SCICOS: a general purpose modeling and simulation environment

M. Najafi[‡] , S. Furic[*], R. Nikoukhah [†‡]

Abstract

Partial support for Modelica is now provided by the general purpose dynamical system simulator Scicos. In particular it is now possible to use component models in Scicos diagrams where the dynamics of the component has been described in Modelica. This paper presents this new extension of Scicos.
KEYWORDS: Dynamic system simulation; Simulation software; Component level modeling; Scilab; Modelica

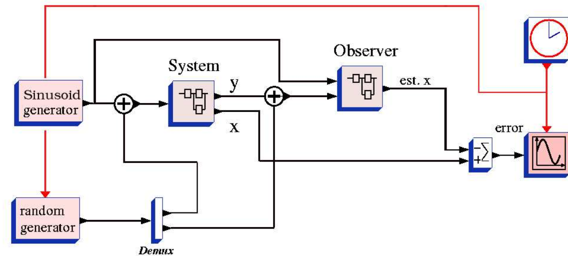

Figure 1: A system modeled in Scicos.

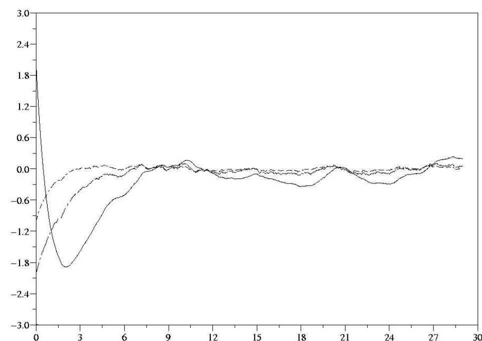

Figure 2: Simulation result of model of Fig. 1 in Scicos.

1 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.
Very general dynamical systems, including hybrid systems, can be modeled in Scicos [3, 4, 5, 6]. A typical Scicos diagram is presented in Fig. 1. This diagram is used to evaluate the performance of an observer by simulation; the simulation results are given in Fig. 2.
The model of Fig. 1 is composed of ”explicit” blocks, i.e., block with explicitly identified inputs and outputs. Modeling with such blocks is called system level modeling. Component level modeling, on the other hand, allows the use of ”implicit” blocks which are blocks with port connections which a-priori are not labeled as inputs or outputs [7]. 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. In Modelica community implicit blocks are called acausal [13].

Contrary to explicit blocks, 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.

*IMAGINE (www.amesim.com)
†Corresponding author (ramine.nikoukhah@inria.fr)
‡M. Najafi, R. Nikoukhah are with INRIA-Rocquencourt, Domaine de Voluceau, 78153 Le Chesnay Cedex, France
[1]Scilab is a free open-source software for scientific computation, see www.scilab.org and www.scicos.org.

2 Modelica and Scicos

Even though Modelica is a rich language having the capacity to handle continuous-time and discrete-time behaviors, for the start, 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 significantly 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 [7]. The mechanism, which can be compared to the way an `amesim`[2] or `Dymola`[3] model is integrated in Simulink, is completely transparent to the user.

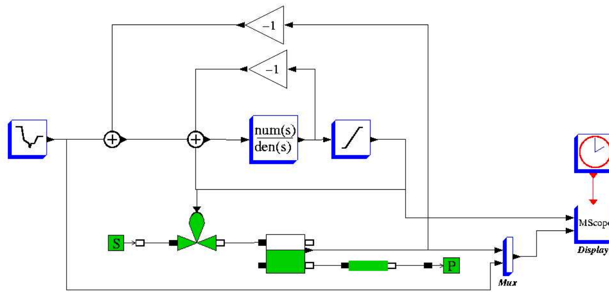

Figure 3: Scicos diagram containing both types of blocks.

Consider for example the Scicos diagram in Fig. 3. Here we have a fluid level control system. To model this system in a natural way, a hydraulic source, a regulated valve, a 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.

2.1 Scicos architecture

To illustrate our method, a simple flowchart given in Fig. 4 shows how Scicos and Modelica interact. A designer can select blocks from either standard or implicit toolboxes. Blocks in implicit toolbox have been developed using Modelica language. As shown in the flowchart, if the model contains an implicit block, after a series of automatic preprocessing steps, implicit part of model is abstracted into a standard block with explicit input/outputs; the resulting model can then be simulated by Scicos [7].

[2]www.amesim.com
[3]www.dymola.com

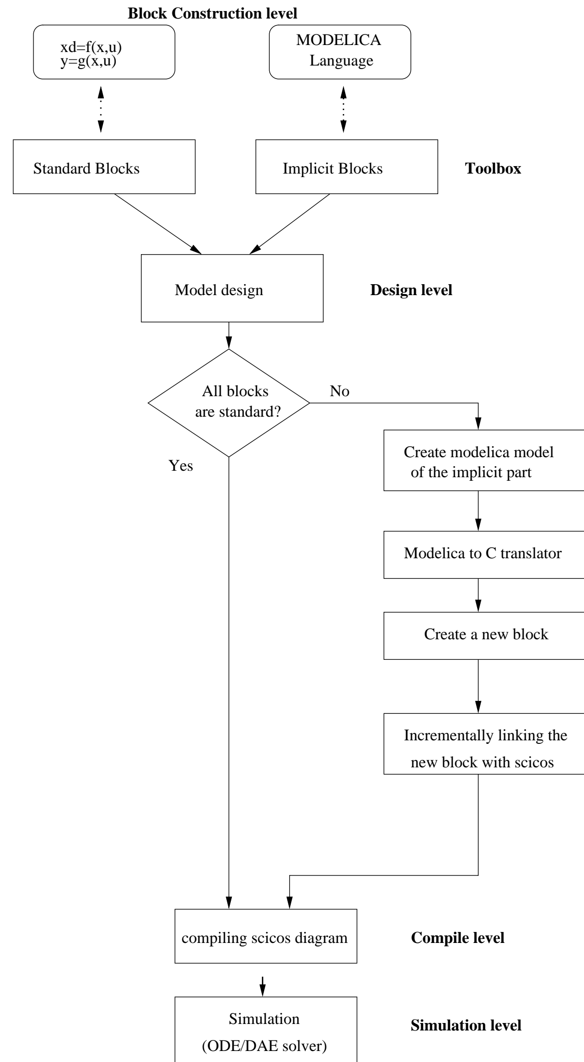

Figure 4: system flowchart

2.2 Available implicit toolboxes

To be able to use implicit blocks in addition to explicit ones in Scicos, several new features have been added to Scicos. So far, only two palettes with implicit blocks are available for testing purposes: the electrical and the thermodynamics palettes. The thermo-hydraulic toolbox and available blocks are shown in Fig. 5.

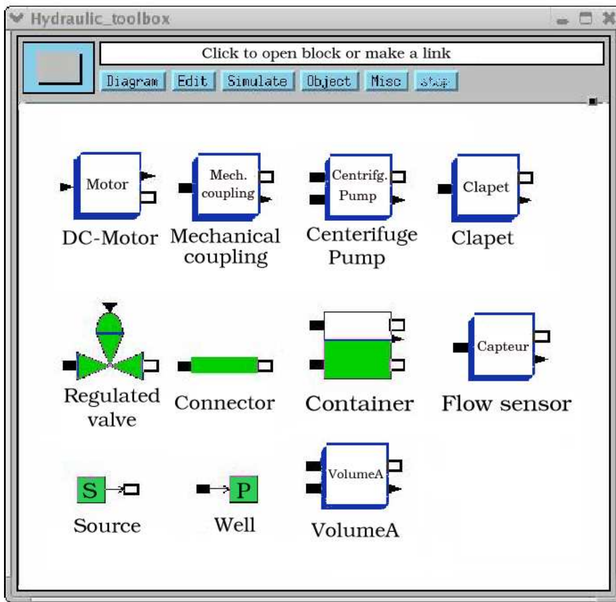

Figure 5: Thematic toolbox

2.3 New link and port type

Implicit blocks or components are interfaced via special links associated with physical quantities such as current or voltage in electronics, or, flow or pressure in hydraulics. It would be meaningless for a link representing a voltage to be connected to another link representing the output value of a PID controller. To distinguish between these two, two different link types have been defined: explicit and implicit links that interconnect explicit and implicit ports respectively. In Fig. 6 we have a hydraulic container which has four implicit ports (marked IP) representing liquid outlets and an explicit port (marked EO) representing a liquid level sensor output.

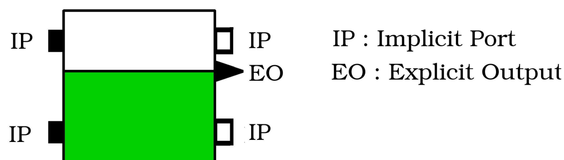

Figure 6: An implicit block can have implicit and explicit ports.

2.4 Compiling a mixed diagram

To compile and simulate a model containing implicit blocks, Scicos groups all implicit blocks into a single block having explicit inputs and outputs. Then it generates a Modelica code expressing the behavior of the new block and save it in a temporary file. This file is then processed by `modelicac` [4] which translates this Modelica code into a C-code describing the behavior of the new Scicos block. Once the C-code is compiled and incrementally linked in Scilab, Scicos sees this new block as a standard explicit block; see Fig. 7. At the end of this procedure, the model is no longer implicit because all blocks are standard explicit blocks, so the model can be compiled and simulated as usual. It should be noted that this procedure is completely transparent to the user [7].

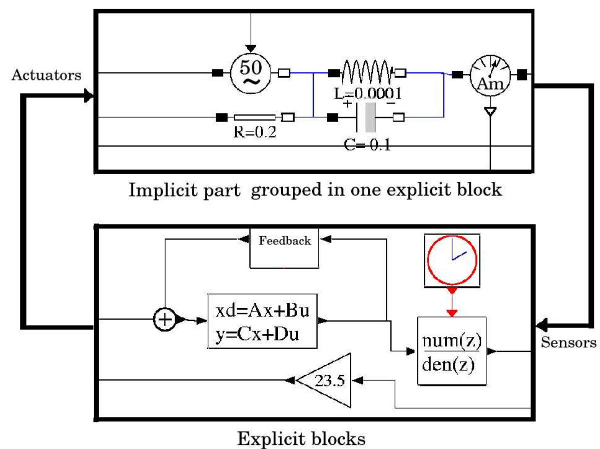

Figure 7: In a precompilation phase, all implicit blocks are grouped to form an explicit block.

2.5 New numerical Solver

Most of the time, after generating C code for implicit part of the model, we end up with a set of Differential-Algebraic Equations (DAE) that no longer can be integrated via ordinary differential equation solvers. It is for this reason that the DAE solver DASKR [8, 9, 10, 11, 12] has been incorporated into Scicos.

3 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 hybrid models under Scicos. Modelicac is an external tool, i.e. it is independent of Scilab, so

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

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 a code generator for another target.

3.1 Modelicac development

Modelicac has been developed in Objective Caml[5] which is a functional programming language developed at INRIA since 1985. This language is distributed with two compiler-development tools (i.e., Ocamllex and Ocamlyacc) that offer some nice facilities to build compilers. Furthermore the Objective Caml compiler is free and open source, that's why we adopted it to develop modelicac [16].

3.2 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 to be used later. 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 simple flowchart shows how modelicac generates a C file from modelica model of a Scicos diagram.

3.3 Supported Modelica subset

As said previously, the current version of modelicac (1.x.x) does not handle the full set of Modelica language constructs. It actually allows only the description of physical models at "equation" level. A physical model is built as the aggregation of sub-models or basic types with constraints between variables, and explicit event declarations ("When"). Currently modelicac has the following main limitations:

- Only "Real" data type is supported.
- Inheritance is not currently supported.
- "Algorithm" is not supported but it can be defined as an external C function.

[5] caml.inria.fr

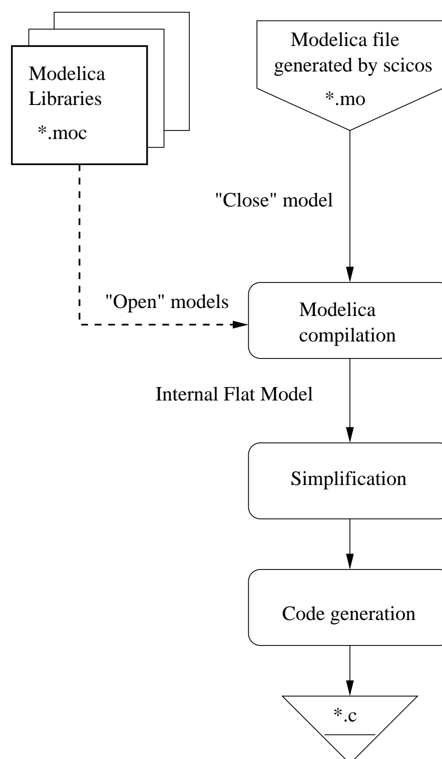

Figure 8: Modelicac translation flowchart

3.3.1 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:

- 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.

Of course, only closed models can be simulated. In order to find classes in the host file system hierarchy, it is required that the name of the file be the same as the name of the enclosed class. To compile the "close" model modelicac searches the libraries used in the current compilation directory and also in user-defined directories.

The following source code describes a simple resistor enclosed in a "Resistor.mo" file:

```
class Resistor
  Pin p, n;
  parameter Real R "Resistance";
```

```
equation
  R*p.i = p.v - n.v;
  p.i = -n.i;
end Resistor;
```

An instance of ”Resistor” has two ”connectors” (”p” and ”n”), that have their own potentials and flows 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 through ”n”.

3.4 Model simplification

The following tasks are fulfilled by modelicac to simplify and generate a C source file from a Modelica source code and library object code files:

• Obtaining a flat model by replacing an aggregation of sub-models by the set of all their variables and equations merged together and replacing connection equations by ordinary equations. Symbolic manipulations in modelicac are performed using classical acyclic graph manipulation techniques

• Simplification of trivial or unnecessary equations using symbolic manipulations e.g. in the following system

$$\begin{cases} \cos(x)+\sin(y)=0 \\ \cos(x)-\sin(y)=0 \\ z-x-y=0 \\ f(x,y,z,v)=0 \end{cases}$$

the first two equations are fully non-linear and only the numerical solver can solve the system for x and y. But the third equation is trivial and z can be obtained in terms of x and y, so in the rest of equations z is replaced by $x+y$. Most of the variables used to connect Modelica components (”connection variables”), are eliminated in this way.

• Causality analysis, i.e. computation of system's Jacobian matrix. It will be explained further.

3.4.1 Causality analysis

Causality analysis performs a few operations in order to find the so-called ”strongly connected components” of a system of equations viewed as a directed bipartite graph [13]:

1. Constructing a bipartite graph whose nodes on the left represent variables in the system and whose nodes on the right represent constraints between variables (i.e., equations). There is an edge between a left-side node and a right-side node if and only if the variable represented by the left-side node appears in the equation represented by the right-side node.

2. Finding a coupling (using the Ford and Fulkerson method for instance)

3. Giving the edges an orientation depending on the results of the previous step. Edges that link two coupled nodes are all oriented in a given direction (either left-to-right or right-to-left) and the other ones in the opposite direction.

4. Finding strongly connected components in the resulting oriented graph (using Tarjan's algorithm for instance)

5. Sorting the resulting nodes in a topological order.

Each strongly connected component represents a subsystem of the whole system and it is now possible to perform symbolic simplification steps in order to reduce the number of variables in the system.

3.4.2 Modelicac simplification strategy

Symbolic simplifications typically involve variants of the Gauss method (to solve linear systems) and simple symbolic simplification methods based on a set of predetermined patterns (for efficiency reasons) to try to solve the remaining equations. In modelicac we focused on the second class of simplification methods. The problem when trying to solve a set of non-linear equations is to determine a coupling in the bipartite graph described above that triggers as many simplifications as possible. So the Ford and Fulkerson (or equivalent) method is not enough for our purposes: instead of taking the first encountered coupling, we want in addition that the coupling satisfy a given criterion (e.g. maximizing the potential number of simplifications in the system). Hence the use of a variant of the Hungarian method which can be seen in modelicac as a method for finding a coupling based on an additional constraint called the ”satisfaction”. Practically, that is done in modelicac by associating a set of pairs (variable, weight) with each equation: given an equation, each weight indicates whether the equation is ”easy” to solve with respect to its associated variable or not. For instance, if an equation contains only one variable, the weight associated with that variable is low whereas the

weight associated with any other variable is infinite. Since modelicac associates low weights with variables that appear in linear systems, the Hungarian method ”discovers” linear systems by itself and symbolic substitution techniques, when applied to those linear systems, achieve the same effect as Gaussian elimination. Even though we did not consider the Gaussian elimination algorithms in modelicac, we got good results.

3.5 A complete (yet simple) example

First, we present the Modelica source code of a few electrical models from electrical library and then show how to use these models to construct and compile elaborated electrical models with modelicac.

3.5.1 Connectors

In Scicos libraries ”connectors” are the most basic open models. Each particular domain (e.g., electrical, hydraulic, etc.) has a its own connectors that connect two or more models and exchange quantities. The are two connector types:

• Internal connectors, that allow connection of two Modelica components, such as ”p” and ”n” pins used in electrical resistor model.

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

• External connectors, that allow communication of a Modelica component with an external environment (Explicit part of model in Scicos environment, for instance). Instances of ”InPutPort” and ”OutPutPort” are examples of these connectros types

```
class InPutPort
  input Real vi;
end InPutPort;

class OutPort
  output Real v;
end OutPort;
```

These types of connectors are used in sensor and actuator blocks that can be seen in Fig. 3 and 9.

3.5.2 Electrical component classes

These models include the ideal resistor, capacitor, inductor, sinusoidal voltage source and ground.

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

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

class Inductor "Ideal electrical inductor"
  Pin p, n;
  Real v;
  Real i;
  parameter Real L "Inductance";
equation
  v = L*der(i);
  v = p.v - n.v;
  0 = p.i + n.i;
  i = p.i;
end Inductor;

class VsourceAC "Sin-wave voltage 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;
```

3.5.3 ”Main class”

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 that 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 specifica-

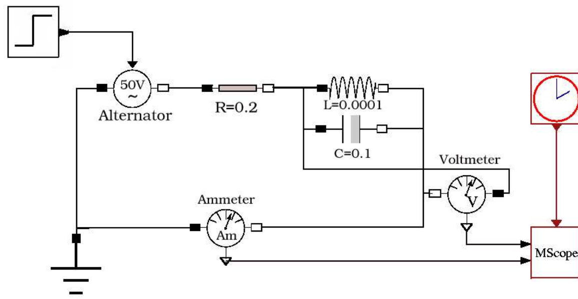

Figure 9: An electrical circuit modeled in Scicos.

tion and invokes modelicac to convert Modelica code into C code. In Fig. 9 there is a model of an electrical circuit modeled in Scicos. Here is its Modelica class automatically generated by Scicos:

```
class imppart_rlc
  parameter Real P1=0.0001;
  parameter Real P2=0.1;
  parameter Real P3=25.0;
  parameter Real P4=0.2;
  parameter Real P5=50.0;
  Inductor      B1(L=P1);
  Capacitor     B2(C=P2, v(start=P3));
  Ground        B3;
  VoltageSensor         B4;
  CurrentSensor         B5;
  Resistor      B6(R=P4);
  VVsourceAC    B7(f=P5);
  OutPutPort    B8;
  OutPutPort    B9;
  InPutPort     B10;
equation
  connect (B5.p,B3.p);
  connect (B7.p,B3.p);
  connect (B2.p,B1.p);
  connect (B4.p,B1.p);
  connect (B6.n,B1.p);
  connect (B2.n,B1.n);
  connect (B4.n,B1.n);
  connect (B5.n,B1.n);
  connect (B7.n,B6.p);
  B4.v = B8.vi;
  B5.i = B9.vi;
  B10.vo = B7.VA;
end imppart_rlc;
```

For this model modelicac generates a C code. This C code is incrementally linked with Scicos to be used as a standard block.

```
/*
number of discrete variables = 0
number of variables = 3
number of inputs = 1
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 rlc(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,v1;

   if (flag == 0) {
     res[0] = x[1]-xd[0]*rpar[0];
     res[1] = x[0]+xd[1]*rpar[1]-x[2];
     v1=get_scicos_time();
     res[2] = x[1]+x[2]*rpar[3]+sin(6.28318530718*v1*rpar[4])*u[0][0];
   } else if (flag == 1) {
       if (get_phase_simulation() == 1) {
          y[0][0] = x[1]; /* main.B8.vo */
          y[1][0] = -x[2]; /* main.B9.vo */
       } else {
          y[0][0] = x[1]; /* main.B8.vo */
          y[1][0] = -x[2]; /* main.B9.vo */
       }
   } else if (flag == 2 && nevprt < 0) {
   } else if (flag == 4) {
       x[0] = 0.0;     /* main.B1.i */
       x[1] = rpar[2]; /* main.B2.v */
       x[2] = 0.0;     /* main.B6.p.i */
       Set_Jacobian_flag(1);
   } else if (flag == 6) {
   } else if (flag == 7) {
       property[0] = 1;   /* main.B1.i (state variable) */
       property[1] = 1;   /* main.B2.v (state variable) */
       property[2] = -1;  /* main.B6.p.i (algebraic variable) */
       set_pointer_xproperty(property);
   } else if (flag == 9) {
   } else if (flag == 10) {
       v0 = Get_Jacobian_parameter();
       res[0] = -rpar[0]*v0;
       res[1] = 1.0;
       res[2] = 0.0;
       res[3] = 1.0;
       res[4] = rpar[1]*v0;
       res[5] = 1.0;
       res[6] = 0.0;
       res[7] = -1.0;
       res[8] = rpar[3];
       res[9] = 0.0;
       res[10] = 0.0;
       res[11] = sin(6.28318530718*get_scicos_time()*rpar[4])
       res[12] = 0.0;
       res[13] = 0.0;
       res[14] = 1.0;
       res[15] = 0.0;
       res[16] = 0.0;
       res[17] = -1.0;
       res[18] = 0.0;
       res[19] = 0.0;
       set_block_error(0);
   }
        return;
}
```

Conclusion

The use of Modelica in Scicos provides a versatile modeling and simulation tool.

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,*Scicos a dynamic system builder and simulator*, IEEE INTERNATIONAL CONFERENCE ON CACSD, Dearborn, Michigan, 1996.

[4] R. Nikoukhah and S. Steer, *Hybrid systems: modeling and simulation*, COSY: MATHEMATICAL MODELLING OF COMPLEX SYSTEM, Lund, Sweden, Sept. 1996.

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

[6] 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.

[7] M. Najafi, A.Azil, and R. Nikoukhah, *Extending scicos from system to component level simulation*, ESMc2004 international conference, Paris, France, October 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. N. Brown, A. C. Hindmarsh, and L. R. Petzold, *Consistent Initial Condition Calculation for Differential-Algebraic Systems*, SIAM J. SCI. COMP., NO. 19, 1998.

[13] P. FRITZSON,*Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*, WILEY-IEEE PRESS, 2004

[14] S.E. MATTSSON, H. ELMQVIST, M. OTTER, AND H. OLSSON, "INITIALIZATION OF HYBRID DIFFERENTIAL-ALGEBRAIC EQUATIONS IN MODELICA 2.0", *2nd Inter. Modelica Conference 2002*, DYNASIM AB, SWEDEN AND DLR OBERPFAFFENHOFEN, GERMANY, 2002, PP. 9–15.

[15] R. NIKOUKHAH AND S. STEER, "CONDITIONING IN HYBRID SYSTEM FORMALISM", *International Conference Automation of Mixed Processes:A D P M Hybrid Dynamic Systems*, DORTMUND, GERMANY, 2000.

[16] WEIS, PIERRE , LEROY, XAVIER, *Le Langage CAML*, 2ND ED , DUNOD PRESS, 1999