We are IntechOpen,
the world's leading publisher of
Open Access books
Built by scientists, for scientists

6,900
Open access books available

186,000
International authors and editors

200M
Downloads

154
Countries delivered to

Our authors are among the

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS
BOOK CITATION INDEX
INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index
in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

S-Function Library for Bond Graph Modeling

B. Umesh Rai
Indian Institute of Science, Bangalore
India

1. Introduction

S-functions are short for system-functions. They are used for extending the capabilities of Simulink®. S-functions allows us to add our own algorithms to Simulink models. The process of creating S-function blocks is quite simple. Simulink provides a S-function API which can be used to write a S-function routine observing a set of laid down rules. The compiled routine is enclosed inside a Simulink block, which can be subsequently customised by masking. A library of customised S-function blocks are created for an application specific task. This library can be subsequently distributed to work in MATLAB® environment.

1.1 How is S-function useful?

S-function can also be described as a computer language description of the Simulink block. S-function is written in any one of the popular languages viz C, C++, Fortran or Ada besides MATLAB's own M programming language and compiled as MEX files, where MEX stands for *MATLAB Executable*. S-functions use a special calling syntax called the S-function API that interacts with the Simulink engine. This interaction is very similar to the interaction that takes place between the engine and built-in Simulink blocks.

In MATLAB S-functions, the S-function routines are implemented as MATLAB functions. In C MEX S-functions, they are implemented as C functions. All the S-function routines available to MATLAB S-functions exist for C MEX S-functions as well. However, Simulink provides a larger set of S-function routines for C MEX S-functions.

S-function routines can be written for continuous, discrete or hybrid systems. A set of S-function blocks created by us can be placed in a tool box or library and distributed for working in MATLAB environment. S-functions allow creation of customised blocks for Simulink. By following a set of rules, any block algorithms can be implemented in an S-function. It can also be deployed for using an existing C code into a simulation. After compiling the S-function, the run time file has to be placed in an S-function block. User interface can then be customised by using masking. An advantage of using S-functions is that a general purpose block can be built that can be used many times in a model, varying parameters with each instance of the block.

The most common usage of S-functions is for creating a set of custom Simulink blocks for an application. Existing C code in the application is easily encapsulated into S-function block and used as a separate Simulink block. This is used alongside the other blocks of Simulink.

If the system model has been already modeled as a set of mathematical equations, it becomes easy to convert each equation into a S-function block, and and develop the system model in Simulink.

1.2 Vectors in S-function

In a Simulink block, a vector of input, u, are processed by a vector of states, x to output a vector of output, y (Fig. 1) (Mathworks, 2011). The state vector may consist of continuous states, discrete states, or a combination of both.

$$u \longrightarrow \boxed{\begin{array}{c} x \\ \text{(states)} \end{array}} \longrightarrow \begin{array}{c} y \\ \text{(output)} \end{array}$$

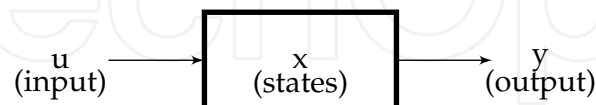$$\begin{array}{c} u \\ \text{(input)} \end{array}$$

Fig. 1. Generalised Simulink block

In S-functions written in the MATLAB programming language, the MATLAB S-function, Simulink partitions the state vector into two spaces. The first part of the state vector is occupied by the continuous state, and the discrete states occupy the second part. But in the other programming language written S-function, MEX-file S-functions, there are two separate state vectors for the continuous and discrete states.

2. Steps in simulation

Routines have to be written in S-function to carry out the simulation steps required by the Simulink engine. To cross-reference the routine required for the simulation step, two different approaches are used. For an MATLAB S-function, Simulink passes a flag parameter to the S-function. The flag indicates the current simulation stage. Routines in M-code calls the appropriate functions for each flag value. For a C MEX S-function, Simulink calls the S-function routines directly. This is done by following a naming convention for the routines.

2.1 Steps in S-function block simulation

Simulink engine first calls the S-function Routine to perform initialisation of all S-functions block in the model. Later, the engine makes repeated calls during simulation loop to each S-function block in the model, directing it to perform tasks such as computing its outputs, updating its discrete states, or computing its derivatives. Finally, the engine invokes a call to each S-function Routine for a termination task (Fig. 2). The tasks required at various stages, include:

- Initialization: Simulink initializes the S-function as a first step. The tasks are:
    - Initialising the SimStruct, a structure that contains information about the S-function.
    - Setting the number and size of input and output ports.
    - Setting the block sample times.
    - And allocating storage areas and the sizes array.
- Calculation of next sample hit - for a variable step integration routine, this stage calculates the time of the next variable hit, that is, it calculates the next stepsize in the variable step.
- Calculation of outputs in the major time step. After this call is complete, all the output ports of the blocks are valid for the current time step.

- Update discrete states in the major time step. In this call, all blocks should perform once-per-time-step activities such as updating discrete states for next time around the simulation loop.
- Integration: This applies to models with continuous states and/or nonsampled zero crossings. If S-function has continuous states, Simulink calculates the derivative of the continuous state at minor time steps. Simulink computes the states for S-function. If S-function (C MEX only) has nonsampled zero crossings, then Simulink will call the output and zero crossings portion of S-function at minor time steps, so that it can locate the zero crossings.
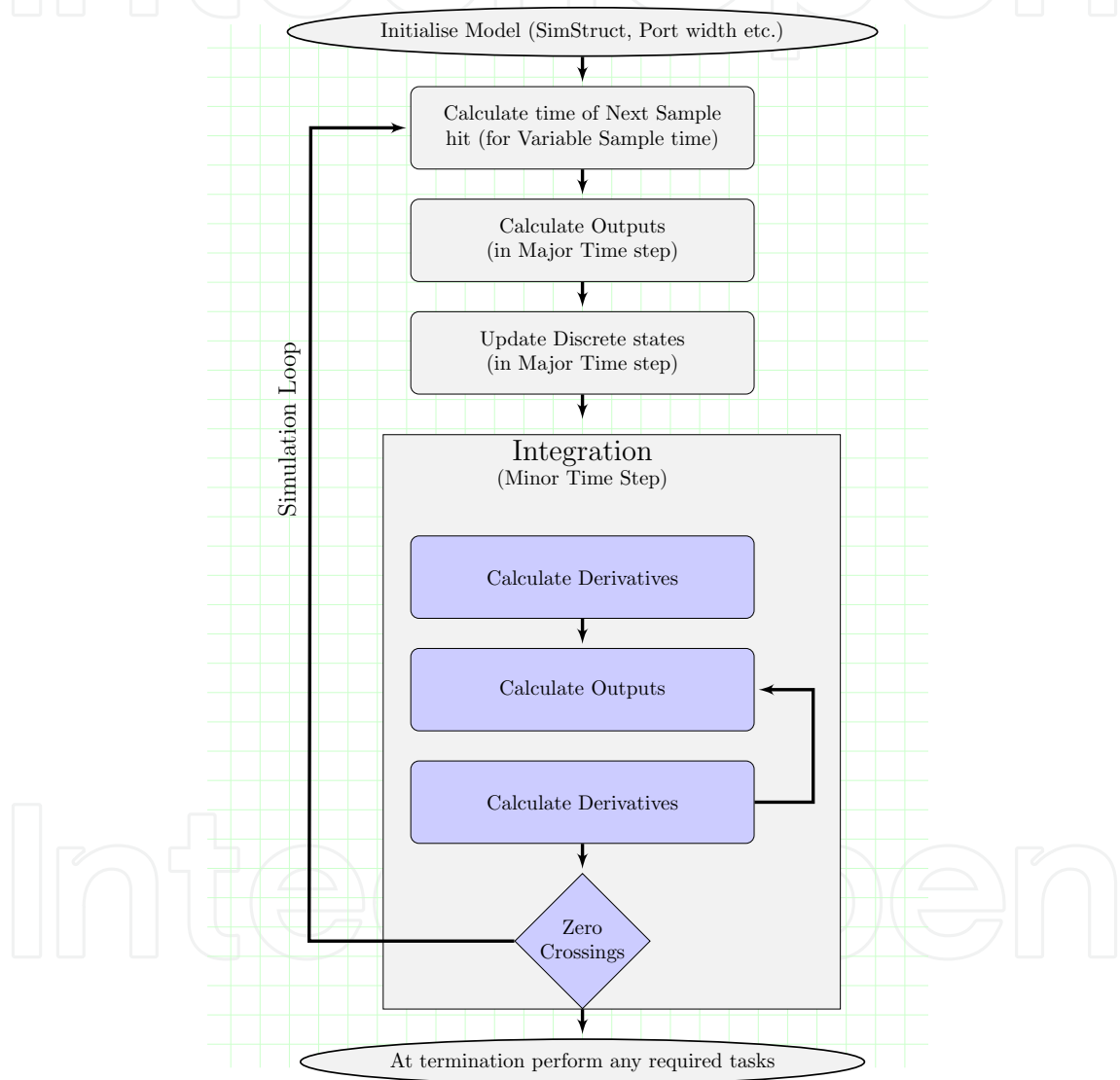

Fig. 2. S-function simulation cycle

2.2 Flags in MATLAB S-function

An MATLAB file that defines an S-Function block must provide information about the model. Simulink needs this information during simulation. The MATLAB S-function has to be of the

following form:

$$[sys, x0, str, ts] = f(t, x, u, flag, p1, p2, ...)$$

where f is the name of the S-function. Simulink passes t, the current time, x, state vector, u, input vector, integer flag in argument to S-function. In an MATLAB S-function flags are used for indicating the current simulation stage. S-function code calls the appropriate functions for each flag value. Table 1 lists the simulation stages, the corresponding S-function routines, and the associated flag value for MATLAB S-functions.

Simulation Stage	S-Function	Routine Flag
Initialization	mdlInitializeSizes	flag = 0
Calculation of derivatives of continous state variables	mdlDerivatives	flag = 1
Update discrete states, sample times	mdlUpdate	flag = 2
Calculation of outputs	mdlOutputs	flag = 3
Calculation of next sample hit (Only when discrete-time sample time specified)	mdlGetTimeOfNextVarHit	flag = 4
End of simulation tasks	mdlTerminate	flag = 9

Table 1. M-File flags

An MATLAB S-function returns an output vector having the following elements:

- sys - the values returned depend on the flag value (for flag = 3, sys contains the S-function outputs).
- x0 - the initial state values at flag = 0 (otherwise ignored).
- str - reserved for future use.
- ts - two-column matrix containing sample time and offset.

2.3 C MEX S-function callback methods

As with MATLAB S-functions, Simulink interacts with a C MEX-file S-function by invoking callback methods that the S-function implements. C MEX-file S-functions have the same structure and perform the same functions as MATLAB S-functions. In addition, C MEX S-functions provide more functionality than MATLAB S-functions. C MEX-files can access and modify the data structure that Simulink uses internally to store information about the S-function. This gives it an ability to handle matrix signals and multiple data types.

C MEX-file that defines an S-Function block provides information about the model to Simulink during the simulation. Unlike MATLAB S-functions, no explicit flag parameter is associated with C MEX S-function routine. But the routines have to follow the naming convention. Simulink then automatically calls each S-function routine at the appropriate time during its interaction with the S-function. It defines specific tasks which include defining initial conditions and block characteristics, and computing derivatives, discrete states, and outputs.

Simulink defines in a general way the task of each callback and their sequence (Fig.3) (Mathworks, 2011). The green box in Fig.3 are the compulsorily present routines and the blue box are the optional routines. The S-function is free to perform any task according to the functionality it implements. For example, Simulink specifies that the S-function's mdlOutput method must compute that block's outputs at the current simulation time. It does not specify what those outputs must be. The callback-based API allows to create S-functions, and hence custom blocks, of any desired functionality. The contents of the routines can be as complex and any logic can reside in the S-function routines as long as the routines conform to their required formats.
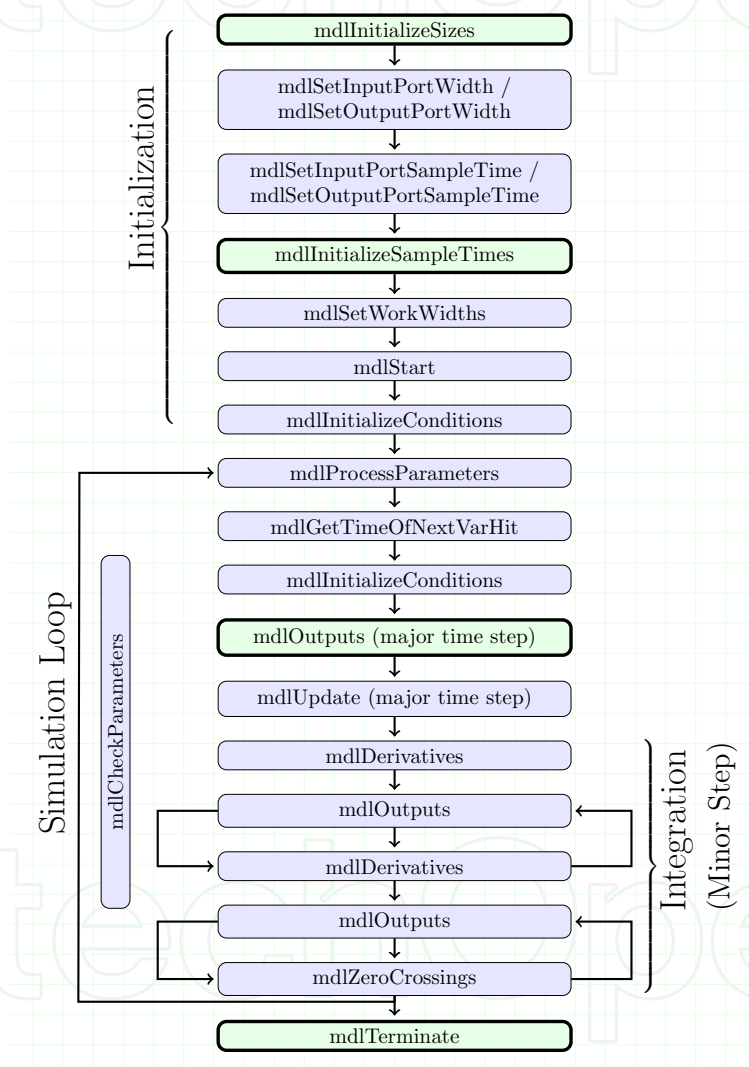

Fig. 3. Calling sequence of C MEX S-function Blocks

2.4 MEX versus MATLAB S-functions

Both the approaches of MATLAB and MEX S-functions development have some inherent advantages due to their origin in the programming language they are coded in. The advantage of MATLAB S-functions is speed of its development. Developing MATLAB S-functions avoids the time consuming compile-link-execute cycle required when developing in a compiled language like C. MATLAB S-functions also have easier access to MATLAB toolbox functions

and can utilize the MATLAB Editor/Debugger. MEX S-functions are more appropriate for integrating legacy code into a Simulink model. For more complicated systems, MEX S-functions may simulate faster than MATLAB S-functions because the MATLAB S-function has to call the MATLAB interpreter for every callback routine.

3. S-function examples

The ease of writing an S-function is demonstrated here by taking a simple example. Both MATLAB file and C Mex approaches will be demonstrated. A block '*timestwo*' is implemented in S-function. This simple block has no states. Functionally, the block takes an input scalar signal, doubles it and outputs to a connected device (Fig.4).
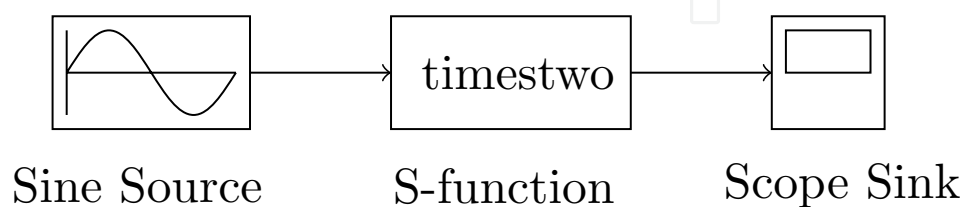

Fig. 4. timestwo S-function in model

3.1 MATLAB S-function implementation of timestwo

The S-function will be defined with four input arguments from Simulink. They are, the current time t, state x, input u, and flag *flag*. The output vector contains the variable listed in Sec.2.2. For the simple example chosen, two routines will be sufficient. A routine for initialisation and another for calculating the output. Below is the MATLAB code for the *timestwo.m* S-function:

```
function [sys,x0,str,ts] = timestwo(t,x,u,flag)

 % Dispatch the flag. The switch function controls the calls to
  % S-function routines at each simulation stage.
  switch flag,

    % Initialization
    case 0
      [sys,x0,str,ts] = mdlInitializeSizes;

    % Calculate outputs
    case 3
      sys = mdlOutputs(t,x,u);

    % Unused flags
    case { 1, 2, 4, 9 }
      sys = [];

    % Error handling
    otherwise
```

```
        error(['Unhandled flag = ',num2str(flag)]);

    end;
% End of function timestwo.
```

The routines that are called by *timestwo.m* are *mdlInitializeSizes* and *mdlOutputs*. The routine *mdlInitializeSizes* passes on the block information to the *size* structure. The information on number of outputs, inputs, continuous and discrete states, sample times and whether direct feed through is present, is passed on to the structure variable. The output variables, *x0, str* and ts are also set to the desired values. The second routine *mdlOutputs* just doubles the input scalar.

```
%===============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%===============================================================

function [sys,x0,str,ts] = mdlInitializeSizes

% Call function simsizes to create the sizes structure.
sizes = simsizes;

% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=1;
sizes.NumInputs=1;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;

% Load the sys vector with the sizes information.
sys = simsizes(sizes);

%
x0 = []; % No continuous states
%
str = []; % No state ordering
%
ts = [-1 0]; % Inherited sample time

% End of mdlInitializeSizes.



%===============================================================
% Function mdlOutputs performs the calculations.
```

```
%===============================================================
function sys = mdlOutputs(t,x,u)

sys = 2*u;

% End of mdlOutputs.
```

The timestwo MATLAB S-function can now be used in the Simulink model, by first dragging an S-Function block from the User-Defined Functions block library into the model. Then entering the name *timestwo* in the S-function name field of the S-Function block's Block Parameters dialog box (Fig. 5).
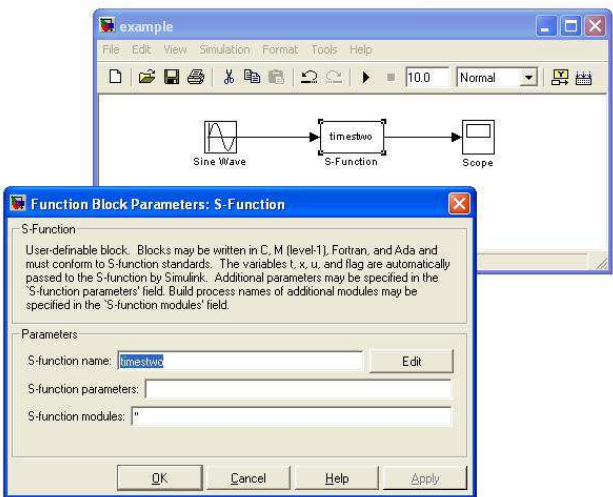

Fig. 5. timestwo S-function in Simulink model file

3.2 C MEX S-function implementation of timestwo

C MEX S-function will contain the callback methods mdlInitializeSizes, mdlInitializeSampleTimes, mdlOutputs and mdlTerminate. The simulation steps will have setting of intial condition by the first two blocks, the simulation loop in the third block and final termination by the last block (Fig.6). The code is reproduced below:
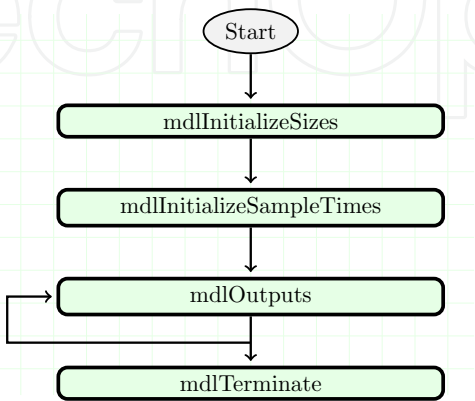

Fig. 6. C MEX S-function timestwo Blocks

```
#define S_FUNCTION_NAME timestwo /* Defines and Includes */
#define S_FUNCTION_LEVEL 2

#include "simstruc.h"

static void mdlInitializeSizes(SimStruct *S) {
  ssSetNumSFcnParams(S, 0);
  if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
    return; /* Parameter mismatch reported by the Simulink engine*/
  }
  if (!ssSetNumInputPorts(S, 1)) return;
  ssSetInputPortWidth(S, 0, DYNAMICALLY_SIZED);
  ssSetInputPortDirectFeedThrough(S, 0, 1);
  if (!ssSetNumOutputPorts(S,1)) return;
  ssSetOutputPortWidth(S, 0, DYNAMICALLY_SIZED);
  ssSetNumSampleTimes(S, 1);
  ssSetOptions(S, SS_OPTION_EXCEPTION_FREE_CODE);
}

static void mdlInitializeSampleTimes(SimStruct *S) {
  ssSetSampleTime(S, 0, INHERITED_SAMPLE_TIME);
  ssSetOffsetTime(S, 0, 0.0);
}

static void mdlOutputs(SimStruct *S, int_T tid) {
  int_T i;
  InputRealPtrsType uPtrs = ssGetInputPortRealSignalPtrs(S,0);
  real_T *y = ssGetOutputPortRealSignal(S,0);
  int_T width = ssGetOutputPortWidth(S,0);
  for (i=0; i<width; i++) {
    *y++ = 2.0 *(*uPtrs[i]);
  }
}

static void mdlTerminate(SimStruct *S){}

#ifdef MATLAB_MEX_FILE/*Is this file being compiled as a MEX-file?*/
#include "simulink.c" /* MEX-file interface mechanism */
#else
#include "cg_sfun.h" /* Code generation registration function */
#endif
```

3.3 Explanation of C MEX timestwo code

We start by two define statements, and give the name to our S-function and tell the Simulink engine that the code is written in Level 2 format. Later we include the header file *simstruc.h*,

which is a header file that defines a data structure, called the SimStruct, that the Simulink engine uses to maintain information about the S-function. A more complex S-function will include more header files.

The callback routine *mdlInitializeSizes* tells the Simulink engine that the function has no parameters, has one input and output port. It also declares that only one sample time will be specified later in *mdlInitializeSampleTimes*. Simulink is also informed that the code is exception free code. This declaration speeds up the execution time. The next callback routine *mdlInitializeSampleTimes* declares the sample time to be inherited i.e the block executes whenever the driving block executes.

The callback routine *mdlOutputs* calculates outputs at each time step. The input and output port signals is accessed through a vector of pointers. The width of the output port which is defined to be dynamically set is then read and the program loops for the width while calculating the output signal to be two times the input signal. The final callback mandatory routine *mdlTerminate* performs the end of the simulation task, which in the present case is a *NIL* set. following this routine the mandatory trailer code for compiler is present. Its absence will lead to compile errors.

The C MEX S-function has to be now compiled. Simulink gives a choice of C compiler to be used. We can use either the built in MEX compiler or any other C compiler already loaded in the system. The following command at the command line

```
mex -setup
```

allows us to locate all the compilers available in the system and the option to use one for compiling and linking in the MATLAB environment. Later the command

```
mex timestwo.c
```

compiles and links the timestwo.c file to create a dynamically loadable executable for the Simulink software to use. The resulting executable is referred to as a MEX S-function. The MEX file extension varies from platform to platform. For example, on a 32-bit Microsoft Windows system, the MEX file extension is .mexw32. The compiled run time file is then put into the S-function block similer to MATLAB S-function file (Fig. 5).

4. Bond graph modeling

4.1 Analogous behaviour of physical systems

The bond graph approach to physical system modeling was conceptualized by Hank Paynter on April 24, 1959 (Paynter & Briggs, 1961), inspired by the earlier work of Gabriel Kron (Kron, 1962). Bond graph language is a port based graphical approach for modeling energy exchange between subsystems. This technique was further developed by Karnopp and Rosenberg (Karnopp et al., 1990; 2006; Karnopp & Rosenberg, 1968; Rosenberg & Karnopp, 1983). Several books, special issues and articles on bond graph technique have popularised it for growing usage (Borutzky, 2009; Borutzky et al., 2004; Breedveld, 1984; 1991; 2004; Breedveld et al., 1991; Cellier et al., 1995; Dauphin-Tanguy, 2000; Gawthrop, 1995; Gawthrop & Smith, 1996; Mukherjee & Karmakar, 2000; Thoma, 1990; Thoma & Perelson, 1976).

Energy domain	f flow	e effort	q Generalised momentum	p Generalised displacement
Electromagnetic	i, [A] current	v, [V] voltage	q, [C] charge	λ, [V-s] linked flux
Mechanical translation	V, [m/s] Velocity	F, [N] Force	x, [m] Displacement	p, [N-s] Momentum
Angular translation	ω, [rad/s] Velocity	T, [N m] Torque	θ, [rad] Angle	p_ω, [N-m-s] Momentum
Hydraulic	φ, [m^3/s] Volume flow	P, [N/m^2] Pressure	V, [m^3] Volume	Γ, [N-s/m^2] Momentum of flow tube
Thermal	T, [K] Temperature	F_S, [J/K/s] Entopy flow	S, [J/K] Entropy	
Chemical	μ, [mol/s] Molar flow	F_N, [J/mol] Chemical potential		N, [mol] Number of moles

Table 2. Flow and effort variables in different domains

Behaviour of a physical system is constrained, either implicitly or explicitly by laws of physics viz. mass and energy conservation, laws of momentum and positive entropy production. Furthermore, various physical domains are each characterized by a particular quantity that is conserved. Each of these domains have analogous ideal behaviour with respect to energy (Table 2). This analogy led to the concept of energy port, the building block of bond graph modeling language. Here, the interaction between physical systems is through energy port and is always bidirectional. There will be an input signal and a consequent output signal ('*back effect*') and their product will signify the '*power that is transacted*'. From a computational point of view, the effort could be computed by 'Port 1', while the flow is computed in 'Port 2'. It could be the other way around as well. Apriori the computational direction of signal is not known, except the fact that they are in opposite direction (Fig.7).

4.2 Bond graph elements

Bond graphs are labelled, directed graphs. The vertices of a bond graph denote subsystems, system components or elements, while the edges, called power bonds or bonds for short, represent energy flows between them. The nodes of a bond graph have power ports where energy can enter or exit. As energy can flow back and forth between power ports of different nodes, a half arrow is added to each bond indicating a reference direction of the energy flow. The amount of power, *P(t)*, at each given time, *t*, is given by the product of the two conjugate variables, which are called effort, *e*, and flow, *f*, respectively.

$$P(t) = e(t) \cdot f(t) \tag{1}$$

There can be five groups of physical behaviour by elements handling energy:

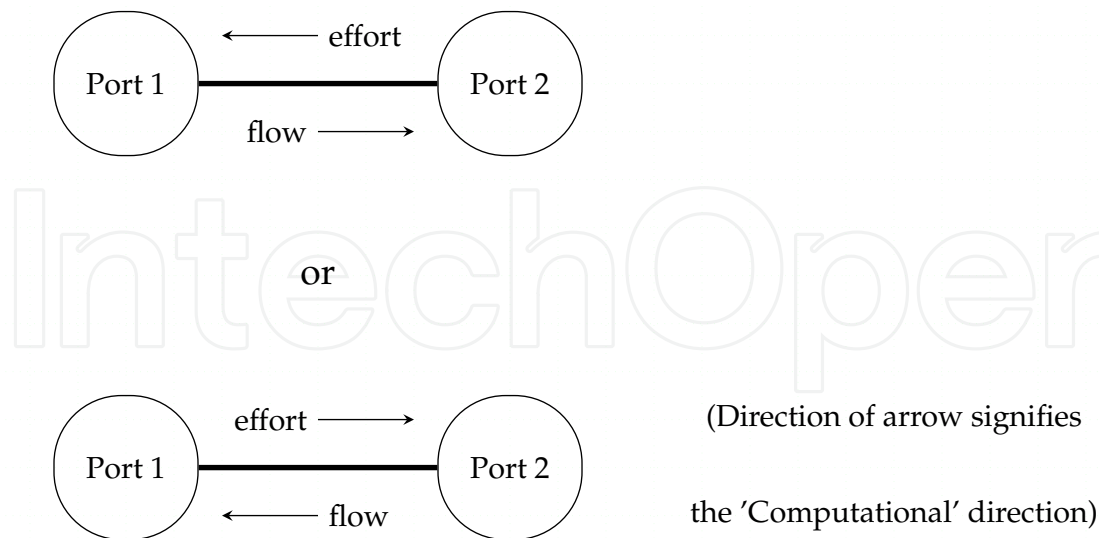1. **Storing** of energy.
2. **Supply** on demand.

Fig. 7. Computational direction of flow and effort between ports

3. **Reversible** transformation (including inter-domain transfers).
4. **Irreversible** transformation (positive entropy production).
5. **Distribution** to connected ports.

These five behaviour are represented by nine basic elements.

- Two types of *storage* elements, effort or *Capacitive*- storage and flow or *Inductive* - storage.
- Two types of *sources*, **Source - effort** and **Source - flow**.
- Two types of Reversible transformators, non-mixing, reciprocal *Transformer* or **TF**-type transducer and mixing anti- reciprocal *Gyrator* or **GY**-type transducer.
- Irreversible transducer is an energy *Dissipater* or can be also classified as entropy producing **R**-type transducer.
- Distributor junction are also in dual form, **0**-junction and **1**-junction. The 0-junction represents a generalised domain independent Kirchoff current law and similarly a 1-junction represents a generalised domain independent Kirchoff voltage law.

The elements can also be segregated based on their port structure:

- Five one-port elements: $C, L(orI), R, S_e, S_f$.
- Two two-port elements: *TF, GY*.
- Two n-port junctions: *0-junction, 1-junction*.

4.3 Constitutive relations

A constitutive relation with a constant parameter characterises each element. For sources, the imposed variable is independent of the conjugate variable, and for the rest of elements, the relationship is algebraic between its conjugate variables. The storage elements are classified as *memory* elements. The preferred constitutive equation is integration with respect to time.

If differentiation with respect to time is used, the information on initial condition or *history* is lost. If sensors are included in the bond graph modeling and used for determining the factor of the algebraic relationship between the conjugate variable, we rename the bond as *modulated* bond with prefix *M* for the modulated element name.
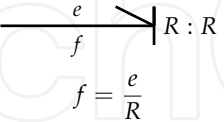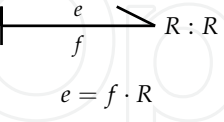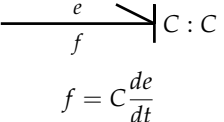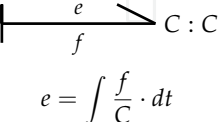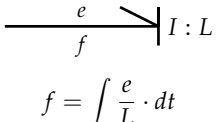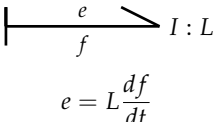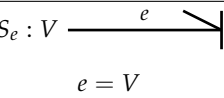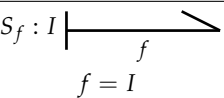
Element	Effort causal	Flow causal
Resistor, R	$R : R$ $f = \dfrac{e}{R}$	$R : R$ $e = f \cdot R$
Capacitor, C	$C : C$ $f = C\dfrac{de}{dt}$	$C : C$ $e = \int \dfrac{f}{C} \cdot dt$
Inductor, I	$I : L$ $f = \int \dfrac{e}{L} \cdot dt$	$I : L$ $e = L\dfrac{df}{dt}$
Source, $S_{e/f}$	$S_e : V$ $e = V$	$S_f : I$ $f = I$

Table 3. One-port elements
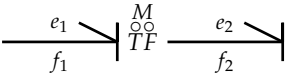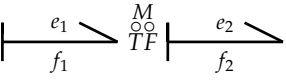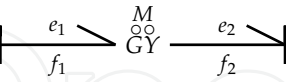
Element	Effort causal	Flow causal
Transformer, TF modulating parameter = M	$\overset{M}{TF}$ $e_2 = M \cdot e_1;\ f_1 = M \cdot f_2$	$\overset{M}{TF}$ $e_1 = \dfrac{e_2}{M};\ f_2 = \dfrac{f_1}{M}$
Gyrator, GY modulating parameter = M	$\overset{M}{GY}$ $f_1 = \dfrac{e_2}{M};\ f_2 = \dfrac{e_1}{M}$	$\overset{M}{GY}$ $e_1 = M \cdot f_2;\ e_2 = M \cdot f_1$

Table 4. Two-port elements

5. Computational causality

Each bond connects two power ports of different primitive elements and carries two power variables as can be seen in Fig.7. One of the two power variables may be determined by one of the two sub-models, while the other is determined by the other model. A short stroke, called *causal stroke*, perpendicular to the bond is placed at one of its ends of the bond. This indicates the computational direction of the effort variable. Consequently the other *open* end is the decider of the flow variable. The nine basic elements with their constitutive relationships that are dependent on their causal stroke are shown in Table 3,4 and 5

Element	Symbol	Governing Law
0-junction	$r1:R$ $1:1$ e_{r1} f_{r1} e_l f_l $S_e:V_s \xrightarrow{\ e_s\ }_{f_s} 0 \xrightarrow{\ e_{r2}\ }_{f_{r2}} R:r2$	*Effort Law* General equation $\sum_{i=1}^{m} f_i = 0$ and $e_1 = e_2 = e_3 = ... = e_m$ For bond graph model at left $f_s - f_{r1} - f_l - f_{r2} - ... = 0$ and $e_l = e_{r1} = e_{r2} = ... = e_s$ The element having causal bar toward the junction decides the effort for all bonds associated with the junction.
1-junction	$r1:R$ $1:1$ e_{r1} f_{r1} e_l f_l $S_e:V_s \xrightarrow{\ e_s\ }_{f_s} 1 \xrightarrow{\ e_{r2}\ }_{f_{r2}} R:r2$	*Flow Law* General equation $\sum_{i=1}^{m} e_i = 0$ and $f_1 = f_2 = f_3 = ... = f_m$ For bond graph model at left $V_s - e_{r1} - e_l - e_{r2} - ... = 0$ and $f_s = f_{r1} = f_{r2} = ... = f_l$ The element having causal bar away from the junction decides the flow for all bonds associated with the junction.

Table 5. Junction elements

5.1 S-function implementation for C bond

We will now use the C MEX S-function code to develop a continuous state Simulink block. An One-port element which stores energy is chosen for illustration. Effort causal Capacitor C element in Table 3 is one such element. The code, which is an extension of code at Sec. 3.2 (added routines for continuous state) is given below:

```
#define S_FUNCTION_NAME   C_complex_bond
#define S_FUNCTION_LEVEL 2

#define NUM_INPUTS          1 /* Input Port  0 */
#define IN_PORT_0_NAME      u0
#define INPUT_0_WIDTH       DYNAMICALLY_SIZED
#define INPUT_0_FEEDTHROUGH 0

#define NUM_OUTPUTS         1 /* Output Port  0 */
#define OUT_PORT_0_NAME     y0
#define OUTPUT_0_WIDTH      DYNAMICALLY_SIZED

#define NPARAMS             2 /* Parameter  Capacitance */
#define PARAMETER_0_NAME    C /* Capacitance Value*/
#define PARAMETER_1_NAME    bias /* Initial Charge */
```

```
#define SAMPLE_TIME_0        CONTINUOUS_SAMPLE_TIME
#define NUM_CONT_STATES      2
#define CONT_STATES_IC       [0]

#include "simstruc.h"

#define PARAM_DEF0(S) ssGetSFcnParam(S, 0)
#define PARAM_DEF1(S) ssGetSFcnParam(S, 1)

static void mdlInitializeSizes(SimStruct *S) {
    ssSetNumSFcnParams(S, NPARAMS);

    if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
        return; /* Parameter mismatch will be reported by Simulink */
    }
    ssSetNumContStates(S, NUM_CONT_STATES);

    if (!ssSetNumInputPorts(S, NUM_INPUTS)) return;
    ssSetInputPortWidth( S, 0, INPUT_0_WIDTH);
    ssSetInputPortComplexSignal( S, 0, COMPLEX_INHERITED);
    ssSetInputPortDirectFeedThrough(S, 0, INPUT_0_FEEDTHROUGH);
    ssSetInputPortRequiredContiguous(S, 0, 1); /*direct input signal
    access*/

    if (!ssSetNumOutputPorts(S,NUM_OUTPUTS)) return;
    ssSetOutputPortWidth(S, 0, OUTPUT_0_WIDTH);
    ssSetOutputPortComplexSignal(S, 0, COMPLEX_INHERITED);

    ssSetNumSampleTimes(S, 1);
}

static void mdlInitializeSampleTimes(SimStruct *S) {
    ssSetSampleTime(S, 0, SAMPLE_TIME_0);
    ssSetOffsetTime(S, 0, 0.0);
}

static void mdlInitializeConditions(SimStruct *S) {
   real_T *xC   = ssGetContStates(S);

    xC[0] =  0;
    xC[1] =  0;
}

static void mdlOutputs(SimStruct *S, int_T tid) {
    boolean_T   yIsComplex = ssGetOutputPortComplexSignal(S, 0) ==
    COMPLEX_YES;
```

```
    real_T              *y0    = ssGetOutputPortRealSignal(S,0);
    const real_T        *xC    = ssGetContStates(S);
    const real_T        *C     = mxGetData(PARAM_DEF0(S));
    const real_T        *B     = mxGetData(PARAM_DEF1(S));

    y0[0] = *B + xC[0] / (*C);
    if(yIsComplex){ /* Process imag part */
        y0[1] = *B + xC[1] / (*C);
    }
}

static void mdlDerivatives(SimStruct *S) {
    const real_T   *u0  = (const real_T*) ssGetInputPortSignal(S,0);
    real_T         *dx  = ssGetdX(S);

    dx[0]=u0[0] ;
    dx[1]=u0[1] ;
}

static void mdlTerminate(SimStruct *S) {}/* mdlTerminate */

#ifdef MATLAB_MEX_FILE
#include "simulink.c"
#else
#include "cg_sfun.h"
#endif
```

6. Junction algorithm

In a bond graph model a set of elements is connected at the junction. One of the elements in the set takes up the role of *decider* bond. Remaining bonds in the set *per-se* take up the role of *non-decider* bonds. The junction algorithm is illustrated by taking O-junction as an example. The 0-junction block is a common effort junction (Fig.8). The effort is decided by a decider bond attached to it and having the causal bar towards the junction. Similarly for a 1-junction, decider bond will have its causal bar away from the junction, thus complementing the 0-junction behaviour. In the figure, J_{in}'s are the input from the non-decider bonds into the junction, and J_{sum} is the output of the junction.

The governing law of the 0-junction, *Effort Law (or KCL)*, states that the flow of the decider bond is the sum of the flows of all the non-decider bonds. In Fig.8, the decider bond of the junction has J_{sum}, the sum of all J_{in}, as its causal (flow) variable. Value of the conjugate variable (effort) of the decider bond, J_{out} is decided by the bond's constitutive equation. The second part he governing law of the 0-junction, states that the all the bonds connected to the junction share a common effort. Thus the effort of the decider bond becomes the causal variable for all the non-decider bonds. For each non-decider bond, its non-causal variable
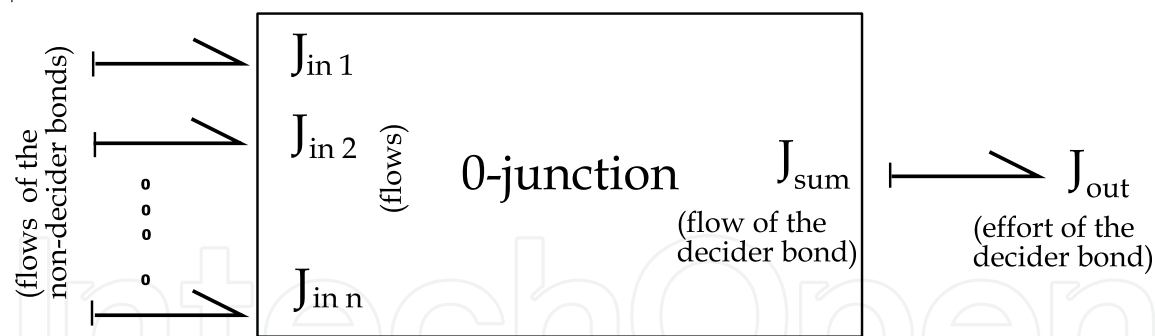
Fig. 8. 0-junction

flow into the junction as individual J_{in}. This completes the cycle for junction. Table 6 tabulates the algorithm for one-port elements.

Element	Decider		Non-decider	
	Causal	Non-causal	Causal	Non-causal
R,C,L	J_{sum}	J_{out}	J_{out}	J_{in}
S_e, S_f		J_{out}		J_{in}

Table 6. One-port element variables

It can be seen that for an element connected to a junction, the causal variable can take either of the two values, J_{sum} or J_{out}, depending on whether the connection bond is a decider or a non-decider, respectively. Similarly the non-causal variable can have either the value of J_{out} or J_{in}, again depending on whether the connection bond is a decider or a non-decider.

A two-port is connected to two junctions at either end. These junction could be either 0-junction or 1-junction. But as a two-port element allows only two combinations of causality out of available four, there will be only 2×2 alternatives for a variable. Looking at the two-port junction in Fig.9, if the element is decider bond for junction J_1, and the junction is 0-junction, then the flow at input port will be J_{1sum}. And if junction is a 1-junction the effort will be J_{1sum}. Similarly for the output junction connected to J_2. Note that the direction of two-port half arrows is reversed in the two junctions. Thus if two-port element with flow causal at input port is decider bond for the first junction, the junction has to necessarily be an 0-junction, but flow casual at output port as decider bond indicates a 1-junction. The modulus of the two-port element decides the value of the variable on the conjugate port, while the junction decides the conjugate variable value of the port. In a similar manner for all combination of causality and decider bond, the variables can be listed out. The algorithm is summarised in Table 7 (Umarikar et al., 2006).
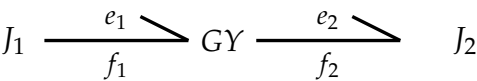
$$J_1 \xrightarrow[f_1]{e_1} GY \xrightarrow[f_2]{e_2} J_2$$

Fig. 9. Algorthim for 2 port element

Port	Input Port				Output Port			
	$J_1 \xrightarrow[f_1]{e_1} TF \xrightarrow[f_2]{e_2} J_2$							
	Decider		Non-decider		Decider		Non-decider	
	e_1	f_1	e_1	f_1	e_2	f_2	e_2	f_2
TF (flow causality)	J_{1out}	J_{1sum}	J_{1in}	J_{1out}	J_{2sum}	J_{2out}	J_{2out}	J_{2in}
TF (effort causality)	J_{1sum}	J_{1out}	J_{1out}	J_{1in}	J_{2out}	J_{2sum}	J_{2in}	J_{2out}
	$J_1 \xrightarrow[f_1]{e_1} GY \xrightarrow[f_2]{e_2} J_2$							
GY (flow causality)	J_{1out}	J_{1sum}	J_{1in}	J_{1out}	J_{2out}	J_{2sum}	J_{2in}	J_{2out}
GY (effort causality)	J_{1sum}	J_{1out}	J_{1out}	J_{1in}	J_{2sum}	J_{2out}	J_{2out}	J_{2in}

Table 7. two-port element variables

7. Linking elements

A bond graph model on paper does not explicitly use a connector as in block diagram model, to link one element to another element. To retain the look and feel of a paper model when transferred to computer terminal, the connection between the elements has to be invisible. The masking properties of the Simulink block is utilised for this purpose in the tool box. A junction is considered as a node, to which the elements are connected. The bond graph element, as in paper model, is placed next to its associated junction. The link to the junction is made by entering the junction label in the mask parameter box. *Shared Memory*' algorithm is then used to implicitly connect the element.

In bond graph modeling two or more elements will be linked to a junction. Their data have to be shared. Shared data structure is used in the toolbox. The memory locations are earmarked for a junction by assigning it a unique label by *character aggregation*' during its first run. An S-function's initialisation callback method is used for memory allocation as this callback is used only once during the simulation run. The associated elements using the notation listed out in Table 6 and Table 7, share their respective memory address, thereby their data.

The elements in the tool box are masked and have screen interfaces. For a one-port element the following details need to be entered.

1. Name of the element.
2. Parametric value.
3. Whether decider bond or not.
4. Associated junction name.

For a two-port element the extra information of the second port is also entered. Similarly a junction screen interface will have all entries of the elements that are linked to it, along with their energy flow direction signs.

7.1 Propagation of data

The propagation of data from one element to the next is by reading and writing into a common block address (Fig.10). The input element, designated as *Provider*, produces the data

through its constitutive equation and writes it on to the labelled memory address. The label, as discussed earlier is unique to a junction. The next element in hierarchy, designated as *Consumer*, in turn reads the data and by using its constitutive equations, *Produces* the next set of data which is written on to the next labelled memory address. There can be many *Consumer* of the data but there is only one unique *Provider*. An analogy to bond graph junction concept of one decider and many non-decider can be clearly seen here. It is also seen that a *Consumer* element of previous step becomes the *Provider* element in the next step. As the model is hierarchical, all the elements are in turn *Provider* and *Consumer* to their respective memory address block in one integration step. The memory location is released and freed when the data is not needed.
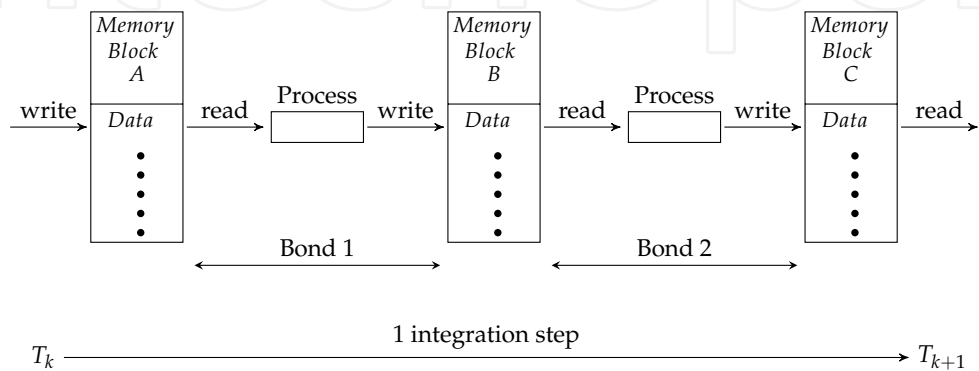

Fig. 10. Propagation of data in a integral time step

7.2 S-function implementation for shared memory link

The port of the bond graph element as discussed above is the pointer to the shared memory address. When we specify a input/output port for a bond graph element, we have to supply three parameters to the S-function. They are:

- The bond graph element name.
- Whether the element is a decider bond or not for the producer - consumer correspondence to the input - output port to be decided.
- The name of the junction to which it is connected.

The complete C MEX S-function code for input junction of a bond is given below:

```
#define S_FUNCTION_NAME   inPort
#define S_FUNCTION_LEVEL 2

#define NPARAMS             3
#define PARAMETER_0_NAME       decider
#define PARAMETER_0_DTYPE      boolean_T
#define PARAMETER_1_NAME       element
#define PARAMETER_2_NAME       junction

#include "shm_com.h"
#include "windows.h"
```

```
#include "mex.h"
#include <malloc.h>
#include "simstruc.h"

#define PARAM_DEF0(S) ssGetSFcnParam(S, 0)
#define PARAM_DEF1(S) ssGetSFcnParam(S, 1)
#define PARAM_DEF2(S) ssGetSFcnParam(S, 2)

#define IS_PARAM_DOUBLE(pVal)
  (mxIsNumeric(pVal) && !mxIsLogical(pVal) &&\
!mxIsEmpty(pVal) && !mxIsSparse(pVal) && !mxIsComplex(pVal)\
   && mxIsDouble(pVal))

static void mdlInitializeSizes(SimStruct *S) {
  ssSetNumSFcnParams(S, NPARAMS);/* Number of expected parameters*/
  if (ssGetNumSFcnParams(S) != ssGetSFcnParamsCount(S)) {
      return; /* Parameter mismatch will be reported by Simulink */
  }
  ssSetNumContStates(S, 0);
  ssSetNumDiscStates(S, 0);
  if (!ssSetNumInputPorts(S, 0)) return;
  if (!ssSetNumOutputPorts(S, 1)) return;
  ssSetOutputPortWidth(S, 0, DYNAMICALLY_SIZED);
  ssSetOutputPortComplexSignal(S, 0, COMPLEX_YES);
  ssSetNumSampleTimes(S, 1);
  ssSetNumRWork(S, 0);
  ssSetNumIWork(S, 0);
  ssSetNumPWork(S, 4);
  ssSetNumModes(S, 0);
  ssSetNumNonsampledZCs(S, 0);
  ssSetOptions(S, 0);
}

static void mdlInitializeSampleTimes(SimStruct *S) {
  ssSetSampleTime(S, 0, CONTINUOUS_SAMPLE_TIME);
  ssSetOffsetTime(S, 0, 0.0);
}

static void mdlStart(SimStruct *S) {
  const boolean_T *decider  = mxGetData(PARAM_DEF0(S));
  void *shared_memory_loc  = NULL;
  void *ishared_memory_loc = NULL;
  HANDLE hMapObject  = NULL;  // handle to real data file mapping
  HANDLE ihMapObject = NULL;  // handle to file mapping
  char_T str[sizeof("fbnbn00")];
  char_T temp[sizeof("fbnbn00")];
```

```c
char_T istr[sizeof("fbnbn00")];
mxGetString(PARAM_DEF1(S),str,sizeof(str));  //element name
mxGetString(PARAM_DEF2(S),temp,sizeof(str));  //junction name
if(*decider){
  strcpy(str,temp); //junction name
  strcat(str,"sum");
} else {
  strcpy(str,temp); //junction name
  strcat(str,"out");
}
strcpy(istr,"j"); // prefix 'j'
strcat(istr,str);
hMapObject = CreateFileMapping(
  INVALID_HANDLE_VALUE, // use paging file
  NULL,                 // no security attributes
  PAGE_READWRITE,       // read/write access
  0,                    // size: high 32-bits
  sizeof(struct shared_struct),        // size: low 32-bits
  str);      // name of map object
ssSetPWorkValue(S,0, hMapObject);

shared_memory_loc = MapViewOfFile(
  hMapObject,    // object to map view of
  FILE_MAP_WRITE, // read/write access
  0,             // high offset:  map from
  0,             // low offset:   beginning
  0);            // default: map entire file
if (shared_memory_loc == NULL )   {
  CloseHandle(hMapObject);
  return;
} else {
  ssSetPWorkValue(S,1, shared_memory_loc);
  memset(shared_memory_loc, '\0', sizeof(struct shared_struct));
}
ihMapObject = CreateFileMapping(
  INVALID_HANDLE_VALUE, // use paging file
  NULL,                 // no security attributes
  PAGE_READWRITE,       // read/write access
  0,                    // size: high 32-bits
  sizeof(struct shared_struct),        // size: low 32-bits
  istr);     // name of map object
ssSetPWorkValue(S,2, ihMapObject);

ishared_memory_loc = MapViewOfFile(
  ihMapObject,    // object to map view of
  FILE_MAP_WRITE, // read/write access
```

```c
    0,                  // high offset:  map from
    0,                  // low offset:   beginning
    0);                 // default: map entire file
  if ( ishared_memory_loc == NULL)   {
    CloseHandle(ihMapObject);
    return;
    } else {
    ssSetPWorkValue(S,3, ishared_memory_loc);
    memset(ishared_memory_loc, '\0', sizeof(struct shared_struct));
  }
}

static void mdlOutputs(SimStruct *S, int_T tid) {
  boolean_T   yIsComplex=ssGetOutputPortComplexSignal(S, 0)
    ==COMPLEX_YES;

  real_T  *y = ssGetOutputPortRealSignal(S,0);
  struct shared_struct *shared_stuff, *ishared_stuff;
  shared_stuff = (struct shared_struct *)ssGetPWorkValue(S,1);
  ishared_stuff = (struct shared_struct *)ssGetPWorkValue(S,3);
  y[0] = shared_stuff->some_data;
  y[1] = ishared_stuff->some_data;
  CloseHandle((HANDLE)ssGetPWorkValue(S,0));
  CloseHandle((HANDLE)ssGetPWorkValue(S,2));
}

static void mdlTerminate(SimStruct *S) {
  struct shared_struct *shared_stuff, *ishared_stuff;
  HANDLE c = (HANDLE) ssGetPWork(S)[0]; // retrieve and destroy C++
  HANDLE ic = (HANDLE) ssGetPWork(S)[2]; // retrieve and destroy C++
  shared_stuff = (struct shared_struct *)ssGetPWorkValue(S,1);
  ishared_stuff = (struct shared_struct *)ssGetPWorkValue(S,3);
  free(c);
  if(ic != NULL) {
    free(ic);
  }
  free(shared_stuff);
  if(ishared_stuff !=NULL) {
    free(ishared_stuff);
  }
}

#ifdef  MATLAB_MEX_FILE    /* Is this file being compiled as
    a MEX-file? */
#include "simulink.c"      /* MEX-file interface mechanism */
#else
```

```
#include "cg_sfun.h"          /*Code generation registration function*/
#endif
```

8. S-function library for bond graph elements

Bond graph modeling is an emerging field especially in electrical, mechatronics and electro-mechanics, and a bond graph engineer may feel the necessity to define his own element while making a model. With the set of callback routine and function available to MEX files, any complex constitutive equation can be written for a new element. To provide support for complex variables and vectors in Bond Graph, a tool library using C-MEX S-functions with data propagation through shared memory, is developed. The MEX file for the library has been written in C/C++. After compiling and debugging, C/C++ MEX S-Function are masked with bond graph icons to distinguish between different elements.

Each element of the bond graph library has two common input/output blocks along with with a middle block (Fig.11). The code for the middle block is specific to the element it implements. After placing the three S-functions block in the subsystem, the subsystem is masked. The element's mask screen has a help at the top and parameter entry text boxes, below. There is a check box to specify whether the bond is decider (Fig.12). The parameters entered in the mask screen are manipulated by the S-functions underneath to initialise the element before the simulation cycle starts.

The capability of S-functions to support arbitrary input dimensions is exploited in the tool box. The actual input dimensions can be determined dynamically when a simulation is started by evaluating the dimensions of the input vector driving the S-function. This feature allows the same element to handle a scalar or a vector input as the case may be, without declaring it apriori.

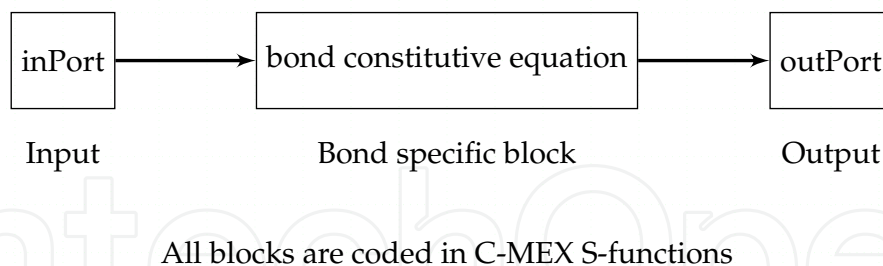
All blocks are coded in C-MEX S-functions

Fig. 11. Typical blocks under the mask of a bond graph element

The library is available in the standard format of simulink (Fig:13). The required elements can be had for *Pick and Place* from library after navigating down (Fig:13(a) and Fig:13(b)). The tools available in the mask's - icon graphics support, is utilised to give a natural iconic representation to the element subsystem.

8.1 Examples of tool box

Using the tool box, the circuit in Fig.14(a) is modeled in bond graph (Fig.14(b)). The simulation results are given in Fig.14(c) and Fig.14(d). As can be observed the library is able to handle complex quantities quite accurately.
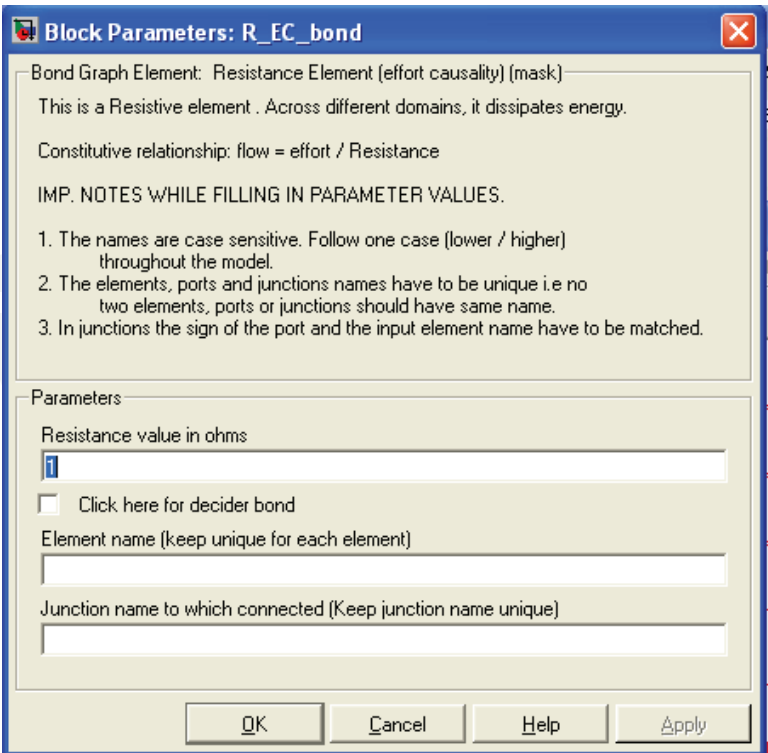
Fig. 12. Input mask screen

For another example of switched junction, one model of the switched mode power converter in Fig.14(f) is used. The circuit is of a boost converter. There are two switches S_1 and S_2 driven by complementary signals. This ensures that only one switch is on at any given time. The circuit is modeled in bond graph using the switched junction as shown in Fig.14(e). The simulation result of the effected state variable is shown in Fig.14(g).

8.2 Simulation results for IM model

A rotating electrical machine can be viewed as a machinery which converts one form of energy into another. More specifically it converts electrical energy into mechanical energy or vice-versa. Magnetic energy is used as a conversion medium between electrical and mechanical energy.

8.2.1 Axis rotator element

This generalised concept for electrical machine modeling needs 'Axis Rotator', a new bond graph element (Umesh Rai & Umanand, 2008; 2009a;b) to mathematically model a electrical commutator (Fig.15). The constitutive relationship for the flow and effort in the bonds is given by Eqn.(2) and Eqn.(3).

$$f_i = \frac{d}{dt}\left(\Lambda_m\left(\sum_{k=1}^{n} e_k \cos\alpha_{(i,k)}\right)\right) \quad where, \quad \alpha_{(i,k)} = \alpha_i - \alpha_k \tag{2}$$

$$\sum_{k=1}^{n} e_k f_k = P_m \tag{3}$$

(a) Bond Graph tool Box level 1

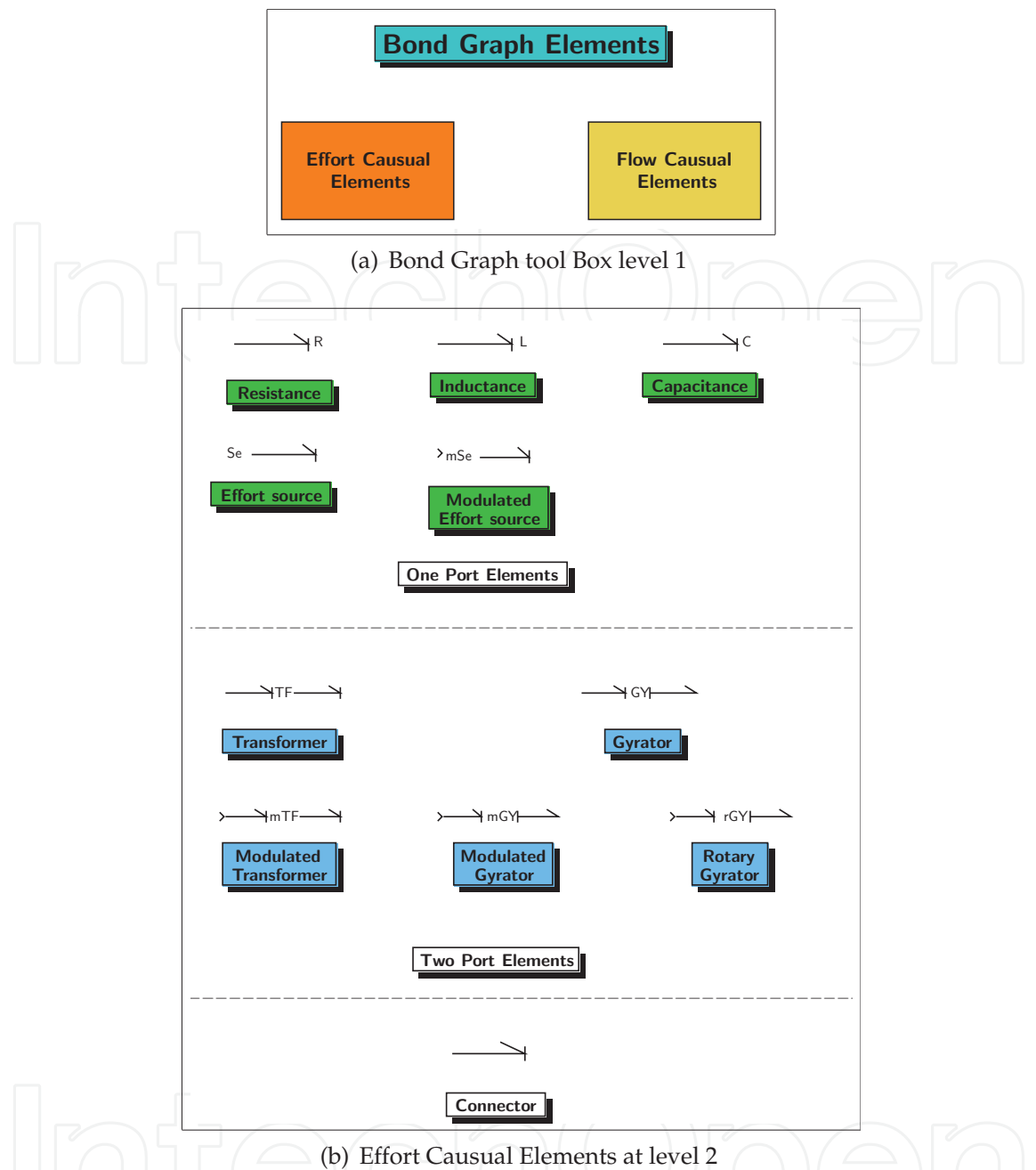(b) Effort Causual Elements at level 2

Fig. 13. Bond Graph library

$\cos \alpha_{(i,k)}$ refers to the spatial angle between the kth winding's axis and the ith winding axis with respect to the bond under consideration. Λ_m is the mutual permeance (inverse of reluctance) of the magnetic core, P_m is the reactive power required to magnetise the core.

8.2.2 Induction motor model

A bond graph model of 3ϕ doubly fed induction motor using the Axis Rotator element is shown in Fig.16. There are six sets of electric energy input ports, three each for stator and rotor, in the model. The motor shaft represents a mechanical output port. The air gap is represented by the AR with six connection bonds terminating at it, each representing a set of

(a) RC circuit

(b) Bond graph model of RC circuit

(c) Plots of C-bond flow

(d) Plots of C-bond effort

(e) Boost converter model built using switched junction

(f) A Boost converter

(g) Plot of effected state variables
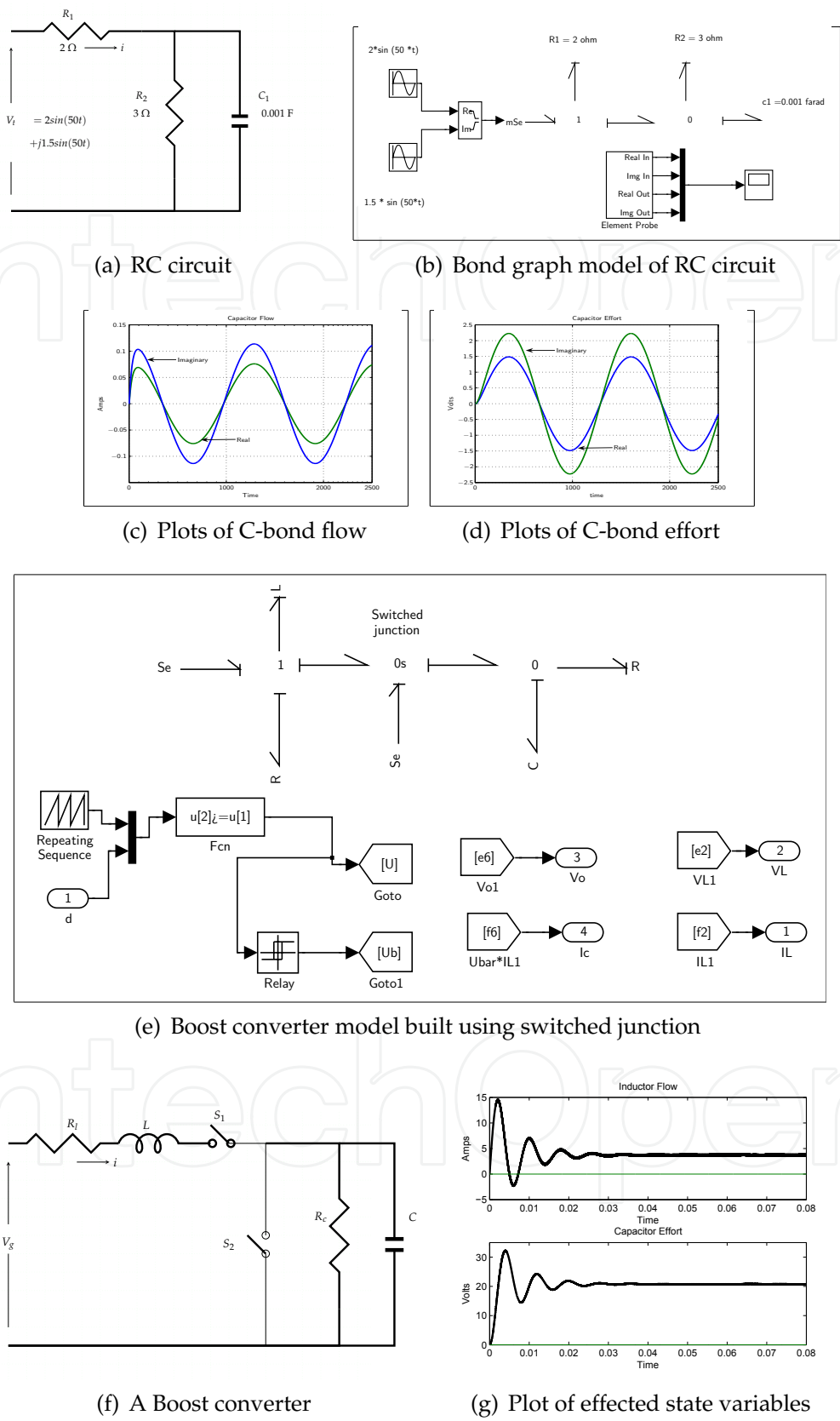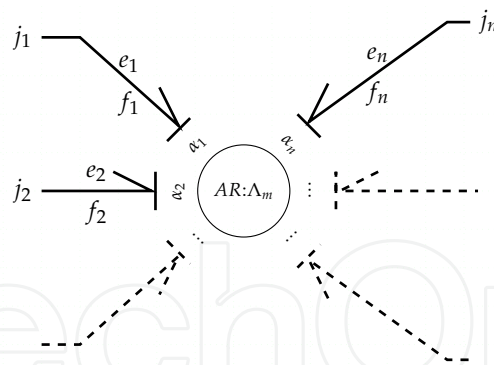
Fig. 14. Bond graph tool box examples

Fig. 15. Axis rotator connected with bonds representing windings

winding. The permeance parameter of the AR represents the mutual coupling effect of the flux. The iron loss is represented by the dissipative port. The stator and rotor energy ports are described with different set of port parameters.

The dissipative, entropy producing fields, R_{as}, R_{bs}, R_{cs}, R_{ar}, R_{br}, R_{cr} and R_i are non-equal non-linear resistances depending on the underlying physical system. R_i represents the iron losses of the core. In a similar manner, the model represents the permeances L_{asl}, L_{bsl}, L_{csl}, L_{arl}, L_{brl}, L_{crl} and L_m . There is no linearity restriction on the above parameters. They could be constants, functions or even a lookup table, without loss of generality. The value of the lumped parameters are different for different phases, dependent on the electric and the magnetic energy they represent. Similarly the energy sources feeding the different windings are represented by s_1, s_2 and s_3. For balanced supply voltages the voltage peaks and frequency would be same, with a phase difference of $(2\pi/3)$ to one another. The three domains of electrical, magnetic and mechanical are clearly brought out in Fig.16.

At the shaft the developed electromagnetic torque as a function of the stator, rotor currents and the angle between them is represented as an effort source. The electromagnetic torque provides the effort at the one junction against the inertial, frictional and load torque components. The feedback information on flow at this junction, which gives the measure of rotor speed is transmitted to the AR for calculating the instantaneous angle of the rotor windings.

8.2.3 S-function model of induction motor

The power of S-function is demonstrated by firstly implementing the complex AR element using C Mex S-function and making it a part of bond graph library. As discussed in the above section, there is no linearity or balance supply constraint on the model. The increased complexity can easily be handled by the bond graph library. The causal model implementation of squirrel cage induction motor is shown in Simulink (Fig.17). The machine starts from stall. A step load is applied to the motor at 0.5sec. The simulation results of speed curve for various step load are presented in Fig.18(a). Similarly the current curves and the torque curve for a specific load are at Fig.18(c) and Fig.18(e) respectively. The transition of rotor currents to slip frequency can be distinctly seen at Fig.18(c). The simulation results of the bond graph model in S-function co-related well with the speed curves obtained by d-q model of the induction motor implemented by functional block in Simulink (Fig.18(b), Fig.18(d) and Fig.18(f)).
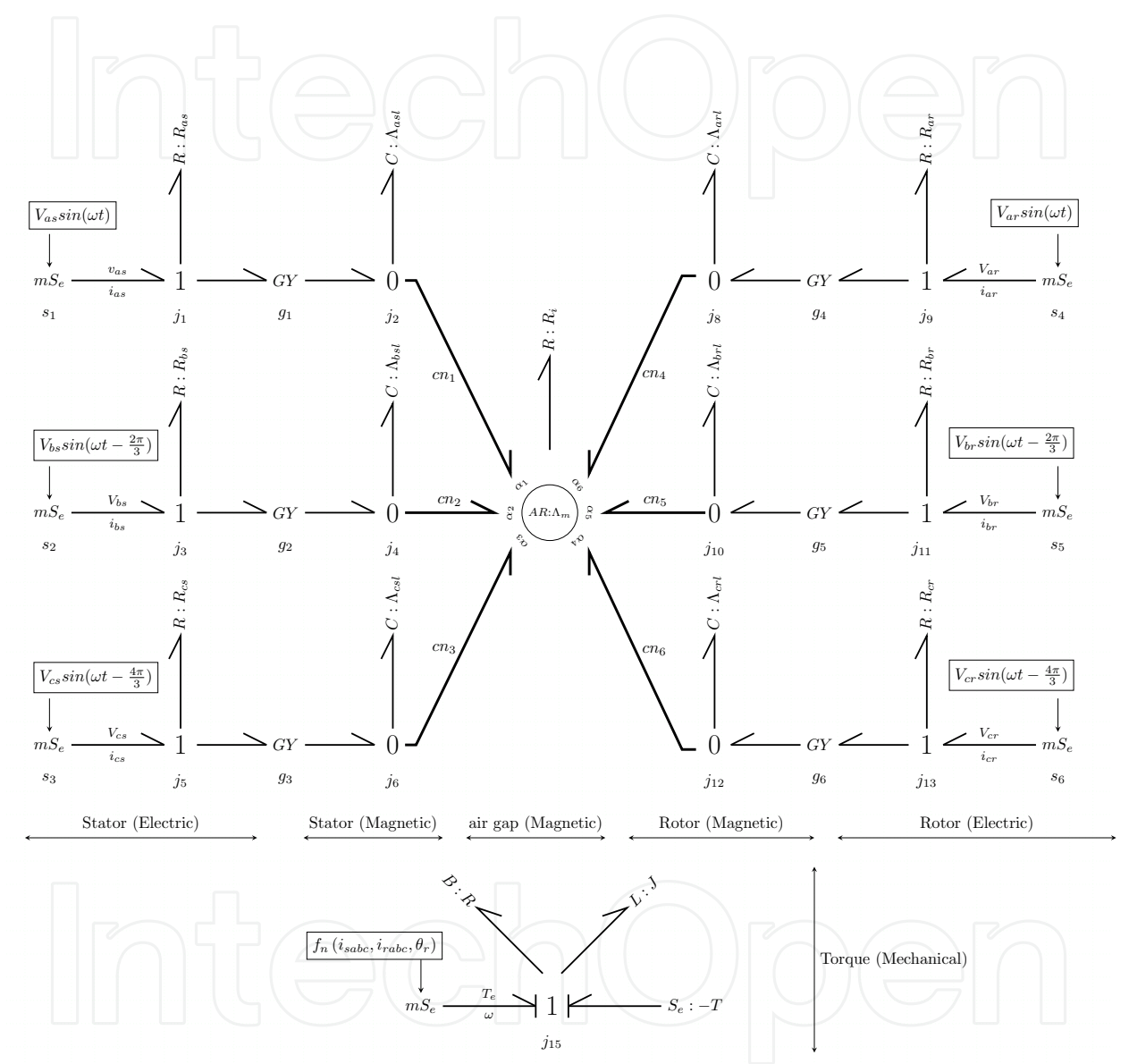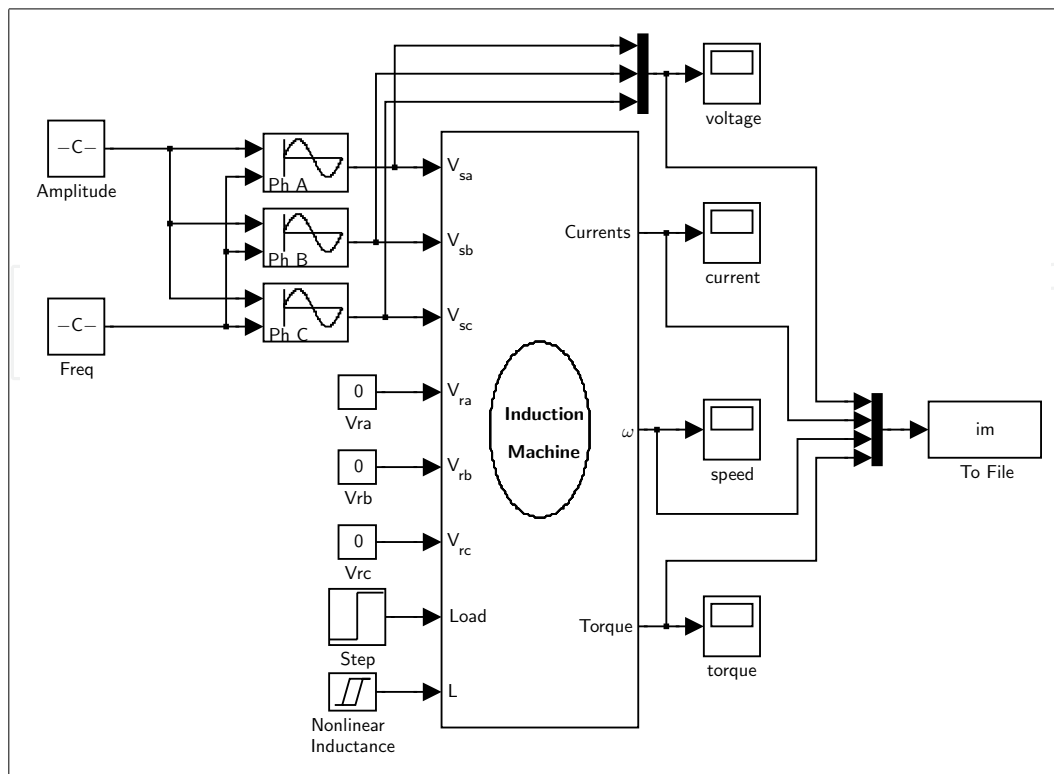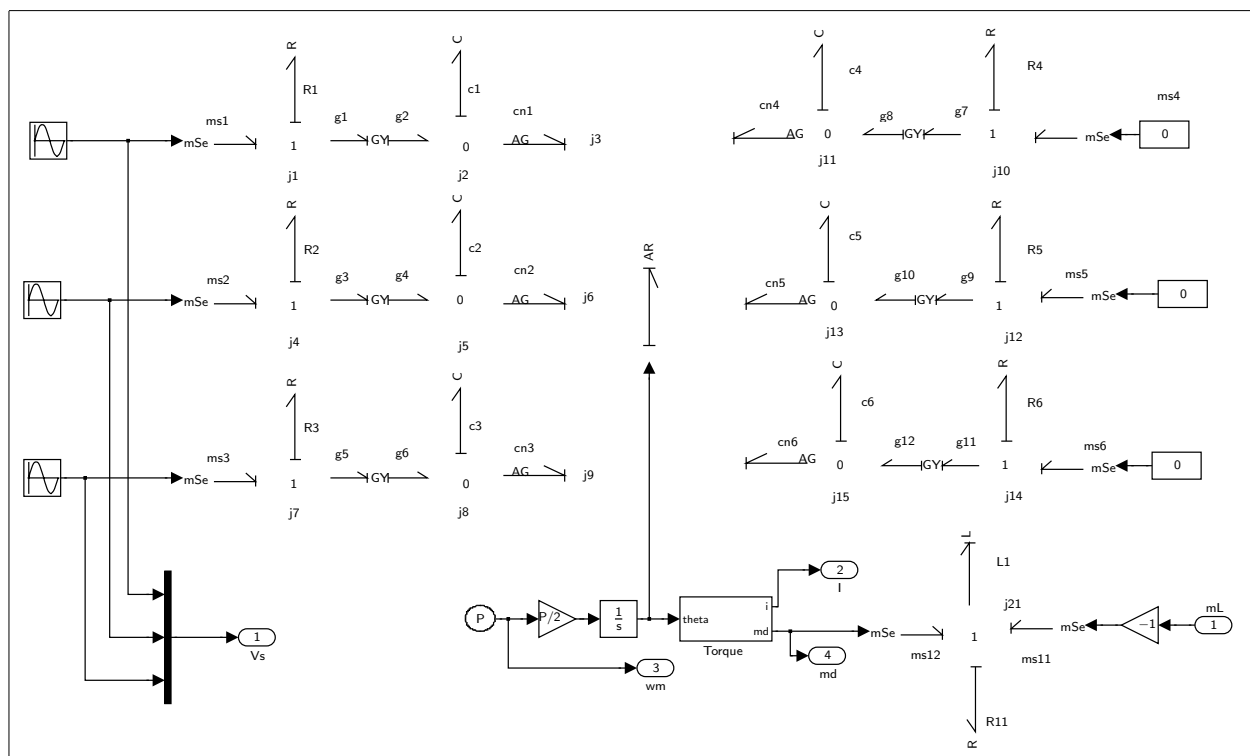
Fig. 16. 3ϕ *DFIM* bond graph model

(a) Model in Solver

(b) Model in MATLAB/Simulink)

Fig. 17. 3ϕ Induction motor model in Matlab

(a) Speed curves proposed model

(b) Speed curves dq model

(c) Current curves proposed model

(d) Current curves dq model
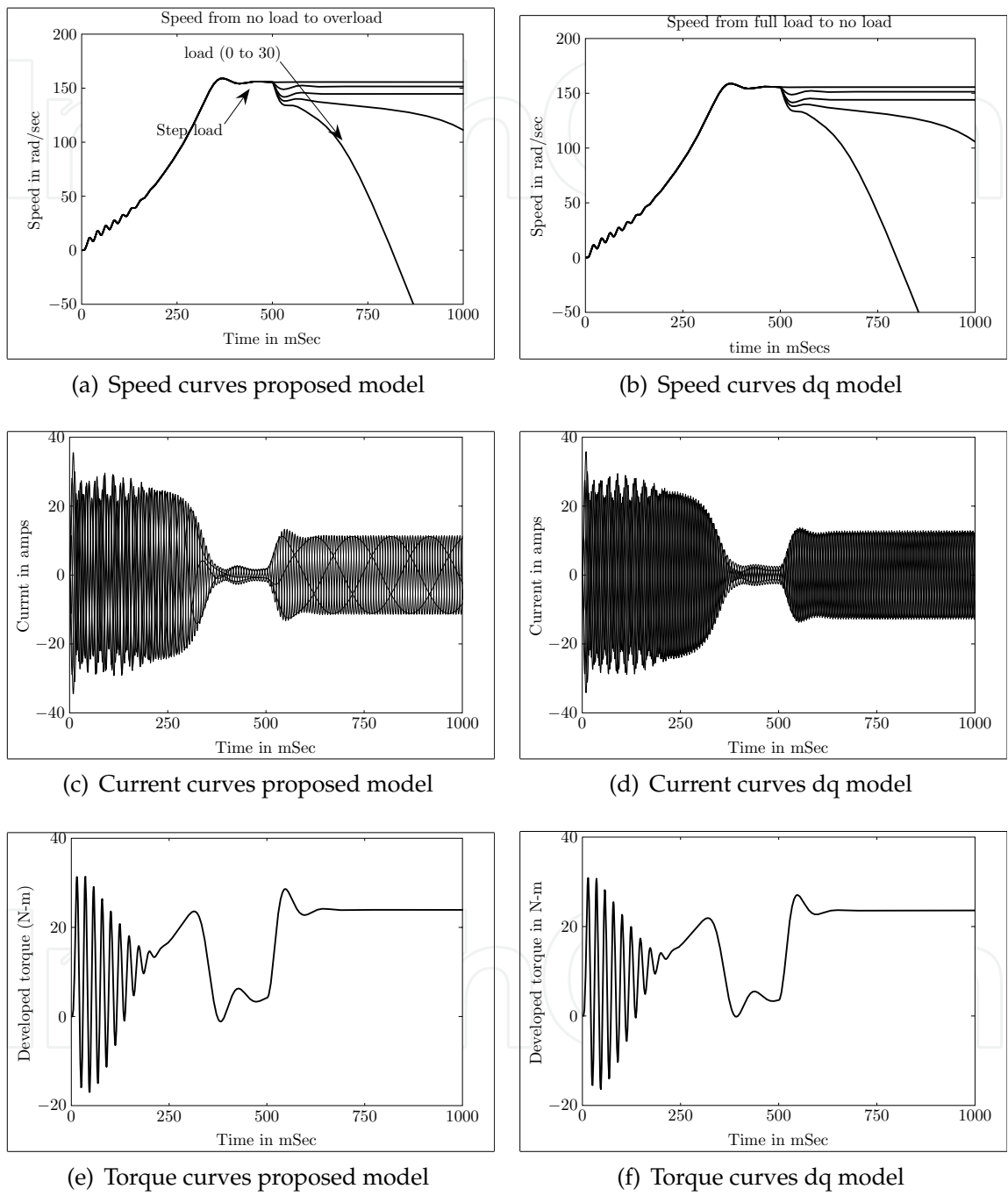
(e) Torque curves proposed model

(f) Torque curves dq model

Fig. 18. Simulation results of the bond graph model

9. Conclusion

The power of S-function in customising MATLAB/Simulink® environment suitable for a specific modeling need is illustrated in this chapter. Two different approaches for implementing a customised Simulink element is then discussed with their advantages and disadvantages. Later the Bond graph approach of modeling is briefly introduced. Level 2 C MEX S-function technique is then used to develop a library of bond graph element.

This library of bond graph elements can handle both the scalar and vector or complex variables without declaring apriori, a distinct advantage. A new element - *Axis Rotator*, used for representing rotating magnetic field is included in the library. The ability to handle complex variables along with the rotation enables the elements in the library to be used for modeling rotating frames as that existing in an electric machine. The library also incorporates switched junctions, which allows for the modeling of switches in any circuit. In developing the library, the *Shared Memory Concept* is used. By using shared memory concept, the memory requirement comes down as only the pointer to the memory location are passed and not the data values.

10. References

Borutzky, W. (2009). *Bond Graph Modelling*, Simulation Modelling Practice and Theory.

Borutzky, W., for Modeling, S. & International, S. (2004). *Bond Graphs: A Methodology for Modelling Multidisciplinary Dynamic Systems*, SCS Publishing House e. V.: Society for Modeling and Simulation International.

Breedveld, P. (1984). *Physical Systems Theory in Terms of Bond Graphs*, THT-Afdelin Electrotechniek.

Breedveld, P. (1991). *Special Issue on Current Topics in Bond Graph Related Research*, Pergamon.

Breedveld, P. (2004). Port-based modeling of mechatronic systems, *Mathematics and Computers in Simulation* 66(2-3): 99–128.

Breedveld, P., Rosenberg, R. & ZHOU, T. (1991). Bibliography of bond graph theory and application, *Franklin Institute, Journal* 328(5): 1067–1109.

Cellier, F., Elmqvist, H. & Otter, M. (1995). Modeling from physical principles, *The Control Handbook* pp. 99–108.

Dauphin-Tanguy, G. (2000). *Les bond graphs*, Hermès science publications.

Gawthrop, P. (1995). Physical model-based control: A bond graph approach, *Journal of the Franklin Institute* 332(3): 285–305.

Gawthrop, P. & Smith, L. (1996). *Metamodelling: for bond graphs and dynamic systems*, Prentice Hall International (UK) Ltd. Hertfordshire, UK, UK.

Karnopp, D., Margolis, D. & Rosenberg, R. (1990). *System Dynamics: A Unified Approach*, John Wiley & Sons, Inc., NY.

Karnopp, D., Margolis, D. & Rosenberg, R. (2006). *System Dynamics: Modeling and Simulation of Mechatronic Systems*, John Wiley & Sons, Inc. New York, NY, USA.

Karnopp, D. & Rosenberg, R. (1968). *Analysis and Simulation of Multiport Systems: The Bond Graph Approach to Physical System Dynamics*, MIT Press.

Kron, G. (1962). *Diakoptics: The Piecewise Solution of Large-scale Systems*, Macdonald.

Mathworks (2011). *Developing S-Functions*, The MathWorks, Inc.
    URL: *www.mathworks.com*

Mukherjee, A. & Karmakar, R. (2000). *Modelling And Simulation of Engineering Systems Through Bondgraphs*, Alpha Science Int'l Ltd.

Paynter, H. & Briggs, P. (1961). *Analysis and Design of Engineering Systems*, MIT Press.

Rosenberg, R. & Karnopp, D. (1983). *Introduction to Physical System Dynamics*, McGraw-Hill, Inc. New York, NY, USA.

Thoma, J. (1990). *Simulation by bondgraphs*, Berlin and New York, Springer-Verlag, 194 p.

Thoma, J. & Perelson, A. (1976). Introduction to Bond Graphs and Their Applications, *Systems, Man and Cybernetics, IEEE Transactions on* 6(11): 797–798.

Umarikar, A., Mishra, T. & Umanand, L. (2006). Bond graph simulation and symbolic extraction toolbox in MATLAB/SIMULINK, *J. Indian Inst. Sci* 86: 45–68.

Umesh Rai, B. & Umanand, L. (2008). Bond graph model of doubly fed three phase induction motor using the Axis Rotator element for frame transformation, *Simulation Modelling Practice and Theory* 16(10): 1704–1712.

Umesh Rai, B. & Umanand, L. (2009a). Bond graph model of an induction machine with hysteresis nonlinearities, *Nonlinear Analysis: Hybrid Systems* 4(3): 395–405.

Umesh Rai, B. & Umanand, L. (2009b). Generalised bond graph model of a rotating machine, *International Journal of Power Electronics* 1(4): 397–413.
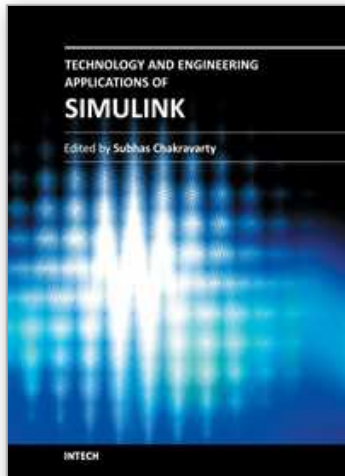
Technology and Engineering Applications of Simulink

Edited by Prof. Subhas Chakravarty

Building on MATLAB (the language of technical computing), Simulink provides a platform for engineers to plan, model, design, simulate, test and implement complex electromechanical, dynamic control, signal processing and communication systems. Simulink-Matlab combination is very useful for developing algorithms, GUI assisted creation of block diagrams and realisation of interactive simulation based designs. The eleven chapters of the book demonstrate the power and capabilities of Simulink to solve engineering problems with varied degree of complexity in the virtual environment.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

B. Umesh Rai (2012). S-Function Library for Bond Graph Modeling, Technology and Engineering Applications of Simulink, Prof. Subhas Chakravarty (Ed.), ISBN: 978-953-51-0635-7, InTech, Available from: http://www.intechopen.com/books/technology-and-engineering-applications-of-simulink/s-function-library-for-bond-graph-modelling

INTECH
open science | open minds