing applications. Scicos editor provides an easy to use graphical editor for building complex models of hybrid systems in a modular way, a compiler which converts the graphical description into Scicos executable code, and a simulator. The simple session presented in Chapter 2, the examples of Chapter 6 and specially the demos provided with the package, should allow new users to start building and simulating simple models very quickly. It is however recommended that users familiarize themselves with basic concepts and elementary building blocks of Scicos by reading Chapters 3, 4 and, at least the first section of Chapter 5.

Scicos Version 1.0 is still a beta test version; it has not been fully tested. Questions, discussions and suggestions should be posted to newsgroup

`comp.soft-sys.math.scilab`

and bug reports should be sent to `scilab@inria.fr`.

2 Getting started

This section presents the steps required for modeling and simulating a simple dynamic system in Scicos. In Scicos, systems are modeled in a modular way by interconnecting subsystems referred to as *blocks*. The model we construct here uses only existing blocks (available in various palettes); the procedure for creating new blocks will be discussed later.

2.1 Constructing a simple model

In Scilab main window, type `scicos();`. This opens up an empty Scilab graphic window with a menu bar on the side (Figure 1). By default, this window is named `Untitled`.

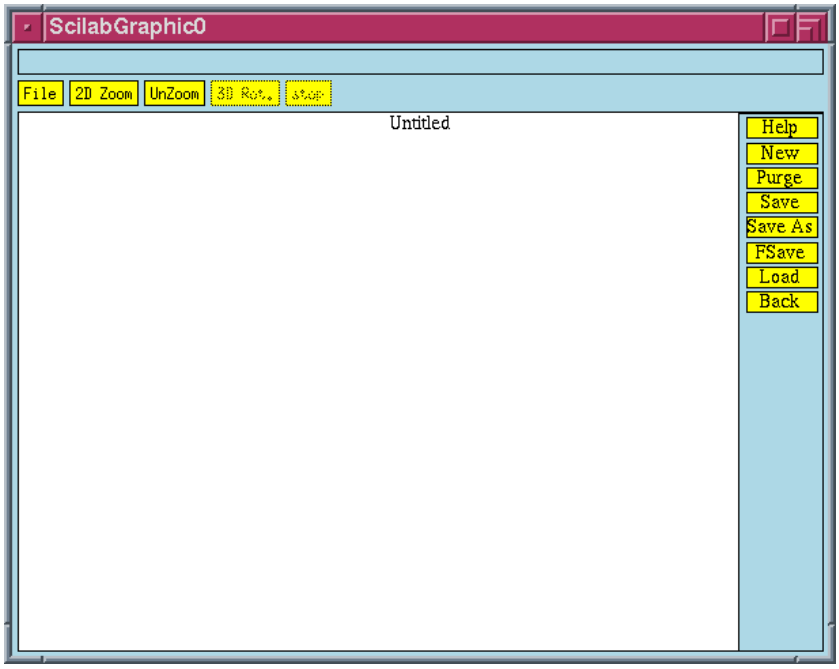


Figure 1: Scicos main window

Click on the `Edit..` button to obtain the `Edit` menu bar. To open up a palette, click on the `Palettes` button. You are then presented with a choice of palettes (Figure 2). Click on `Inputs/Outputs`; this opens up a palette which is a new Scilab graphic window containing a number of blocks (Figure 3). To copy a block, click first on the `copy` button in Scicos main window, then on the block to be copied in the palette, and finally somewhere in the Scicos main window, where the block is to be placed.

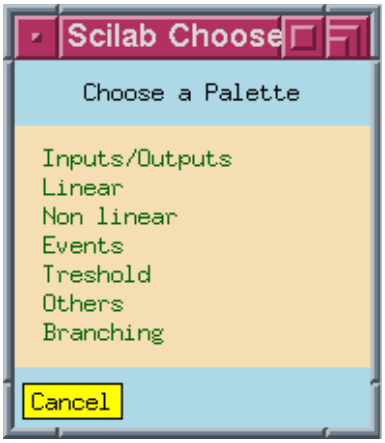


Figure 2: Choice of palettes

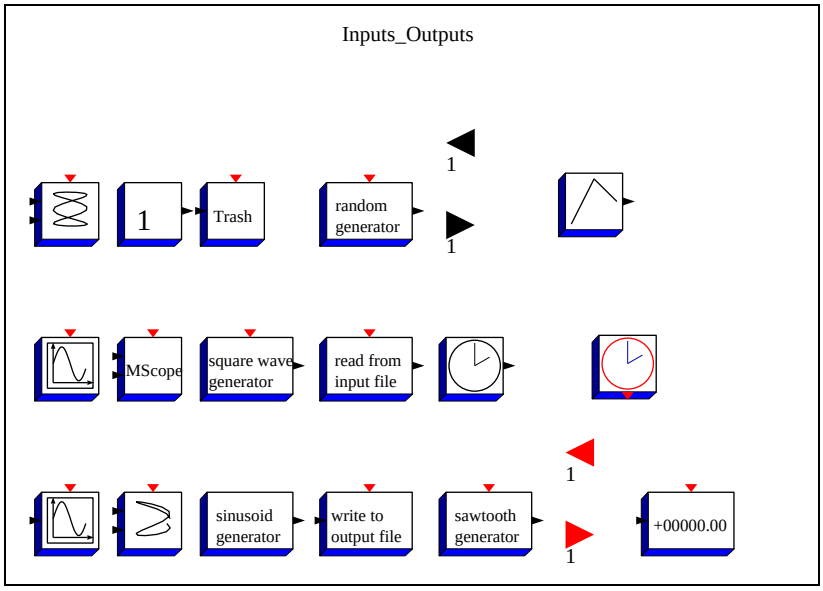


Figure 3: Inputs/Outputs Palette

Using this procedure, copy the `MScope` block, the `sinusoid generator` block and the `Clock` block (the clock with an output port on the bottom) into Scicos main window. The result should look like in Figure 4.

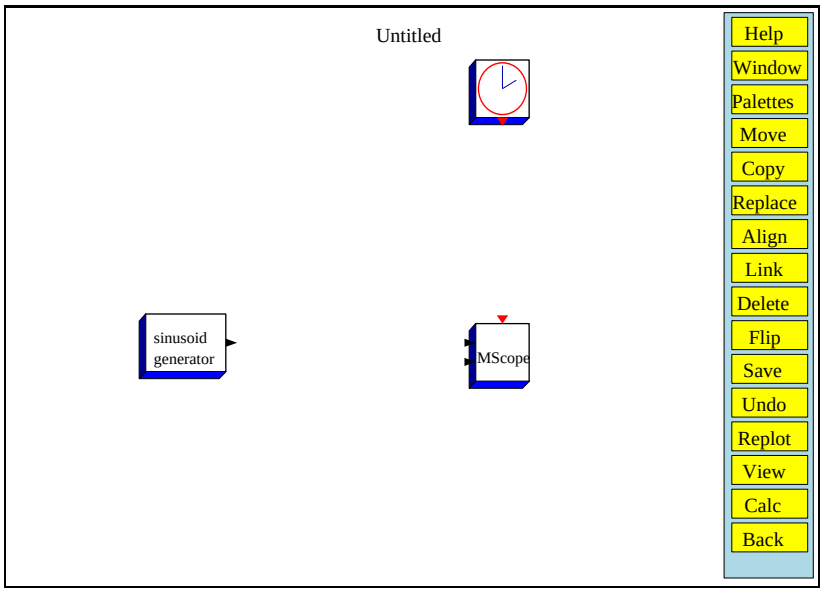


Figure 4: These blocks have been copied from the Inputs/Outputs Palette

Open the `Linear` palette and copy the block `1/s` (integrator) into Scicos main window. Connect the input and output ports of the blocks by clicking first on the `link` button, then on the output port and then on the input port (or on intermediary points if you don't just want a straight line connection), for each new link. Connect the blocks as illustrated in Figure 5. To make a path originate from another path, i.e., be split, click 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).

The `Clock` generates a train of impulses which tells the `MScope` block at what times the value of its inputs must be displayed. To inspect (and if needed change) the `Clock` parameters, click on the `Clock` block. This opens up a dialogue panel as illustrated in Figure 6. At this point the period, i.e. the time period between events and the time of the first event can be changed. Let's leave them unchanged; click on `Cancel` or `OK`. Similarly you can inspect the parameters of other blocks.

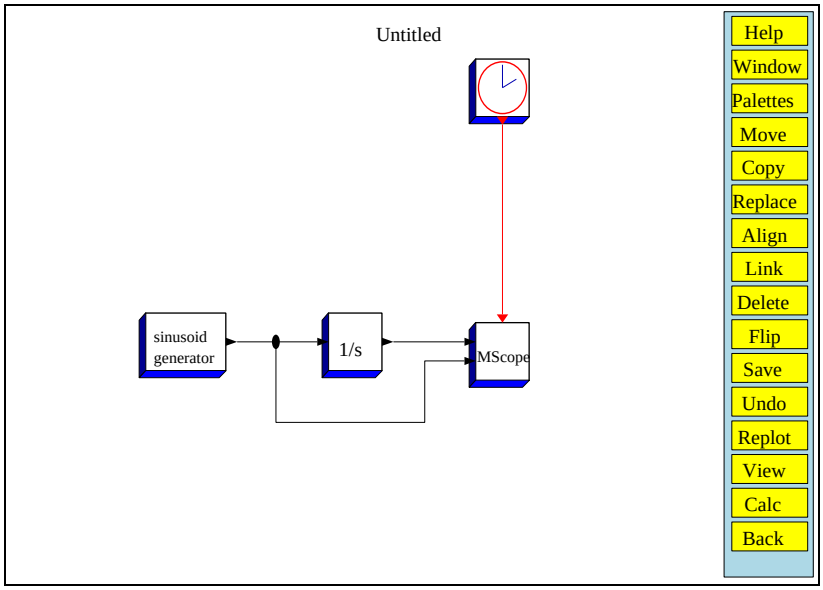


Figure 5: Complete model

You can now save your diagram by clicking on the `Save` button. This saves your diagram in a file called `Untitled.cos` in the directory where Scilab was launched.

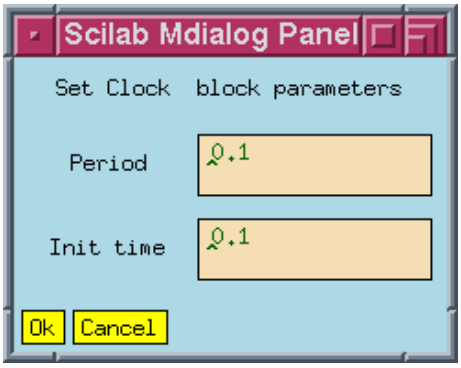


Figure 6: `Clock`'s dialogue panel

2.2 Model simulation

It is now time to do a simulation. For that you need to leave the `Edit..` menu; click on `back`. You are now back to the main menu. Click on `Simulate..`, you have now the simulation menu. Click on `Run`. After a short pause (time of compilation), the simulation starts; see Figure 7.

The simulation can be stopped by clicking on the `stop` button on top of the Scicos main window. It is clear at this point that the `MScope`'s parameters need to be adjusted. So, click on the `MScope` block; this opens up a dialogue box (see Figure 8).

Clearly to improve the display, we must change `Ymin` and `Ymax` of the two plots. The first input ranges from 0 to 2 and the second form -1 to 1; `Ymin` and `Ymax` can now be adjusted accordingly. Increasing the `Buffer size` can speed up the simulation, but can make the display "jerky". The refresh period is the maximum time displayed in a single window. We can also change the colors of the two curves. Modify the parameters in the dialogue box as in Figure 9.

Click now on `OK` to register the new parameters and get back to the Scicos main window. You can now restart the simulation by clicking on `Run` and selecting `Restart` among the proposed choices. The result is depicted in Figure 7. Note that this time the simulation starts right off, no need for re-compilation.

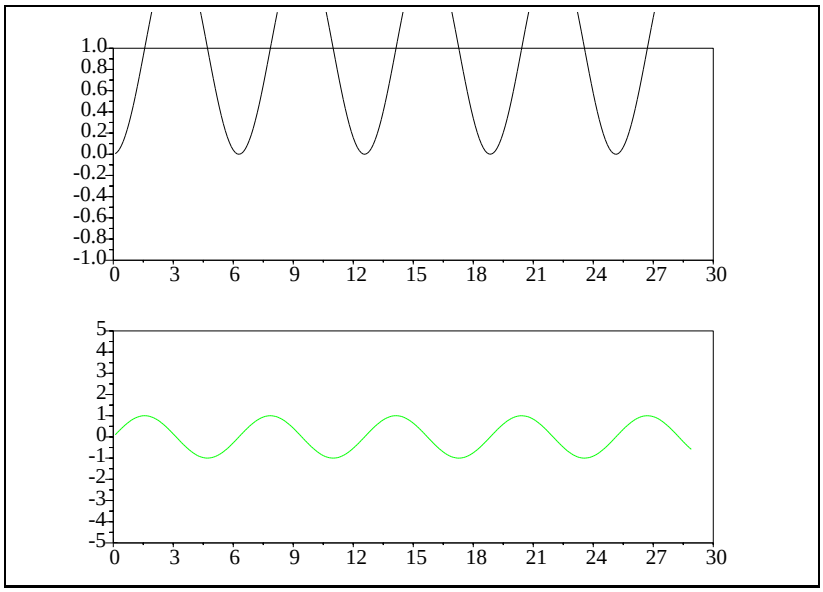


Figure 7: Simulation result

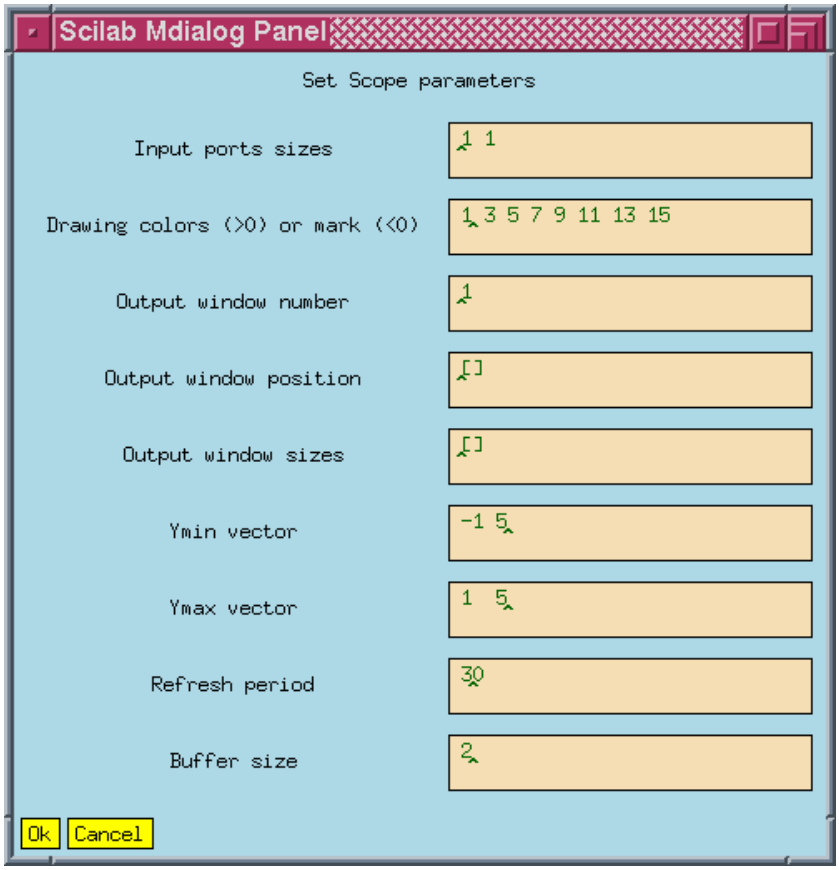


Figure 8: `MScope` original dialogue box

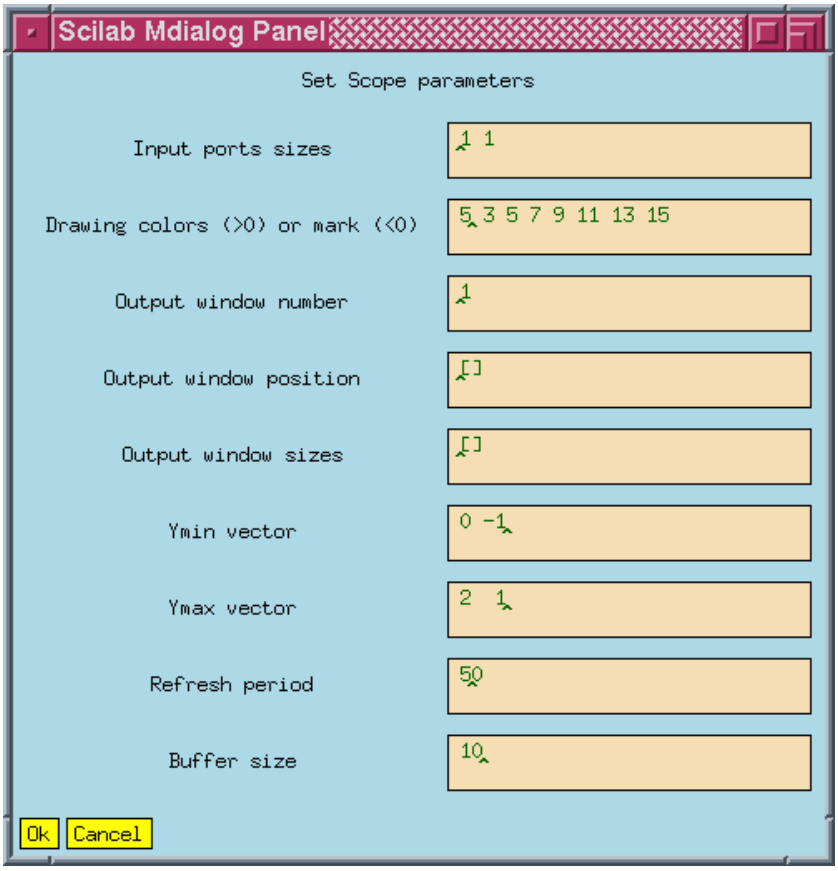


Figure 9: MScope modified dialogue box

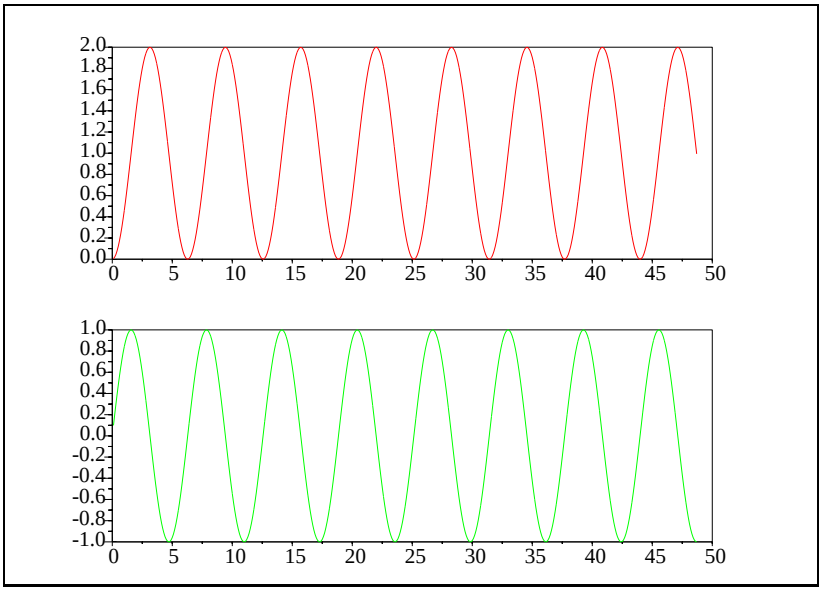


Figure 10: Simulation result after modifications

To make your diagram look good, you can go back to the main menu and click on `Block...` Here you can change the background colors of the blocks (`Color`), the texts or pictures displayed inside them (`Icon`) and their sizes (`Resize`). For example to color the block MScope, click on the button `Color`, then on the block. This opens up a color palette; select the desired color by clicking on it, and confirming by `OK`. To change the color of a link, simply click on it.

You can also place text on your diagram by copying the block `Text` in `Others` palette into your diagram and changing the text, the font and the size by clicking on it.

Finally, you can print and export, in various formats, your diagram using the `File` menu on top of the corresponding graphics window. Make sure to do a replot before to clean up the diagram. The result will be like the diagrams in this document.

2.3 Symbolic parameters and "context"

In the above example, the block parameters seem to be defined numerically. But in fact, even when a number is entered in a block's dialogue, it is first treated and memorized symbolically, and then evaluated. For example in the block `sinusoid generator`, we can enter `2-1` instead of `1` for `Magnitude` and the result would be the same (see Figure 11); opening up the dialogue the next time around would display `2-1`. We can go further and for example use a symbolic expression such as `sin(cos(3)-.1)`.

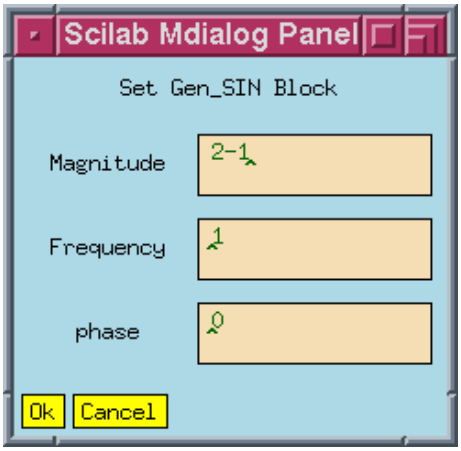


Figure 11: Symbolic expression as parameter

In fact we can use any valid Scilab expression, including Scilab variables. But, these variables (symbolic parameters) should be defined at some point. For that click on the `context` button in the `edit` menu. You are then presented with a "Dialogue Panel" (see Figure 12) in which you can define symbolic parameters (in this case `ampl`). Once you click on `OK`, the variable `ampl` can be used in all the blocks in the diagram. It can for example be used to change, in `sinusoid generator` block, the magnitude of the sine-wave (see Figure 13).

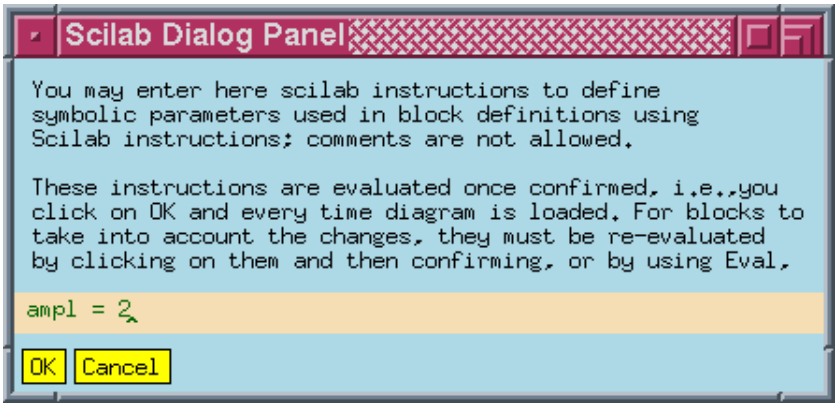


Figure 12: Context is used to give numerical values to symbolic expressions

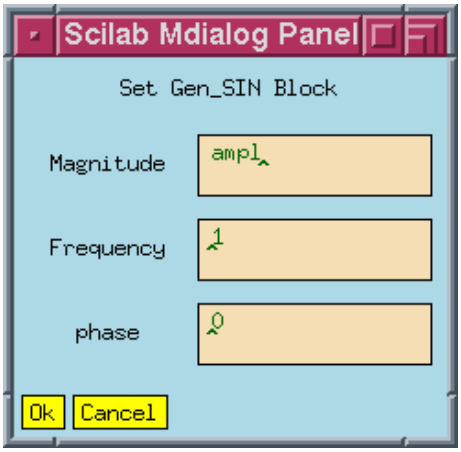


Figure 13: Use of symbolic expressions in block parameter definition

The context of the diagram is saved with the diagram and the expressions are re-evaluated when the diagram is reloaded. Note however that if you change the context, you must re-evaluate the diagram if you want the blocks to take into account the changes. That can be done either by clicking on the `eval` button in the `Simulate` menu, or by clicking on the concerned blocks and confirming.

2.4 Use of Super Block

It would be very difficult to model a comlex system with hundreds of component in one diagram. For that, Scicos provides the possibility of grouping blocks together and defining sub-diagrams called Super Blocks. These blocks have the appearance of regular blocks but can contain an unlimited number of blocks, even other Super Blocks. A Super Block can be duplicated using the `copy` button in the `Edit` menu and used any number of times.

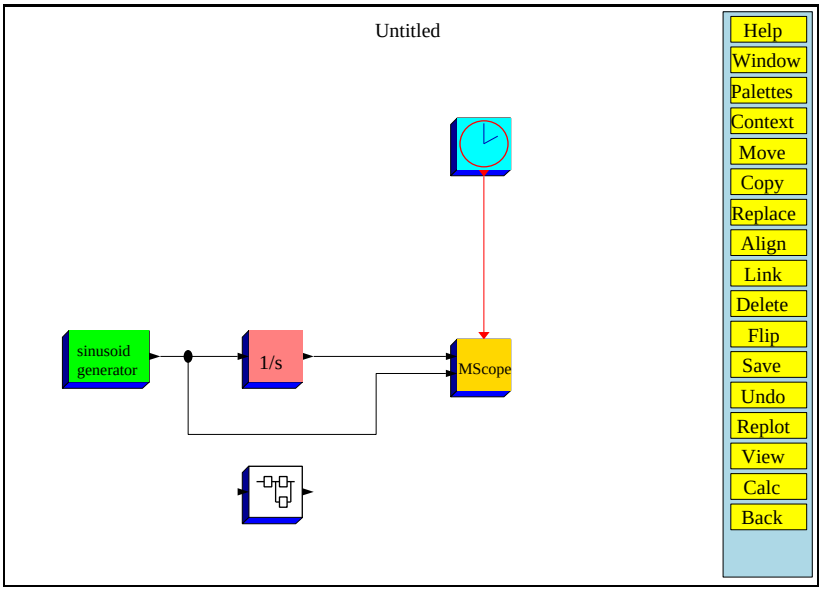


Figure 14: Super Block in the diagram

To define a Super Block, copy `Super Block` from `Others` palette into your diagram (see Figure 14) and click on it. This opens up a new Scicos panel with an empty diagram. Edit this diagram by copying and connecting a `S/H` block (sample and hold), a discrete linear system, and input and output Super Block ports as in Figure 15.

Once you exit (using `Exit` button) from the Super Block panel, you get back to the original diagram. Note that the Super Block now has the right number of inputs and outputs, i.e., one event input port on top, one regular input port and one regular output port. To drive this discrete component, we need a `Clock`; just copy the `Clock` already in the diagram

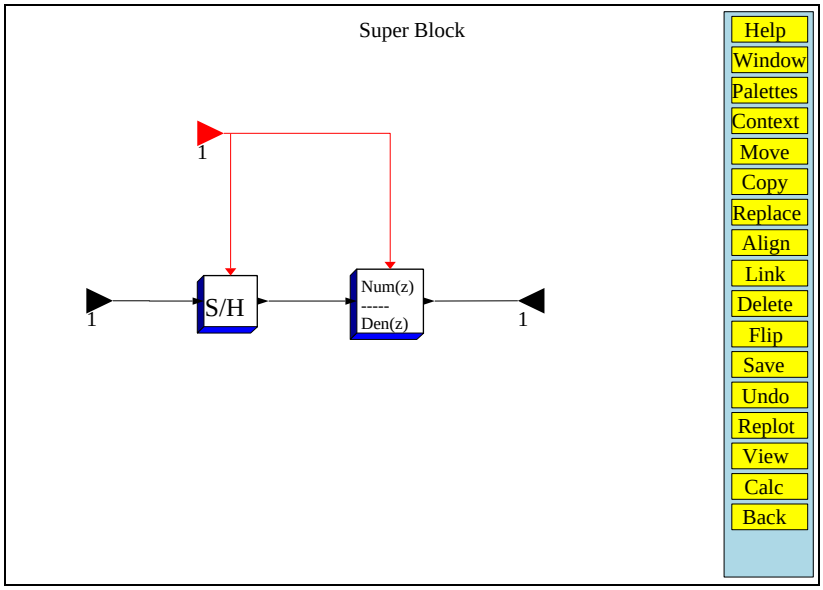


Figure 15: Super Block content

(see Figure 16). Note that there is no need to disconnect the links to `MScope` to change the number of its inputs. Simply update its parameters as in Figure 17; the input ports adjust automatically.

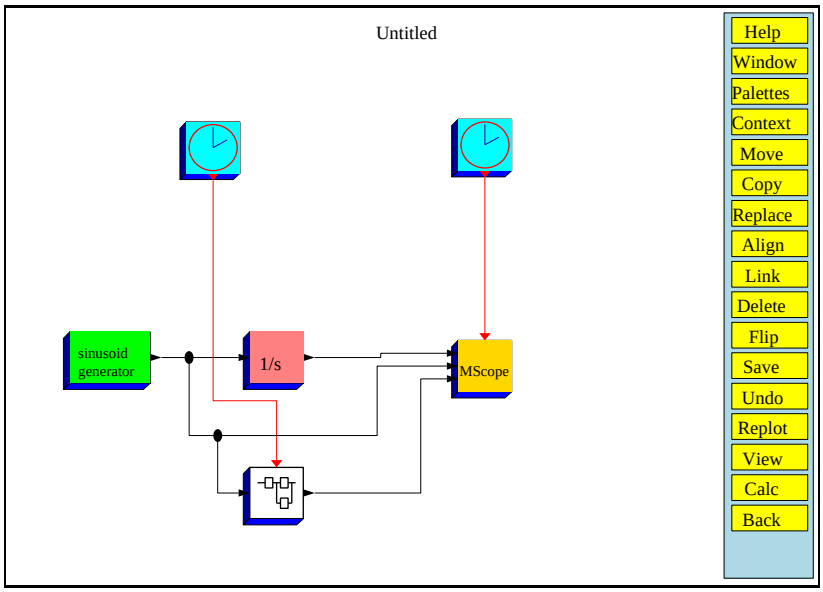


Figure 16: Complete diagram with Super Block

Before simulating, set the period of this new `Clock` to 2 (to observe the discrete behavior, period of discretization must be larger than that of the scope). You can now `Run` the diagram.

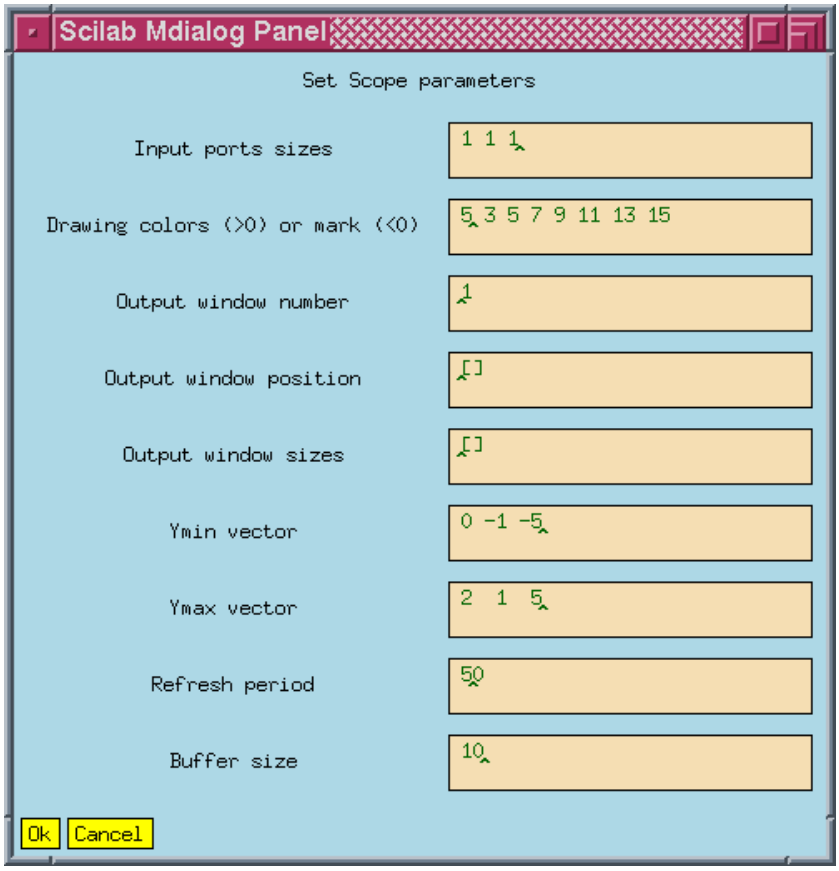


Figure 17: MScope updated dialogue box

3 Basic concepts

Models in Scicos are constructed by interconnection of Basic Blocks. There exist four 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.[1] One way to think of event signals as physical signals 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 behaviors of Basic Blocks is necessary.

3.1 Basic Blocks

There are four types of Basic Blocks in Scicos.

3.1.1 Continuous Basic Block

Continuous Basic Blocks (CBB) can have both regular and event input and output ports. CBB's can model more than just continuous dynamics systems. A CBB can have a *continuous state* x and a *discrete state* z. Let the vector function u denote the regular inputs and y the regular outputs. Then a CBB imposes the following relations

$$\dot{x} = f(t, x, z, u, p) \tag{10}$$
$$y = h(t, x, z, u, p) \tag{11}$$

where f and h are block specific functions, and p is a vector of constant parameters. Constraints (10)-(11) are imposed by the CBB as long as no events (impulses) arrive on its event input ports. An event input can cause a jump in the states of the CBB. Let's say one or more events arrive on CBB's event ports at time t_e, then the states jump according to the following equations:

$$x := g_c(t_e, x(t_e^-), z(t_e^-), u(t_e^-), p, n_{evprt}) \tag{12}$$
$$z := g_d(t_e, x(t_e^-), z(t_e^-), u(t_e^-), p, n_{evprt}) \tag{13}$$

where g_c and g_d are block specific functions; n_{evprt} designates the port(s) through which the event(s) has (have) arrived; x and z are the vectors of continuous state and discrete state. $z(t_e^-)$ is the previous value of the discrete state z; z remains constant between any two successive events.

Finally, CBB's can generate event signals on their event output ports. These events can only be scheduled at the arrival of an input event. If an event has arrived at time t_e, the time of each output event is generated according to

$$t_{evo} := k(t_e, z(t_e), u(t_e), p, n_{evprt}) \tag{14}$$

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 t_e. If an element is less than t_e, it simply means the absence of an output event signal on the corresponding event output port. $t_{evo} = t$ should be avoided, the resulting causality structure is ambiguous. Note that even if $t_{evo} = t$, this does not mean that the output event is synchronized with the input event; synchronization of two events is a very restrictive condition (see Synchro blocks). Two events can have the same time but not be synchronized.

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 CBB with event output ports. 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.

Only one output event can be scheduled on each output event port, initially and in the course of the simulation, 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

[1] Regular paths are vectorized in a sense that each link can carry a set of functions; event links are not.

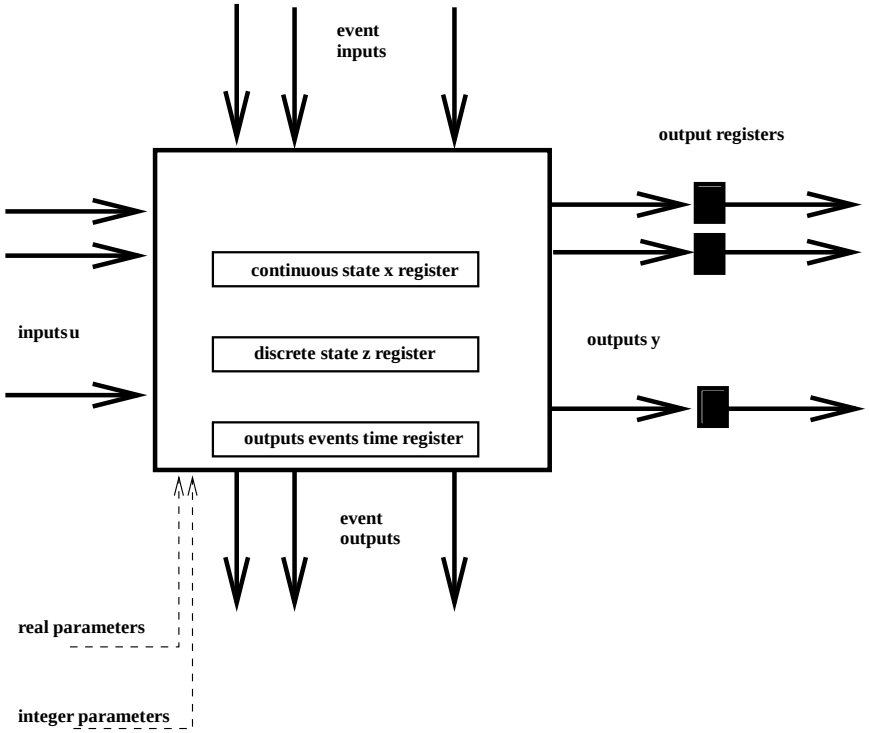


Figure 18: A Continuous Basic Block

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. Another interpretation is that as long as the previously scheduled event has not been fired, the corresponding output port is considered busy, meaning it cannot accept a new event scheduling. If the simulator encounters such a conflict, it stops and returns the error message *event conflict*.

In the unlikely event that a block receives two (or more) successive events having the same time (without them being synchronized). For the second event, the t_e^- should be interpreted as *previous value of*. Note that in that case, the values of x and u at t_e and t_e^- are not uniquely defined; this means that the time t is not the proper independent variable in terms of which we should express system's equations but rather a more general concept of time such as the time of simulation should be used. Expressed in such a time frame, at the arrival of one or more events at time t_e, the variable t freezes and the states are updated and then t goes on.

3.1.2 Discrete Basic Block

The CBB monitors permanently its inputs and updates continuously its outputs and continuous state. In contrast, Discrete Basic Blocks (DBB) act only when they receive an input event and their actions are instantaneous. DBB's can have both regular and event input and output ports but they must have at least one event input port. DBB's can model discrete dynamics systems. A DBB can have a discrete state z but no continuous state. Let u denote the regular inputs and y the regular outputs, then, upon the arrival of an event (or events) at time t_e, the state and the outputs of a DBB change as follows

$$z := f_d(t_e, z(t_e^-), u(t_e^-), p, n_{evprt}) \quad (15)$$

$$y := h_d(t_e, z, u(t_e), p) \quad (16)$$

where f_d and h_d are block specific functions, and p is a vector of constant parameters and n_{evprt} designates the port(s) through which the event(s) has (have) arrived. Needless to say that y remains constant between any two successive events so that the output y and the state z are piece-wise constant, right-continuous functions of time. Like CBB's, DBB's can generate output events according to (14). These events can also be initialized as in the case of CBB's.

The difference between a CBB and a DBB is that a DBB cannot have a continuous state and that its outputs remain constant between two events. It is very simple to emulate a DBB by a CBB, so why a use a DBB? The reason is that by

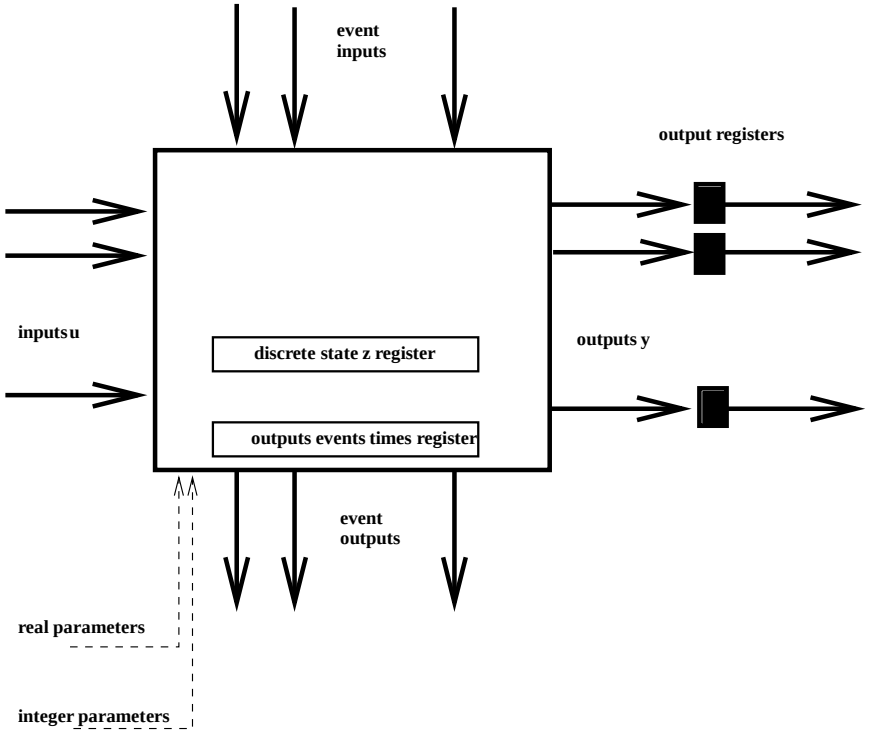


Figure 19: A Discrete Basic Block

specifying the block to be a DBB, the simulator knows that the outputs of this block remain constant between events and uses this information to optimize simulation performance.

Note that the regular output signal of a DBB is always piece-wise constant (we refer to it as discrete signal). Being piece-wise constant does not imply necessarily that a signal is discrete, for example the output of an integrator (which is a CBB with continuous state) can in some special cases be constant; the discrete property characterizes signals that are piece-wise constant based solely on the basic properties of the blocks that generate them. In particular, in Scicos, every regular output signal of a DBB is discrete and every regular output signal of a state-less time invariant CBB receiving only discrete signals on its inputs is also discrete. Thus, the discrete nature of signals in a model can be specified off-line; the compiler does this and uses this information to optimize simulation performance.

Most of the elementary blocks in Scicos are either CBB's or DBB's: the followings are a few examples.

Static blocks A static block is one where the (regular) outputs are static functions of its inputs. 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 CBB's.

Even though Static blocks are CBB's, they can be used even in purely discrete models. It is of course possible to construct static DBB's (i.e. blocks that realize static functions of their inputs on their outputs only when events are received on their event input ports) but it turns out that a static DBB does not necessarily perform any better than a static CBB if its inputs are discrete signals. In particular, knowing the discrete nature of its inputs, the compiler does not make useless repeated calls to the static CBB, it makes only one call every time one of the inputs jumps.

The `Non linear` palette contains a number of examples of Static blocks.

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

$$\xi(k+1) = m(\xi(k), u(k)) \quad (17)$$

$$y(k) = n(\xi(k), u(k)) \quad (18)$$

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

Clocks A clock is a generator of event signals on a periodic basis. CBB's and DBB's cannot act as a clock. The reason is that, except for a possible pre-scheduled initial output event, CBB's and CDD's must receive an event signal on one of their 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 DBB which has no state, no regular input or output. It has one event input port and one event output port. When an event arrives, it schedules an event on its event output port, i.e., 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 20), 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).

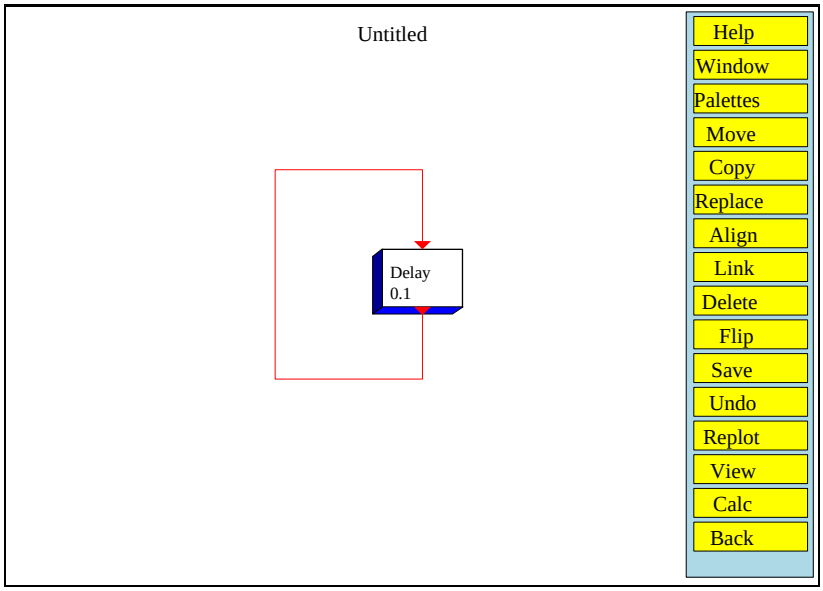


Figure 20: Constructing an event clock using feedback on a delay block

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 this way, so is modeling clocks with variable frequencies, etc...

3.1.3 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 `Threshold` 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.

Inputs of Zero Crossing Basic Blocks should not remain zero. This situation is ambiguous and is declared as an error. Note however that these inputs can start off at zero. Similarly the input of a zero Crossing Basic Block should not jump across zero. If it does, the crossing may or may not be detected.

Zero Crossing Basic Blocks cannot be modeled as CBB's or DBB's because in these Blocks, no output event can be generated unless an input event has arrived beforehand.

3.1.4 Synchro Basic Block

These are the only blocks that generate output events that are synchronized with their input events. These blocks have a unique event input port, a unique (possibly vector) regular input, no state, no parameters, and two or more event output ports. Depending on the value of the regular input, the incoming event input is routed to one of the event output ports. An example of such a block is the `event select` block in the `Branching` palette. The other is the `If-then-else` block in the same palette.

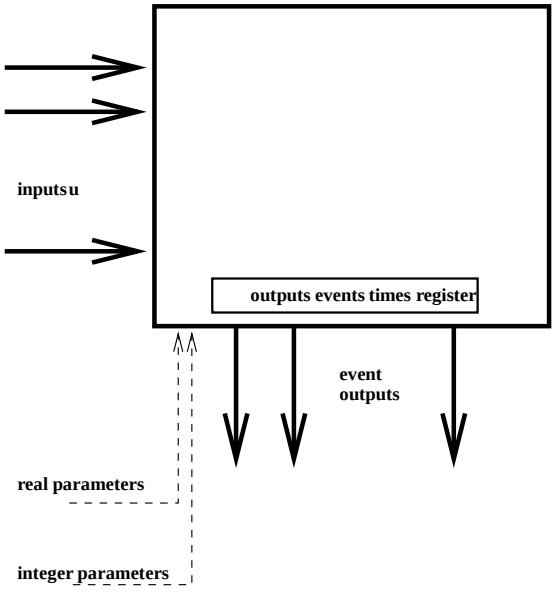


Figure 21: A zero Crossing Basic Block

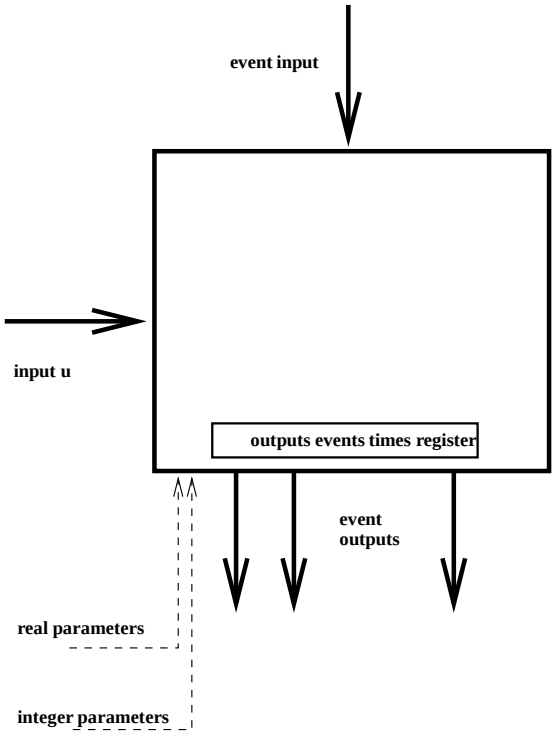


Figure 22: A Synchro Basic Block

These blocks are used for routing and under-sampling event signals.

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 (12)-(13) or (15)-(16) and (14). 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.

3.2.1 Event split

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. Even though it is not implemented as a block, the behavior of the event split resembles that of Synchro blocks.

3.2.2 Event addition

Besides split, there exists another block which operates on event paths: the event addition in the `Branching` palette. 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 but are not synchronized, the output would still consist of two event impulses, having the same time as that of the input event signals. If the two input events are synchronized, only one gets through.

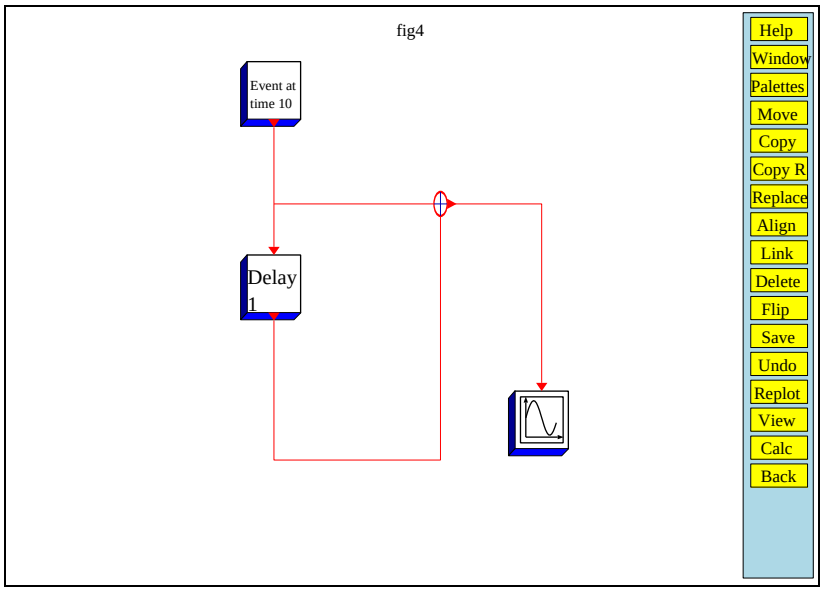


Figure 23: Event links: Split and addition

3.2.3 Synchronization

Synchronization is an important issue in Scicos. As we have seen, 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". The only way two event signals can be synchronized is that they can be traced back, through event paths, event additions, event splits and Synchro blocks 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 24. The events on the second event output port of this Super Block are clearly synchronized with a subset of events on its first ouput event port. In fact the events on the first output event port of the Super Block are the union of the output events on both output event ports of the `M.freq.clock`. This block generates $n-1$ times an event on its first output event port, then one event on its second output event port, always with the same delay d with respect to the input event; and starts over. This means that second ouput event port acts like a clock having a frequency n times lower than the input frequency of the block, which is also the first event output of the Super Block. Specifically, the frequency of the train of events generated on the first output event port is $1/d$ and on the second is n/d.

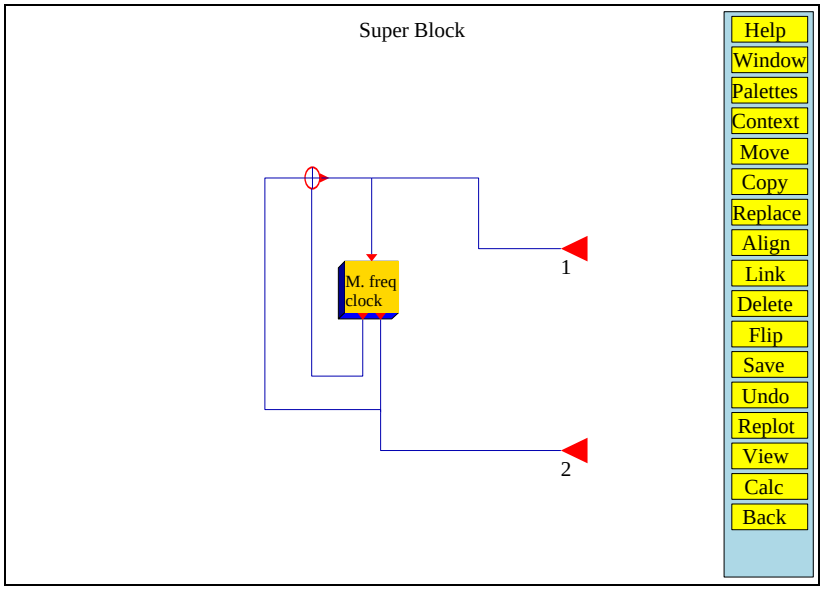


Figure 24: Super Block defining a 2-frequency clock.

4 Block construction

In addition to the blocks available in Scicos' palettes, user can use custom blocks. Super Blocks allow block functionality to be defined by graphically interconnecting existing blocks and new blocks can be constructed by compiling Super Blocks. But the standard way of constructing a new block is by defining a pair of functions: one for handling the user-interface (*Interfacing* function) and the other for specifying its dynamic behavior (*Computational* function). The first function is always written in Scilab but the second can be in Scilab, C or Fortran. C and Fortran routines dynamically linked or permanently interfaced with Scilab can be used and give the best results as far as simulation performance is concerned. The `Scifunc` and `GENERIC` blocks in the `others` palettes are generic *Interfacing* functions, very useful for testing user-developed *Computational* functions in the early phases of development.

4.1 `Super Block`

Not all blocks in Scicos' palettes are Basic Blocks; some are Super Blocks. 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.

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 without any limit on the number or the depth.

A Super Block can be saved in various formats. If the Super Block is of interest in other constructions, it can be saved or converted into a block using the `Newblk` button and placed in a user palette using `Pal editor..` menu.

If the Super Block is only used in a 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 in the `edit..` menu.

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.

4.2 `Scifunc` block

The `Scifunc` block allows using Scilab language for defining Scicos blocks. It provides a generic *Interfacing* function and a generator of *Computational* function. The *Computational* function is generated interactively, user needing only to define block parameters such as the number and sizes of inputs and outputs, initial states, type of the block, the initial firing vector and Scilab expressions defining various functions defining the dynamic behavior of the block. This is done interactively by clicking on the `Scifunc` block, once copied in the Scicos window. Besides its performance (the generated function being a Scilab function) the main disadvantage of `Scifunc` is that the dialogue for updating block parameters cannot be customized.

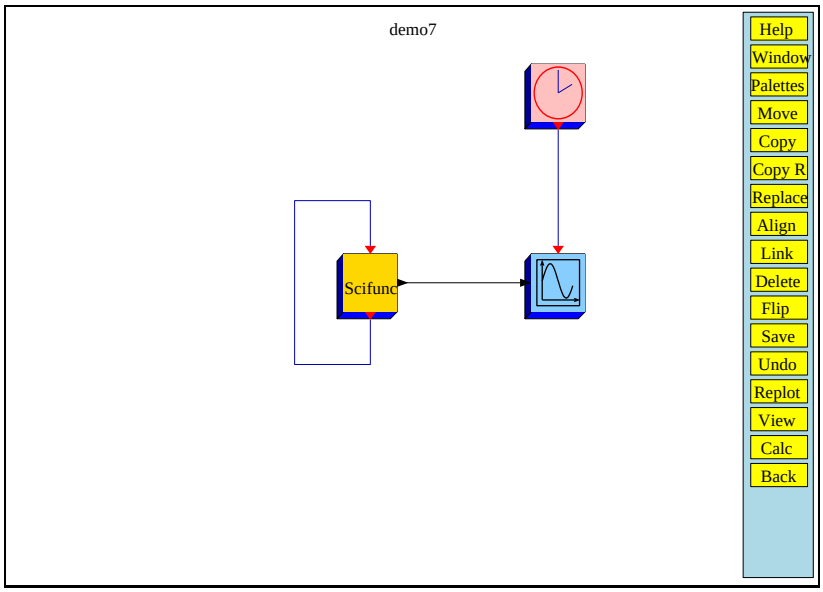


Figure 25: Diagram with a Scifunc block

4.3 `GENERIC` block

The `GENERIC` block also provides a generic *Interfacing* function but the *Computational* function needs to be defined separately, either as a Scilab function, or a Fortran or C function. Compared to the `Scifunc` block, `GENERIC` block requires more information on the properties of the corresponding *Computational* function. Besides the name of the function, user should specify information such as the type, and whether or not the block contains a direct feed-through term it is time-varying (i.e., at least one of the outputs depend explictly on one of the inputs or on time).

Another important difference with `Scifunc` is that the *Computational* function of `Scifunc` is part of the data structure of the diagram, and thus it is saved along with the diagram. But in the case of `GENERIC` block, only the name of the function figures in the data structure of the diagram. This means that the functions realizing *Computational* functions of `GENERIC` blocks of a Scicos diagram must be saved along with the diagram, and loaded or dynamically linked before simulation. One way to make such a diagram self-contained, if the *Computational* functions of all `GENERIC` blocks are Scilab functions, is to define these functions in the "contexts" of the diagram, more on that later.

4.4 *Interfacing* function

For defining a fully customizable basic block, user must define a Scilab function for handling the user interface. This function, referred to as the *Interfacing* function, not only determines the geometry, color, number of ports and their sizes, icon, etc..., but also the initial states, parameters, and also handles the dialogue for updating them.

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

4.4.1 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 unused.
  - `x,y,typ` are unused.

  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 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 (1 for regular and 2 for event).

  In general we 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 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 unused.
  - `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 unused.

  In general we can use the `standard_origin` function.
- job==’set’: function opens up a dialogue for block parameter acquisition, if any.
  - `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 (name of corresponding *Computational* function, type, number and sizes of inputs and outputs, etc...).
  - `arg1, arg2` are unused.
  - `x` is the data structure of the block.
  - `y` is unused.
  - `typ` is unused.

4.4.2 Block data-structure definition

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

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

where `GUI_function` is a string containing hte name of the corresponding *Interfacing* function and `graphics` is the structure containing the graphics 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 width
- h: block height
- orient: boolean, specifies if block is flipped
- 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 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.
- gr_i: character string vector, Scilab instructions used to draw the icon.

and `model` is the data structure relative to simulation

```
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 simulation function name (fortran, C, or Scilab function). Second element is an integer specifying the type. The type of a *Computational* function specifies essentially its calling sequence; more on that later.
- #input: vector of size equal to the number of 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 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 event input ports. All entries must be equal to 1. Scicos does not support (for the moment) vectorized event links.
- #clk_output: vector of size equal to the number of event output ports. All entries must be equal to 1. Scicos does not support (for the moment) vectorized event links.
- state: vector (column) of initial continuous state
- dstate: vector (column) of initial discrete state
- rpar: vector (column) of real parameters passed to corresponding *Computational* function.

- ipar: vector (column) of integer parameters passed to corresponding *Computational* function.
- typ: string: 'c' if continuous, 'd' if discrete, 'z' if zero-crossing, and 'l' if Synchro basic block
- firing: vector of initial firing times of size -number of clock outputs- which 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 system output depends explicitly on time
- label: character string, used as block identifier. This field may be set by the `label` button in `Block` menu (see Appendix B).

4.4.3 Examples

Example of a static block The block `ABSBLK` that realizes the absolute value function in the `Non linear` palette has a simple *Interfacing* function because there is no parameters to be set in this case.

```
function [x,y,typ]=ABSBLK_f(job,arg1,arg2)
//Absolute value block
x=[];y=[];typ=[];
select job
case 'plot' then
  standard_draw(arg1)
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;
case 'define' then
  in=-1 // ports have equal undefined dimension
  model=list(list('absblk',1),in,in,[],[],[],[],..
                 [],[],'c',[],[%t %f],' ',list())
  gr_i='xstringb(orig(1),orig(2),''Abs'',sz(1),..
                                    sz(2),''fill'')'
  x=standard_define([2 2],model,' ',gr_i)
end
```

Example of a static block with parameter The `LOGBLK` block interfacing function is somewhat more complicated because the basis of the log function can be set.

```
function [x,y,typ]=LOGBLK_f(job,arg1,arg2)
x=[];y=[];typ=[];
select job
case 'plot' then
  standard_draw(arg1)
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;
  dlg=x(2)(4) // symbolic parameters
  while %t do
    // open dialogue window
    [ok,a,dlg]=getvalue('Set log block parameters',..
         'Basis (>1)',list('vec',1),dlg)
    if ~ok then break,end
    // Check user answers consistency
    if a<=1 then
      x_message('Basis must be larger than 1')
    else
      if ok then
        // update block's data structure
        x(2)(4)=dlg //parameter expressions
        x(3)(8)=a  //parameter value
        break
      end
    end
  end
case 'define' then
  a=%e
  model=list({'logblk',0},-1,-1,[],[],[],[],a,[],..
                          'c',[],[%t %f],' ',list())
  // symbolic parameters
  dlg='%e'
  //initial icon definition
  gr_i=['xstringb(orig(1),orig(2),''Log'',sz(1),sz(2),..
                                          ''fill'');']
  sz=[2 2] //initial block size
  x=standard_define(sz,model,dlg,gr_i)
end
```

Example of a complex CBB The following is the *Interfacing* function associated with the `Jump (A,B,C,D)` block in the `Linear` palette. This block realizes a continuous-time linear state-space system with the possibility of jumps in the state. The number of inputs to this block is two. The first input vector is the standard input of the system, the second is of size equal to the size of the continuous state x. When an event arrives at the unique event input port of this block, the state of the system jumps to the value of the second input of the block. This system is defined by four matrices A, B, C and D which are coded into `rpar` and an initial condition x_0.

```
function [x,y,typ]=TCLSS_f(job,arg1,arg2)
x=[];y=[];typ=[]
select job
case 'plot' then
  standard_draw(arg1)
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
  dlg=x(2)(4) // symbolic expressions
  while %t do
   // Expose dialogue window
   [ok,A,B,C,D,x0,dlg]=..
     getvalue('Set system parameters',..
```

```
     ['A matrix';'B matrix';'C matrix';..
      'D matrix';'Initial state'],..
     list('mat',[-1,-1],
          'mat',['size(x1,2)','-1'],..
          'mat',['-1','size(x1,2)'],..
          'mat',[-1 -1],..
          'vec','size(x1,2)'),dlg)
    if ~ok then break,end
    // Check user answers consistency
    out=size(C,1);if out==0 then out=[],end
    in=size(B,2);if in==0 then in=[],end
    [ms,ns]=size(A)
    if ms<>ns then
      x_message('A matrix must be square')
    else
      //update block input and output
      [model,graphics,ok]=check_io(x(3),..
                  x(2),[in;ms],out,1,[])
      if ok then
        // update block's data structure
        x(2)=graphics
        x(3)=model
        x(2)(4)=dlg; //symbolic parameters
        x(3)(8)=[A(:);B(:);C(:);D(:)];//set values
        x(3)(6)=x0(:) // set new initial state
        //update input dependency
        if D<>[] then
          if norm(D,1)<>0 then
            x(3)(12)=[%t %f];
          else
            x(3)(12)=[%f %f];
          end
        else
          x(3)(12)=[%f %f];
        end
        break
      end
    end
  end
case 'define' then
  //initial values of symbolic parameters
  x0=0;A=0;B=1;C=1;D=0;
  in=1;nx=size(x0,'*');out=1;
  model=list(list('tcslti',1),[in;nx],out,1,[],x0,..
          [],[A;B;C;D],[],'c',[],[%f %f],' ',list())
  // symbolic parameters
  dlg=[strcat(sci2exp(A));
        strcat(sci2exp(B));
        strcat(sci2exp(C));
        strcat(sci2exp(D));
        strcat(sci2exp(x0))]
  // initial icon definition
  gr_i=['txt=[''Jump'';''(A,B,C,D)''];';
    'xstringb(orig(1),orig(2),txt,sz(1),sz(2),''fill'')']
  sz=[3 2] //initial block size
  x=standard_define(sz,model,dlg,gr_i)
end
```

The only hard part of defining an *Interfacing* function is the `'set'` case which handles user dialogue and determines system parameters. For example, in the case of `TCLSS_f`, the interfacing function should determine whether or not the block has a direct feedthrough term. The `define` case on the other hand is only used once when the block is first copied into a palette.

4.5 *Computational* function

The *Computational* function should evaluate 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.

4.5.1 Behavior

There are different tasks that need to be performed for the simulator by the *Computational* function:

Initialization The *Computational* function is called once right at the beginning for initialization. At this point, the continuous and discrete states and the outputs of the block can be initialized (if necessary – note that they are already initialized by the interfacing function). Other tasks can also be performed at this occasion, for example blocks that read or write data from and to files, open the corresponding files on disk, `scope` block initializes the graphic window, memory allocation can be done, etc... The inputs of the block are arbitrary at this point

Re-initialization This is another occasion to initialize the states and the outputs of the block. But this time, the inputs are available. A block can be called a number of times for re-initialization.

Outputs update The simulator is requesting the values of the outputs which means that the *Computational* function should compute them as a function of the states, the inputs and the input events (if any).

States update Upon arrival of one or more events, the simulator calls the *Computational* function which must update the states x and z of the block. The states are updated in place (this avoids useless and time consuming copies, in particular when part, or all of x or z are not to be changed).

State derivative computation During the simulation, the solver calls the *Computational* function (very often) for the value of $\dot{x}$. This means the function f in (10).

Output events timing If a block has output event ports, then upon the arrival of an event, the simulator calls the *Computational* function inquiring the timing of the outgoing events. The *Computational* function should return t_{evo} as defined in (14).

Ending Once the simulation is done, the simulator calls the *Computational* function once. 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, to free allocated memory, etc...

The simulator specifies which task should be performed through a flag (see Table 1).

Flag	Task
1	Outputs update
2	States update if event(s) received
2	State derivative computation in absence of event
3	Output events timing
4	Initialization
5	Ending
6	Re-initialization

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

4.5.2 Types

There exist various types of *Computational* functions supported by Scicos (see Table 2). *Computational* function type is given by the second field of `eqns` (see Section 4.4.2).

Function type	Scilab	Fortran	C	comments
0	yes	yes	yes	Obsolete. Calling sequence fixed
1	no	yes	yes	Varying calling sequence
2	no	no	yes	Calling sequence fixed
3	yes	no	no	Inputs/outputs are Scilab lists.

Table 2: Different types of the *Computational* functions

***Computational* function type 0** In this case, the simulator constructs a unique input vector by stacking up all the input vectors, and expects the outputs, stacked up in a unique vector. This type is supported for backward compatibility but should be avoided since it is not efficient.

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

***Computational* function type 1** The simplest way of explaining this type is by considering a special case: for a block with two input vectors and three output vectors, the *Computational* function has the following synopsis.

Fortran case

```
      subroutine myfun(flag,nevprt,t,xd,x,nx,z,nz,tvec,
     &   ntvec,rpar,nrpar,ipar,nipar,u1,nu1,u2,nu2,
     &   y1,ny1,y2,ny2,y3,ny3)
c
      double precision t,xd(*),x(*),z(*),tvec(*),rpar(*)
      double precision u1(*),u2(*),y1(*),y2(*),y3(*)
      integer flag,nevprt,nx,nz,ntvec,nrpar,ipar(*)
      integer nipar,nu1,nu2,ny1,ny2,ny3
```

See Tables 3 for the definitions of the arguments.

C case Type 1 *Computational* functions can also be written in C language, the same way. Simply, arguments must be passed as pointers; see Example on page 33.

The best way to learn how to write these functions is to examine the routines in Scilab directory "routines/Scicos". There you will find the *Computational* functions of all Scicos blocks available in various palettes, most of which are fortran type 0 and 1.

Example The following is *Computational* function associated with the `Abs` block; it is of type 1 (also 0 since the input and output are unique).

```
      subroutine absblk(flag,nevprt,t,xd,x,nx,z,nz,
     &   tvec,ntvec,rpar,nrpar,ipar,nipar,u,nu,y,ny)
c     Scicos block simulator
c     returns Absolute value of the input vector
c
      double precision t,xd(*),x(*),z(*),tvec(*),rpar(*)
      double precision u(*),y(*)
      integer flag,nevprt,nx,nz,ntvec,nrpar,ipar(*)
      integer nipar,nu,ny
c
      do 15 i=1,nu
```

I/O	Args.	description
I	flag	1, 2, 3, 4, 5, 6 indicating what function should do, see Table 1.
I	nevprt	binary coding of event port numbers receiving events, e.g. events received on ports 2 and 3 imply 011, i.e. nevprt= 6
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 if flag=3.
I	ntvec	number of event output ports
I	rpar	parameter
I	nrpar	size of rpar
I	ipar	parameter
I	nipar	size of ipar
I	ui	ith input (regular), i=1,2,..
I	nui	size of ith input
O	yj	jth output (regular), j=1,2,..
I	nyj	size of jth output

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

```
         y(i)=abs(u(i))
 15   continue
      end
```

This example is particularly simple because absblk is only called with flag equal to 1,4 and 6. That is becaue this block has no state and no output event port.

The C version of this block would be:

```
#include "<SCIDIR>/routines/machine.h"
void absblk(flag,nevprt,t,xd,x,nx,z,nz,
        tvec,ntvec,rpar,nrpar,ipar,nipar,u,nu,y,ny)
/*
Returns Absolute value of the input vector
*/
double  *t,xd[],x[],z[],tvec[],rpar[],u[],y=[] ;
integer *flag,*nevprt,*nx,*nz,*ntvec,*nrpar,ipar[],*nipar;
integer *nu,*ny ;
{
int i ;
for (i==0;i<*nu;i++)
  y[i]=abs(u[i]) ;
}
```

Example The following is the *Computational* function associated with the Jump (A,B,C,D) block. This function is clearly of type 1.

```
      subroutine tcslti(flag,nevprt,t,xd,x,nx,z,nz,tvec,
     &   ntvec,rpar,nrpar,ipar,nipar,u1,nu1,u2,nu2,y,ny)
c     continuous state space linear system with jump
c     rpar(1:nx*nx)=A
c     rpar(nx*nx+1:nx*nx+nx*nu)=B
c     rpar(nx*nx+nx*nu+1:nx*nx+nx*nu+nx*ny)=C
```

```
c     rpar(nx*nx+nx*nu+nx*ny+1:nx*nx+nx*nu+nx*ny+ny*nu)=D
c!
c
      double precision t,xd(*),x(*),z(*),tvec(*),rpar(*)
      double precision u1(*),u2(*),y(*)
      integer flag,nevprt,nx,nz,ntvec,nrpar,ipar(*)
      integer nipar,nu1,nu2,ny
c
      la=1
      lb=nx*nx+la
c
      if(flag.eq.1) then
c     y=c*x+d*u1
         ld=lc+nx*ny
         lc=lb+nx*nu1
         call dmmul(rpar(lc),ny,x,nx,y,ny,ny,nx,1)
         call dmmul1(rpar(ld),ny,u1,nu1,y,ny,ny,nu1,1)
      elseif(flag.eq.2.and.nevprt.eq.1) then
c     x+=u2
         call dcopy(nx,u2,1,x,1)
      elseif(flag.eq.2.and.nevprt.eq.0) then
c     xd=a*x+b*u1
         call dmmul(rpar(la),nx,x,nx,xd,nx,nx,nx,1)
         call dmmul1(rpar(lb),nx,u1,nu1,xd,nx,nx,nu1,1)
      endif
      return
      end
```

***Computational* function type 2** This *Computational* function type is specific to C code. The synopsis is

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

integer *flag,*nevprt,*nx,*nz,*ntvec,*nrpar;
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.

Example The following is the *Computational* function associated with the `Selector` block. It is assumed here that at most one event can arrive on input event ports of this block, at a time.

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

integer *flag,*nevprt,*nx,*nz,*ntvec,*nrpar;
integer ipar[],*nipar,insz[],*nin,outsz[],*nout;

double x[],xd[],z[],tvec[],rpar[];
double *inptr[],*outptr[],*t;
{
    int k;
    double *y;
```

I/O	Args.	description
I	*flag	indicates what the function should do.
I	*nevprt	binary coding of event port numbers receiving events
I	*t	time
O	xd	derivative of the continuous state if flag=2 and nevprt= 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 if flag=3.
I	*ntvec	number of event 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 regular input
I	insz	insz[i] is the size of the ith regular input
I	*nin	number of regular input ports
I	outptr	outptr[j] is pointer to beginning of jth regular output
I	outsz	outsz[j] is the size of the jth regular output
I	*nout	number of regular output ports

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

```
    double *u;
    int nev,ic;
    ic=z[0];
    if ((*flag)==2) {
    /* store index of input event port fired */
        ic=-1;
        nev=*nevprt;
        while (nev>=1) {
            ic=ic+1;
            nev=nev/2;
        }
        z[0]=ic;
    }
    else {
    /* copy selected input on the output */
        if (*nin>1) {
            y=(double *)outptr[0];
            u=(double *)inptr[ic];
            for (k=0;k<outsz[0];k++)
                *(y++)=*(u++);
        }
        else {
            y=(double *)outptr[ic];
            u=(double *)inptr[0];
            for (k=0;k<outsz[0];k++)
                *(y++)=*(u++);
        }
    }
}
```

***Computational* function type 3** This *Computational* function type is specific to Scilab code. 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.

I/O	Args.	description
I	`flag`	indicating what the function should do (scalar)
I	`nevprt`	binary coding of event port numbers receiving events (scalar)
I	`t`	time (scalar)
I	`x`	continuous state (vector)
I	`z`	discrete state (vector)
I	`rpar`	parameter (any type of scilab variable)
I	`ipar`	parameter (vector)
I	`u`	`u(i)` is the vector of ith regular input (list)
O	`y`	`y(j)` is the vector of j-th regular output (list)
O	`x`	new x if `flag`=2 and `nevprt`> 0, or `flag`=4,5 or 6
O	`z`	new z if `flag`=2 and `nevprt`> 0, or `flag`=4,5 or 6
O	`xd`	derivative of x if `flag`=2 and `nevprt`$= 0$, [ ] 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.

There is no predefined computational function written in Scilab code (for efficiency, all blocks are written in C and Fortran). The examples below are just toy codes.

Example The following is the *Computational* function associated with a block that displays in Scilab window, every time it receives an event, the number of events it has received up to the current tie 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
    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)
```

5 Evaluation, compilation and simulation

5.1 Evaluation

We have seen in Section 2.3 that block parameters in Scicos can be defined symbolically. The real utility of this symbolic capability is that we can use Scilab instructions for evaluating Scicos block parameters without having to modify the Scicos diagram. Let us say the variable `para` has been defined in the Scilab environment, before Scicos was invoked. In that case, we can use `para` for setting parameters of one or more blocks. When the diagram is saved, the name `para` and its current value are saved. This means that the next time we invoke Scicos, the value used for simulation is this original value even if the Scilab variable `para` has a different value. To tell the block that it should adopt the new value of `para`, we should click on the block (open the dialogue) and confirm (click on `OK`). This re-evaluates the symbolic parameters of the block. To re-evaluate all the block parameters in a diagram, we can use the `Eval` button in the `Simulation` menu.

The possiblity of using symbolic expressions for defining parameters allows us to construct generic diagrams. For example a unique diagram can realize an LQG conroller feedback diagram for all linear systems; even the number of inputs and outputs of the system can be parameterized. To implement a particular LQG setup, it suffices to define, in Scilab environment, variables having the same names as those of symbolic parameters in question, invoke Scicos, load the generic diagram and then re-evaluate. An example of a generic diagram is given in Section 6.2.

Another situation where the symbolic capability is useful, is when the same parameter is used in different blocks. It would be very tedious to visit every such block to update the numerical value of the parameter. Defining symbolically such a parameter by using the same name, in all of the blocks concerned, we simply need to update the value of parameter in Scilab environment and re-evaluate the diagram.

To redefine a symbolic parameter, we can of course leave scicos (after saving the diagram), redefine the parameter (variable) in Scilab environment, invoke Scicos, reload the diagram and re-evaluate (the diagram or just the corresponding blocks). This procedure is in general not very practical. Another way to update a parameter is by clicking on `Calc` in the `Edit` menu. This stops Scicos and returns the control to Scilab. We can then redefine our Scilab variable. However, since this variable is only locally defined, we need to send it up to Scicos environment. This can be done by using the `resume` command. For example to assign the value of 3 to parameter `para`, you should use the following instruction in Scilab:

```
-1-> para = resume(3)
```

To assign values to more than one parameter, say to assign 3 and 4 to `para1` and `para2`, the instruction is

```
-1-> [para1,para2] = resume(3,4)
```

The most versatile way to define symbolic parameters is by using `context` (in the `Edit` menu). `context` is a set of Scilab instructions associated to a diagram which can be used to define symbolic parameters. By clicking on `context`, we are presented with a Dialogue Panel in which we can write, in Scilab language (except for comments which are not allowed), instructions which are evaluated once we confirm (click on `OK`) and everytime the diagram is reloaded. The advantages of this mehtod are that we do not need to leave the Scicos environment to change numerical values of symbolic parameters, and more importantly, the instructions that evaluate them are part of diagram's data structure so they are saved along with the diagram.

Note that, each Super Block has a `context` of its own. When a block is re-evaluated, it looks first in the `context` of its diagram. If it does not find its symbolic parameters, it looks up at the `context` of the diagram that contains the Super Block defined by its diagram (if any), and so on.

5.2 Compilation

When the diagram is completed, the graphical information describing the diagram is first converted into a compact data structure using the function `c_pass1`. 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`.

The result of the compilation contains essentially information concerning the blocks and the order in which these blocks should be called in 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 2, then block j and l with flag 1 and finally blocks i, j, l with flag 3. The order only matters for the case of flag 1. This information is stored in "scheduling tables". Similarly, the list of blocks that should be called during "continuous operation", i.e., between two events, are also computed and put into scheduling tables.

5.2.1 Scheduling tables

There are a number of scheduling tables constructed by the compiler. A brief desciption can be found in table 6.

Table's name	flag	Short description
cord	1	list of blocks whose outputs evolve continuously
oord	1	subset of cord whose outputs affect computation of continuous state derivatives
zord	1	subset of cord whose outputs affect computation of zero-crossing surfaces
ordclk	1	list of blocks whose outputs may change when an event is fired
execlk	2,3	list of blocks and corresponding input event coding whose discrete state or output event times need to be updated when an event is fired
exexd	2	list of blocks having continuous states
zcro	-	list of zero-crossing blocks

Table 6: Scheduling tables generated by the compiler

Scheduling tables associated with flag 1 are ordered lists, i.e., blocks called with flag 1 must be called in the specific order indicated in the Table. But Scheduling tables associated with flags 2 and 3 are not ordered.

Table **ordclk** is a vector of integers; it contains the list of blocks which must be called with flag 1 when events are fired. In particular it contains the list of blocks which must be called with flag 1 when event number 1 (i.e., event associated with the first output event port of the first block containing an event output port) is fired, followed by the list of blocks which must be called with flag 1 when event number 2 is fired, and so on.

Table **execlk** is a two-column matrix of integers; the first column contains the list of blocks which must be called with flag 2 when events are fired coded as in the case of **ordclk.** The second column of **execlk** contains the corresponding binary coding of events received on the input event ports of the corresponding block.

A two-column matrix (**ordptr**) is used to indicate which section of **ordclk** and **execlk** correspond to which event: the first column of **ordptr** is a pointer to **ordclk** and the second is a pointer to **execlk**. See Figure 26.

The information concerning which output event port of which block corresponds to which event number is stored in vector **clkptr**. In particular, the event number associated with event output port **j** of block **i** is **clkptr(i)+j-1**.

Finally, **exexd** is a vector of integers containing block numbers of blocks with continuous states. In the current version, the blocks are organized such that these blocks are placed at the beginning so **exexd** is simply **[1:nxblk]** where **nxblk** is the number of such blocks. Similarly, the zero-crossing blocks are placed at the end and thus **zcro** is **[nblk-ndcblk+1:nblk]** where **ndcblk** is the number of zero-crossing blocks and **nblk** the total number of blocks.

5.2.2 Memory management

Scicos compiler (function `c_pass2`) also initializes the memory used during simulation and sets up the appropriate pointers to it. This memory contains blocks' states and links' registers. The compiler stacks up all the continuous states of all the blocks into one vector **x** and similarly the discrete states are stored in **z.** Pointer vectors **xptr** and **zptr** are used to specify respectively which parts of **x** and **z** belong to which block. For example, the continuous state of the **i**-th block can be found at **x([xptr(i):xptr(i+1)-1])**.

The situation for link registers is more complicated because each link has one register but can be the input and output of a number of blocks (because of splits). The links are numbered and their registers are stacked up into one vector (**outtb**). Accessing these registers is done through two levels of indirection as indicated in Figure 27. Note that the tables in this figure contain the necessary information to reconstruct the complete connection diagram of regular paths.

5.2.3 Agenda

The simulator uses an agenda to schedule events. This agenda is composed of two vectors. The first one (**evtspt**) is a vector of integers; if event **i** is the next event scheduled to be fired, **evtspt(i)** contains the number of next scheduled event, or zero if no other event is scheduled. The time of the next event is stored in the second vector, **tevts(i)**. Note that **evtspt** is a self pointing pointer vector. See Figure 28. The compiler sets up the agenda and initializes it using the `init firing` vectors of different blocks.

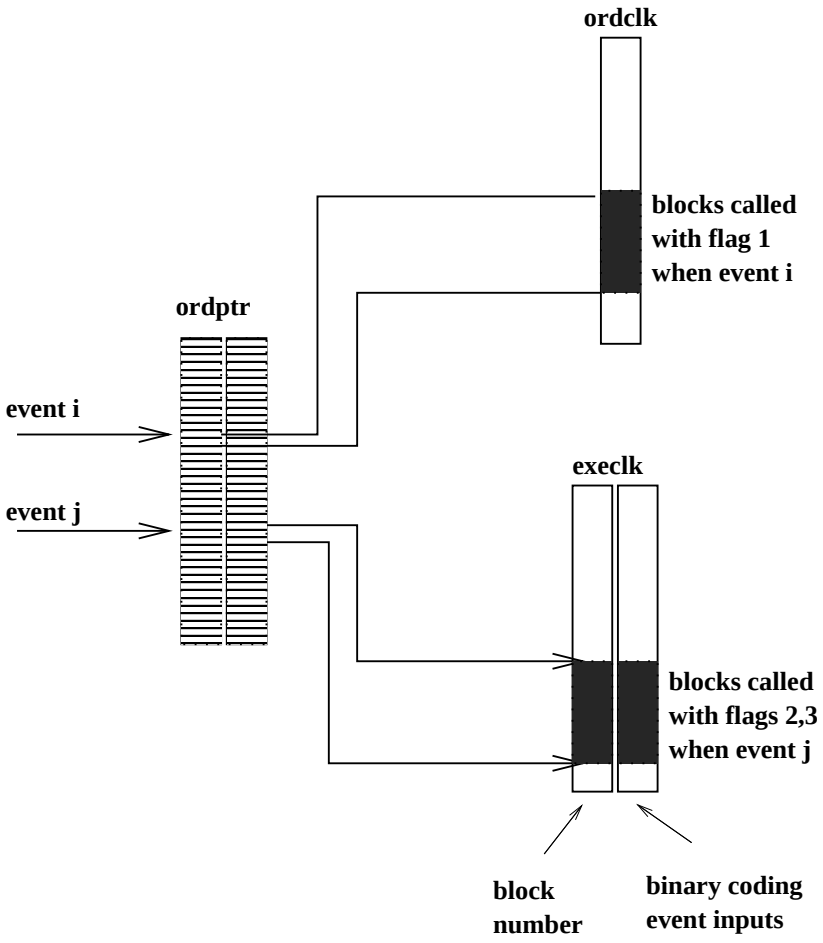


Figure 26: Graphical representation of **execlk, ordclk** and **ordptr**.

5.2.4 Compilation result

The result of the compilation is stored in a list

```
cpr=list(state,sim,cor,corinv)
```

where `state` is a list containing block **states**, **agenda** and **outtb**, i.e. things that evolve in time. `sim` is a list containing pointers (**xptr**, **zptr**, **ipptr**, **inplnk**, **lnkptr**,..), scheduling tables (**cord**, **oord**, **execlk**,..), parameters, function names and their types. `sim` remains constant during simulation, so do `cor` and `corinv` which are lists that allow making the correspondance between block numbers in `c_pass2` and block numbers in the Scicos original diagram.

5.3 Simulation

The function `scicosim` is Scicos's main simulation function for compiled Scicos diagrams (see Appendix B for details). This function uses the scheduling tables generated by the compiler.

The simulation starts off with an initialization phase depicted in Figure 29. At this stage, using a fixed-point iteration scheme, the initial states and initial values of the link registers are computed.

In the simulation phase, the simulator keeps an agenda of all scheduled events. When it is time for an event to be fired, the simulator uses the scheduling table corresponding to this particular event and calls the *Computational* functions of corresponding blocks, first to update their states, then to update their outputs (link registers), and finally to schedule their output events. At this point, the simulator removes the event just fired from the agenda and looks for the next scheduled event.

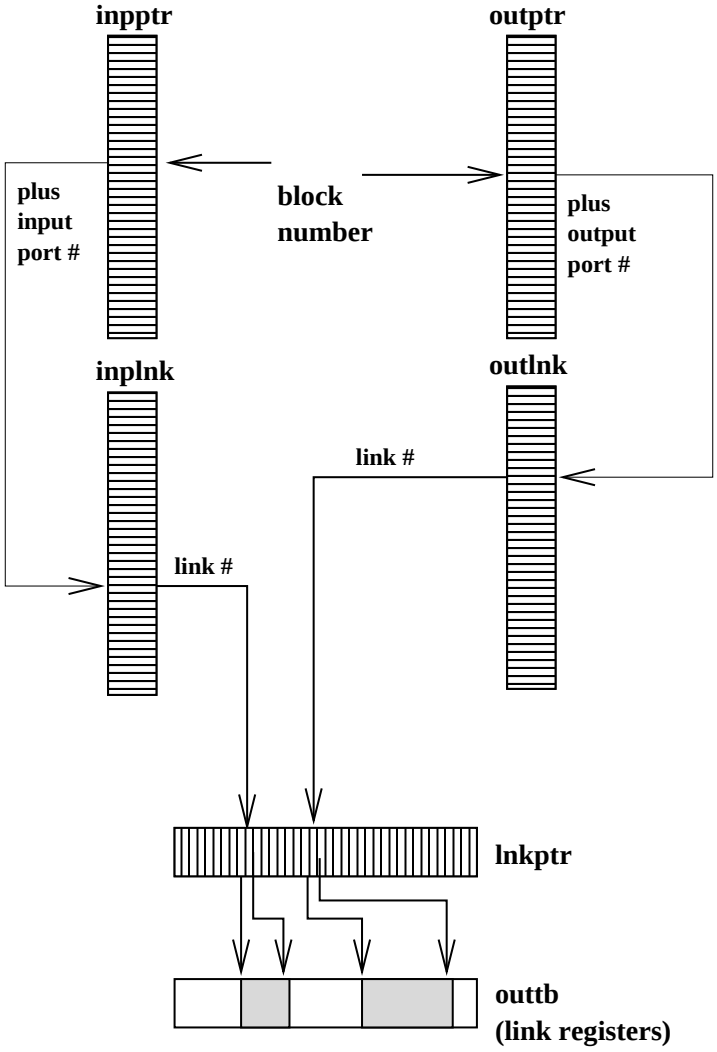


Figure 27: Memory management of link registers. The link number of the link connected to input **i** of block **j** is **l=inplnk(inpptr(j)+i-1)**. Similarly, that of the link connected to output **i** of block **j** is **l=outlnk(outptr(j)+i-1)**. The memory allocated to link l is **outtb([lnkptr(l):lnkptr(l+1)-1])**.

If the next event is scheduled at current time, the simulator fires it. If the next event is scheduled at some future date, the simulator calls the differential equation solver `lsodar` [5, 6] to advance the time, up to this future date. During this period, the outputs of all the discrete blocks are held constant.

The solver either succeeds in going all way to this future date, or it stops if it detects a zero-crossing. At this point the continuous states of all CBB's are updated and the simulator, using precomputed tables, updates the link registers which are affected by these changes.

If the solver stops because of a zero-crossing, it obtains the time of the events that may need to be scheduled by calling the corresponding zero-crossing block(s). These events are then scheduled in the agenda.

At this point the simulator checks to make sure the final time has not been reached and that a pause has not been requested by the user. In that case it starts over again with the next scheduled event. See Figures 30 and 31.

When time t reaches the final simulation time, the ending phase is executed and all the blocks are called with flag 5 (Figure 32).

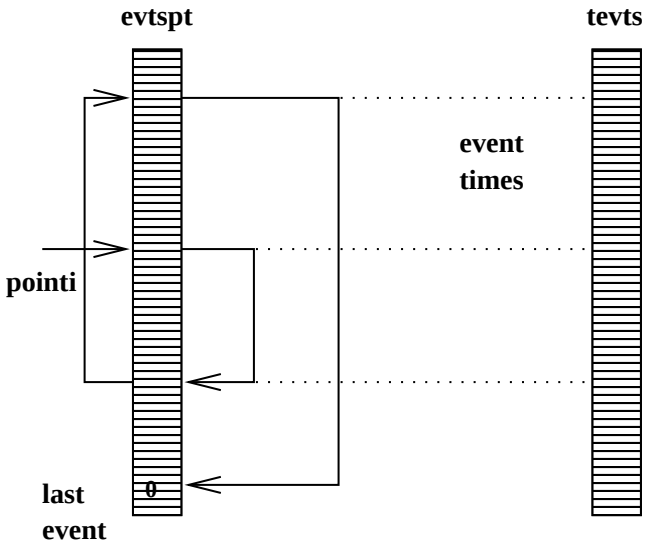


Figure 28: Agenda is composed to two vectors. The number of next event is stored in **pointi**.

6 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.

6.1 Simple examples

Figures 33 through 36 illustrate some of the simple examples provided in the demos.

6.2 An example using "context"

To define a generic diagram in Scicos, symbolic parameters should be defined in the "context". For a generic System-Observer diagram for example, we may have a "context" as in Figure 38, where `A`, `B` and `C` are system parameters, `dt` is the sampling period, `x0` is the initial condition of the system and `K` the observer gain matrix.

The corresponding Scicos diagram is given in Figure 39; the observed system is the Super Block in Figure 40) and the hybrid observer is the Super Block in Figure 41).

The `A` and `B` matrices are used both in the definition of the linear system and the `Jump linear system` used in the two Super Blocks; the difference is that in the case of the system, the initial condition `x0` is used (Figure 42) whereas in the case of observer, the initial condition is set to zero. The `C` and `K` matrices are used in the `Gain` blocks. In particular, the observer gain is given in Figure 43. The `dt` parameter is used in the `Clock`. The result of the simulation (which gives the observation error of all the components of the state) is given in Figure 44.

Note that in this diagram, not only the system matrices, but also their dimensions are arbitrary. You can for example change the `C` matrix so that the number of outputs of the system changes.

This example is part of Scicos demos. You can experiment by changing the context (don't forget to `Eval` for your modifications to be effective).

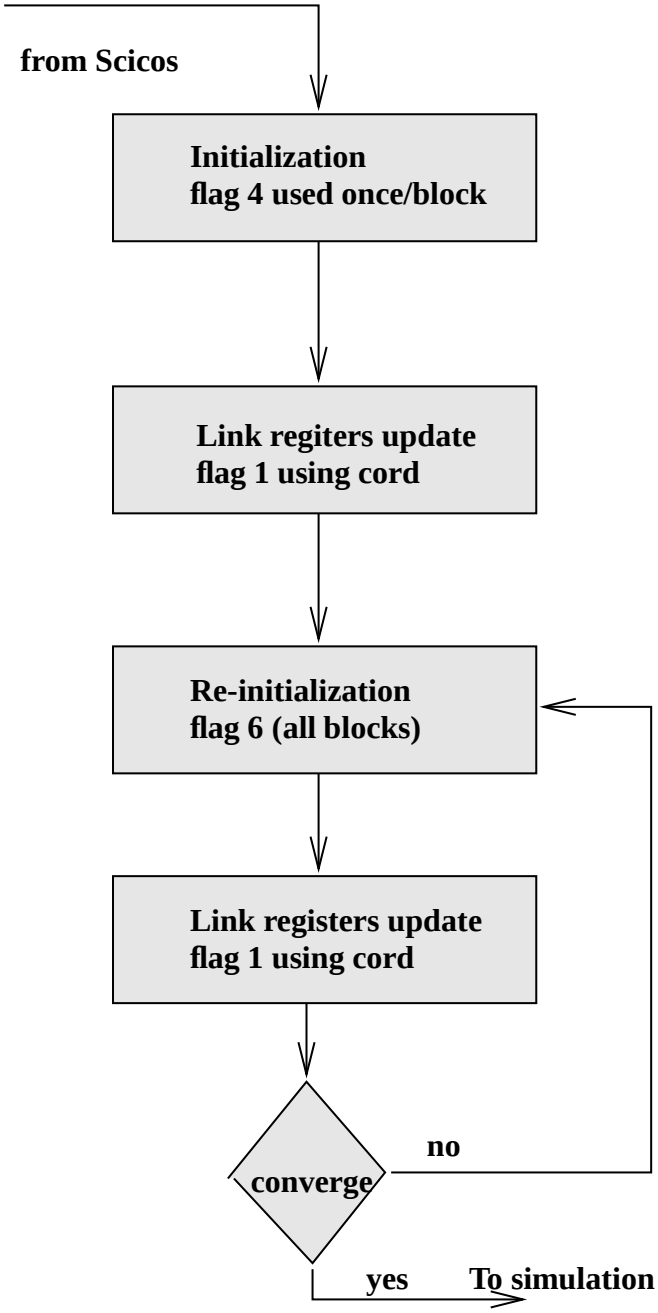


Figure 29: Initialization phase.

7 Future developments

The development of Scicos is actively pursued. There are a number of new features which should be operational in the near future. A few of these developments are presented in this chapter.

7.1 Different types of links and states

In the current version of Scicos, all the link registers are coded as “double precision” numbers. This means that for the moment, blocks inputs and outputs cannot be of type integer, boolean, etc... In the next version of Scicos, each link will have its own type (we are considering fortran types only, but that can be changed). The link type is inherited from input and output types of the blocks connected to it. This means that in defining a block, in addition to specifying the size of

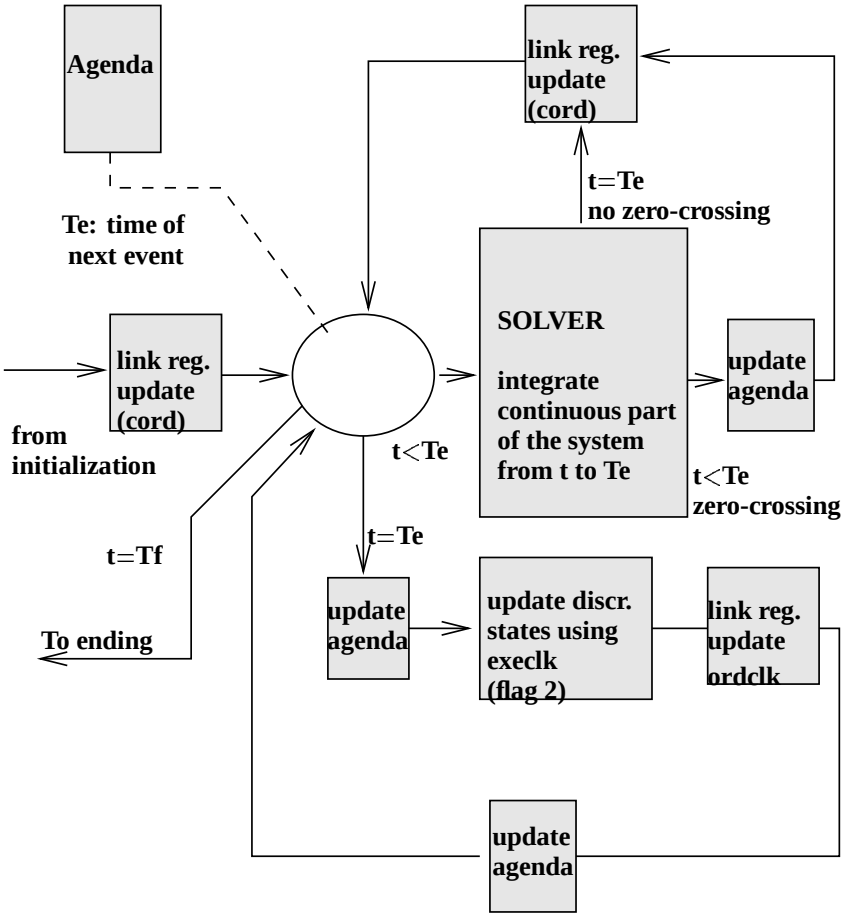


Figure 30: Simulation phase.

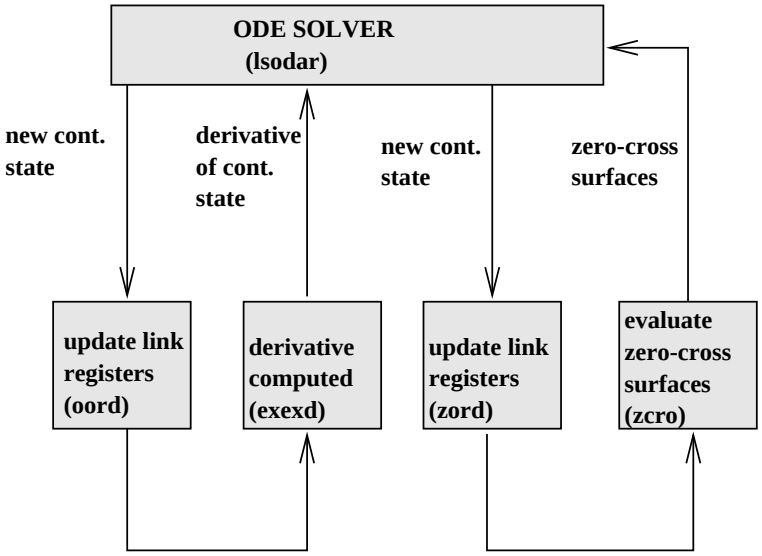


Figure 31: Continuous part evolved by the solver. Note that only relevent link registers are updated during integration.

each input and output, the type of each input and output must also be specified. The compiler of course verifies that ports of different types are not connected together.

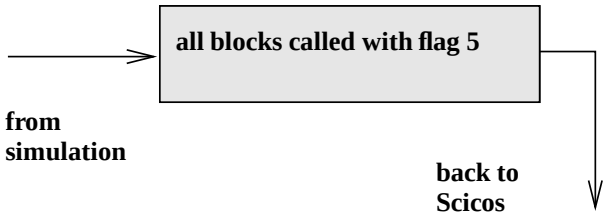


Figure 32: Ending phase.

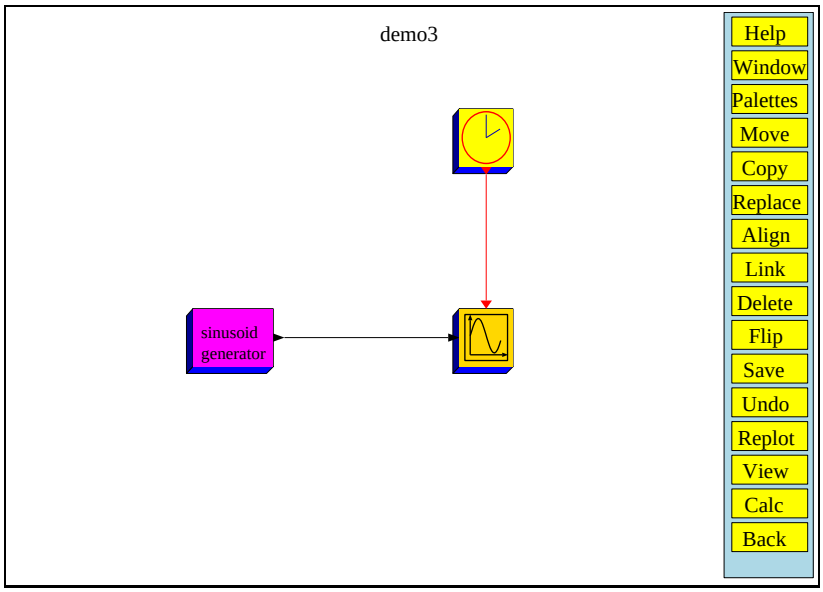


Figure 33: A simple diagram: a sine wave is generated and visualized. Note that Scope need to be driven by a clock!

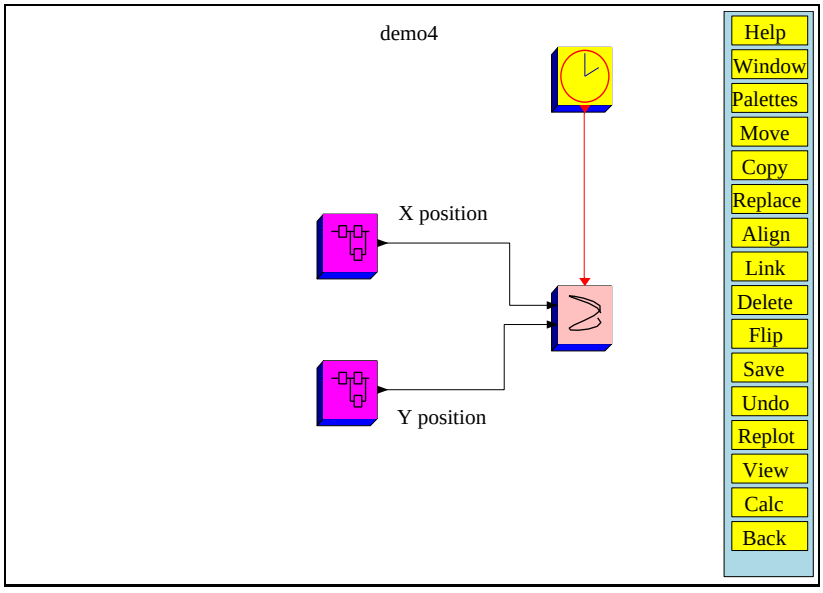


Figure 34: A ball trapped in a box bounces off the floor and the boundaries. The x and y dynamics of the ball are defined in Super Blocks

Similarly the discrete states will be allowed to be composed of subsets of different types. The continuous states however remain as they are.

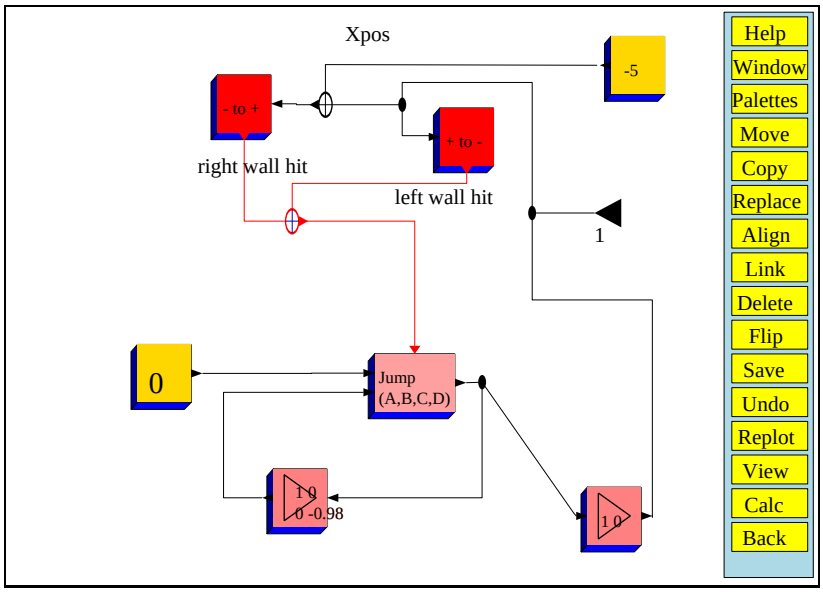


Figure 35: The `X position` Super Block of Figure 34

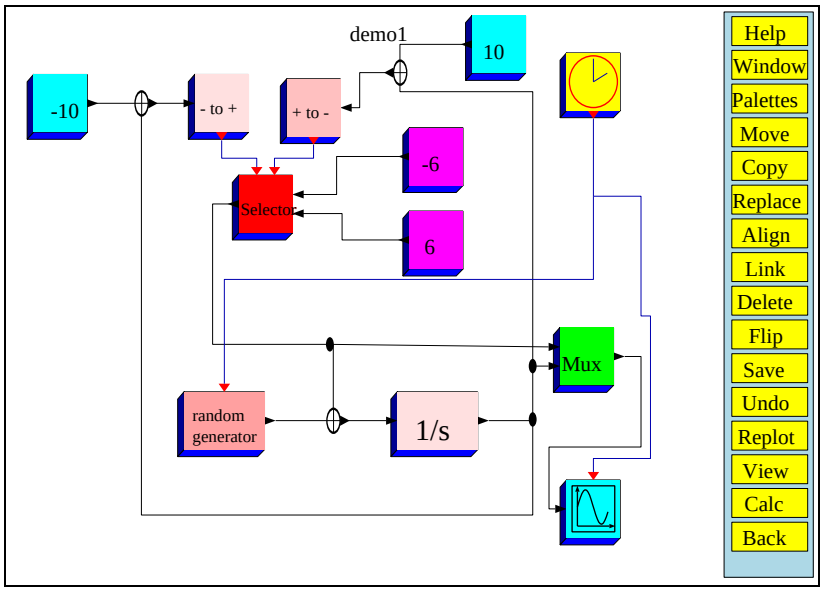


Figure 36: A thermostat controls a heater/cooler unit in face of random perturbation

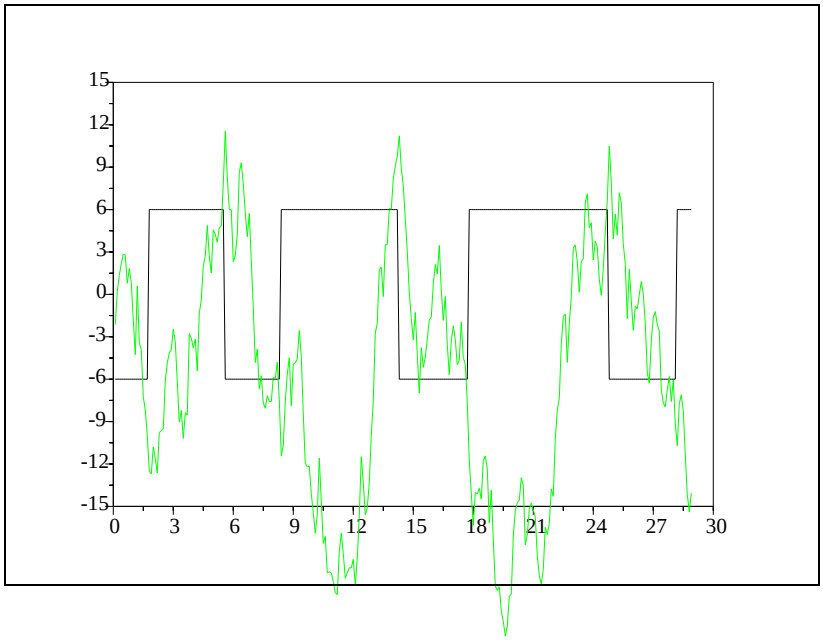


Figure 37: Simulation result corresponding to the thermostat controller in Figure 36

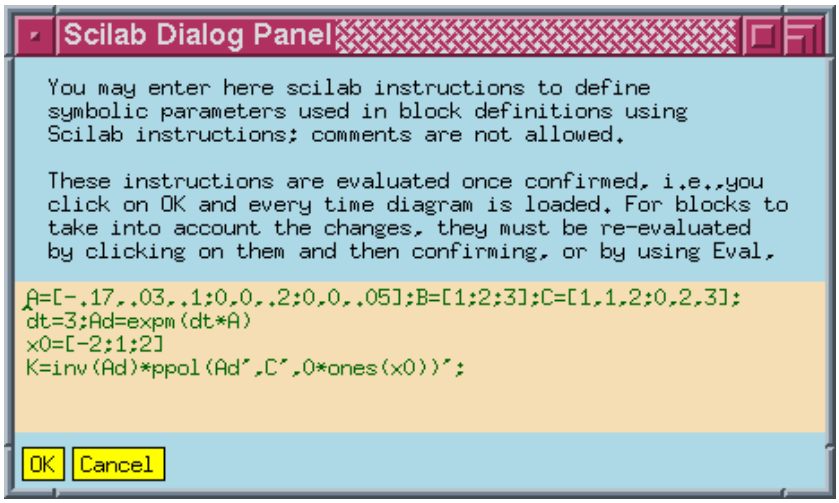


Figure 38: Context of the diagram.

7.2 Real-time code generation

To be able to generate real-time code for various processors realizing Scicos diagrams, an interface to SynDEx is currently being developed (for information on SynDEx, see [9, 10]).

The objective is to have an environment where user can select a portion of his Scicos diagram, usually coresponding to a a discrete controller, click on a button and end up in SynDEx environment loaded with the graph corresponding to this portion. From there, user can generate optimized multi-processor real-time code for almost any target processor.

7.3 Blocks imposing implicit relations

Currently, all the CBB's are explicit in a sense that they cannot impose implicit constraints among signals on different ports; there is clear distinction between inputs and outputs, even though as far as memory management is concerned, inputs and outputs are handled in a completely analogous way in Scicos. In a future version, it would be possible to define blocks imposing implicit relations (algebraic constraints). This often leads to having to solve differential-algebraic equations (DAE), for which we shall use `dassl` [7, 8] as solver.

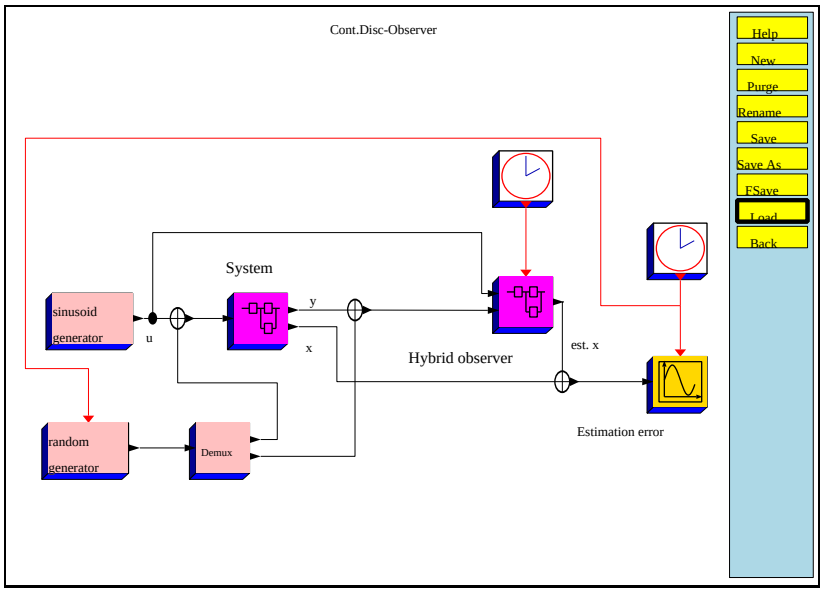


Figure 39: Linear system with a hybrid observer

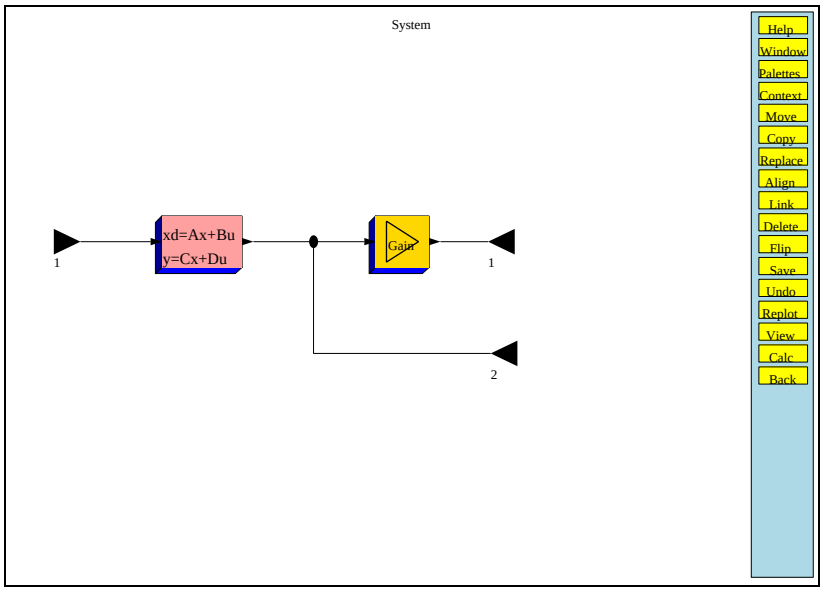


Figure 40: Model of the system. The two outputs are y and x. The gain is C.

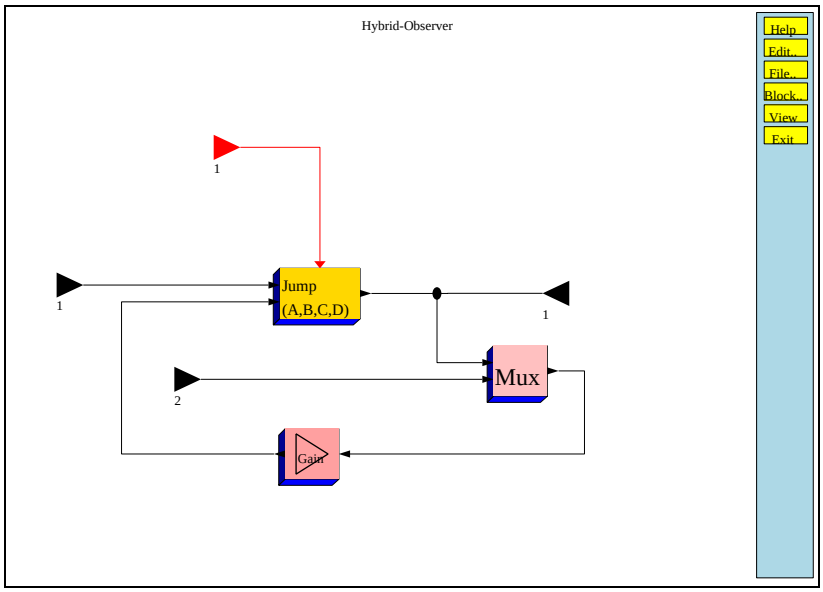


Figure 41: Model of the hybrid observer. The two inputs are `u` and `y`.

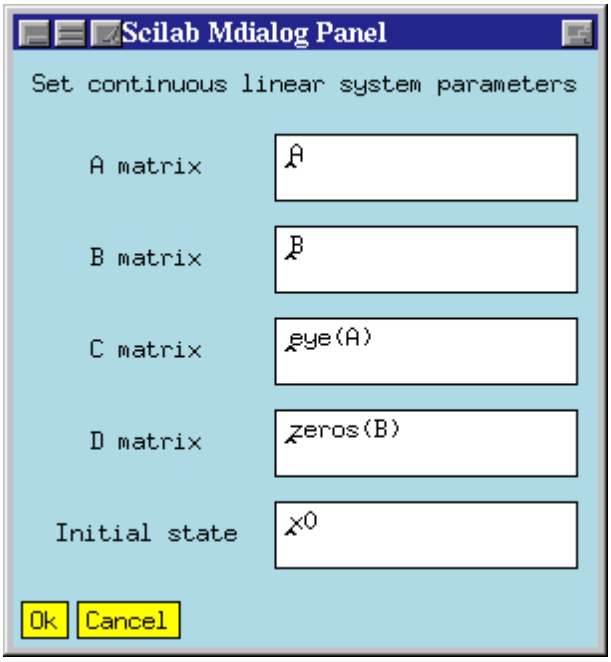


Figure 42: Linear system dialogue box

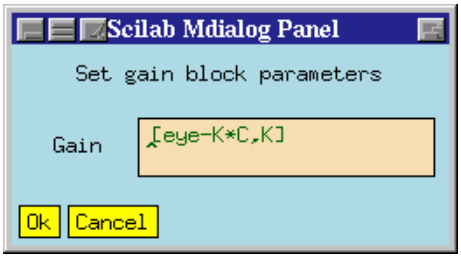


Figure 43: `Gain` block dialogue box

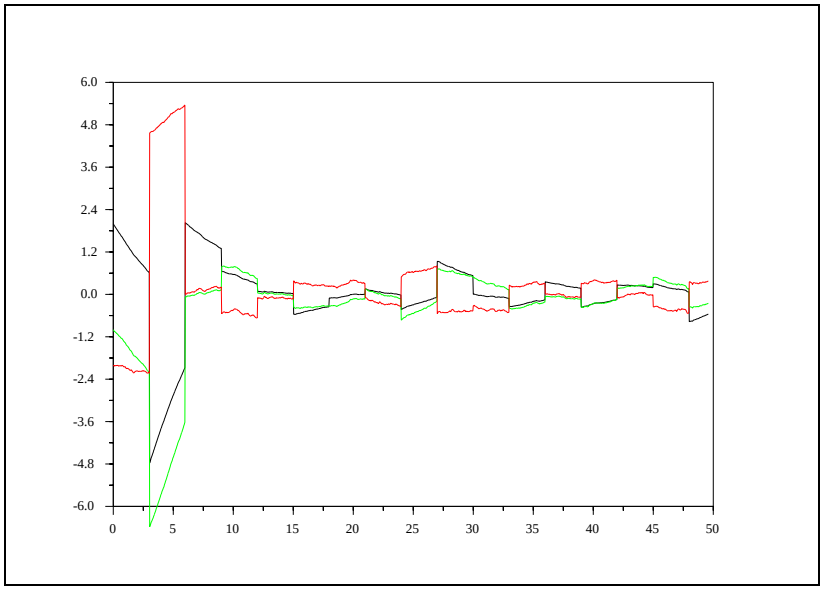


Figure 44: Simulation result.

A Using the graphical user interface

A.1 Overview

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 (this can also be done directly by using the string containing the name of the file containing the existing model, as argument of the `scicos` command) 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 by clicking on various buttons in the Scicos windows. A detailed description is given in Appendix B. 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.

A.1.1 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. See Tables 7 through 13 for a list and a short description of available blocks in different palettes.

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 or not the block is flipped).

A.1.2 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

Interfacing function	*Computational* function	Short description
AFFICH_f	affich.f	display input value in diagram
ANIMXY_f	scopxy.f	animation; inputs are x-y coord.
CLOCK_f	*super block*	generate events periodically
CONST_f	cstblk.f	constant output
CURV_f	intplt.f	signal defined by interpolation
EVENTSCOPE_f	evscpe.f	visualization of events
GENSIN_f	gensin.f	sinusoid generator
GENSQR_f	gensqr.f	square wave generator
MSCOPE_f	mscope.f	multi-display scope
RAND_f	rndblk.f	random generator
RFILE_f	readf.f	read form file
SAWTOOTH_f	sawtth.f	sawtooth generator
SCOPE_f	scope.f	scope; one (vector) input
SCOPXY_f	scopxy.f	plots second input as function of first
TIME_f	timblk.f	output is time
WFILE_f	writef.f	write to file

Table 7: Blocks in Inputs_Outputs palette

Interfacing function	*Computational* function	Short description
CLR_f	csslti.f	cont. lin. sys. (transfer function)
CLSS_f	csslti.f	cont. lin. sys. (state space)
DELAYV_f	delayv.f	variable delay
DELAY_f	*super block*	fixed delay
DLR_f	dsslti.f	discr. lin. sys. (transfer function)
DLSS_f	dsslti.f	discr. lin. sys. (state space)
DOLLAR_f	dollar.f	single scalar register
GAIN_f	gain.f	matrix gain
INTEGRAL_f	integr.f	scalar integrator
REGISTER_f	delay.f	shift register
SAMPLEHOLD_f	samphold.f	sample and hold block
SOM_f	sum2 *and* sum3	addition
TCLSS_f	tcslti.f	cont. lin. sys. with jump

Table 8: Blocks in Linear palette

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 ports may depend on block parameters. In this case, user should adjust block parameters before connecting its ports (it is not possible to remove connected 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).

A.1.3 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 clicking left on the object to be removed (or right to delete a region). 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 help on the `Undo` button.

Interfacing function	*Computational* function	Short description
ABSBLK_f	absblk.f	abs value of vector
COSBLK_f	cosblk.f	cosine operation
DLRADAPT_f	dlradp.f	
EXPBLK_f	expblk.f	exponentiation
INVBLK_f	invblk.f	inversion
LOGBLK_f	logblk.f	logarithm
LOOKUP_f	lookup.f	lookup table
MAX_f	maxblk.f	maximum of inputs
MIN_f	minblk.f	minimum of inputs
POWBLK_f	powblk.f	computes to the power of
PROD_f	prod.f	multiplication
QUANT_f	qzrnd.f	
SAT_f	lusat.f	saturation
SINBLK_f	sinblk.f	sine operation
TANBLK_f	tanblk.f	tangent operation

Table 9: Blocks in Non Linear palette

Interfacing function	*Computational* function	Short description
ANDLOG_f	andlog.f	
EVTGEN_f	*none*	generates event at specified time
EVTDLY_f	evtdly.f	delay on event
HALT_f	hltblk.f	stop simulation if event received
MCLOCK_f	*super block*	multi frequency clock
MFCLCK_f	mfclck	
TRASH_f	trash.f	does nothing

Table 10: Blocks in Events palette

Interfacing function	*Computational* function	Short description
GENERAL_f	zcross.f	detects conditional zero crossing
NEGTOPOS_f	zcross.f	zero crossing with positive slope
POSTONEG_f	zcross.f	zero crossing with negative slope
ZCROSS_f	zcross.f	detects zero crossing

Table 11: Blocks in Treshold palette

Interfacing function	*Computational* function	Short description
scifunc_block	*generated*	used to define blocks in Scilab
generic_block	*user supplied*	generic *Interfacing* function
SUPER_f	*super block*	
TEXT_f	text.f	used to include text in diagram

Table 12: Blocks in Others palette

A.1.4 Save model and simulate

Once the Scicos model is constructed, it can be saved, compiled and simulated. Normally a finished model should not contain any unconnected input ports (if some regular input ports are left unconnected, the corresponding blocks are deactivated). To simulate the model, user should click on the `Run` button in the `Simulation..` menu. Simulation parameters can be set before 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

Interfacing function	*Computational* function	Short description
DEMUX_f	demux.f	one vector input demultiplexed
ESELECT_f	eselect.f	input event directed to one output
IFTHEL_f	ifthel.f	
MUX_f	mux.f	inputs multiplexed to one output
SELECT_f	selector.c	one of regular inputs gets out

Table 13: Blocks in Branching palettes

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.

A.1.5 Editing palettes

Scicos provides a number of predefined palettes (see Section A.1.1). The Palette editor can be used to create and modify new palettes. To use it, user should click on the `Pal editor..` button of the main Scicos menu. A new window appears with a menu similar to Scicos' main menu. This is the Palette editor.

Creating and modifying a palette To modify an existing palette, user should load it in the Palette editor window by clicking first on `File..` and then `Load` (Scicos predefined palettes may be found in `<SCIDIR>/macros/scicos/*.cosf` files). It is then possible to copy blocks from palettes or from main Scicos window using `Copy` button in the `Edit..` menu, or add newly defined blocks. For that, user should click on the `AddNew` button in the `Edit..` menu (in the Palettes editor mode). A dialogue box inquires then the name of the *Interfacing* function associated with the new block.[2]

If the *Interfacing* function is not yet defined in Scilab, a file selection dialogue opens up and user is requested to select the file that contains it; in this case the corresponding `getf` instruction needed to load the function for further use is added automatically to the `.scilab` startup file in user's home directory.

To have modifications taken into account user should save the modified palette (using `Save` if it has the desired name, or `Save As` or `FSave` buttons in the `File..` menu) before leaving the Palette edition mode (using `Exit` button).

Creating a new palette can be done exactly the same way (except that there is no palette to load initially). At the end, Scicos attempts to customize user's `.scilab` startup file by adding the path and the name of the new palette and if necessary the instructions necessary to get block *Interfacing* functions (user is asked to confirm or refuse these modifications).

Converting a Super Block into a block If user wants to freeze a Super block in a single block before adding it to a palette, he should first click on it to open the Super block's diagram. Then it must click on the `Newblk` button in the `File..` menu. A dialogue box will appear asking user to set the path of the directory where he wants to place the file containing the new block's *Interfacing* function. The name of the created file is `<path><window name>.sci`. To change the window name before freezing the Super block, it is necessary to save the super block in a file using `Save As` button in the `File..` menu.

Once frozen, Super block's structure cannot be changed. Clicking on itk opens one after the other, all `set` dialogues of every block present in the Super block.

It is possible to recover the Super block by modifying (using a text editor) the generated *Interfacing* function by replacing the `'csuper'` character string by `'super'` in `model(1)` definition.

[2] Not the name of the file that contains it.

B Reference guide

C Scicos editor

C.1 scicos: Block diagram editor and GUI for the hybrid simulator scicosim

CALLING SEQUENCE :

```
sys=scicos()
sys=scicos(sys,[menus])
sys=scicos(file,[menus])
```

PARAMETERS :

`sys` : a Scicos data structure

`file` : a character string. The path of a file containing the image of a Scicos data structure. These files may have `.cos` or `.cosf` extensions.

`menus` : a vector of character strings. It allows to select some of the Scicos menus. If `menus==[]` Scicos draws the diagram and the contents of each super blocks in separate windows without menu bar. This option is useful to print diagrams.

DESCRIPTION :

Scicos is a visual editor for constructing models of hybrid dynamical systems. Invoking Scicos with no argument opens up an empty Scicos window. Models can then be assembled, loaded, saved, compiled, simulated, using GUI of Scicos. The input and ouput arguments are only useful for debugging purposes. Scicos serves as an interface to the various block diagram compilers and the hybrid simulator scicosim.

SEE ALSO : `scicosim` 77, `scicos_main` 71, `scicos_menu` 53

C.2 scicos_menu: Scicos menus description

DESCRIPTION :

Here is a list of operations available in Scicos:

`Main menu` :

- `Help` : To get help on an object or menu buttons, click first on Help button and then on the selected object or menu item.
- `Edit..` : Click on this button to open the diagram edition menu.
- `Simulate..` : Click on this button to open the compilation/execution menu.
- `File..` : Click on this button to open the file management menu.
- `Block..` : Click on this button to open the block management menu.
- `Pal editor..` : Click on this button to open the palette edition window and palette management menu.
- `View` : To shift the diagram to left, right, up or down, click first on the View button, then on a point in the diagram where you want to appear in the middle of the graphics window.
- `Exit` : Click on the Exit button to leave Scicos and return to Scilab session. Save your diagram before leaving Scicos or it will be lost.

`Diagram edition menu.` : This menu allows to edit diagram and palettes

- `Help` : To get help on an object or menu buttons, click first on Help button and then on the selected object or menu item.
- `Window` : opens up a dialogue where user may change the Scicos edition window size. Use this instead of standard window manager way.
- `Palettes` : opens up a selection dialogue where user may select a desired palette among all defined palettes.
- `Context` : opens up a dialogue where user may enter and modify Scilab instructions to be executed when diagram is loaded (`Edit../Load` menu) or evaluated (`Simulate../Eval` menu) (of course instructions are also evaluated when dialogue returns).These instructions may be used to define Scilab variables whose names are used in the block parameters definition expressions.
- `Move` : To move a block in main Scicos window, click first on the Move button, then click on the selected block, drag the mouse to the desired block position and click again to fix the position.

`Copy :` To copy a block in main Scicos window, click first on the Copy button, then click left 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 Scicos window.

To copy a region in main Scicos window, click first on the Copy button, then click right on a corner of the desired region (in Scicos window or in a palette) , drag to select the desired region, click to fix the selected region and finally click where you want the copy to be placed in Scicos window. If source diagram is a big region, selection may take a while.

`Align` : To obtain nice diagrams, you can align ports of different blocks, vertically and horizontally. Click first on the Align button, then on the first port and finally on the second port. The block corresponding to the second port is moved. Connected blocks cannot be aligned.

`AddNew` : This button is exists in Palette edition mode. To add a newly defined block to the current palette click first on this button, a dialogue box will pop up asking for the name of the GUI function associated with the block. If this function is not already loaded it is searched in the current directory. The user may then click at the desired position of the block in the palette.

`Link` : This button is defined only in diagram edition mode. To connect an output port to an input port, click first on the Link button, then on the intermediate points, if necessary, and finally on the input port. Scicos tries to draw horizontal and vertical lines to form links.

To split a link, click first on the Link button, then on the link where the split should be placed, and finally on an input port. Only one link can go from and to a port. Link color can be changed directly by clicking on the link.

`Delete` : To delete a block or a link, click first on the Delete button, then click left on the selected object. If you delete a block all links connected to it are deleted as well.

To delete a region in main Scicos window click first on the Delete button, then click right on a corner of the desired region (in Scicos window or in a palette), drag to select de desired region, click to fix the selected region. If source diagram is a big region, selection may take a while.

`Flip` : To reverse the positions of the (regular) inputs and outputs of a block placed on its sides, click on the Flip button first and then on the selected block. This does not affect the order, nor the position of the input and output event ports which are numbered from left to right. A connected block cannot be flipped.

`Save` : See `Save` button in `File..` menu below.

`Undo` : Click on the Undo 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 or palette. Replot diagram before printing or exporting Scicos diagrams.

`View` : See the description of `View` in the main menu above.

`Calc` : When you click on this button you switch Scilab to the pause mode (see the help on pause). In the Scilab main window and you may enter Scilab instructions to compute whatever you want. to go back to Scicos you need to enter ""return"" or "[...]=return(...)" Scilab instruction. ' If you use "[...]=return(...)" Scilab instruction take care not to modify Scicos variables such as "scs_m","scs_gc", "menus","datam",... ' If you have modified Scicos graphic window you may restore it using the Scicos "Replot" menu.

`Back` : go back to the main menu.

`Simulation menu` :

`Help` : See Help button above.

`Setup` : In the main Scicos window, clicking on the Setup button invokes a dialogue box that allows you to change integration parameters: absolute and relative error tolerances for the ode solver, the time tolerance (the smallest time interval for which the ode solver is used to update continuous states), and the maximum time increase realized by a single call to the ode solver.

`Compile` : 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 modifications are not correctly updated and a manual compilation may be needed before a Restart or a Continue. Click on this button to compile the block diagram. Please report if you encounter such a case.

`Context` : See `Context` definition below.

`Eval` : blocks dialogues answers can be defined using Scilab expressions. These expressions are evaluated immediately and they are also stored as character strings. Click on the Eval button to have them re-evaluated according to the new values of underlying Scilab variables defined by context for example.

`Run` : To start the simulation. If the system has already been simulated, a dialogue 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 or restart the simulation with the new values.

`Save` : See `Save` button in `File..` menu below.

`Calc` : See `Calc` button in `Edit..` menu above.

`Back` : go back to the main menu.

`File menu` :

`Help` : See `Help` button in main menu above.

`New` : Clicking on the New button creates an empty diagram in the main Scicos window. If the previous content of the window is not saved, it will be lost.

`Purge` : Suppress deleted blocks out of Scicos data structure. This menu changes block indexing and implies compilation of the diagram before compilation.

`Rename` : Click on this button to change the diagram or palette's name. A dialogue window will pop up.

`Newblk` : Click on this button to save the Super Block as a new Scicos block. A Scilab function is generated and saved in <window_name>.sci file in the desired directory. <window_name> is the name of the Super Block appearing on top of the window. A dialogue allows choosing the directory. This block may be added to a palette using `Pal editor../Edit/AddNew` menu.

`Save` : Saves the block diagram in the current binary file selected by a previous call to `SaveAs` or `Load` button. If no current binary file, diagram is saved in the current directory as <`window_name`>`.cos`.

`Save As` : Saves the block diagram in a binary file. A file selection dialogue will pop up.

`FSave` : Save the diagram in a formatted ascii file. A dialogue box allows choosing the file name which must have a .cosf extension.

Formatted save is slower than regular save but has the advantage that the generated file is system independent (usefull for exchanging data on different computers.

`Load :` Loads an ascii or binary file containing a saved block diagram. A file selection dialogue will pop up.

`Back` : Go back to the main menu.

`Block menu` :

`Help` : See `Help` button in main menu above.

`Resize` : To change the size of a block , click first on this button, click next on the desired block. A dialogue appears that allows you to change the width and/or height of the block shape.

`Icon` : To change the icon of a block drawn by `standard_draw`, click first on this button, click next on the desired block. A dialogue appears that allows you to enter Scilab instructions used to draw the icon. These instructions may refer to `orig` and `sz` variables and more generaly to the block data structure named `o` in this context (see `scicos_block`). If Icon description selects colors for drawing, it is necessary to get it through `scs_color` function to have `Color` button work properly.

`Color` : To change the background of a block drawn by `standard_draw`, click first on this button, click next on the selected block. A color palette appears where user may select the block background color.

`Label` : To change or define the blocks label, click first on this button, click next on the desired block. A dialogue appears that allows you to enter the desired label. Labels may be used within blocks computational functions as an identification (see `getlabel` function).

`Back` : Go back to the main menu.

`Pal editor menu` : The `Pal editor..` menu is similar to the main Scicos menu. Clicking on this button opens a new Scicos window in which palettes can be edited. The `Edit..`, `File..`, `Block..`, `View`, and `Exit` menus can be used to create, save, load and modify palettes. Palettes have same data structure as diagrams and menus act almost exactly the same way. The only differences are:

`Addnew` : this button in `Edit..` menu allows user to add a new block in a palette. A dialogue box will appear where user may input the name of the interfacing function. If the required interfacing function is not already present in Scilab environment, Scicos opens up a file selection window to get the file containing the interfacing function.

`Save and SaveAs` : When a palette is saved, if necessary, Scicos proposes to add lines to the file `~/.scilab` which is the user Scilab startup file.

SEE ALSO : `scicos` 53

D Blocks

D.1 ABSBLK_f: Scicos abs block

DIALOGUE PARAMETERS :
None.

DESCRIPTION :
This block realizes element-wise vector absolute value operation. This block has a single input and a single output port. Port dimension is determined by the context.

D.2 AFFICH_f: Scicos numerical display

DIALOGUE PARAMETERS :

`font` : integer, the selected font number (see xset)
`fontsize` : integer, the selected font size (set xset)
`color` : integer, the selected color for the text (see xset)
`Total numer of digits` : an integer greater than 3, the maximum number of digits used to represent the number (sign, integer part and rational part)
`rational part number of digits` : an integer greater than or equal 0, the number of digits used to represent the rational part

DESCRIPTION :
This block displays the value of its unique input inside the block (in the diagram) during simulation. The block must be located in the main Scicos window.

Warning: each time the block is moved user must click on it to set its parameters. The display position is then automatically updated.

SEE ALSO : `SCOPE_f` 67

D.3 ANDLOG_f: Scicos logical AND block

DIALOGUE PARAMETERS :
None.

DESCRIPTION :
This block, with two event inputs and a regular output, outputs +1 or -1 on its regular ouput depending on input events.

`+1` : When events are synchronously present on both event input ports
`-1` : When only one event is present.

SEE ALSO : `IFTHEL_f` 63

D.4 ANIMXY_f: Scicos 2D animated visualization block

DESCRIPTION :
This block realizes the visualization of the evolution of the two regular input signals by drawing the second input as a function of the first at instants of events on the event input port.

DIALOGUE PARAMETERS :

`Curve colors` : an integer. It is the color number (>=0) or marker type (<0) used to draw the evolution of the input port signal. See `xset()` for color (dash type) definitions.
`Line or mark size` : an integer.
`Output window number` : The number of graphic window used for the display. It is often good to use high values to avoid conflict with palettes and Super Block windows. If you have more than one scope, make sure they don't have the same window numbers (unless superposition of the curves is desired).
`Output window position` : a 2 vector specifying the coordinates of the upper left corner of the graphic window. Answer [] for default window position.
`Output window size` : a 2 vector specifying the width and height of the graphic window. Answer [] for default window dimensions.

`Xmin, Xmax` : Minimum and maximum values of the first input; used to set up the X-axis of the plot in the graphics window.

`Ymin, Ymax` : Minimum and maximum values of the second input; used to set up the Y-axis of the plot in the graphics window.

`Buffer size` : an integer. In order to minimize the number of graphics outputs, data may buffered.

REMARKS :

Output window number, Output window size, Output window position are only taken into account at the initialisation time of the simulation.

SEE ALSO : `SCOPE_f` 67, `EVENTSCOPE_f` 60, `SCOPXY_f` 68

D.5 BIGSOM_f: Scicos addition block

DIALOGUE PARAMETERS :

`Input signs` : a vector `sgn` of weights (generaly +1 or -1). The number of input signs fix the number of input ports.

DESCRIPTION :

This block realize weighted sum of the input vectors. The output is vector `k`th component is the sum of the `k`th components of each input ports weighted by `sgn(k)`.

SEE ALSO : `GAIN_f` 61, `SOM_f` 69

D.6 CLINDUMMY_f: Scicos dummy continuous system with state

DESCRIPTION :

This block should be placed in any block diagram that contains a zero-crossing block but no continuous system with state. The reason for that is that it is the ode solver that find zero crossing surfaces.

SEE ALSO : `ZCROSS_f` 71

D.7 CLKIN_f: Scicos Super Block event input port

DESCRIPTION :

This block must only be used inside Scicos Super Blocks to represent an event input port.
In a Super Block, the event input ports must be numbered from 1 to the number of event input ports.

DIALOGUE PARAMETERS :

`Port number` : an integer defining the port number.

SEE ALSO : `IN_f` 63, `OUT_f` 65, `CLKOUT_f` 57

D.8 CLKOUT_f: Scicos Super Block event output port

DESCRIPTION :

This block must only be used inside Scicos Super Blocks to represent an event output port.
In a Super_Block, the event output ports must be numbered from 1 to the number of event output ports.

DIALOGUE PARAMETERS :

`Port number` : an integer giving the port number.

SEE ALSO : `IN_f` 63, `OUT_f` 65, `CLKIN_f` 57

D.9 CLKSOM_f: Scicos event addition block

DIALOGUE PARAMETERS :

None.

DESCRIPTION :

This block is an event addition block with up to three inputs. The output reproduces the events on all the input ports. Strictly speaking, CLKSOM is not a Scicos block because it is discarded at the compilation phase. The inputs and output of CLKSOM are synchronized.

D.10 CLKSPLIT_f: Scicos event split block

DIALOGUE PARAMETERS :

None.

DESCRIPTION :

This block is an event split block with an input and two outputs. The outputs reproduces the event the input port on each output ports. Strictly speaking, CLKSPLIT is not a Scicos block because it is discarded at the compilation phase. This block is automatically created when creating a new link issued from a link.

The inputs and output of CLKSPLIT are synchronized.

D.11 CLOCK_f: Scicos periodic event generator

DESCRIPTION :

This block is a Super Block constructed by feeding back the output of an event delay block into its input event port. The unique output of this block generates a regular train of events.

DIALOGUE PARAMETERS :

`Period` : scalar. One over the frequency of the clock. Period is the time that separates two output events.
`Init time` : scalar. Starting date. if negative the clock never starts.

SEE ALSO : `EVTDLY_f` 61

D.12 CLR_f: Scicos continuous-time linear system (SISO transfer function)

DIALOGUE PARAMETERS :

`Numerator` : a polynomial in s.
`Denominator` : a polynomial in s.

DESCRIPTION :

This block realizes a SISO linear system represented by its rational transfer function `Numerator/Denominator`. The rational function must be proper.

SEE ALSO : `CLSS_f` 58, `INTEGRAL_f` 63

D.13 CLSS_f: Scicos continuous-time linear state-space system

DESCRIPTION :

This block realizes a continuous-time linear state-space system.

```
xdot=A*x+B*u
y   =C*x+D*u
```

The system is defined by the (A,B,C,D) matrices and the initial state x0. The dimensions must be compatible.

DIALOGUE PARAMETERS :

`A` : square matrix. The A matrix
`B` : the `B` matrix, `[]` if system has no input
`C` : the `C` matrix , `[]` if system has no output
`D` : the `D` matrix, `[]` if system has no `D` term.
`x0` : vector. The initial state of the system.

SEE ALSO : `CLR_f` 58, `INTEGRAL_f` 63

D.14 CONST_f: Scicos constant value(s) generator

DIALOGUE PARAMETERS :

`constants` : a real vector. The vector size gives the size of the output port. The value `constants(i)` is assigned to the ith component of the output.

DESCRIPTION :

This block is a constant value(s) generator.

D.15 COSBLK_f: Scicos cosine block

DIALOGUE PARAMETERS :
None.

DESCRIPTION :
This block realizes vector cosine operation. `y(i)=cos(u(i))`. The port input and output port sizes are equal and determined by the context.

SEE ALSO : `SINBLK_f` 69, `GENSIN_f` 62

D.16 CURV_f: Scicos block, tabulated function of time

DIALOGUE PARAMETERS :
Tabulated function is entered using a graphics curve editor (see edit_curv in Scilab documentation)

DESCRIPTION :
This block defines a tabulated function of time. Between mesh points block performs a linear interpolation. Outside tabulation block outputs last tabulated value.

User may define the tabulation of the function using a curve editor.

D.17 DELAYV_f: Scicos time varying delay block

DIALOGUE PARAMETERS :

`Number inputs` : size of the delayed vector (-1 not allowed)
`Register initial state` : register initial state vector. Dimension must be greater than or equal to 2
`Max delay` : Maximum delay that can be produced by this block

DESCRIPTION :
This block implements a time varying discretized delay. The value of the delay is given by the second input port. The delayed signal enters the first input port and leaves the unique output prot.

The first event output port must be connected to unique input event port if auto clocking is desired. But the input event port can also be driven by outside clock. In that case, the max delay is size of initial condition times the period of the incoming clock.

The second output event port generates an event if the second input goes above the maximum delay specified. This signal can be ignored. In that case the output will be delayed by max delay.

SEE ALSO : `DELAY_f` 59, `EVTDLY_f` 61, `REGISTER_f` 66

D.18 DELAY_f: Scicos delay block

DIALOGUE PARAMETERS :

`Discretization time step` : positive scalar, delay discretization time step
`Register initial state` : register initial state vector. Dimension must be greater than or equal to 2

DESCRIPTION :
This block implements as a discretized delay. It is in fact a Scicos super block formed by a shift register and a clock.

value of the delay is given by the discretization time step multiplied by the number of states of the register minus one

SEE ALSO : `DELAYV_f` 59, `EVTDLY_f` 61, `REGISTER_f` 66

D.19 DEMUX_f: Scicos demultiplexer block

DIALOGUE PARAMETERS :

`number of output ports` : positive integer less than or equal to 8.

DESCRIPTION :
Given a vector valued input this block splits inputs over vector valued outputs. So `u=[y1;y2...;yn]`, where `yi` are numbered from top to bottom. Input and Output port sizes are determined by the context.

SEE ALSO : `MUX_f` 65

D.20 DLRADAPT_f: Scicos discrete-time linear adaptive system

DIALOGUE PARAMETERS :

`Vector of p mesh points` : a vector which defines `u2` mesh points. Numerator roots : a matrix, each line gives the roots of the numerator at the corresponding mesh point.
`Denominator roots` : a matrix, each line gives the roots of the denominator at the corresponding mesh point.
`gain` : a vector, each vector entry gives the transfer gain at the corresponding mesh point.
`past inputs` : a vector of initial value of past `degree(Numerator)` inputs
`past outputs` : a vector of initial value of past `degree(Denominator)` outputs

DESCRIPTION :
This block realizes a SISO linear system represented by its rational transfer function whose numerator and denominator roots are tabulated functions of the second block input. The rational function must be proper.
Roots are interpolated linearly between mesh points.

SEE ALSO : `DLSS_f` 60, `DLR_f` 60

D.21 DLR_f: Scicos discrete-time linear system (transfer function)

DIALOGUE PARAMETERS :

`Numerator` : a polynomial in z.
`Denominator` : a polynomial in z.

DESCRIPTION :
This block realizes a SISO linear system represented by its rational transfer function (in the symbolic variable z). The rational function must be proper.

SEE ALSO : `DLSS_f` 60, `DLRADAPT_f` 60

D.22 DLSS_f: Scicos discrete-time linear state-space system

DESCRIPTION :
This block realizes a discrete-time linear state-space system. The system is defined by the (A,B,C,D) matrices and the initial state x0. The dimensions must be compatible. At the arrival of an input event on the unique input event port, the state is updated.

DIALOGUE PARAMETERS :

`A` : square matrix. The A matrix
`B` : the B matrix
`C` : the C matrix
`x0` : vector. The initial state of the system.

SEE ALSO : `DLR_f` 60, `INTEGRAL_f` 63, `CLSS_f` 58, `DLSS_f` 60

D.23 EVENTSCOPE_f: Scicos event visualization block

DESCRIPTION :
This block realizes the visualization of the input event signals.

DIALOGUE PARAMETERS :

`Number of event inputs` : an integer giving the number of event input ports colors : a vector of integers. The i-th element is the color number (>=0) or dash type (<0) used to draw the evolution of the i-th input port signal. See `xset` for color (dash type) definitions.
`Output window number` : The number of graphic window used for the display. It is often good to use high values to avoid conflict with palettes and Super Block windows. If you have more than one scope, make sure they don't have the same window numbers (unless superposition of the curves is desired). Output window position : a 2 vector specifying the coordinates of the upper left corner of the graphic window. Answer [] for default window position.

`Output window size` : a 2 vector specifying the width and height of the graphic window. Answer [] for default window dimensions.
`Refresh period` : Maximum value on the X-axis (time). The plot is redrawn when time reaches a multiple of this value.

REMARKS :
Output window number, Output window size, Output window position are only taken into account at the initialisation time of the simulation.

SEE ALSO : `SCOPXY_f` 68, `SCOPE_f` 67, `ANIMXY_f` 56

D.24 EVTDLY_f: Scicos event delay block

DESCRIPTION :
One event is generated `Delay` after an event enters the unique input event port. Block may also generate an initial output event.

DIALOGUE PARAMETERS :

`Delay` : scalar. Time delay between input and output event.
`Auto-exec` : scalar. If `Auto-exec>=0` block initially generates an output event at date `Auto-exec`.

SEE ALSO : `CLOCK_f` 58

D.25 EVTGEN_f: Scicos event firing block

DESCRIPTION :
One event is generated on the unique output event port if `Event time` is larger than equal to zero, if not, no event is generated.

DIALOGUE PARAMETERS :

`Event time` : scalar. date of the initial event

SEE ALSO : `CLOCK_f` 58, `EVTDLY_f` 61

D.26 EXPBLK_f: Scicos aˆu block

DIALOGUE PARAMETERS :

`a :` real positive scalar

DESCRIPTION :
This block realizes `y(i)=aˆu(i)`. The input and output port sizes are determined by the compiler.

D.27 GAIN_f: Scicos gain block

DIALOGUE PARAMETERS :

`Gain` : a real matrix.

DESCRIPTION :
This block is a gain block. The output is the Gain times the regular input (vector). The dimensions of Gain determines the input (number of columns) and output (number of rows) port sizes.

D.28 GENERAL_f: Scicos general zero crossing detector

DESCRIPTION :
Depending on the sign (just before the crossing) of the inputs and the input numbers of the inputs that have crossed zero, an event is programmed (or not) with a given delay, for each output. The number of combinations grows so fast that this becomes unusable for blocks having more than 2 or 3 inputs. For the moment this block is not documented.

DIALOGUE PARAMETERS :

`Size of regular input` : integer.
`Number of output events` : integer.
`the routing matrix` : matrix. number of rows is the number of output events. The columns correspond to each possible combination of signs and zero crossings of the inputs. The entries of the matrix give the delay for generating the output event (<0 no event is generated).

SEE ALSO : `NEGTOPOS_f` 65, `POSTONEG_f` 65, `ZCROSS_f` 71

D.29 GENERIC_f: Scicos generic interfacing function

DESCRIPTION :
This block can realize any type of block. The computational function must already be defined in Scilab, Fortran or C code.

DIALOGUE PARAMETERS :

`simulation function` : a character string, the name of the computational function
`function type` : a non negative integer, the type of the computational function
`input port sizes` : a vector of integers, size of regular input ports.
`output port sizes` : a vector of integers, size of regular output ports.
`input event port sizes` : a vector of ones, size of event input ports. The size of the vector gives the number of event input ports.
`output event port sizes` : a vector of ones, size of event output ports. The size of the vector gives the number of of event output ports.
`Initial continuous state` : a column vector.
`Initial discrete state` : a column vector.
`System type` : a string: c,d, z or l (CBB, DBB, zero crossing or synchro).
`Real parameter vector` : column vector. Any parameters used in the block can be defined here as a column vector.
`Integer parameter vector` : column vector. Any integer parameters used in the block can be defined here as a column vector.
`initial firing` : vector. Size of this vector corresponds to the number of event outputs. The value of the i-th entry specifies the time of the preprogrammed event firing on the i-th output event port. If less than zero, no event is preprogrammed.
`direct feedthrough` : character ”y” or ”n”, specifies if block has a direct input to output feedthrough.
`Time dependance` : character ”y” or ”n”, specifies if block output depends explicitly on time.

SEE ALSO : `scifunc_block` 71

D.30 GENSIN_f: Scicos sinusoid generator

DESCRIPTION :
This block is a sine wave generator: M*sin(F*t+P)

DIALOGUE PARAMETERS :

`Magnitude` : a scalar. The magnitude M.
`Frequency` : a scalar. The frequency F.
`Phase` : a scalar. The phase P.

SEE ALSO : `GENSQR_f` 63, `RAND_f` 66, `SAWTOOTH_f` 67

D.31 GENSQR_f: Scicos square wave generator

DESCRIPTION :
This block is a square wave generator: output takes values `-M` and `M`. Every time an event is received on the input event port, the output switches from `-M` to `M`, or `M to -M`.

DIALOGUE PARAMETERS :

`Amplitude` : a scalar `M`.

SEE ALSO : `GENSIN_f` 62, `SAWTOOTH_f` 67, `RAND_f` 66

D.32 HALT_f: Scicos Stop block

DIALOGUE PARAMETERS :

`State on halt` : scalar. A value to be placed in the state of the block. For debugging purposes this allows to distinguish between different halts.

DESCRIPTION :
This block has a unique input event port. Upon the arrival of an event, the simulation is stopped and the main Scicos window is activated. Simulation can be restarted or continued (Run button).

D.33 IFTHEL_f: Scicos if then else block

DIALOGUE PARAMETERS :
None.

DESCRIPTION :
One event is generated on one of the output event ports when an input event arrives. Depending on the sign of the regular input, the event is generated on the first or second output.

This is a synchro block, i.e., input and output event are synchronized.

D.34 INTEGRAL_f: Scicos simple integrator

DESCRIPTION :
This block is an integrator. The output is the integral of the input.

DIALOGUE PARAMETERS :

`Initial state` : a scalar. The initial condition of the integrator.

SEE ALSO : `CLSS_f` 58, `CLR_f` 58

D.35 INVBLK_f: Scicos inversion block

DIALOGUE PARAMETERS :
None.

DESCRIPTION :
This block computes `y(i)=1/u(i)`. The input (output) size is determined by the context

D.36 IN_f: Scicos Super Block regular input port

DESCRIPTION :
This block must only be used inside Scicos Super Blocks to represent a regular input port. The input size is determined by the context.
In a Super Block, regular input ports must be numbered from 1 to the number of regular input ports.

DIALOGUE PARAMETERS :

`Port number` : an integer giving the port number.

SEE ALSO : `CLKIN_f` 57, `OUT_f` 65, `CLKOUT_f` 57

D.37 LOGBLK_f: Scicos logarithm block

DIALOGUE PARAMETERS :

`a : real scalar greater than 1`

DESCRIPTION :
This block realizes `y(i)=log(u(i))/log(a)`. The input and output port sizes are determined by the context.

D.38 LOOKUP_f: Scicos Lookup table with graphical editor

DESCRIPTION :
This block realizes a non-linear function defined using a graphical editor.

D.39 MAX_f: Scicos max block

DIALOGUE PARAMETERS :
None.
DESCRIPTION :
The block outputs the maximum of the input vector: `y=max(u1,...un)`. The input vector size is determined by the compiler according to the connected blocks port sizes.

SEE ALSO : `MIN_f` 64

D.40 MCLOCK_f: Scicos 2 frequency event clock

DESCRIPTION :
This block is a Super Block constructed by feeding back the outputs of an MFCLCK block into its input event port. The two outputs of this block generate regular train of events, the frequency of the first input being equal to that of the second output divided by an integer n. The two outputs are synchronized (this is impossible for standard blocks; this is a Super Block).
DIALOGUE PARAMETERS :

`Basic period` : scalar. equals 1/f, f being the highest frequency.
`n` : an intger >1. the frequency of the first output event is f/n.

SEE ALSO : `MFCLCK_f` 64, `CLOCK_f` 58

D.41 MFCLCK_f: Scicos basic block for frequency division of event clock

DESCRIPTION :
This block is used in the Super Block MCLOCK. The input event is directed once every n times to output 1 and the rest of the time to output 2. There is a delay of "Basic period" in the transmission of the event. If this period >0 then the second output is initially fired. It is not if this period=0. In the latter case, the input is driven by an event clock and in the former case, feedback can be used.
DIALOGUE PARAMETERS :

`Basic period` : positive scalar.
`n` : an integer greater than 1.

SEE ALSO : `MCLOCK_f` 64, `CLOCK_f` 58

D.42 MIN_f: Scicos min block

DIALOGUE PARAMETERS :
None.
DESCRIPTION :
The block outputs the minimum of the input vector: `y=min(u1,...un)`. The input vector size is determined by the compiler according to the connected blocks port sizes.

SEE ALSO : `MAX_f` 64

D.43 MUX_f: Scicos multiplexer block

DIALOGUE PARAMETERS :

`number of output ports` : integer greater than or equal to 1 and less than 8

DESCRIPTION :
Given `n` vector valued inputs this block merges inputs in an single output vector. So `y=[u1;u2....;un]`, where `ui` are numbered from top to bottom. Input and Output port sizes are determined by the context.

SEE ALSO : `MUX_f` 65

D.44 NEGTOPOS_f: Scicos negative to positive detector

DESCRIPTION :
An output event is generated when the unique input crosses zero with a positive slope.

SEE ALSO : `POSTONEG_f` 65, `ZCROSS_f` 71, `GENERAL_f` 62

D.45 OUT_f: Scicos Super Block regular output port

DIALOGUE PARAMETERS :

`Port number` : an integer giving the port number.

DESCRIPTION :
This block must only be used inside Scicos Super Blocks to represent a regular output port.
In a Super Block, regular output ports must be numbered from 1 to the number of regular output ports.
size of the output is determined by the compiler according to the connected blocks port sizes.

SEE ALSO : `CLKIN_f` 57, `IN_f` 63, `CLKOUT_f` 57

D.46 POSTONEG_f: Scicos positive to negative detector

DESCRIPTION :
An output event is generated when the unique input crosses zero with a negative slope.

SEE ALSO : `NEGTOPOS_f` 65, `ZCROSS_f` 71, `GENERAL_f` 62

D.47 POWBLK_f: Scicos u^a block

DIALOGUE PARAMETERS :

`a :` real scalar

DESCRIPTION :
This block realizes `y(i)=u(i)^a`. The input and output port sizes are determined by the compiler according to the connected blocks port sizes.

D.48 PROD_f: Scicos element wise product block

DESCRIPTION :
The output is the element wize product of the inputs.

D.49 QUANT_f: Scicos Quantization block

DIALOGUE PARAMETERS :

`Step` : scalar, Quantization step
`Quantization method` : scalar with possible values 1,2,3 or 4

- `1` : Round method
- `2` : Truncation method
- `3` : Floor method
- `4` : Ceil method

DESCRIPTION :
This block outputs the quantization of the input according to a choice of methods
for `Round` method
`y(i)=Step*(int(u(i)/Step+0.5)-0.5)` if `u(i)<0`.
`y(i)=Step*(int(u(i)/Step-0.5)+0.5)`. if `u(i)>=0`.
For truncation method
`y(i)=Step*(int(u(i)/Step+0.5))` if `u(i)<0`.
`y(i)=Step*(int(u(i)/Step-0.5))` if `u(i)>=0`.
For floor method
`y(i)=Step*(int(u(i)/Step+0.5))` .
For ceil method
`y(i)=Step*(int(u(i)/Step-0.5))` .

D.50 RAND_f: Scicos random wave generator

DESCRIPTION :
This block is a random wave generator: each output component takes piecewise constant random values. Every time an event is received on the input event port, the outputs take new independent random values.
output port size is given by the size of `A` and `B` vectors

DIALOGUE PARAMETERS :

`flag` : 0 or 1. 0 for uniform distribution on `[A,A+B]` and 1 for normal distribution `N(A,B*B)`.
`A` : scalar
`B` : scalar

SEE ALSO : `GENSIN_f` 62, `SAWTOOTH_f` 67, `GENSQR_f` 63

D.51 REGISTER_f: Scicos shift register block

DESCRIPTION :
This block realizes a shift register. At every input event, the register is shifted one step.

DIALOGUE PARAMETERS :

`Initial condition` : a column vector. It contains the initial state of the register.

SEE ALSO : `DELAY_f` 59, `DELAYV_f` 59, `EVTDLY_f` 61

D.52 RFILE_f: Scicos "read from file" block

DIALOGUE PARAMETERS :

`Time record Selection` : an empty matrix or a positive integer. If an integer `i` is given the `i`th element of the read record is assumed to be the date of the output event. If empty no output event exists.
`Output record selection` : a vector of positive integer. `[k1,..,kn]`,The `ki`th element of the read record gives the value of `i`th output.
`Input file name` : a character string defining the path of the file
`Input Format` : a character string defining the Fortran format to use or nothing for an unformatted (binary) write

`Buffer size` : To improve efficiency it is possible to buffer the input data. read on the file is only done after each `Buffer size` call to the block.
`size of output` : a scalar. This fixes the number of "value" read.

DESCRIPTION :
This block allows user to read datas in a file, in formatted or binary mode. `Output record selection` and `Time record Selection` allows the user to select data among file records.
Each call to the block advance one record in the file.

SEE ALSO : `WFILE_f` 71

D.53 SAMPLEHOLD_f: Scicos Sample and hold block

DIALOGUE PARAMETERS :
None.
DESCRIPTION :
Each time an input event is received block copy its input on the output and hold it until input event. For periodic Sample and hold, event input must be generated by a `Clock`.

SEE ALSO : `DELAY_f` 59, `CLOCK_f` 58

D.54 SAT_f: Scicos Saturation block

DESCRIPTION :
This block realizes the non-linear function: saturation.
DIALOGUE PARAMETERS :

`Min` : a scalar. Lower saturation bound
`Max` : a scalar. Upper saturation bound
`Slope` : a scalar. The slope of the line going through the origin and describing the behaviour of the function around zero.

SEE ALSO : `LOOKUP_f` 64

D.55 SAWTOOTH_f: Scicos sawtooth wave generator

DESCRIPTION :
This block is a sawtooth wave generator: output is (t-t_i) from ti to t_(i+1) where t_i and t_(i+1) denote the times of two successive input events.
DIALOGUE PARAMETERS :
None.

SEE ALSO : `GENSIN_f` 62, `GENSQR_f` 63, `RAND_f` 66

D.56 SCOPE_f: Scicos visualization block

DESCRIPTION :
This block realizes the visualization of the evolution of the signals on the standard input port(s) at instants of events on the event input port.
DIALOGUE PARAMETERS :

`Curve colors` : a vector of integers. The i-th element is the color number (>0) or dash type (<0) used to draw the evolution of the i-th input port signal. See `plot2d` for color (dash type) definitions.
`Output window number` : The number of graphic window used for the display. It is often good to use high values to avoid conflict with palettes and Super Block windows. If you have more than one scope, make sure they don't have the same window numbers (unless superposition of the curves is desired).
`Output window position` : a 2 vector specifying the coordinates of the upper left corner of the graphic window. Answer [] for default window position.
`Output window size` : a 2 vector specifying the width and height of the graphic window. Answer [] for default window dimensions.

`Ymin, Ymax` : Minimum and maximum values of the input; used to set up the Y-axis of the plot in the graphics window.
`Refresh period` : Maximum value on the X-axis (time). The plot is redrawn when time reaches a multiple of this value.
`Buffer size` : To improve efficiency it is possible to buffer the input data. The drawing is only done after each `Buffer size` call to the block.
`Accept herited events` : if 0 `SCOPE_f` draws a new point only when an event occurs on its event input port. if 1 `SCOPE_f` draws a new point when an event occurs on its event input port and when it's regular input changes due to an event on an other upstrem block (herited events).

REMARKS :
Output window number, Output window size, Output window position are only taken into account at the initialisation time of the simulation.

SEE ALSO : `SCOPXY_f` 68, `EVENTSCOPE_f` 60, `ANIMXY_f` 56

D.57 SCOPXY_f: Scicos visualization block

DESCRIPTION :
This block realizes the visualization of the evolution of the two regular input signals by drawing the second input as a function of the first at instants of events on the event input port.

DIALOGUE PARAMETERS :

`Curve colors` : an integer. It is the color number (>0) or dash type (<0) used to draw the evolution of the input port signal. See `plot2d` for color (dash type) definitions.
`Line or mark size` : an integer.
`Output window number` : The number of graphic window used for the display. It is often good to use high values to avoid conflict with palettes and Super Block windows. If you have more than one scope, make sure they don't have the same window numbers (unless superposition of the curves is desired).
`Output window position` : a 2 vector specifying the coordinates of the upper left corner of the graphic window. Answer [] for default window position.
`Output window size` : a 2 vector specifying the width and height of the graphic window. Answer [] for default window dimensions.
`Xmin, Xmax` : Minimum and maximum values of the first input; used to set up the X-axis of the plot in the graphics window.
`Ymin, Ymax` : Minimum and maximum values of the second input; used to set up the Y-axis of the plot in the graphics window.
`Buffer size` : To improve efficiency it is possible to buffer the input data. The drawing is only done after each `Buffer size` call to the block.

REMARKS :
Output window number, Output window size, Output window position are only taken into account at the initialisation time of the simulation.

SEE ALSO : `SCOPE_f` 67, `EVENTSCOPE_f` 60, `ANIMXY_f` 56

D.58 SELECT_f: Scicos select block

DIALOGUE PARAMETERS :

`number of inputs` : a scalar. Number of regular and event inputs.
`initial connected input` : an integer. It must be between 1 and the number of inputs.

DESCRIPTION :
This block routes one of the regular inputs to the unique regular output. the choice of which input is to be routed is done, initially by the "initial connected input" parameter. Then, every time an input event arrives on the i-th input event port, the i-th regular input port is routed to the regular output.

D.59 SINBLK_f: Scicos sine block

DIALOGUE PARAMETERS :
None.

DESCRIPTION :
This block realizes vector sine operation. `y(i)=sin(u(i))`. The input and output port sizes are equal and determined by the context.

D.60 SOM_f: Scicos addition block

DIALOGUE PARAMETERS :

`Input signs` : a (1x3) vector of +1 and -1. If -1, the corresponding input is multiplied by -1 before addition.

DESCRIPTION :
This block is a sum. The output is the element-wise sum of the inputs.

Input ports are located at up, left or right and down position. You must specify 3 gain numbers but if only two links are connected only the first values are used, ports are numbered anti-clock wise.

SEE ALSO : `GAIN_f` 61

D.61 SPLIT_f: Scicos regular split block

DIALOGUE PARAMETERS :
None.

DESCRIPTION :
This block is a regular split block with an input and two outputs. The outputs reproduces the input port on each output ports. Strictly speaking, SPLIT is not a Scicos block because it is discarded at the compilation phase. This block is automatically created when creating a new link issued from a link.

Port sizes are determined by the context.

D.62 STOP_f: Scicos Stop block

DIALOGUE PARAMETERS :

`State on halt` : scalar. A value to be placed in the state of the block. For debugging purposes this allows to distinguish between different halts.

DESCRIPTION :
This block has a unique input event port. Upon the arrival of an event, the simulation is stopped and the main Scicos window is activated. Simulation can be restarted or continued (Run button).

D.63 SUPER_f: Scicos Super block

DESCRIPTION :
This block opens up a new Scicos window for editing a new block diagram. This diagram describes the internal functions of the super block.
Super block inputs and outputs (regular or event) are designated by special (input or output) blocks.
Regular input blocks must be numbered from 1 to the number of regular input ports. Regular input ports of the super block are numbered from the top of the block shape to the bottom.

Regular output portss must be numbered from 1 to the number of regular output ports. Regular output ports of the super block are numbered from the top of the block shape to the bottom.

Event input blocks must be numbered from 1 to the number of event input ports. Event input ports of the super block are numbered from the left of the block shape to the right.
Event output ports must be numbered from 1 to the number of event output ports. Event output ports of the super block are numbered from the left of the block shape to the right.

SEE ALSO : `CLKIN_f` 57, `OUT_f` 65, `CLKOUT_f` 57, `IN_f` 63

D.64 TANBLK_f: Scicos tan block

DIALOGUE PARAMETERS :
None.

DESCRIPTION :
This block realizes vector tangent operation. input (output) port size is determined by the compiler.

SEE ALSO : `SINBLK_f` 69

D.65 TCLSS_f: Scicos jump continuous-time linear state-space system

DESCRIPTION :
This block realizes a continuous-time linear state-space system with the possibility of jumps in the state. The number of inputs to this block is two. The first input is the regular input of the linear system, the second carries the new value of the state which is copied into the state when an event arrives at the unique event input port of this block. That means the state of the system jumps to the value present on the second input (of size equal to that of the state). The system is defined by the (A,B,C,D) matrices and the initial state x0. The dimensions must be compatible. The sizes of inputs and outputs are adjusted automatically.

DIALOGUE PARAMETERS :

`A` : square matrix. The A matrix
`B` : the B matrix
`C` : the C matrix
`D` : the D matrix
`x0` : vector. The initial state of the system.

SEE ALSO : `CLSS_f` 58, `CLR_f` 58

D.66 TEXT_f: Scicos text drawing block

DIALOGUE PARAMETERS :

`txt` : a character string, Text to be displayed
`font` : a positive integer less than 6, number of selected font (see xset)
`siz` : a positive integer, selected font size (see xset)

DESCRIPTION :
This special block is only use to add text at any point of the diagram window. It has no effect on the simulation.

D.67 TIME_f: Scicos time generator

DIALOGUE PARAMETERS :
None.

DESCRIPTION :
This block is a time generator. The unique regular output is the current time.

D.68 TRASH_f: Scicos Trash block

DIALOGUE PARAMETERS :
None

DESCRIPTION :
This block does nothing. It simply allows to safely connect the outputs of other blocks which should be ignored. Useful for sinking outputs of no interest. The input size is determined by the compiler.

D.69 WFILE_f: Scicos "write to file" block

DIALOGUE PARAMETERS :

`input size` : a scalar. This fixes the input size
`Output file name` : a character string defining the path of the file
`Output Format` : a character string defining the Fortran format to use or nothing for an unformatted (binary) write
`Buffer size` : To improve efficiency it is possible to buffer the input data. write on the file is only done after each `Buffer size` calls to the block.

DESCRIPTION :
This block allows user to save data in a file, in formatted and binary mode. Each call to the block corresponds to a record in the file. Each record has the following form: `[t,V1,...,Vn]` where `t` is the value of time when block is called and `Vi` is the ith input value

SEE ALSO : `RFILE_f` 66

D.70 ZCROSS_f: Scicos zero crossing detector

DESCRIPTION :
An output event is generated when all inputs (if more than one) cross zero simultaneously.
DIALOGUE PARAMETERS :

`Number of inputs` : a positive integer.

SEE ALSO : `POSTONEG_f` 65, `GENERAL_f` 62

D.71 scifunc_block: Scicos block defined interactively

DESCRIPTION :
This block can realize any type of Scicos block. The function of the block is defined interactively using dialogue boxes and in Scilab language. During simulation, these instructions are interpreted by Scilab; the simulation of diagrams that include these types of blocks is slower. For more information see Scicos reference manual.
DIALOGUE PARAMETERS :

`number of inputs` : a scalar. Number of regular input ports
`number of outputs` : a scalar. Number of regular output ports
`number of input events` : a scalar. Number of input event ports
`number of output events` : a scalar. Number of output event ports
`Initial continuous state` : a column vector.
`Initial discrete state` : a column vector.
`System type` : a string: c or d (CBB or DBB, other types are not supported).
`System parameter` : column vector. Any parameters used in the block can be defined here a column vector.
`initial firing` : vector. Size of this vector corresponds to the number of event outputs. The value of the i-th entry specifies the time of the preprogrammed event firing on the i-th output event port. If less than zero, no event is preprogrammed.
`Instructions` : other dialogues are opened consecutively where used may input Scilab code associated with the computations needed (block initialization, outputs, continuous and discrete state, output events date, block ending),

SEE ALSO : `GENERIC_f` 62

E Data Structures

E.1 scicos_main: Scicos editor main data structure

DEFINITION :

```
scs_m=list(params,o_1,....,o_n)
```

PARAMETERS :

`params` : Scilab list, `params=list(wpar,title,tol,tf,context))`

`wpar` : viewing parameters: `[w,h,Xshift,Yshift]`

`w` : real scalar,Scicos editor window width
`h` : real scalar,Scicos editor window height
`Xshift` : real scalar, diagram drawing x offset within Scicos editor window
`Yshift` : real scalar, diagram drawing y offset within Scicos editor window

`title` : character string, diagram title and default name of save file name

`tol` : 1 x 4 vector `[atol,rtol,ttol,maxt]`, where `atol`, `rtol` are respectively absolute and relative tolerances for the ode solver, `ttol` is the minimal distance between to different events time and `maxt` is maximum integration time interval for a single call to the ode solver.

`tf` : real scalar, final time for simulation.

`context` : vector of character strings, Scilab instructions used to define Scilab variables used in block definitions as symbolic parameters.

`o_i` : block or link or deleted object data structure.

See `scicos_block` and `scicos_link`).

Deleted object data structure is marked `list('Deleted')`.

`scs_m` : main Scicos structure

DESCRIPTION :
Scicos editor uses and modifies the Scicos editor main data structure to keep all information relative to the edited diagram. Scicos compiler uses it as a input.

SEE ALSO : `scicos` 53, `scicos_block` 72, `scicos_link` 74

E.2 scicos_block: Scicos block data structure

DEFINITION :

```
blk=list('Block',graphics,model,void,gui)
```

PARAMETERS :

`"Block"` : keyword used to define list as a Scicos block representation
`graphics` : Scilab list, graphic properties data structure
`model` : Scilab list, system properties data structure.
`void` : unused, reserved for future use.
`gui` : character string, the name of the graphic user interface function (generally written in Scilab) associated with the block.
`blk` : Scilab list, Scicos block data structure

DESCRIPTION :
Scicos editor creates and uses for each block a data structure containing all information relative to the graphic interface and simulation part of the block. Each of them are stored in the Scicos editor main data structure. Index of these in Scicos editor main data structure is given by the creation order.

For Super blocks `model(8)` contains a data structure similar to the `scicos_main` data structure.

SEE ALSO : `scicos_graphics` 72, `scicos_model` 73

E.3 scicos_graphics: Scicos block graphics data structure

DEFINITION :

```
graphics=list(orig,sz,flip,exprs,pin,pout,pein,peout,gr_i)
```

PARAMETERS :

`orig` : 2 x 1 vector, the coordinate of down-left point of the block shape.

`sz` : vector `[w,h]`, where `w` is the width and `h` the height of the block shape.
`flip` : boolean, the block orientation. if true the input ports are on the left of the box and output ports are on the right. if false the input ports are on the right of the box and output ports are on the left.
`exprs` : column vector of strings, contains expressions answered by the user at block set time.
`pin` : column vector of integers. If `pin(k)<>0` then `k`th input port is connected to the `pin(k)<>0` block, else the port is unconnected. If no input port exist `pin==[]`.
`pout` : column vector of integers. If `pout(k)<>0` then `k`th output port is connected to the `pout(k)<>0` block, else the port is unconnected. If no output port exist `pout==[]`.
`pein` : column vector of ones. If `pein(k)<>0` then `k`th event input port is connected to the `pein(k)<>0` block, else the port is unconnected. If no event input port exist `pein==[]`.
`peout` : column vector of integers. If `peout(k)<>0` then `k`th event output port is connected to the `epout(k)<>0` block, else the port is unconnected. If no event output port exist `peout==[]`.
`gr_i` : column vector of strings, contains Scilab instructions used to customize the block graphical aspect. This field may be set with `"Icon"` sub_menu.
`graphics` : Scilab list, Scicos block graphics data structure.

DESCRIPTION :
Scicos block graphics data structure contains all information relative to graphical display of the block and to user dialogue. Fields may be fixed by block definition or set as a result of user dialogue or connections.

SEE ALSO : `scicos` 53, `scicos_model` 73, `scicos_main` 71

E.4 scicos_model: Scicos block functionality data structure

DEFINITION :

```
model=list(sim,in,out,evtin,evtout,state,dstate,..
      rpar,ipar,blocktype,firing,dep_ut,label,import)
```

PARAMETERS :

`sim` : list(fun,typ) or fun. In the latest case typ is supposed to be 0.

- `fun` : character string, the name of the block simulation function (a linked C or Fortran procedure or a Scilab function).
- `typ` : integer, calling sequence type of simulation function (see documentation for more precision).

`in` : column vector of integers, input port sizes indexed from top to bottom of the block. If no input port exist `in==[]`.
`out` : column vector of integers, output port sizes indexed from top to bottom of the block. If no output port exist `in==[]`.
`evtin` : column vector of ones, the size of `evtin` gives the number of event input ports. If no event input port exists `evtin` must be equal to `[]`.
`evtout` : column vector of ones, the size of `evtout` gives the number of event output ports. If no event output port exists `evtout` must be equal to `[]`.
`state` : column vector, the initial continuous state of the block. Must be `[]` if no continuous state.
`dstate` : column vector, the initial discrete state of the block. Must be `[]` if no discrete state.
`rpar` : column vector, the vector of floating point block parameters. Must be `[]` if no floating point parameters.
`ipar` : column vector, the vector of integer block parameters. Must be `[]` if no integer parameters.
`blocktype` : a character with possible values:

- `:` 'c' block output depend continuously of the time.
- `:` 'd' block output changes only on input events.
- `:` 'z' zero crossing block
- `:` 'l' logical block

`firing` : a vector whose size is equal to the size of `evtout`> It contains output initial event dates (Events generated before any input event arises). Negative values stands for no initial event on the corresponding port.
`dep_ut` : 1x 2 vector of boolean `[dep_u, dep_t]`, `dep_u` must be true if output depends continuously of the input, `dep_t` must be true if output depends continuously of the time.
`label` : a character string, used as an identifier.
`import` : Unused.
`model` : Scilab list, Scicos block model data structure.

DESCRIPTION :
Scicos block model data structure contains all information relative to the simulation functionality of the block. Fields may be fixed by block definition or set.

If block is a super block, the fields `state`,`dstate`,`ipar`,`blocktype`,`firing`, `dep_ut`, are unused.

The `rpar` field contains a data structure similar to the `scicos_main` data structure.

SEE ALSO : `scicos` 53, `scicos_model` 73, `scicos_main` 71

E.5 scicos_link: Scicos link data structure

DEFINITION :

```
lnk=list('Link',xx,yy,'drawlink',' ',[0,0],ct,from,to)
```

PARAMETERS :

`"Link"` : keyword used to define list as a Scicos link representation
`xx` : vector of x coordinates of the link path.
`yy` : vector of y coordinates of the link path.
`ct` : 2 x 1 vector, `[color,typ]` where `color` defines the color used for the link drawing and `typ` defines its type (0 for regular link ,1 for event link).
`from` : 2 x 1 vector, `[block,port]` where `block` is the index of the block at the origin of the link and `port` is the index of the port.
`to` : 2 x 1 vector, `[block,port]` where `block` is the index of the block at the end of the link and `port` is the index of the port.

DESCRIPTION :
Scicos editor creates and uses for each link a data structure containing all information relative to the graphic interface and interconnection information. Each of them are stored in the Scicos editor main data structure. Index of these in Scicos editor main data structure is given by the creation order.

SEE ALSO : `scicos` 53, `scicos_main` 71, `scicos_graphics` 72, `scicos_model` 73

E.6 scicos_cpr: Scicos compiled diagram data structure

DEFINITION :

```
cpr=list(state,sim,cor,corinv)
```

PARAMETERS :

`state` : Scilab `tlist` contains initial state.

`state('x')` : continuous state vector.
`state('z')` : discrete state vector.
`state('tevts')` : vector of event dates
`state('evtspt')` : vector of event pointers
`state('pointi')` : pointer to next event state('npoint') : not used yet state('outtb') : vector of inputs/outputs initial values.

`sim` : Scilab `tlist`. Usually generated by Scicos `Compile` menu. Some useful entries are:

`sim('rpar')` : vector of blocks' floating point parameters
`sim('rpptr')` : (nblk+1) x 1 vector of integers,
`sim('rpar')(rpptr(i):(rpptr(i+1)-1))` is the vector of floating point parameters of the `i`th block.
`sim('ipar')` : vector of blocks' integer parameters
`sim('ipptr')` : (nblk+1) x 1 vector of integers,
`sim('ipar')(ipptr(i):(ipptr(i+1)-1))` is the vector of integer parameters of the `i`th block.
`sim('funs')` : vector of strings containing the names of each block simulation function
`sim('xptr')` : (nblk+1) x 1 vector of integers,
`state('x')(xptr(i):(xptr(i+1)-1))` is the continuous state vector of the `i`th block.

`sim('zptr')` : (nblk+1) x 1 vector of integers,
`state('z')(zptr(i):(zptr(i+1)-1))` is the discrete state vector of the `i`th block.
`sim('inpptr')` : (nblk+1) x 1 vector of integers,
`inpptr(i+1)-inpptr(i)` gives the number of input ports. `inpptr(i)`th points to the beginning of `i`th block inputs within the indirection table `inplnk`.
`sim('inplnk')` : nblink x 1 vector of integers,
`inplnk(inpptr(i)-1+j)` is the index of the link connected to the `j`th input port of the `i`th block. where `j` goes from `1` to `inpptr(i+1)-inpptr(i))`.
`sim('outptr')` : (nblk+1) x 1 vector of integers,
`outptr(i+1)-outptr(i)` gives the number of output ports. `outptr(i)`th points to the beginning of `i`th block outputs within the indirection table `outlnk`.
`sim('outlnk')` : nblink x 1 vector of integers,
`outlnk(outptr(i)-1+j)` is the index of the link connected to the `j`th output port of the `i`th block. where `j` goes from `1` to `outptr(i+1)-outptr(i))`.
`sim('lnkptr')` : (nblink+1) x 1 vector of integers,
`k`th entry points to the beginning of region within `outtb` dedicated to link indexed `k`.
`sim('funs')` : vector of strings containing the names of each block simulation function
`sim('funtyp')` : vector of block block types.

`cor` : is a list with same recursive structure as scs_m each leaf contains the index of associated block in `cpr` data structure.

`corinv` : corinv(i) is the path of `i` th block defined in `cpr` data structure in the `scs_m` data structure.

DESCRIPTION :
Scicos compiled diagram data structure contains all information needed to simulate the system (see `scicosim`).

SEE ALSO : `scicos` 53, `scicos_model` 73, `scicos_main` 71, `scicosim` 77

F Useful Functions

F.1 standard_define: Scicos block initial definition function

CALLING SEQUENCE :

```
o=standard_define(sz,model,dlg,gr_i)
```

PARAMETERS :

`o` : Scicos block data structure (see scicos_block)
`sz` : 2 vector, giving the initial block width and height
`model` : initial model data structure definition (see scicos_model)
`dlg` : vector of character strings,initial parameters expressions
`gr_i` : vector of character strings, initial icon definition instructions

DESCRIPTION :
This function creates the initial block data structure given the initial size `sz`, this initial model definition `model`, the initial parameters expressions `dlg` and initial icon definition instructions `gr_i`

SEE ALSO : `scicos_model` 73

F.2 standard_draw: Scicos block drawing function

CALLING SEQUENCE :

```
standard_draw(o)
```

PARAMETERS :

`o` : Scicos block data structure (see scicos_block)

DESCRIPTION :
`standard_draw` is the Scilab function used to display standard blocks in interfacing functions.
It draws a block with a rectangular shape with any number of regular or event input respectively on the left and right faces of the block (if not flipped), event input or output respectively on the top and bottom faces of the block. Number of ports, size, origin, orientation, background color, icon of the block are taken from the block data structure `o`.

SEE ALSO : `scicos_block` 72

F.3 standard_input: get Scicos block input port positions

CALLING SEQUENCE :

```
[x,y,typ]=standard_input(o)
```

PARAMETERS :

`o` : Scicos block data structure (see scicos_block)
`x` : vector of x coordinates of the block regular and event input ports
`y` : vector of y coordinates of the block regular and event output ports
`typ` : vector of input ports types (+1 : regular port; -1:event port)

DESCRIPTION :
`standard_input` is the Scilab function used to get standard blocks input port position and types in interfacing functions.
Port positions are computed, each time they are required, as a function of block dimensions.

SEE ALSO : `scicos_block` 72

F.4 standard_origin: Scicos block origin function

CALLING SEQUENCE :

```
[x,y]=standard_draw(o)
```

PARAMETERS :

`o` : Scicos block data structure (see scicos_block)
`x` : x coordinate of the block origin (bottom left corner)
`y` : y coordinate of the block origin (bottom left corner)

DESCRIPTION :
`standard_origin` is the Scilab function used to get standard blocks position in interfacing functions.

SEE ALSO : `scicos_block` 72

F.5 standard_output: get Scicos block output port positions

CALLING SEQUENCE :

```
[x,y,typ]=standard_output(o)
```

PARAMETERS :

`o` : Scicos block data structure (see scicos_block)
`x` : vector of x coordinates of the block regular and event output ports
`y` : vector of y coordinates of the block regular and event output ports
`typ` : vector of output ports types (+1 : regular port; -1:event port)

DESCRIPTION :
`standard_output` is the Scilab function used to get standard blocks output port position and types in interfacing functions.
Port positions are computed, each time they are required, as a function of block dimensions.

SEE ALSO : `scicos_block` 72

F.6 scicosim: Scicos simulation function

CALLING SEQUENCE :

```
[state,t]=scicosim(state,0,tf,sim,'start' [,tol])
[state,t]=scicosim(state,tcur,tf,sim,'run' [,tol])
[state,t]=scicosim(state,tcur,tf,sim,'finish' [,tol])
```

PARAMETERS :

`state` : Scilab `tlist` contains scicosim initial state. Usually generated by Scicos `Compile` or `Run` menus (see scicos_cpr for more details).
`tcur` : initial simulation time
`tf` : final simulation time (Unused with options `'start'` and `'finish'`
`sim` : Scilab `tlist`. Usually generated by Scicos `Compile` menu (see scicos_cpr for more details).
`tol` : 4 vector `[atol,rtol,ttol,deltat]` where `atol`, `rtol` are respectively the absolute and relative tolerances for ode solver (see ode), `ttol` is the precision on event dates. `deltat` is maximum integration interval for each call to ode solver.
`t` : final reached time

DESCRIPTION :
Simulator for Scicos compiled diagram. Usually `scicosim` is called by `scicos` to perform simulation of a diagram. But `scicosim` may also be called outside Scicos. Typical usage in such a case may be:

1 Use Scicos to define a block diagram, compile it.
2 Save the compiled diagram using `Save,SaveAs` Scicos menus .
3 In Scilab, load saved file using `load` function. You get variables `scicos_ver`, `scs_m`, `cpr`

`scs_m` is the diagram Scicos main data structure.
`cpr` is the data structure `list(state,sim,cor,corinv)` if the diagram had been compiled before saved, else `cpr=list()`

4 Extract `state`, `sim` out of `cpr`
5 Execute `[state,t]=scicosim(state,0,tf,sim,'start' [,tolerances])` for initialisation.
6 Execute `[state,t]=scicosim(state,0,tf,sim,'run' [,tolerances])` for simulation from `0` to `tf`. Many successives such calls may be performed changing initial and final time.
7 Execute `[state,t]=scicosim(state,0,tf,sim,'finish' [,tolerances])` at the very end of the simulation to close files,...

For advanced user it is possible to "manually" change some parameters or state values

SEE ALSO : `scicos` 53, `scicos_cpr` 74

F.7 curblock: get current block index in a Scicos simulation function

CALLING SEQUENCE :

```
k=curblock()
```

PARAMETERS :

`k` : integer, index of the block corresponding to the Scilab simulation function where this function is called.

DESCRIPTION :
During simulation it may be interesting to get the index of the current block to trace execution, to get its label, to animate the block icon according to simulation...
For block with a computational function written in Scilab, Scilab primitive function `curblock()` allows to get the index of the current block in the compiled data structure.
To obtain path to the block in the Scicos main structure user may uses the `corinv` table (see scicos_cpr).
For block with a computational function written in C user may uses the C function `k=C2F(getcurblock)()`. Where `C2F` is the C compilation macro defined in <`SCIDIR`>`/routines/machine.h`
For block with a computational function written in Fortran user may uses the integer function `k=getcurblock()`.
SEE ALSO : `getblocklabel` 78, `getscicosvars` 78, `setscicosvars` 79, `scicos_cpr` 74, `scicos_main` 71

F.8 getblocklabel: get label of a Scicos block at running time

CALLING SEQUENCE :

```
label=getblocklabel()
label=getblocklabel(k)
```

PARAMETERS :

`k` : integer, index of the block. if `k` is omitted `k`is supposed to be equal to `curblock()`.
`label` : a character string, The label of `k`th block (see `Label` button in `Block` menu.

DESCRIPTION :
For display or debug purpose it may be usefull to give label to particular blocks of a diagram. This may be done using Scicos editor (`Label` button in `Block` menu). During simulation, value of these labels may be obtained in any Scilab block with `getblocklabel` Scilab primitive function.
For C or fortran computational functions, user may use `C2F(getlabel)` to get a block label. See `routines/scicos/import.c` file for more details

Block indexes are those relative to the compile structure`cpr`.

SEE ALSO : `curblock` 77, `getscicosvars` 78, `setscicosvars` 79

F.9 getscicosvars: get Scicos data structure while running

CALLING SEQUENCE :
v=getscicosvars(name)

PARAMETERS :

`name` : a character string, the name of the required structure
`v` : vector of the structure value

DESCRIPTION :
This function may be used in a Scilab block to get value of some particular global data while running. It allows to write diagram monitoring blocks.
for example the instruction `disp(getscicosvars('x'))` displays the entire continuous state of the diagram.

```
x=getscicosvars('x');
xptr=getscicosvars('xptr');
disp(x(xptr(k):xptr(k+1)-1))
```

displays the continuous state of the `k` block

name	data structure definition
'x'	continuous state
'xptr'	continuous state splitting vector
'z'	discrete state
'zptr'	discrete state splitting vector
'rpar'	real parameters vector
'rpptr'	rpar splitting vector
'ipar'	integer parameters vector
'ipptr'	ipar splitting vector
'outtb'	vector of all input/outputs values
'inpptr'	inplnk splitting vector
'outptr'	outlnk splitting vector
'inplnk'	vector of input port values address in lnkptr
'outlnk'	vector of output port values address in lnpkpr
'lnkptr'	outtb splitting vector

See `scicos_cpr` for more detail on these data structures.
For C or fortran computational function the C procedure `C2F(getscicosvars)` may used. See `routines/scicos/import.c` file for more details.

SEE ALSO : `setscicosvars` 79, `scicosim` 77, `curblock` 77, `scicos_cpr` 74, `getblocklabel` 78

F.10 setscicosvars: set Scicos data structure while running

CALLING SEQUENCE :
setscicosvars(name,v)

PARAMETERS :

`name` : a character string, the name of the required structure
`v` : vector of the new structure value

DESCRIPTION :
This function may be used in a Scilab block to set value of some particular global data while running. It allows to write diagram supervisor blocks.
for example the instructions

```
x=getscicosvars('x');
xptr=getscicosvars('xptr');
x(xptr(k):xptr(k+1)-1)=xk
setscicosvars('x',x)
```

Changes the continuous state of the `k` block to `xk`.

name	data structure definition
'x'	continuous state
'xptr'	continuous state splitting vector
'z'	discrete state
'zptr'	discrete state splitting vector
'rpar'	real parameters vector
'rpptr'	rpar splitting vector
'ipar'	integer parameters vector
'ipptr'	ipar splitting vector
'outtb'	vector of all input/outputs values
'inpptr'	inplnk splitting vector
'outptr'	outlnk splitting vector
'inplnk'	vector of input port values address in lnkptr
'outlnk'	vector of output port values address in lnpkpr
'lnkptr'	outtb splitting vector

See `scicos_cpr` for more detail on these data structures.
For C or fortran computational function the C procedure `C2F(setscicosvars)` may used. See `routines/scicos/import.c` file for more details.
Warning: The use of this function requires a deep knowledge on how scicosim works, it must be used very carefully. Unpredicted parameters, state, link values changes may produce erroneous simulations.

SEE ALSO : `getscicosvars` 78, `scicosim` 77, `curblock` 77, `scicos_cpr` 74, `getblocklabel` 78

References

[1] *SIMULINK User's guide*, The MathWorks, Inc., 1993.

[2] *SystemBuild User's Guide*, Integrated Systems Inc., 1994.

[3] Grossman, R. L., Nerode A., Ravn, A. P. and Rischel H. (Eds.), *Hybrid Systems*, Lecture Notes in Computer Science 736, Springer Verlag, Berlin Heidelberg, 1993.

[4] Andersson, M., *Object-oriented modeling and simulation of hybrid systems*, Ph. D. Thesis, Lund Institute of Technology, 1994.

[5] Hindmarsh, A. C., "Lsode and Lsodi, two new initial value ordinary differential equation solvers," *ACM-Signum Newsletter*, vol. 15, no. 4, 1980.

[6] Petzold, L. R., "Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations," *SIAM J. Sci. Stat. comput.*, 4, 1983.

[7] Petzold, L. R., "A description of DASSL: a differential/algebraic system solver," in *Scientific Computing*, eds. R. S. Stepleman et al., North-Holland, 1983.

[8] Brenan, K. E., Campbell, S. L., and Petzold, L. R., *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, North-Holland, 1989.

[9] Sorel, Y., "Real-time embedded image processing applications using the A3 methodology," *Proc. IEEE Internat. Conf. Image Processing*, Lausanne, Switzerland, Sept. 1996.

[10] Sorel, Y., "Massively Parallel Computing Systems with Real-Time Constraints: the Algorithm Architecture Adequation Methodology," *Proc. Conf. MPCS94*, Ischia, Italy, May 1994.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399