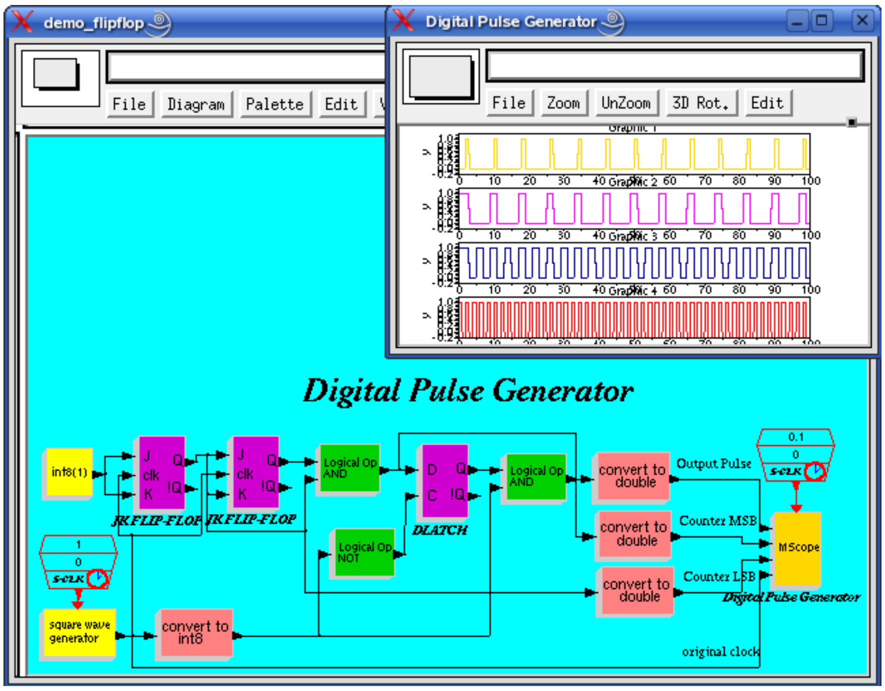
Scicos Block

Fady NASSIF

1 Introduction

Scicos is a scilab toolbox for modeling and simulating dynamical systems. It is particularly useful for modeling systems where continuous-time and discrete-time components are disconnected. The graphic editor of the toolbox Scicos allows the user to describe his dynamic system in a completely modular way. One of the basic module from which scicos diagram is constructed is the Block. A block in Scicos defines an operation that can evolve continuously or discretely in time. By interconnecting blocks with links, the user can construct his algorithm. A large number of mostly used blocks is saved in palettes,these blocks provide elementary operations needed to construct models diagram. Still users of scicos can create their own blocks. In this report, we will explain how to create a new basic scicos block and we will give a brief description about scicos structure.



2 Creating a basic scicos block

Each Scicos block is defined by two functions. The first one is the interfacing function, it is written in Scilab and it defines the geometry of the block (inputs,outputs,Icon, size, shape,...). It is also the function that handle the user interface (parameter, states,...). The second function is the computational function, it is mostly written in C but it can be also written in scilab. This function defines the behavior of the block during the simulation phase. A general example of the interfacing function is the `GENERIC` block in the `Others` palette. This block can be used by the user to create any operational block he wants. An interfacing function can support more than one computational function, and a computational function can be used by more than one interfacing function. Let us take the example of the computational function named `selector_m`, this function is used by the block `SELECT` and `ISELECT` in the `Branching` palette. Another example that shows the use of multiple computational function by one interfacing function is the `SUMMATION` block in the `Linear` palette.

2.1 Interfacing Function

The interfacing function is only used by the editor. It is used to initialize the block, to draw it and to modify its parameters. It defines also the initial state of the block and its computational function used by the simulator. The inputs of the interface function are job and $arg1$. The outputs of the function x, y and typ depends on the input flag job.

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

2.1.1 *job* the task parameter of `GUI_function`:

The parameter job can take the following values:

- **plot**: This function draws the block.
  - $arg1$ is the data structure of the block.
  - x,y,typ are unused.

  In general, we used the `standard_draw` function to draw a rectangular block, as well as its inputs and outputs ports. It also manages the size, the icon and the colors of the block.

- **getinputs**: This function returns the position and the type of inputs ports (regular or event).
  - $arg1$ is the data structure of the block.
  - x is the vector of x coordonates of inputs ports.
  - y is the vector of y coordonates of inputs ports.
  - typ is the vector of type of inputs ports (1 for regular and 2 for event).

In general, we use the `standard_input` function.

- **getoutputs**: This function returns the position and the type of outputs ports (regular or event).
  - *arg*1 is the data structure of the block.
  - *x* is the vector of x coordonates of outputs ports.
  - *y* is the vector of y coordonates of outputs ports.
  - *typ* is the vector of type of outputs ports (1 for regular and 2 for event).

  In general, we use the `standard_output` function. The `standard_input`, `standard_output` functions place the regular input/output port of a block on its side, the event inputs ports on its top and the event outputs ports on its bottom.

- **getorigin**: This function returns the coordinates of the lower left point of the rectangular containing the block's shape.
  - *arg*1 is the data structure of the block.
  - *x* is the x coordonate of the lower left point of the block.
  - *y* is the y coordonate of the lower left point of the block.
  - *typ* is unused.

  In general, we use the `standard_origin` function.

- **set**: This function open a dialog box for the block parameter acqisition, if it exists. This is done when we double click on the block. The programme must check the coherence of the parameters provided by the user.
  - *arg*1 is the data structure of the block.
  - *x* is the new data structure of the block.
  - *y* is unused.
  - *typ* is unused.

- **define**: This function initializes the data structure of the block (name of the computational function, the type, input/output number, sizes and data type,etc...).
  - *arg*1 is unused.
  - *x* is the data structure of the block.
  - *y* is unused.
  - *typ* is unused.

Example: the Interfacing Function of the Unit Delay Block

```
 function [x,y,typ]=DOLLAR_m(job,arg1,arg2)
// Copyright INRIA
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;
  graphics=arg1.graphics;exprs=graphics.exprs
  model=arg1.model;
  //backward compatibility
   if size(exprs,'*')<2 then exprs(2)='0'; end
   while %t do
    [ok,a,inh,exprs]=getvalue('Set 1/z block parameters',..
        ['initial condition';'Inherit (no:0, yes:1)'],...
                              list('mat',[-1 -2],'vec',-1),exprs)
     if ~ok then break, end
    out=[size(a,1) size(a,2)]; if out==0 then out=[], end
    in=out
    model.sim=list('dollar4_m',4)
    model.odstate=list(a);
    model.dstate=[];
     if (type(a)==1) then
         if isreal(a) then
            it=1;
            ot=1;
             if (size(a,1)==1 | size(a,2)==1) then
                model.sim=list('dollar4',4);
                model.dstate=a(:);
                model.odstate=list();
             end
         else
             it=2;
             ot=2;
         end
     elseif (typeof(a)=="int32") then
             it=3;
             ot=3;
     elseif (typeof(a)=="int16") then
             it=4;
             ot=4;
```

```
      elseif (typeof(a)=="int8")  then
              it=5;
              ot=5;
      elseif (typeof(a)=="uint32")  then
              it=6;
              ot=6;
      elseif (typeof(a)=="uint16")  then
              it=7;
              ot=7;
      elseif (typeof(a)=="uint8")  then
              it=8;
              ot=8;
      else message ("type is not recognized"); ok=%f;
      end
      if ok  then
       [model,graphics,ok]=set_io(model,graphics,list(in,it),list(out,ot),ones(1-inh,1),[])
      end
      if ok  then
       graphics.exprs=exprs;
       x.graphics=graphics;x.model=model
        break
      end
    end
  case 'define'  then
   z=0
   inh=0
   in=1
   exprs=string([z;inh])
   model=scicos_model()
   model.sim=list('dollar4',4)
   model.in=in
   model.out=in
   model.evtin=1-inh
   model.dstate=z
   model.blocktype='d'
   model.dep_ut=[%f %f]
   gr_i='xstringb(orig(1),orig(2),''1/z'',sz(1),sz(2),''fill'')'
   x=standard_define([2 2],model,exprs,gr_i)
  end
  endfunction
```

The data structure used in this function will be explained later. We are going to study the 'set' case of this example. We have in this case two majors and basics function of the 'set' case. The `getvalue` and the `setio`. The `getvalue` function is used to create a dialog for the block. It must be used in all user blocks, since it is overloaded when `Eval` is performed. In our example, the function is defined by:

```
[ok,a,inh,exprs]=getvalue('Set 1/z block parameters',..
        ['initial condition';'Inherit (no:0, yes:1)'],...
                              list('mat',[-1 -2],'vec',-1),exprs)
```

Where the 'initial condition' and 'Inherit' are the parameters that appear in the dialog box. (for more information on `getvalue` execute *help getvalue* in Scilab prompt).

The `setio` function update the model and the graphic of a block by adjusting its input/output number, size, type and data type. The input parameter of this function are *model*, *graphics*, *input*, *output*, *eventinput*, *eventoutput*. The regular input/output are defined as a list where the first element is the input/output dimensions and the second is the input/output data type (third and forth input arguments of the function `setio`). The event input/output ports are defined by the fifth/sixth argument of the `setio`. Each one is defined as a column vector of ones where the size is equal to the number of event input/output. In our example:

```
[model,graphics,ok]=set_io(model,graphics,list(in,it),list(out,ot),..
                                 ones(1-inh,1),[])
```

- *in* is the input ports dimensions. The number of lines of *in* defines the number of regular inputs.
- *it* is the input data type. It is a line vector where the number of columns is equal to the number of regular inputs. The values of this vector are integer between 1 and 8.
- *out* is the output ports dimensions. The number of lines of *out* defines the number of regular outputs.
- *ot* is the output data type. It is a line vector where the number of columns is equal to the number of regular outputs. The values of this vector must be integer between 1 and 8.

In addition to the input/output update, the 'set' case updates also the states of the block, the computational function, etc... Here for example we charge the object state of the block by the 'initial value'.

2.2 Computational Function

The computational function defines the operation of the block. It is called by the simulator in various ways. These ways are characterize by the *typ* of the simulation function. Most of new blocks use type 4 or type 5. Type 4 is used for programs written in C and type 5 is used for programming in Scilab. In the C programmation case, the function must have two inputs arguments: a structure containing information on the block and the flag.

```
#include "scicos_block4.h"

void sim_func(scicos_block *block,int flag)
```

```
{
}
```

We will briefly introduce, in the next paragraphs, the two parameters of the computational functions, as well as a set of macros used in almostly all of the new computational functions.

2.2.1 flag

The computational function performs different jobs according to the flag parameter. Till now the following jobs exists in the simulation:

- **Initialization:** At the begining of the simulation, this job is called under flag=4 to initialize continuous and discrete states (if necessary). It also initialize the output port of the block. This function is not used in all of the blocks, it is used for blocks that needed dynamically allocated memory (the allocation is done under this flag), for blocks that read and write data from files, for opening a file, or by the scope to initialize the graphics window.
- **Reinitialization:** This job is called under flag=6. In this case, the values of the inputs are available and the function create a fixed point iteration for the outputs of the block. On this occasion, the function can also reinitialize its initial states.
- **Outputs update:** This job is called when flag=1, it is called only if the block has regular ports. In this case, the simulator requests the values of the outputs. The computational function uses the information given by the block structure to calculate the output and put it at the address given by the block structure. If the block contains different mode then the calculation depends on the *simulation phase*.
- **States update:** This job is called when flag=2, it is called only if the block has event input. In this case, the computation function update the value of the continuous or discrete state. This flag is called only if the block has an event activation (`nevprt` $\neq 0$, or an internal zero-crossing (`nevprt`$= -1$). In the second case a vector `jroot` specifies which surface has been crossed and in which direction. for exemple, if the ith entry of `jroot` is equal to 0 it means that there is no crossing, whereas if it is equal to $+1$ (respectively -1), then the crossing is with a positive (respectivly negative) slope.
- **Integrator calls:** This job is called when flag=0. In this case, the simulation function computes the derivative state $\dot{x}$ and place its value in the address provided by the block's structure.
- **Mode and zero-crossing:** This job is called when flag=9. In this case, we set the mode and evaluate the zero-crossing. To set the mode, information about the nonsmoothness of the model must be given to the solver.

flag	inputs	outputs	description
0	`t`, `nevprt`, `x`, `z`, `inptr`, `mode`, `phase`	`xd`	compute the derivative of continuous time state
1	`t`, `nevprt`, `x`, `z`, `inptr`, `mode`, `phase`	`outptr`	compute the outputs of the block
2	`t`, `nevprt>0`, `x`, `z`, `inptr`	`x`, `z`	update states due to external activation
2	`t`, `nevprt=-1`, `x`, `z`, `inptr`, `jroot`	`x`, `z`	update states due to internal zero-crossing
3	`t`, `x`, `z`, `inptr`, `jroot`	`evout`	program activation output delay times
4	`t`, `x`, `z`	`x`, `z`, `outptr`	initialize states and other initializations
5	`x`, `z`, `inptr`	`x`, `z`, `outptr`	final call to block for ending the simulation
6	`t`, `nevprt`, `x`, `z`, `inptr`, `mode`, `phase`	`x`, `z`,`outptr`	reinitialization (if needed).
7			only used for internally implicit blocks
9	`t`, `phase=1`, `nevprt`, `x`, `z`, `inptr`	`g`, `mode`	compute zero-crossing surfaces and set modes
9	`t`, `phase=2`, `nevprt`, `x`, `z`, `inptr`	`g`	compute zero-crossing surfaces

Table 1: This tables illustrates the jobs that the computational function must perform for different flags.

Let us consider the case of the of the `abs` block. In this case we have two `modes`, the first one is when the output is equal to the input, and the second one is when it is the input's negative. The nonsmoothness point zero is detected by the zero-crossing and the model is updated after such detection. the number of zero-crossing may be greater than the number of modes (i.e `zcross` block).

- **Events schedular:** This job is called when flag=3. In this case, the simulator updates the next event output time of the block.
- **Ending:** This job is called when flag=5 at the end. This case is used to close files opened by the block at the begining or during the simulation, to free the allocated memory,etc..

Table 1 summarizes the role of each `flag`. In most cases, the computation functions are not called with all flags. For example, a block with no event port is not called under flag=2.

In addition to the block structure, the computational function recieves inputs from the following functions:

- **double get_scicos_time()**: returns the current time `t`.
- **int get_phase_simulation()**: returns the simulation phase (1 or 2).
- **int get_block_number()**: returns the block number in the structure `%cpr`.
- **void set_block_error(int)**: used by the block to signal an error to the simulator.
- **void do_cold_restart()**: used to force a cold restart of the numerical solver (almost never used since Scicos determines automatically when cold restart is needed).
- **void set_pointer_xproperty(int* pointer)**: used only for internally implicit blocks to designate which states are algebraic and which ones are differential.
- **void *scicos_malloc(size_t)**: used to allocate memory for workspace if needed.
- **void scicos_free(void *p)**: used to free allocated memory.
- **double Get_Jacobian_parameter(void)**
- **double Get_Scicos_SQUR(void)**
- **void Set_Jacobian_flag(int flag)**

These functions are defined in `scicos_block4.h` in the `'routine/scicos/'` directory.

2.2.2 Block Structure

In C programming, the block structure is defined as follows:

```
 /* scicos_block structure definition */
typedef struct {
    int nevprt;      /* Binary coding of activation inputs, -1 for internal activation */
    voidg funpt ;    /* Pointer to the computational function */
    int type;        /* Type of the computational function, in this case type4 */
    int scsptr;      /* Not used for C programming */
    int nz;          /* Size of discrete-time state vector */
    double *z;       /* Vector of discrete-time state */
    int noz;         /* Number of object states */
    int *ozsz;       /* Vector of sizes of object states */
    int *oztyp;      /* Vector of data types of object states */
    void **ozptr;    /* Table of pointers to object states */
    int nx;          /* Size of continuous-time state vector */
    double *x;       /* Vector of continuous-time state */
    double *xd;  /* Vector of the derivative of the continuous-state, same size nx */
    double *res; /* Only used for internally implicit blocks, vector of size nx */
    int nin;        /* Number of regular input ports */
    int *insz;      /* Vector of sizes of regular input ports*/
    void **inptr; /* Tables of pointer to the regular input ports */
    int nout;       /* Number of regular output ports */
    int *outsz;     /* Vector of sizes of regular output ports */
    void **outptr; /* Tables of pointers to the regular output ports */
    int nevout;     /* Number of event output ports */
    double *evout; /* Delay time of output events */
    int nrpar;      /* Size of real parameters vector */
    double *rpar; /* Vector of real parameters */
    int nipar;      /* Size of integer parameters vector */
    int *ipar;      /* Vector of integer parameters */
    int nopar;      /* Number of object parameters */
    int *oparsz;    /* Vector of sizes of object parameters */
    int *opartyp; /* Vector of data types of object parameters*/
    void **oparptr; /* Table of pointers to the object parameters */
    int ng;         /* Size of zero-crossing surfaces vector*/
    double *g;      /* Vector of zero-crossing surfaces*/
    int ztyp;       /* Boolean, True only if the block may have zero-crossing surfaces */
    int *jroot;     /* Vector of size ng indicates the presence and the direction of the crossing */
    char *label;    /* Block label */
    void **work; /* Table of pointers to the block workspace (if allocation done by the block) */
    int nmode;      /* Size of modes vector */
    int *mode;      /* Vector of modes */
} scicos_block;
```

2.2.3 Useful macros for C programming

Almostly all the new computational functions of type 4 use at least one of the following macros:

```
/* scicos_block macros definition :*/
GetNin(blk)               /* Get Number of Regular Input Port */
GetInPortPtrs(blk,x)      /* Get Regular Input Port Pointer of port number x */
GetNout(blk)              /* Get Number of Regular Output Port */
GetOutPortPtrs(blk,x)     /* Get Regular Output Port Pointer of port number x */
GetInPortRows(blk,x)      /* Get number of Rows (first dimension) of Regular Input Port number x */
GetInPortCols(blk,x)      /* Get number of Columns (second dimension) of Regular Input Port number x */
GetInPortSize(blk,x,y)    /* Get Regular Input Port Size */
                          /* usage :
                          *  GetInPortSize(blk,x,1) : get first dimension of input port number x
                          *  GetInPortSize(blk,x,2) : get second dimension of input port number x
                          */
GetInType(blk,x)          /* Get Type of Input Port */
GetOutPortRows(blk,x)     /* Get number of Rows (first dimension) of Regular Output Port number x */
GetOutPortCols(blk,x)     /* Get number of Columns (second dimension) of Regular Output Port number x */
GetOutPortSize(blk,x,y)   /* Get Regular Output Port Size */
GetOutType(blk,x)         /* Get Type of Output Port */
GetRealInPortPtrs(blk,x) /* Get Pointer of Real Part of Regular Input Port */
GetImagInPortPtrs(blk,x) /* Get Pointer of Imaginary Part of Regular Input Port */
GetRealOutPortPtrs(blk,x) /* Get Pointer of Real Part of Regular Output Port */
GetImagOutPortPtrs(blk,x) /* Get Pointer of Imaginary Part of Regular Output Port */
Getint8InPortPtrs(blk,x) /* Get Pointer of int8 typed Regular Input Port */
Getint16InPortPtrs(blk,x) /* Get Pointer of int16 typed Regular Input Port */
```

Getint32InPortPtrs(blk,x)	*/* Get Pointer of int32 typed Regular Input Port */*
Getuint8InPortPtrs(blk,x)	*/* Get Pointer of uint8 typed Regular Input Port */*
Getuint16InPortPtrs(blk,x)	*/* Get Pointer of uint16 typed Regular Input Port */*
Getuint32InPortPtrs(blk,x)	*/* Get Pointer of uint32 typed Regular Input Port */*
Getint8OutPortPtrs(blk,x)	*/* Get Pointer of int8 typed Regular Output Port */*
Getint16OutPortPtrs(blk,x)	*/* Get Pointer of int16 typed Regular Output Port */*
Getint32OutPortPtrs(blk,x)	*/* Get Pointer of int32 typed Regular Output Port */*
Getuint8OutPortPtrs(blk,x)	*/* Get Pointer of uint8 typed Regular Output Port */*
Getuint16OutPortPtrs(blk,x)	*/* Get Pointer of uint16 typed Regular Output Port */*
Getuint32OutPortPtrs(blk,x)	*/* Get Pointer of uint32 typed Regular Output Port */*
GetNipar(blk)	*/* Get Number of Integer Parameters */*
GetIparPtrs(blk)	*/* Get Pointer of Integer Parameters */*
GetNrpar(blk)	*/* Get Number Real Parameters */*
GetRparPtrs(blk)	*/* Get Pointer of Real Parameters */*
GetWorkPtrs(blk)	*/* Get Pointer of Work */*
GetNdstate(blk)	*/* Get Number of Discrete state */*
GetDstate(blk)	*/* Get Pointer of Discrete state */*
GetNev(blk)	*/* Get Nevprt of the block */*
GetNopar(blk)	*/* Get Number of Object Parameters */*
GetOparType(blk,x)	*/* Get Type of Object Parameters */*
GetOparSize(blk,x,y)	*/* Get Size of Object Parameters */*
GetOparPtrs(blk,x)	*/* Get Pointer of Object Parameters */*
GetRealOparPtrs(blk,x)	*/* Get Pointer of Real Object Parameters */*
GetImagOparPtrs(blk,x)	*/* Get Pointer of Imaginary Part of Object Parameters */*
Getint8OparPtrs(blk,x)	*/* Get Pointer of int8 typed Object Parameters */*
Getint16OparPtrs(blk,x)	*/* Get Pointer of int16 typed Object Parameters */*
Getint32OparPtrs(blk,x)	*/* Get Pointer of int32 typed Object Parameters */*
Getuint8OparPtrs(blk,x)	*/* Get Pointer of uint8 typed Object Parameters */*
Getuint16OparPtrs(blk,x)	*/* Get Pointer of uint16 typed Object Parameters */*
Getuint32OparPtrs(blk,x)	*/* Get Pointer of uint32 typed Object Parameters */*
GetNoz(blk)	*/* Get Number of Object State */*
GetOzType(blk,x)	*/* Get Type of Object State */*
GetOzSize(blk,x,y)	*/* Get Size of Object State */*
GetOzPtrs(blk,x)	*/* Get Pointer of Object State */*
GetRealOzPtrs(blk,x)	*/* Get Pointer of Real Object State */*
GetImagOzPtrs(blk,x)	*/* Get Pointer of Imaginary Part of Object State */*
Getint8OzPtrs(blk,x)	*/* Get Pointer of int8 typed Object State */*
Getint16OzPtrs(blk,x)	*/* Get Pointer of int16 typed Object State */*
Getint32OzPtrs(blk,x)	*/* Get Pointer of int32 typed Object State */*
Getuint8OzPtrs(blk,x)	*/* Get Pointer of uint8 typed Object State */*
Getuint16OzPtrs(blk,x)	*/* Get Pointer of uint16 typed Object State */*
Getuint32OzPtrs(blk,x)	*/* Get Pointer of uint32 typed Object State */*
GetSizeOfOz(blk,x)	*/* Get The sizeof of the Object State */*
GetSizeOfOpar(blk,x)	*/* Get The sizeof of the Object Parameters */*
GetSizeOfOut(blk,x)	*/* Get The sizeof of the Output */*
GetSizeOfIn(blk,x)	*/* Get The sizeof of the Input */*
SCSREAL_N	*/* Corresponds to the Scilab real number (10) */*
SCSCOMPLEX_N	*/* Corresponds to the Scilab complex number (11) */*
SCSINT_N	*/* Corresponds to the Scilab integer number (80) */*
SCSINT8_N	*/* Corresponds to the Scilab int8 number (81) */*
SCSINT16_N	*/* Corresponds to the Scilab int16 number (82) */*
SCSINT32_N	*/* Corresponds to the Scilab int32 number (84) */*
SCSUINT_N	*/* Corresponds to the Scilab unsigned integer number (800) */*
SCSUINT8_N	*/* Corresponds to the Scilab uint8 number (811) */*
SCSUINT16_N	*/* Corresponds to the Scilab uint16 number (812) */*
SCSUINT32_N	*/* Corresponds to the Scilab uint32 number (814) */*
SCSUNKNOW_N	*/* Corresponds to the Scilab unknown number (-1) */*

```
SCSREAL_COP        /* Corresponds to the C real operator       (double) */
SCSCOMPLEX_CO      /* Corresponds to the C complex operator    (double) */
SCSINT_COP         /* Corresponds to the C integer operator    (int) */
SCSINT8_COP        /* Corresponds to the C int8 operator       (char) */
SCSINT16_COP       /* Corresponds to the C int16 operator      (short) */
SCSINT32_COP       /* Corresponds to the C int32 operator      (long) */
SCSUINT_COP        /* Corresponds to the C unsigned integer operator (unsigned int) */
SCSUINT8_COP       /* Corresponds to the C uint8 operator      (unsigned char) */
SCSUINT16_COP      /* Corresponds to the C uint16 operator     (unsigned short) */
SCSUINT32_COP      /* Corresponds to the C uint32 operator     (unsigned long) */
SCSUNKNOW_COP      /* Corresponds to the C unknown operator    (double) */
```

2.2.4 Examples

A simple example of a computational function that shows how to use the C macros is one of the computational functions of the SUMMATION block.

```
#include <math.h>
#include "scicos_block4.h"
#include <stdio.h>
extern int sciprint();
void summation_ui16e(scicos_block *block,int flag)
{
 if((flag==1)|(flag==6)) {
    int j,k;
    int nu,mu,nin;
    unsigned short *y;
    int *ipar;
    double v,l;
    unsigned short *u;

    y=Getuint16OutPortPtrs(block,1);
    nu=GetInPortRows(block,1);
    mu=GetInPortCols(block,1);
    ipar=GetIparPtrs(block);
    nin=GetNin(block);
    l=pow(2,16);
    if (nin==1){
      v=0;
      u=Getuint16InPortPtrs(block,1);
      for (j=0;j<nu*mu;j++) {
        v=v+(double)u[j];
      }
      if ((v>=l)|(v<0))
         {sciprint("overflow error");
          set_block_error(-4);
          return;}
      else y[0]=(unsigned short)v;
    }
    else {
      for (j=0;j<nu*mu;j++) {
        v=0;
        for (k=0;k<nin;k++) {
          u=Getuint16InPortPtrs(block,k+1);
          if(ipar[k]>0){
             v=v+(double)u[j];
          }
        else{
          v=v−(double)u[j];}
     }
     if ((v>=l)|(v<0))
        {sciprint("overflow error");
        set_block_error(−4);
        return;}
     else y[j]=(unsigned short)v;
   }
  }
 }
}
```

In this function, we use the C-macros to get the input x, the integer parameter $ipar$, the number of inputs nin, the inputs' dimensions m, n, and to charge the output y. In case of overflow, the `set_block_error` function is called to indicate the error. This function is called under flag 1 and flag 6.

We will discuss now another example that calls the computational function under flag 3. Let us consider the *Multi-frequency* computational function.

```
#include "scicos_block4.h"
#if WIN32
#define NULL 0
#endif
void m_frequ(scicos_block *block,int flag)
{
  double *mat;
  double *Dt;
  double *off;
  long *icount;
  double t;
  long long *counter;
  int m;
  mat=GetRealOparPtrs(block,1);
  Dt=GetRealOparPtrs(block,2);
  off=GetRealOparPtrs(block,3);
  icount=Getint32OparPtrs(block,4);
  m=GetOparSize(block,1,1);
  switch(flag)
  {
   case 4 : {/* the workspace is used to store discrete counter value */
             if ((*block−>work=scicos_malloc(sizeof(long long int)*2))==NULL) {
               set_block_error(−16);
               return;
             }
             counter=*block−>work;
             if (*icount!=0) (*counter)=(int)mat[0];
             else *counter=0;
             (*(counter+1))=*icount;
             break;
            }
   /* event date computation */
   case 3 : {
             counter=*block−>work;
             t=get_scicos_time();
             *counter+=(int)mat[*(counter+1)]; /*increase counter*/
             block->evout[(int)mat[*(counter+1)+m]-1]=*off+((double)*counter/(*Dt))-t;
             (*(counter+1))++;
             *(counter+1)=*(counter+1)%m;
             break;
            }
   /* finish */
   case 5 : {
             scicos_free(*block->work); /*free the workspace*/
             break;
            }
   default : break;
  }
}
```

This computation function uses the dynamically allocated memory's function `scicos_malloc` to charge the workspace with data, and at the end (flag=5), the function free this memory with the `scicos_free` function. This computational function uses also the `get_scicos_time()` function, to get the current scicos time and then calculates the next output activation time and charge it in the `block->evout`(under flag=3).

```
block->evout[(int)mat[*(counter+1)+m]-1]=*off+((double)*counter/(*Dt))-t;
```

where t is the current time. **To compute the activation delay you must always substract the current time**.

Another example, a little bit more complicated is the computational function associated with the `ABS` block.

```
#include "scicos_block.h"
#include <math.h>

void absolute_value(scicos_block *block,int flag)
{
  int i,j;
  if (flag==1){
    if( block->ng>0){
      for(i=0;i<block->insz[0];++i){
        if (get_phase_simulation()==1) {
          if (block->inptr[0][i]<0){
            j=2;
          } else{
            j=1;
          }
        }else {
          j=block->mode[i];
        }
        if (j==1){
          block->outptr[0][i]=block->inptr[0][i];
        } else{
          block->outptr[0][i]=-block->inptr[0][i];
        }
      }
    }else{
      for(i=0;i<block->insz[0];++i){
        if (block->inptr[0][i]<0){
          block->outptr[0][i]=-block->inptr[0][i];
        }else{
          block->outptr[0][i]=block->inptr[0][i];
        }
      }
    }
  }else if (flag==9){
    for(i=0;i<block->insz[0];++i){
      block->g[i]=block->inptr[0][i];
      if (get_phase_simulation()==1) {
        if(block->g[i]<0){
          block->mode[i]=2;
        }else{
          block->mode[i]=1;
        }
      }
    }
  }
}
```

This example is not very simple because this block is nonsmooth and thus uses modes. Note, however, that the modes are used only if the block generates continuous-time signals affecting the continuous-time state. The compiler determines whether this is the case, so the computational function must be prepared to function both with and without modes. It is the value of `block->ng` (or equivalently in this case `block->nmode`) that determines whether modes should be used. Note that if `block->ng` is zero, then a simple absolute value operation is performed in flag 1 and the block is never called with flag 9.

If, on the other hand, `block->ng` is not zero, then modes must be used. An important function to use in this case is `get_phase_simulation`. It returns 1 or 2 to specify the simulation phase. If the numerical solver is at work advancing the time, then the computational function must produce a smooth signal and thus must use the mode to generate its output. When `get_phase_simulation` returns 1, then the output must be computed normally by computing the absolute value of the input. In the flag 9 case, the zero-crossing surface is computed all the time, but the mode is set only in simulation phase 1.

Note that using modes in this case means that during the simulation, the output of the `ABS` block can become negative because at zero the solver has to step back and forth for pinpointing the zero-crossing. In some cases, this could be a problem, for example, if the `ABS` block is followed by a `SQRT` block that computes the square root. It is for this reason that in the `ABS` block we have the option of using, or not using, zero-crossings (modes). This is true for most blocks using modes.

2.2.5 Scilab computational function

This function is called when type is equal to 5. As input's arguments, it uses the same block structure as the type 4, and the flag. The fields of block are:

```
scicos_block  nevprt  funpt  type  scsptr  nz  z noz ozsz oztyp ozptr
nx  x  xd  res  nin  insz  inptr  nout  outsz  outptr  nevout  evout
nrpar  rpar  nipar  ipar nopar oparsz opartyp oparptr  ng  g  ztyp
jroot  label  work  nmode  mode
```

In this type block.nz, block.nx,etc... gives the size of the memory used by the Scilab object. It doesn't gives the size of the vector but rather the size of the vector +2. For this reason, it is recomended to use the Scilab command `size` to calculates the size of the vector. For example:

```
nz=size(block.z)
```

The second argument flag is a scalar and has the same role as flag in C programming. The type 5 programation has also very important input functions to obtain additional operation. These functions are defined as follows:

- **curblock()**: returns the current block number in the structure `%cpr`.
- **scicos_time()**: gives the current time.
- **phase_simulation()**: returns the simulation phase.
- **set_blockerror(i)**: set the error flag if the computational function encounters an error.
- **pointer_xproperty**: used for internally implicit blocks.

A simple example of programming with type 5 is the `sin5` computational function.

```
function block=lcm5(block,flag)
  if flag==1  then
     for j=1:(block.insz(1)*block.insz(1+block.nin))
          v=[];
          for i=1:block.nin
             v=[v;block.inptr(i)(j)]
          end
         block.outptr(1)(j)=lcm(v);
     end
  end
endfunction
```

2.3 Saving and using the new block

After creating the computational and interfacing function, how to use it in your diagram. Consider that the computational function of your block is `sim_func.c` and the the interface function is `MYBLK.sci`. First of all, you have to charge the interfacing function. To do so, you can use the `getf` scilab function. Then you have to compile and link your computational function, another scilab function allow you to do it: `ilib_for_link`.

```
--> getf('.../MYBLK.sci');
--> ilib_for_link('sim_func','sim_func.o',[],'c');
--> exec loader.sce
```

After linking the two functions, you can use your block in a diagram by using the `AddNewBlock` menu by giving the name of the corresponding interface function.

3 Scicos structure

A lot of macros and functions used in the previous paragraph are very strange for users who are not adapted to Scicos. For this reason, we will give an overview on how the Scicos compiler and editor are implemented. Our objective in this section is to give enough information so that the code will be comprehensible. This requires explaining the Data structures used and giving the structure of the main programs.

The data structures used in Scicos are all Scilab variables. Thus they can be examined and altered using standard Scilab functions. There are in particular two variables associated with a Scicos diagram that are of great importance:

- `scs_m` contains all the information concerning the Scicos diagram. It is constructed by the editor.
- `%cpr` contains the result of the compilation; it is used by the simulator.

We start with a brief presentation of the editor and the way `scs_m` is constructed. We then look at the compilation phase, where `scs_m` is used to obtain `%cpr`. `%cpr` is used by the simulator to perform simulation.

3.1 Scicos Editor

The Scicos editor is written entirely in the Scilab language, including the graphical user interface, which uses standard Scilab graphics. This facilitates customization of the editor by the user. This section provides the necessary information.

3.1.1 Main Editor Function

The main Scilab function implementing the Scicos editor is `scicos`, which can be found in file `SCI/macros/auto/scicos.sci`. This function takes as input argument a Scicos diagram (`scs_m`) and *opens* it. Invoked without any argument, it opens an empty diagram. This function is called recursively when a Super block is opened.

The Scicos editor can be parameterized by initializing the following variables:

- `scicos_pal`: list of palettes. An $n \times 2$ matrix of strings, where the first column contains the names of the palettes and the second the file (`.cos` or `.cosf`) in which the palette is defined. In fact, a palette is nothing but a Scicos diagram.

- `%scicos_menu`: Scicos menus. A Scilab list including vectors of strings, where the first element of each vector contains the name of the menu, and the rest, the corresponding items of the menu. Each item corresponds to an operation (such as move, copy, save) and has a corresponding Scilab function.

- `%scicos_short`: keyboard shortcuts. An $n \times 2$ matrix of strings, where the first column contains the character to be used for the shortcut (only lowercase characters can be used) and the second, the name of an operation, which must be an item in one of the menus.

- `%scicos_help`: manual pages associated with operations. A Scilab `tlist` where for each operation, a vector of strings contains the corresponding manual page.

- `%scicos_display_mode`: this scalar is used to specify whether backstoring for graphical display of the diagrams should be used by the editor.

- `modelica_libs`: a vector of strings containing the list of directories (full path), where implicit Modelica blocks are defined.

- `scicos_pal_libs`: a vector of strings containing the list of directories inside
  `SCI/macros/scicos_blocks` to be loaded. These directories contain the interfacing functions of the blocks in Scicos palettes. The user should not, in general, edit this variable.

These variables are initialized by the function `initial_scicos_tables`, which can be found in the directory `SCI/macros/util`. This function is executed as a script by `scilab.star`, which is executed at the startup of Scilab. They can be modified by the user subsequently.

The function `scicos` contains the main event loop of the editor. When an item in one of the menus is invoked to perform an operation, `scicos` calls the corresponding Scilab function. The name of the Scilab function is derived from the name of the menu item by removing special characters "`/`", "`.`", and "`-`", and adding a trailing "`_`". For example, the function corresponding to the item `Open/set` in the `Object` menu is `OpenSet_`. Functions associated with editor operations have no input-output arguments and are executed as scripts. Scicos comes with default menus providing many elementary editor operations. The associated functions can be found in the directory `SCI/macros/scicos`.

3.1.2 Structure of `scs_m`

`scs_m` is a Scilab object of type `diagram` having two entries:

- `props:` Diagram properties. Scilab object of type `params`,

- `objs:` list of objects included in the Scicos diagram.

Diagram Properties `params`

The diagram properties are:

- `wpar:` This vector is not currently used. It may be used in the future to code window sizes of the editor.
- `title:` A string containing the name of the diagram. The default value is `"Untitled"`
- `tol:` A vector containing simulation parameters including various tolerances used by the solver:
  - `atol`: Integrator absolute tolerance for the numerical solver.
  - `rtol`: Integrator relative tolerance for the numerical solver.
  - `ttol`: Tolerance on time. If an integration period is less than `ttol`, the numerical solver is not called.
  - `deltat`: Maximum integration time interval. If an integration period is larger than `deltat`, the numerical solver is called more than once in such a way that for each call the integration period remains below `deltat`.
  - `scale`: Real-time scaling; the value 0 corresponds to no real-time scaling. It associates a Scicos simulation time to the real time in seconds. A value of 1 means that each Scicos unit of time corresponds to one second.
  - `solver`: Choice of numerical solver. The value 0 implies `LSODAR` and 100 implies `DASKR`.
  - `hmax`: Maximum step size for the numerical solver. 0 means no limit.

  The default value is `[0.0001,1.000E-06,1.000E-10,100001,0,0]`.
- `tf:` Final integration time. The simulation stops at this time. The default value is `100000`.
- `context:` A vector of strings containing Scilab instructions defining variables to be used inside block GUIs. All valid Scilab instructions can be used but not comments.
- `void1:` Not used.
- `options:` Scilab object of type `scsopt` defining graphical properties of the editor such as background color and link color. The fields are the following:
  - `3D:` A list with two entries. The first one is a boolean indicating whether or not blocks should have 3D aspect. The second entry indicates the color in the current colormap to be used to create the 3D effect. The default is 33, which corresponds to gray added by Scicos

to the standard colormap, which contains 32 colors. The default value is `list(%t,33)`.

  - `Background:` Vector with two entries: background and foreground colors. The default value is `[8,1]`.
  - `Link:` Default link colors for regular and activation links. These colors are used only at link construction. Changing them does not affect already constructed links. The default value is `[1,5]`, which corresponds to black and red if the standard Scilab colormap is used.
  - `ID:` A list of two vectors including font number and sizes. The default value is `list([5,1],[4,1])`.
  - `Cmap:` An $n \times 3$ matrix containing RGB values of colors to be added to the colormap. The default value is `[0.8,0.8,0.8]`, i.e., the color gray.

- `void2:` Not used .
- `void3:` Not used.
- `doc:` Used for documenting the diagram.

Diagram Content

The field `objs` contains a Scilab list of the objects within the diagram. The objects can be of type `Block`, `Link`, or `Text`. A `Block` can be a basic block or a Super block.

Scicos Block It is a structure including the following fields.

- `graphics:` Scilab object of type `graphics` including graphical information concerning the features of the block. The fields are:
  - `orig:` Vector `[xo,yo]`, where `xo` is the x coordinate of the block origin and `yo` is the y coordinate of the block origin.
  - `sz:` Vector `[w,h]`, where `w` is the block width and `h` the block height.
  - `flip:` Boolean indicating block orientation. It is used to switch the ports on the lef- hand side and those on the right-hand side of the block.
  - `exprs:` A vector of strings including formal expressions (usually including numbers and variable names) used in the dialog of the block.
  - `pin:` Vector with `pin(i)`, the number of the link connected to the `i`th regular input port (counting from one), or 0 if this port is not connected.
  - `pout:` Vector with `pout(i)`, the number of the link connected to the `i`th regular output port (counting from one), or 0 if this port is not connected.

  - `pein:` Vector with `pein(i)`, the number of the link connected to the `i`th event input port (counting from one), or 0 if this port is not connected.
  - `peout:` Vector with `peout(i)`, the number of the link connected to the `i`th event output port (counting from one), or 0 if this port is not connected.
  - `gr_i:` Vector of strings including Scilab graphics expressions for drawing block's icon.
  - `id:` A string including an identification for the block. The string is displayed underneath the block in the diagram.
  - `in_implicit:` A vector of strings including `"E"` and `"I"`. `E` and `I` stand respectively for explicit and implicit port, and this vector indicates the nature of each input port. For regular blocks (not implicit), this vector is empty or contains only `E`'s.
  - `out_implicit:` Similar to `in_implicit` but for the output ports.

- `model:` Scilab object of type `model` including the following fields.

  - `sim:` A list containing two elements. The first element is a string containing the name of the computational function (C, Fortran, or Scilab). The second element is an integer specifying the type of the computational function. Currently type 4 and 5 are used, but older types continue to work to ensure backward compatibility.
  - `in:` A vector specifying the number and size of the first dimension of regular inputs.
  - `in2:` A vector specifying the number and size of the second dimension of regular inputs. *in* with *in2* formed the regular input sizes matrices.
  - `intyp:` A vector specifying the types of regular inputs. Its sizes is equal to the sizes of *in*.
  - `out:` A vector specifying the number and size of the first dimension of regular outputs.
  - `out2:` A vector specifying the number and size of the second dimension of regular outputs. *out* with *out2* formed the regular output sizes matrices.
  - `outtyp:` A vector specifying the types of regular outputs. Its sizes is equal to the sizes of *out*.
  - `evtin:` A vector specifying the number and sizes of activation inputs. Currently activation ports can be only of size one.
  - `evtout:` A vector specifying the number and sizes of activation outputs.
  - `state:` Vector containing initial continuous-time state.
  - `dstate:` Vector containing initial discrete-time state.

  - `odstate:` List containing initial object discrete-time state.
  - `rpar:` Vector of real parameters passed to associated computational function.
  - `ipar:` Vector of integer parameters passed to associated computational function.
  - `ipar:` List of object parameters passed to associated computational function.
  - `blocktype:` It can be set to `c` or `d` indifferently for regular blocks. `x` is used if we want to force the computational function to be called during the simulation phase even if the block does not contribute to computation of the state derivative.
  - `firing:` Vector of initial event firing times of size equal to the number of activation output ports. A value ≥ 0 programs an activation (event) at the corresponding port to be fired at the specified time.
  - `dep_ut:` Boolean vector `[timedep udep]`.
    - `timedep` boolean: true if block is *always active*.
    - `udep` boolean: true if block has direct feed-through, i.e., at least one of the outputs depends directly (not through the states) on one of the inputs. In other words, when the computational function is called with flag 1, the value of an input is used to compute the output.
  - `label:` A string. The label can be used to identify a block in order to access or modify its parameters during simulation.
  - `nzcross:` Number of zero-crossing surfaces.
  - `nmode:` Number of modes. Note that this gives the size of the vector mode and not the total number of modes in which a block can operate in. Suppose a block has 3 modes and each mode can take two values, then the block can have up to $2^3 = 8$ modes.
  - `equations:` Used in case of Implicit blocks.

- `gui:` The name of the Scilab GUI function associated with the block.
- `doc:` Used for documentation of the block.

Scicos Link. It is a Scilab list including the following fields:

- `xx:` A vector. A link is defined as a polyline line. `xx` defines the x-coordinate of the points characterizing the polyline.
- `yy:` A vector having the same size as `xx`. It defines the y-coordinate of the points characterizing the polyline.
- **id:** A string corresponding to the name of the function drawing the link. Default value is `"drawlink"`.

- **thick:** Vector of size two defining line thickness.
- **ct:** A vector. The first entry designates the color, and the second, the nature. The second entry is 1 for a regular link, -1 for an activation link, and 2 for an implicit link.
- **from:** Vector of size three including the block number, port number, and port type (0 for output, 1 for input) at the origin of the link. Note that the third entry may be 1 if the link is implicit; otherwise it is zero.
- **to:** Vector of size three including the block number, port number, and port type at the destination of the link.

Once the diagram has been successfully edited, `scs_m` can be passed on to the compiler.

3.2 Scicos Compiler

Scicos diagram compilation is done in two stages. These two stages are implemented by the Scilab functions `c_pass1` and `c_pass2`.

3.2.1 First Compilation Stage

The first stage of the compilation consists in removing the hierarchy from the diagram and constructing a flat description. This is done by `c_pass1`, which has the following calling sequence:

```
[blklst,cmat,ccmat,cor,corinv,ok,flgcdgen,scs_m,freof]=c_pass1(scs_m,flgcdgen)
```

The inputs of this function are defined by:

- `scs_m:` The structure of the diagram given by the editor.
- `flgcdgen:` This is used only when the compiler is called with the `Generation Code`, indicating the number of event input of the Super block. In other cases, this flag take the value -1.

The outputs are defined by:

- `blklst:` is a list of blocks present in the diagram. It contains block information relevant to simulation. Block properties such as color, icon, size, and location, which are not useful for simulation, have been stripped.
- `cmat:` is an $n \times 6$ matrix. Each row corresponds to a regular link and contains the block number, port number, and port type (explicit or implicit) of the source block and the same information regarding the destination block.
- `ccmat:` is an $n \times 4$ matrix. Each row corresponds to an activation link and contains the block number and port number of the source block and the block number and port number of the destination block.

- `cor` and `corinv` are correspondence tables (coded as lists) used to find the correspondence between blocks in `blklst` and `scs_m`.
- `flgcdgen`: It is used only when the compiler is called with the `Generation Code`. It is incremented if there is at least one `SampleCLK` in the super block's diagram.
- `scs_m`: This variable is modified only when the compiler is called with the `Generation Code` and the super block's diagram contains at least one `SampleCLK`
- `freof`: This variable is modified only when the compiler is called with the `Generation Code` and the super block's diagram contains at least one `SampleCLK`, it is a vector composed by the frequency and the offset of the major `SampleCLK`.

3.2.2 Second Compilation Stage

The second stage is done by the main compilation function `c_pass2`, which constructs all the scheduling tables and other information needed for simulation and code generation. The calling sequence is as follows:

```
%cpr=c_pass2(blklst,connectmat,ccmat,cor,corinv)
```

The input arguments are all generated by `c_pass1`. `connectmat` is simply `cmat`, but the third and last columns have been removed. The output `%cpr` is a Scilab structure containing all the information needed by the simulator. We will not explain how `c_pass2` computes `%cpr`. Instead we give a detailed description of `%cpr` in the next section.

3.2.3 Structure of `%cpr`

The Scilab object `%cpr` contains the result of the compilation. The simulator uses only `%cpr`. It is thus important for an advanced user to understand how compilation results are coded in `%cpr`.

- `state`: Scilab object of type `xcs`. It contains all the states of the model, that is, everything than can evolve during the simulation. It contains in particular
  - `x`: The continuous-time state, which is obtained by concatenating the continuous-time states of all the blocks.
  - `z`: The discrete-time state, which is obtained by concatenating the discrete-time states of all the blocks.
  - `oz`: The object discrete-time state, which is obtained by concatenating the object discrete-time states of all the blocks.
  - `iz`: Vector of size equal to the number of blocks. If a block needs to allocate memory at initialization, the associated pointer is saved here.

  - `tevts`: Vector of size equal to the number of activation sources. It contains the scheduled times for programmed activations.
  - `evtspt`: Vector of size equal to the number of activation sources. It is an event scheduler.
  - `pointi`: The number of the next programmed event.
  - `outtb`: Vector containing all the link memories. Link memories hold block output values.

- `sim`: Scilab object of type `scs`. It contains information that does not evolve during the simulation.
  - `funs`: A vector containing the name of the computational functions.
  - `xptr`: A vector pointer to the continuous time state `x`. The continuous-time state of block `i` is
    `%cpr.state.x(%cpr.sim.xptr(i):%cpr.sim.xptr(i+1)-1)`.
  - `zptr`: A vector pointer to the discrete-time state `z`, which is similar to `xptr`.
  - `ozptr`: A vector pointer to the object discrete-time state `z`, which is similar to `xptr`.
  - `zcptr`: A vector pointer to the zero-crossing surfaces.
  - `inpptr`: A vector pointer used to find the link number and consequently the part of `outtb` corresponding to a given input port.
  - `outptr`: Similar to `inpptr` but for output ports.
  - `inplnk`: Similar to `inpptr`.
  - `outlnk`: Similar to `inplnk` but for output ports.
  - `rpar`: Vector of real parameters that is obtained by concatenating the real parameters of all the blocks.
  - `rpptr`: A vector pointer to real parameters `rpar`. The real parameters of block `i` are
    `%cpr.sim.rpar(%cpr.sim.rpptr(i):%cpr.sim.rpptr(i+1)-1)`.
  - `ipar`: Vector of integer parameters, similar to `rpar`.
  - `ipptr`: A vector pointer to integer parameters, similar to `rpptr`.
  - `opar`: List of object parameters, similar to `rpar`.
  - `opptr`: A vector pointer to object parameters, similar to `rpptr`.
  - `clkptr`: A vector pointer to output activation ports.
  - `ordptr`: A vector pointer to `ordclk` designating the part of `ordclk` corresponding to a given activation.
  - `execlk`: Not used.
  - `ordclk`: An $n \times 2$ matrix associated to blocks activated by output activation ports. The first column contains the block number, and the second, the event code by which the block should be called.

  - `cord:` An $n \times 2$ matrix associated to always active blocks. The first column contains the block number, and the second, the event code by which the block should be called.
  - `oord:` Subset of `cord`, which affects the continuous-time state derivative.
  - `zord:` Subset of `cord`, which affects the computation of zero-crossing surfaces.
  - `critev:` A vector of size equal to the number of activations and containing zeros and ones. The value one indicates that the activation is critical in the sense that the continuous-time solver must be cold restarted.
  - `nb:` Number of blocks. Note that the number of blocks may differ from the original number of blocks in the diagram because `c_pass2` may duplicate some conditional blocks.
  - `ztyp:` A vector of size equal to the number of blocks. A 1 entry indicates that the block may have zero-crossings, even if it doesn't in the context of the diagram. Usually not used by the simulator.
  - `nblk:` Not used. Set to `nb`.
  - `ndcblk:` Not used.
  - `subscr:` Not used.
  - `funtyp:` A vector of size equal to the number of blocks indicating the type of the computational function of the block. Block type can be 0 through 5. Currently only type 4 (C language) and type 5 (Scilab language) computational functions should be used. But older blocks can also be used.
  - `iord:` An $n \times 2$ matrix associated to blocks that must be activated at the start of the simulation. This includes blocks inheriting from constant blocks and always active blocks.
  - `labels:` A string vector of size equal to the number of blocks containing block labels.
  - `modptr:` A vector pointer to the block modes.

- `cor:` Scilab list with a hierarchy identical to that of the original diagram and including block numbers in the compiled structure. It allows finding block number in `%cpr` from position in `scs_m`.

- `corinv:` Scilab list containing vectors. It allows finding position in `scs_m` from block number in `%cpr`.

3.2.4 Partial Compilation

To avoid long time compilation, Scicos does a partial compilation if minor modifications are made in a compiled diagram. The partial compilation is performed when changing block's paramters doesn't affect the scheduling tables, which means that some elements of %cpr such as %cpr.sim.opar are updated without recomputing scheduling tables %cpr.sim.ordclk, which can be time-consuming.

Anytime a block parameter is modified, Scicos determines the effect of the modification on the compiled structure by setting the level of compilation required. This level is coded in the Scilab variable needcompile. Four different levels of compilation are used in scicos:

- needcompile= 0: This level is coded for a freshly compiled diagram.
- needcompile= 1: This level is coded when a parameter changes the sizes of the link. In this case, pointer vectors may need to be recomputed. However, scheduling tables remain valid and are not recomputed.
- needcompile= 2: This level is coded when the modification is more complex and involves, for example, preprogrammed activation signals, then it still may not be necessary to recompute scheduling tables. In this case, Scilab function c_pass3 recomputes all the entries from scs_m, except for scheduling tables, before simulation.
- needcompile= 4: This level is coded when a diagram is not yet compiled, or when it is absolutely necessary, for example following any operation modifying the diagram (adding or deleting a block for example).

In many cases, changing a block parameter, for example the gain value of the Gain block, does not affect needcompile. The block is simply placed in the variable newparameters so that the new parameters of the block can be copied from scs_m into %cpr before simulation. This is done by the Scilab function modipar. To be note that if the diagram contains

The Scilab function do_update handles the partial compilation. Note that a recompilation can always be imposed by the user by choosing the menu item Compile.

3.3 Scicos Simulator

The interface to the Scicos simulator in Scilab is the function scicosim. This function is used to initialize, run, and end Scicos simulations. These operations are done by the function do_run (in the SCI/macros/scicos directory), which is called when the user runs a simulation in Scicos.

The Scicos simulator is a complex C program. We do not study it here. The simulation functions are in the file SCI/routines/scicos/scicos.c. The main function is scicos, which calls the routines cosini, cossim, and cosend depending on whether initialization, simulation, or ending has been requested.

3.4 Conclusion

In this report, we explain the construction of a new basic scicos block by introducing its interface and computational function. Then we give a brief description on the structure of Scicos describing it's editor and compilor. The informations of this report are basicly taken from the book `Modeling and Simulation in Scilab/Scicos` writen by *Stephen L. Campbell, Jean-Philippe Chancelier* and *Ramine Nikoukhah*, published by `Springer Science+Business Media.Inc.2006`.