Proceedings of the
2005 IEEE Conference on Control Applications
Toronto, Canada, August 28-31, 2005

New features and new challenges in modeling and simulation in Scicos

Masoud Najafi Ramine Nikoukhah Serge Steer Sebastien Furic

Abstract— **The dynamic system simulator Scicos, has been extended to partially support the Modelica language. Modelica is a programming language for modeling dynamical systems at component level. Adding this feature has introduced several challenges at compiling and simulation levels which will be described in this paper.**

KEYWORDS: Dynamic system simulation; Simulation software; Scicos; Modelica

I. Introduction

Scicos is a Scilab[1] toolbox for modeling and simulation of dynamical systems [1], [2]. Scicos provides a hierarchical graphical editor for the construction of complex dynamical systems, a simulator and a code generator. For many applications, the Scilab/Scicos environment provides a free open-source alternative to Matlab/Simulink and MatrixX [3], [4], [5].

Standard Scicos is not well suited for physical component level modeling [6]. For example, when modeling an electrical circuit, it is not possible to construct a Scicos diagram with a one to one correspondance between the electrical components (resistor, diode, capacitor,...) and the blocks in the Scicos diagram. In fact, the Scicos diagram does not resemble the original electrical circuit . Consider the electrical circuit depicted in Figure 1. This circuit contains a voltage source, a resistor and a capacitor. To model and simulate this diagram in Scicos, we have to express the dynamics explicitly. We can use Kirkhoff's law which states that the loop voltage around a circuit must add up to zero, we obtain

$$V_s + Ri + \frac{1}{C}\int i\,dt = 0 \qquad (1)$$

where V_s is the voltage across the votage source and i the current through the circuit. The voltage V across the capacitor, is given by:

$$V = -\frac{1}{C}\int i\,dt. \qquad (2)$$

M. Najafi, R. Nikoukhah, S. Steer are with INRIA-Rocquencourt, Domaine de Voluceau, 78153 Le Chesnay Cedex, France

S. Furic is with the IMAGINE SA, 5, rue Brison, 42300 Roanne, France. www.amesim.com

Corresponding Author: ramine.nikoukhah@inria.fr

[1]Scilab is a free open-source software for scientific computation, see www.scilab.org and www.scicos.org.

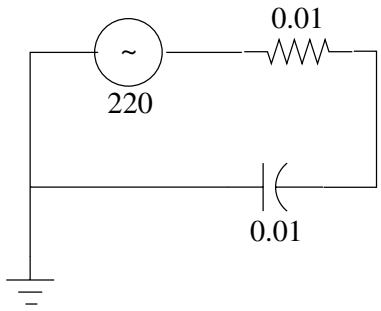

Fig. 1. A simple electrical circuit.

We can now implement these two equations in Scicos by using an integrator block receiving i as input. The resulting Scicos diagram is depicted in Figure 2. Note that this Scicos diagram does not look anything like the original electrical circuit in Figure 1.

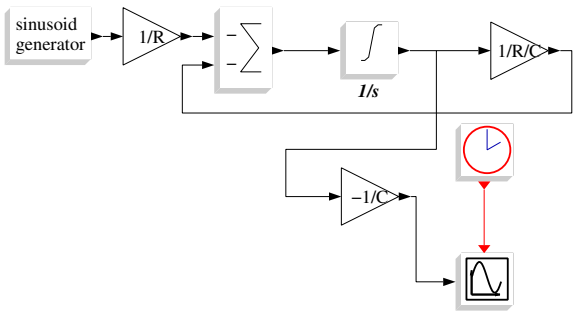

Fig. 2. Scicos diagram realizing the dynamics of the electrical circuit in Figure 1.

Recently an extension of Scicos has been developed to allow modeling of physical components directly within the standard Scicos diagrams. This has been done, in particular, by lifting the causality constraint on Scicos' blocks and by introducing the possibility of describing block behaviors in the Modelica language. This extension allows us to model naturally not only systems containing electrical components but also mechanical, hydraulic, thermal, and other systems. For example, the electrical circuit in Figure 1 can be modeled and simulated by constructing the Scicos diagram in Figure 3. The electrical components

0-7803-9354-6/05/$20.00 ©2005 IEEE

come from the `Electrical` palette.

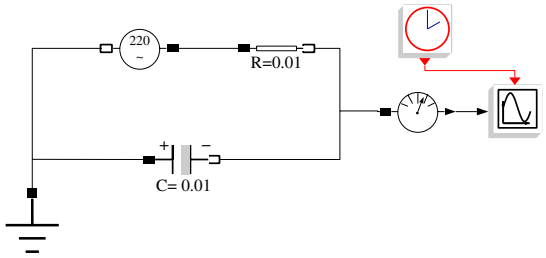

Fig. 3. Scicos diagram using Modelica component.

II. Implicit blocks

Implicit blocks have implicit ports. An implicit port is different from an input or an output port in that connecting two such ports imposes a constraint on the values at these ports but does not imply the transfer of information in an a-priory known direction as it is the case when we connect an output port to an input port. For example, the implicit block `Capacitor` used in diagram of Figure 3 has current and voltage values on its ports but there is no way a-priori, without analyzing the full diagram, to designate any of them as input or output.

Implicit blocks and the construction of diagrams based on these blocks is fully in the spirit of the object oriented Modelica language [12]. Even the description of block behavior is done in Modelica[2]

Links are constructed as in the regular case. It is, of course, not possible to connect an implicit port to an explicit port (it would be meaningless anyway). The connection to the explicit world can only come from the implicit blocks by having one or more explicit ports such as the voltage sensor in Diagram 3.

Implicit blocks give a nice facility to model physical systems without worrying about simplifying the equations and making them explicit; it is done automatically, all that the user should do is selecting the components and connecting them. Implicit blocks are essential for constructing models which include physical components such as resistors, capacitors, etc., in electricity, or pipes, nozzles, etc., in hydraulics. They are also useful in many other areas such as mechanics and thermodynamics.

Contrary to explicit blocks, whose behavior is given by an ODE or a DAE, implicit blocks cannot be modeled as black box objects. The equations realizing the behavior of an implicit block must be available to the compiler for system reduction and code generation. To describe the behavior of these blocks in Scicos, the Modelica language has been adopted.

[2]For the moment only a subset of the language is implemented.

III. Modelica language

Modelica is an object oriented programming language for modeling physical systems. To model a complex system in Modelica, first the model of the composing components should be developed and then the components can be connected to construct the overall model. As an example, suppose that you need to model the electrical system given in Fig. 4

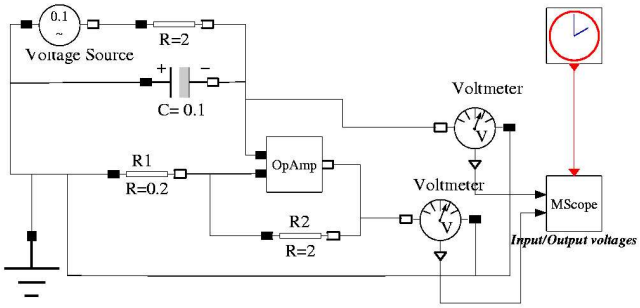

Fig. 4. Scicos diagram containing the new operational amplifier block.

The first step is to model the basic electrical components in Modelica. The Modelica models of ground, capacitor, operational amplifier, and sinusoidal voltage source follow.

```
class Ground "Ground"
  Pin p;
equation
  p.v = 0;
end Ground;

class Capacitor
  Pin p, n;
  Real v;
  Real i;
  parameter Real C=0.1 "Capacitance";
equation
  C*der(v) = i;
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end Capacitor;

class IdealOpAmp3Pin
  Pin in_p "Positive pin of the input port";
  Pin in_n "Negative pin of the input port";
  Pin out "Output pin";
equation
  in_p.v = in_n.v;
  in_p.i = 0;
  in_n.i = 0;
end IdealOpAmp3Pin;

class VsourceAC "Sin-wave V-source"
  Pin p, n;
  Real v;
  Real i;
  parameter Real VA=220 "Amplitude";
  parameter Real f=50 "Frequency";
  parameter Real PI=3.1415926 "PI";
equation
  v = VA*2*PI*f*time;
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end VsourceAC;
```

To construct the complete model, the component are connected as they are connected in a real electrical circuit. This can be compared to the way an engineer models an electrical circuit in SPICE[3]. The following listing is the Modelica program for the model of Circuit of Fig 4. As we will see in section V this code is automatically generated by Scicos from the diagram of Fig 4.

```
class imppart_test
  parameter Real P1;
  parameter Real P2;
  parameter Real P3;
  parameter Real P4;
  parameter Real P5;
  parameter Real P6;
  parameter Real P7;
  Ground        B1;
  VoltageSensor         B2;
  VsourceAC     B3(VA=P1, f=P2);
  Resistor      B4(R=P3);
  IdealOpAmp3Pin        B5;
  Resistor      B6(R=P4);
  Resistor      B7(R=P5);
  VoltageSensor         B8;
  Capacitor     B9(C=P6, v(start=P7));
  OutPutPort    B10;
  OutPutPort    B11;
equation
  connect (B9.n,B4.n);
  connect (B5.in_p,B4.n);
  connect (B2.n,B4.n);
  connect (B6.p,B1.p);
  connect (B3.p,B1.p);
  connect (B9.p,B1.p);
  connect (B8.p,B1.p);
  connect (B2.p,B1.p);
  connect (B7.n,B5.out);
  connect (B8.n,B5.out);
  connect (B5.in_n,B6.n);
  connect (B7.p,B6.n);
  connect (B4.p,B3.n);
  B2.v = B10.vi;
  B8.v = B11.vi;
end imppart_test;
```

IV. Scicos/Modelica interaction

Even though Modelica is a rich language having the capacity to handle continuous-time and discrete-time behaviors, currently, we are mainly using Modelica and implicit blocks to model continuous-time dynamics; only minimal support is provided for discrete-time behavior. The discrete-time behavior, in the Scicos environment, is provided via explicit blocks.

The addition of implicit blocks has been done without changing signifi cantly Scicos formalism. Even though implicit blocks can be used anywhere inside a Scicos diagram, they are grouped and replaced with a single block in a precompilation phase [6] as shown in Fig. 5. The mechanism, which can be compared to the way an `AMESim`[4] or `Dymola`[5] model is integrated in Simulink, is completely transparent to the user.

Consider for example the Scicos diagram in Fig. 5. Here we have a fluid level control system. To model this system in a natural way, a hydraulic source, a regulated valve, a

[3]www.eecs.berkeley.edu/

[4]www.amesim.com

[5]www.dymola.com

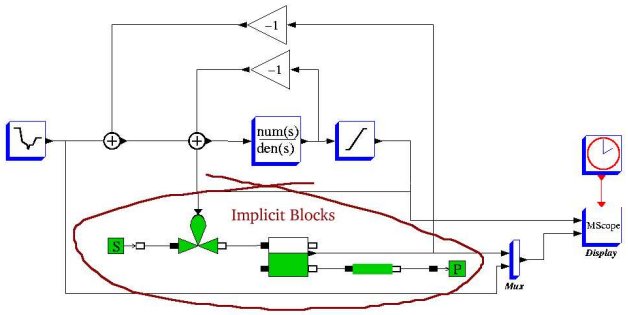

Fig. 5. Scicos diagram containing both types of blocks.

container, a tube, and a well have been used. The container has a built-in level sensor which makes the interface with the explicit part of the system, similarly the valve is regulated through an input signal from the explicit part of model. The controller and the display mechanism have been implemented using explicit blocks and the blocks in gray are implicit blocks that have been developed in Modelica language.

A. Scicos compiler

Implicit and regular Scicos blocks can be used in the same diagram. The way implicit blocks are handled by the Scicos editor is similar to the way regular blocks are. The compilation is again transparent for the user, however, the compiler performs a fi rst stage compilation by grouping all the implicit blocks into a single internally implicit block (see Appendix). This is done by generating a Modelica program for the implicit part of the diagram.

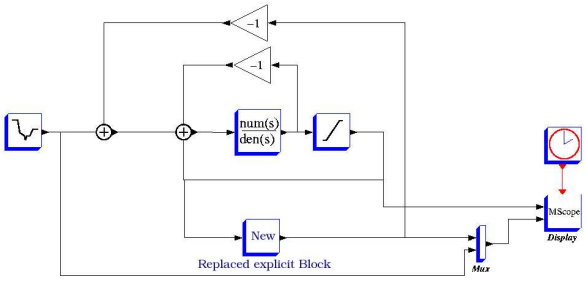

Fig. 6. Scicos diagram of Fig 5; implicit parts grouped into one explicit block.

The generated code expresses the behavior of the implicit part and is saved in a temporary fi le. The imppart_rlc.mo cited in section III is the Modelica model generated automatically by Scicos.

This fi le is then processed by `Modelicac` [6] which translates that Modelica code into a C code describing

[6]A Modelica compiler and C code generator written in `Objective Caml` and included in the Scilab distribution.

the behavior of the implicit part of the model. Once the C code is compiled (this requires a C compiler) and incrementally linked with Scilab, Scicos sees this new block as a standard explicit block. In fact the new explicit (internally implicit) block replaces the implicit part of the diagram.
At the end of this procedure, the model is composed of only explicit blocks and can be compiled and simulated as usual. To illustrate our method, a flowchart given in Fig. 7 shows the compile process in Scicos.

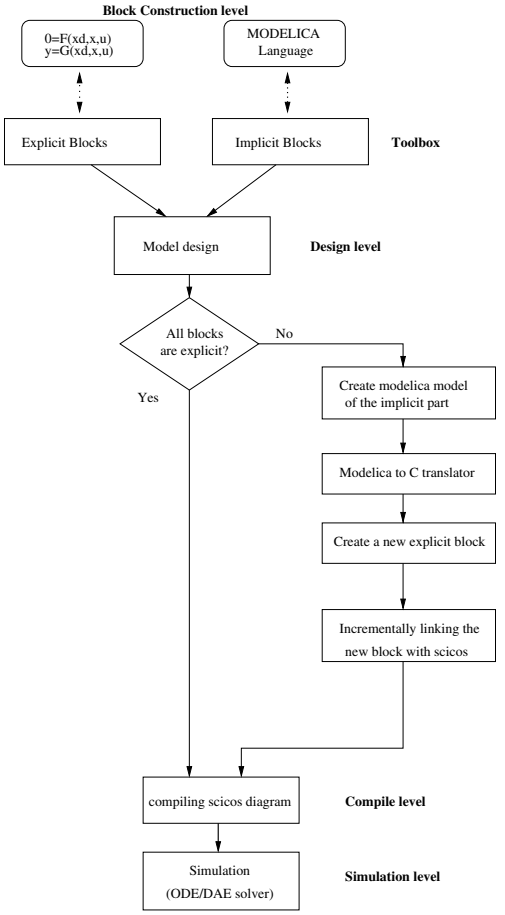

Fig. 7. Scicos' Compiling and simulation flowchart

V. Modelicac, a MODELICA COMPILER

Modelicac (acronym of "Modelica compiler") is a compiler for the subset of the Modelica language we felt necessary to handle in order to cover the needs of simulating of hybrid dynamical systems[7] in Scicos. Modelicac is an external tool, i.e. it is independent of Scilab, so one may use it like an ordinary compiler e.g., like a C compiler. By default, Modelicac comes with a module that generates C code for the Scilab target. However, since Modelicac is free and open source, it is possible to develop code generators for other targets as well.

A. Modelicac *development*

Modelicac has been developed in Objective Caml[8] which is a general purpose programming language developed at INRIA since 1985. This language is distributed with two compiler-development tools (Ocamllex and Ocamlyacc) which offer facilities to build compilers. Furthermore the Objective Caml compiler is free and open source, that is why we adopted it to develop Modelicac [13].

B. *Modelica compilation using* Modelicac

Modelicac is invoked for two purposes: compiling basic models from libraries and generating code for the target simulation environment. To fulfill the first task, like generating an object file with a C compiler, Modelicac is invoked with the appropriate options from the command line to generate an object file with "*.moc" extension. The second task of Modelicac is compiling the "main" Modelica model (here provided by Scicos) and generating a code for the target (here, a C code). In this phase instead of generating an object file, Modelicac performs several simplification steps to generate a code as compact as possible. In Fig. 8 a flowchart shows how Modelicac generates a C file from modelica model of a Scicos diagram.

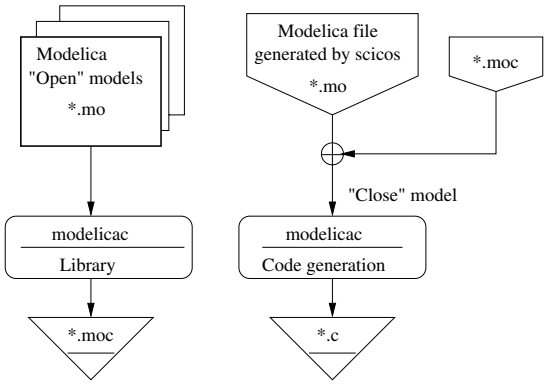

Fig. 8. Modelicac translation flowchart

C. *Modelica source files*

Modelica source files must contain only one class declaration, introduced either by the "class" keyword or by the "function" keyword. So a Modelica source file may define one of the following things:

[7] A dynamical system composed of discrete-event and continuous time blocks

[8] caml.inria.fr

• An ”open” model is a model with free variables. There are more variables than equations (e.g. the model of a resistor in electrical library). The open models are introduced by the ”class” keyword,
• A ”close” model is a model with equal number of variables and constraints. It is also introduced by the ”class” keyword,
• An external function, introduced by the ”function” keyword.

It should be noted that only closed models can be simulated. To compile the ”close” model `Modelicac` searches the classes used in the current compilation directory and also in user-defined directories.

The following source code describes a simple resistor defined in ”Resistor.mo” file:

```
class Pin
   Real v;
   flow Real i;
end Pin;

class Resistor
   Pin p, n;
   parameter Real R "Resistance";
equation
   R*p.i = p.v - n.v;
   p.i + n.i = 0;
end Resistor;
```

An instance of ”Resistor” has two ”connectors” (”p” and ”n”), which have their own potential and flow variables (here, the voltage and the current, respectively). A resistor has also a resistance parameter imposed by the component through the first equation. The second equation simply states that the current that flows in the resistor through ”p” is equal to the current that flows out of ”n”.

D. A code generation example

In section III we saw Modelica source codes of some electrical components from electrical library. Here we will show how to use these models to construct and compile elaborated electrical models with `Modelicac`.

In order to perform the simulation of an electrical circuit one normally has to describe the circuit using Modelica by defining the components involved (i.e. giving their names and the value of their parameters) and the connections to establish. Then, `Modelicac` should be invoked with the appropriate options and arguments. This task is done by Scicos, provided the appropriate library exist in Scicos.

In fact it is not necessary to write any Modelica code to build a circuit: one can assemble components using the Scicos editor and then Scicos automatically builds the Modelica source code from the graphical specification and invokes `Modelicac` to convert Modelica code into C code. Fig. 4 contains a model of an electrical circuit modeled in Scicos. Its Modelica class is automatically generated by Scicos.

For this model `Modelicac` generates a C code. This C code is incrementally linked with Scicos to be used as a standard block. Here is the of generated code for the model in Fig. 4. The code fragment inside *” if (flag == 0)”* represents the residual computations for DAE, in *” if (flag == 1)”* it computes the outputs of block, in *” if (flag == 7)”* it specify which states are static and which ones are dynamic, and the code inside *” if (flag == 10)”* computes the analytical Jacobian of the DAE.

```
/*
number of discrete variables = 0
number of variables = 3
number of inputs = 0
number of outputs = 2
number of modes = 0
number of zero-crossings = 0
I/O direct dependency = false
*/

#include <math.h>
#include <scicos/scicos_block.h>


void imppart_test(scicos_block *block, int flag)
{
   double *rpar = block->rpar;
   double *z = block->z;
   double *x = block->x;
   double *xd = block->xd;
   double **y = block->outptr;
   double **u = block->inptr;
   double *g = block->g;
   double *res = block->res;
   int *jroot = block->jroot;
   int *mode = block->mode;
   int nevprt = block->nevprt;
   int property[3];

   /* Intermediate variables */
   double v0;

   if (flag == 0) {
       v0 = -x[2];
       res[0] = sin(6.2831853*get_scicos_time()
              *rpar[1])*rpar[0]+v0-rpar[2]*x[0];
       res[1] = v0-rpar[3]*x[1];
       res[2] = xd[2]*rpar[5]-x[0];
   } else if (flag == 1) {
       if (get_phase_simulation() == 1) {
           y[0][0] = x[2];
           y[1][0] = x[2]-x[1]*rpar[4];
       } else {
           y[0][0] = x[2]; /* main.B10.vo */
           y[1][0] = x[2]-x[1]*rpar[4];
       }
   } else if (flag == 2 && nevprt < 0) {
   } else if (flag == 4) {
       x[0] = 0.0; /* main.B4.n.i */
       x[1] = 0.0; /* main.B6.n.i */
       x[2] = rpar[6]; /* main.B9.v */
       Set_Jacobian_flag(1);
   } else if (flag == 6) {
   } else if (flag == 7) {
       property[0] = -1; /* main.B4.n.i (algebraic variable)*/
       property[1] = -1; /* main.B6.n.i (algebraic variable)*/
       property[2] = 1; /* main.B9.v (state variable)*/
       set_pointer_xproperty(property);
   } else if (flag == 9) {
   } else if (flag == 10) {
       res[0] = -rpar[2];
       res[1] = 0.0;
       res[2] = -1.0;
       res[3] = 0.0;
       res[4] = -rpar[3];
       res[5] = 0.0;
       res[6] = -1.0;
       res[7] = -1.0;
       res[8] = rpar[5]*Get_Jacobian_parameter();
       res[9] = 0.0;
       res[10] = 0.0;
       res[11] = 0.0;
```

```
        res[12] = -rpar[4];
        res[13] = 1.0;
        res[14] = 1.0;
        set_block_error(0);
    }
    return;
}
```

VI. Simulation

At the end of compilation phase, simulation is performed. For the design in Fig 4, the output of the Mscope is illustrated in Fig 9.

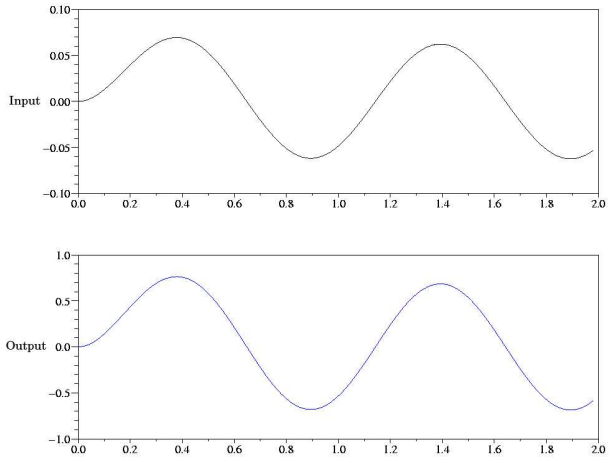

Fig. 9. Simulation result of circuit in Figure 4.

Note that according to

$$Gain = 1 + R2/R1$$

the input signal is amplified by afactor of 11.

Conclusion

In this paper we have the new extention of Scicos which allows more natural modeling of components using the Modelica language. Another modification has been the use of analytical Jacobian in Scicos. For some non-linear models, the numerical Jacobian provided by DASKR is not accurate enough and the integration fails even for low dimension systems. This problem can be avoided by using analytical Jacobian to enhance the simulation performance. The integration of analytical Jacobian will be subject of a futur paper

References

[1] C. Bunks, J. P. Chancelier, F. Delebecque, C. Gomez(ed.), M. Goursat, R. Nikoukhah and S. Steer, *Engineering and Scientific Computing with Scilab*, Birkhauser, 1999.

[2] J. P. Chancelier, F. Delebecque, C. Gomez, M. Goursat, R. Nikoukhah, and S. Steer, *Introduction à Scilab*, Springer-Verlag, 2002.

[3] R. Nikoukhah and S. Steer, *Hybrid systems: modeling and simulation*, COSY: Mathematical Modelling of Complex System, Lund, Sweden, Sept. 1996.

[4] A. Benveniste, *Compositional and Uniform Modeling of Hybrid Systems*, IEEE Trans. Automat. Control, AC-43, 1998.

[5] R. Nikoukhah and S. Steer, *Scicos: A Dynamic System Builder and Simulator, User's Guide - Version 1.0*, INRIA Technical Report, RT-0207, June 1997.

[6] M. Najafi, A.Azil, and R. Nikoukhah, *Extending scicos from system to component level simulation*, ESMc2004 international conference, Paris, France, October 2004.

[7] M. Najafi, R. Nikoukhah, S. L. Campbell, *Computation of consistent initial conditions for multi-mode DAEs: Application to Scicos*,Proc. Computer Aided Control Design, Taipei, 2004

[8] K. E. Brenan, S. L. Campbell, and L. R. Petzold, *Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations*, SIAM publication, Philadelphia, 1996.

[9] A. C. Hindmarsh, "LSODE and LSODI, Two New Initial Value Ordinary Differential Equation Solvers", *ACM-Signum Newsletter*, Vol. 15, 1980, pp. 10–11.

[10] L. R. Petzold, "Automatic selection of methods for solving stiff and nonstiff systems of ordinary differential equations", *SIAM J. Sci. Stat. Comput.*, No. 4, 1983.

[11] L. R. Petzold. "A Description of DASSL: A Differential/Algebraic System Solver", *In Proceedings of the 10th IMACS World Congress*, Montreal, 1982, pp. 8-13.

[12] P. Fritzson,*Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, Wiley-IEEE Press, 2004

[13] Weis, Pierre , Leroy, Xavier, *Le Langage CAML*, 2nd ed , Dunod Press, 1999

Appendix

Implicit blocks are identical to standard Scicos blocks (explicit blocks) except for the fact that the internal continuous-time dynamics are allowed to be implicit. Internally implicit blocks have explicit input-output ports and are used in Scicos exactly the same way as standard blocks are. This means, in particular, that the editor and, except for some minor modifications, the compiler were not affected by the introduction of these blocks. But the simulator had to be extended because a diagram containing even a single internally implicit block results in a system of Differential Algebraic Equations (DAE) which cannot be solved with an ODE solver. It is for this reason that DAE solver DASKR had been interfaced with Scicos simulator. DASKR was chosen both because of its ability to solve many DAEs but also because of the root finding option which is important for hybrid systems [9], [10], [11]

Solving DAE systems poses specific problems, in particular, that of finding consistent initial conditions. In the case of a DAE, and in particular an internally implicit block, the state is not just x but the pair $(x,\dot{x})$, and not all pairs of $(x,\dot{x})$ are consistent with the system equations. Finding consistent initial conditions can be a complex problem in some cases. This is particularly difficult in the multimode case which is often encountered in the Scicos environment [7]. To help the DAE solver find consistent initial conditions, an internally implicit block must be able to furnish information about the nature of each state variable. It should specify in particular which state variables are static and which ones are dynamic [8]. This is done by the computational function of the block when it is called with flag 7.