iSim ESL

ESL Simulation Software

User Guide and Tutorial

Document Information
Version: 1.9.2
Date Published: February 2022

This document relates to ESL version 8.2.5

ISIM welcomes any suggestions to improve
the ESL Simulation Software and documentation.

If you have any suggestions, or would like to point out
any errors or omissions, please contact us:

ISIM International Simulation Limited
161 Claremont Road
Salford
M6 8PA
UK

Tel: +44 (0) 161-736-5283

Email: info@isimsimulation.com
Web: https://www.isimsimulation.com

Table of Contents

CHAPTER 1

Introduction

The purpose of the guide is to introduce ESL by means of a series of tutorial exercises.

1.1 Background

The simulation package, ESL, has evolved from a series of contracts undertaken by ISIM International Simulation Limited and the University of Salford for the European Space Agency (ESA) over a period of some twenty years. The initial contract, which was a research study of simulation algorithms suitable for parallel architecture hardware, produced a proposal for a new Continuous System Simulation Language (CSSL) standard. The proposed standard included features that allowed the decomposition of a large simulation into segments that could, in principle, be executed on parallel hardware. (Although initially this feature was emulated on a single processor, distributed simulation over a network of computers was fully implemented at a later date).

A second contract saw the implementation of a minimal software suite, which supported the proposed CSSL standard and the first version of ESL was born. There then followed a series of contracts in which the language was extended and enhanced to meet the requirements of ESA. These extensions included the addition of graphical interfaces for block diagram model description, interactive simulation execution control and graphical analysis of results. Other enhancements included embedded and remote simulation capability, C++ translation, matrix arithmetic and both single and multiple-variable transfer function representation of dynamic elements.

During recent stages of the product development, a completely new graphical user interface has been added providing an Integrated Simulation Environment (ISE) from which all phases of the simulation process can be managed.

Although initially developed for the European Space Agency, ESL is a general-purpose continuous system simulation tool, with discrete event capability, that finds applications in the non-space sector as well as the space sector. The following are a selection of past and recent applications:

- Design of the Giotto (Halley's Comet probe) "Despin" antenna system.
- Investigation of thermal vibrations in the solar panels of the Hubble Space Telescope.
- Modelling of Nickel Cadmium battery systems for ERS1 (Earth Resource Satellite).
- Attitude Control Computer's Environment Simulation (ACC EnvSim) for the XMM Software Validation Facility (SVF).
- Dynamics Simulation Library of the Model Library for Software Validation (MOLISOVA) developed by GMV.
- Software Validation Facility for the Attitude and Orbit Control Subsystem (AOCS) software of the scientific satellite ISO and the integrated data handling and AOCS software for communications satellite Artemis.
- Off-shore Gas-Rig training simulator.
- Gas Compressor station simulation.
- Rapid Gravity Water Filter-Bed simulation.

1.2 The Style of the Guide

The guide is not intended to be an exhaustive reference manual for ESL. Rather it aims to introduce the main features of the software and provide enough information to get you started using ESL. Detailed information all topics introduced will be found in the extensive on-line help, which you are encouraged to consult at each stage.

CHAPTER 2

A Simple Example

You will create a simple single-input, single-output feedback control system application. This will introduce the following features:

- layout of the Integrated Simulation Environment (ISE)
- simulation elements
- use of the graphical editor to create a top-level block diagram
- setting simulation element attributes
- use of the display elements
- running and interacting with the simulation
- post-run plotting

The example to be considered is the feedback control system shown in block diagram form in Figure 1:

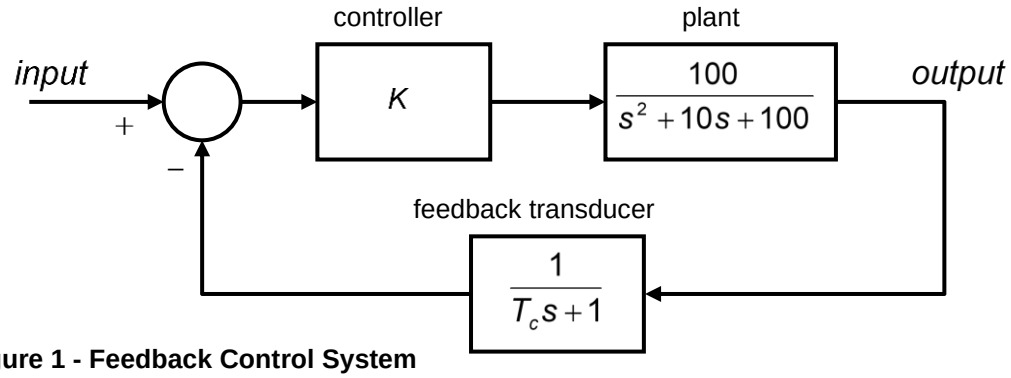

Figure 1 - Feedback Control System

We are interested in the response of the system for different types of input (step, sinusoidal) while varying the values of the gain (K) and feedback time constant (T_c). Initially, K = 2.0 and T_c = 0.1 seconds.

2.1 Creating the Block Diagram

Start ESL ISE - usually from the Start Menu. The appearance of the main window of ISE will be similar to Figure 2.

Note: *To get the ESL ISE windows and dialogs to appear as they are throughout this document, load **derwent.opt** off the View>Load Options… menu.*

The window is divided into the following areas:

- the Canvas - on which block diagrams are created
- the Palette - containing standard simulation elements
- the Message Window - in which informative text is displayed
- the Menu Bar - primary means of selecting ISE options
- the Status bar - displays name of selected palette object and information about certain operations

ISE runs in one of two modes: Edit mode and Run mode. On start-up it is in Edit mode in which you create simulation models.

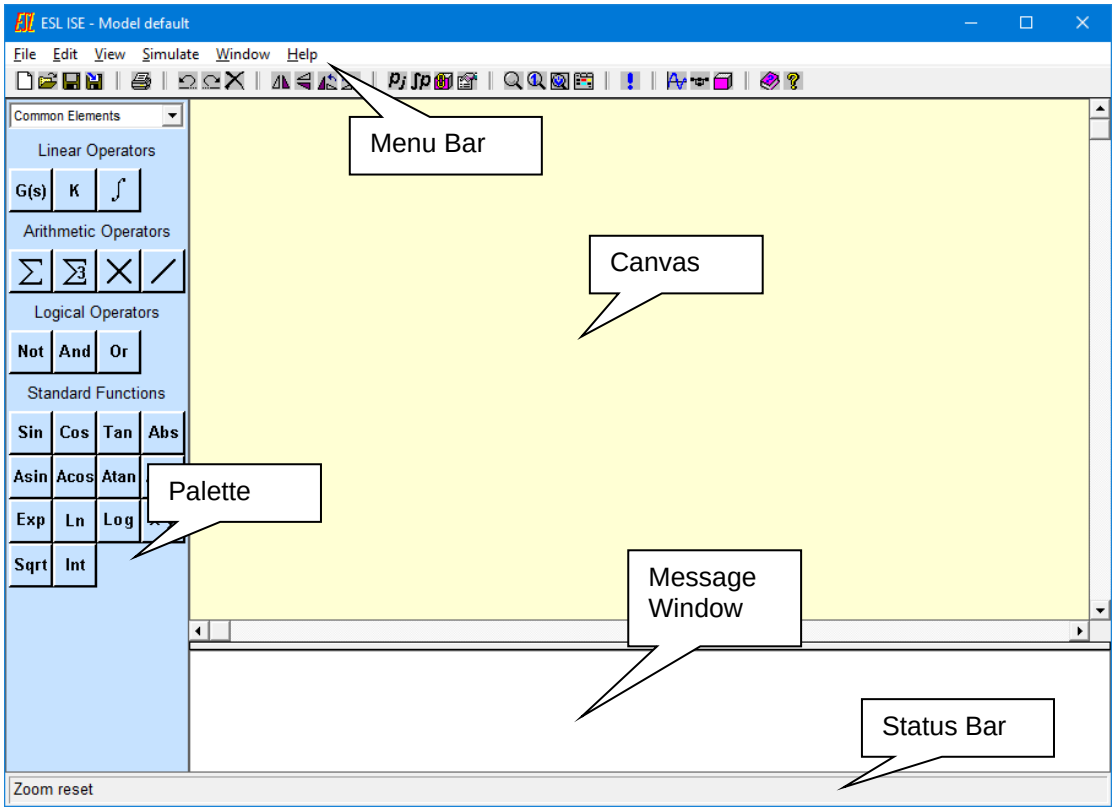

Figure 2 - ISE Main Window

By selecting the *Input/Output* and *Common Elements* panels from the palette combo-box, drag simulation elements onto the canvas in the arrangement shown in Figure 3.

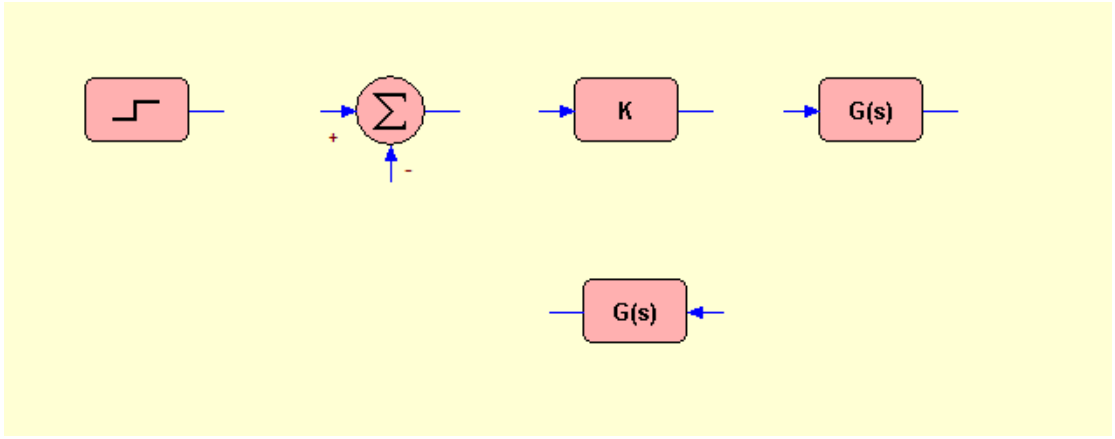

Figure 3 - Positioning Simulation Elements

To drag a simulation element onto the palette - position the pointer over the required element on the palette, hold the left mouse button down, move to the desired position and release the button (notice the name of the simulation element displayed in the Status bar during this operation). Elements can be repositioned on the canvas by selecting and dragging in a similar manner. The horizontal orientation of the lower Transfer Function element has been reversed by choosing a Left/Right Flip from the element short-cut menu (right mouse click).

The simulation elements can now be interconnected as shown in Figure 4. To do this, select a simulation element termination, left mouse click, extend the signal line to the next termination and complete the connection with a further left mouse click. Intervening nodes are created by additional left mouse clicks. Note the advisory messages that appear in the Status bar during these operations.

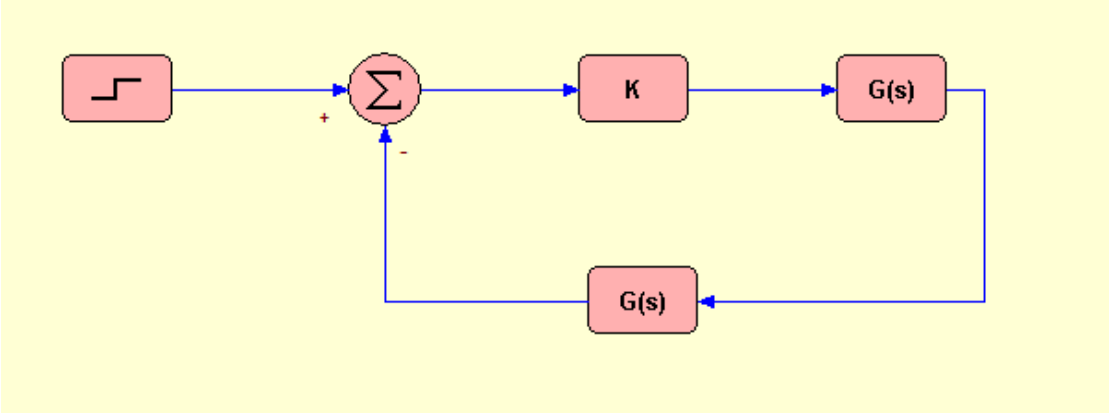

Figure 4 - Interconnecting Simulation Elements

Signal lines started in error can be abandoned by a right mouse click. Existing signal lines and nodes can be removed through their appropriate short-cut menu option.

2.2 Setting Simulation Element Attributes

The next stage is to set the simulation element attributes. You open the Attributes dialog for a simulation element by a left mouse double click or selecting Attributes from the short-cut menu (right mouse click). In the diagram you have constructed so far, all the simulation elements except the summer have associated attributes.

2.2.1 Step Input Attributes

The Step Input Attributes dialog has the appearance of Figure 5.

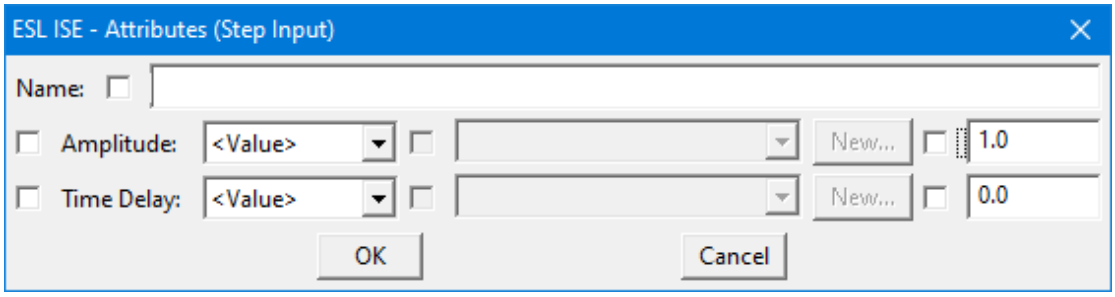

Figure 5 - Step Input Attribute dialog

There are three attributes: Name; Amplitude and Time Delay.

- Name is simply an identification that you want to give to this particular simulation element. It plays no functional part in the running of the simulation but will be displayed on the canvas below the simulation element (if you check its annotation box).
- Amplitude is the amplitude of the step input. It has a default value of 1.0.
- Time Delay is an optional delay from the start of the simulation run to the occurrence of the step input. It has a default value of 0.0, i.e. no delay.

The small boxes next to the attribute names and values are annotation boxes. Checking these causes attribute details to be displayed on the canvas close to the simulation element.

An attribute can be specified as:

- A literal value – simply accept or change the default value.
- A parameter – which can be changed interactively when ISE is in Run mode.
- A package variable (see later section on packages).
- An output – that is the output of another simulation element that has had its attributable box checked (see *Naming Simulation Element Outputs*).

The first drop-down box following the name of the attribute allows you to choose which option to use. The second drop-down box will show the names of any existing parameters, package variables or outputs that can be assigned to the attribute. A new parameter can be created by clicking the New button. In this case we will specify a Name, accept the default values for Amplitude and Delay and click the annotation check boxes to cause the Name and Amplitude attributes to be displayed on the canvas. The completed dialog is shown in Figure 6 and the resultant appearance on the canvas in the completed diagram Figure 11. Click OK when complete.

Note: *A Parameter is an ESL data item that has scope within the program module in which it is declared. It must have an initial value but this can be changed at run time through ISE, the ESL Interact service or a driver file.*

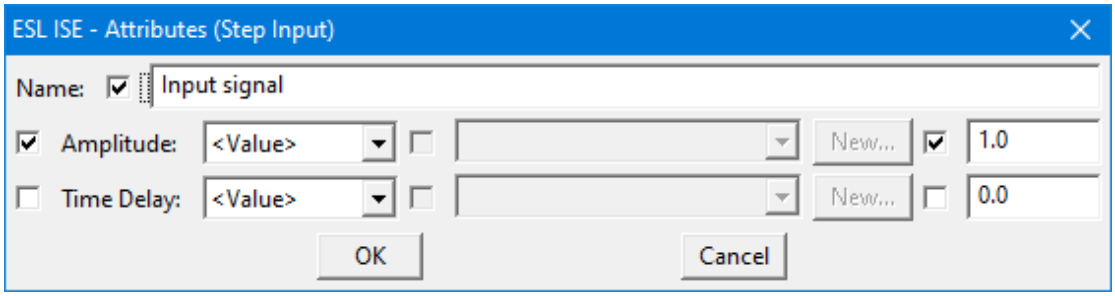

Figure 6 - Completed Step Input Attributes

2.2.2 Constant Multiplier Attributes

In the case of the Constant Multiplier simulation element, we will assign a Parameter called Gain with a value of 2.0 to the single Coefficient attribute. To do this select Parameter from the first drop-down box, click the New button in the Constant Multiplier Attributes dialog and enter the details as shown in Figure 7. On closing the New Parameter dialog, the Parameter and value check boxes in the Attributes dialog should be checked to give the appearance on the canvas shown in Figure 11.

Note: *The New Parameter dialog lets you specify a "true" Parameter, i.e. a value that can be changed interactively when ISE is in Run mode; a Constant, whose value can only be changed in Edit mode and a Variable which can be changed programmatically.*

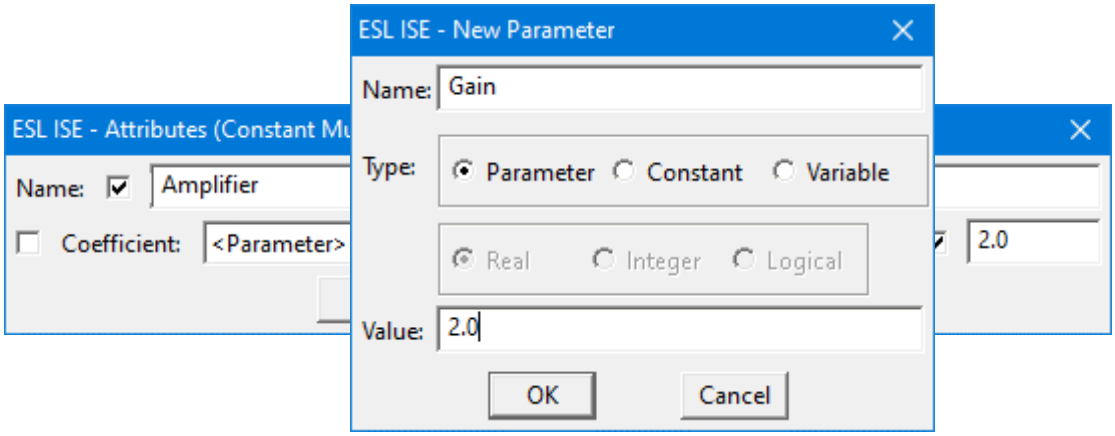

Figure 7 - Constant Multiplier Attributes

2.2.3 Transfer Function Attributes

The Transfer Function Attributes dialog differs from that of the other simulation elements and has the appearance of Figure 8. You can enter the numerator and denominator of the transfer function to be represented. Figure 8 shows how you should complete the dialog for the Transfer Function element in the forward path of the control system loop, representing the plant to be controlled. (See the on-line help for details of transfer function syntax.)

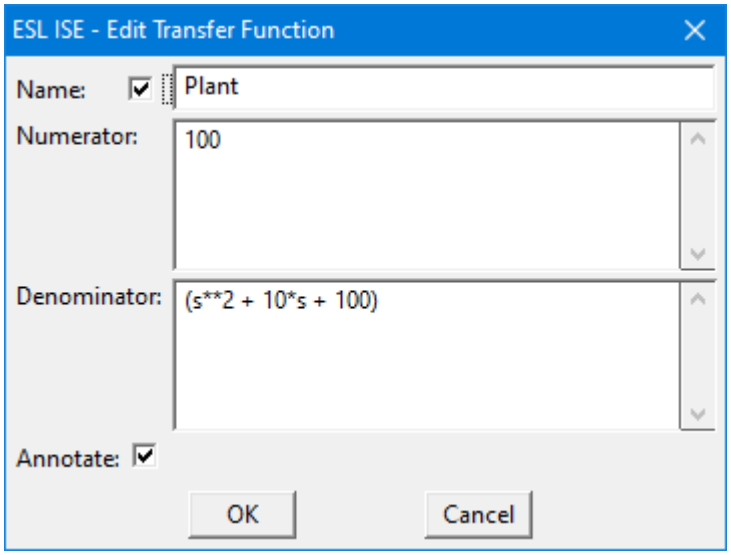

Figure 8 - Forward Path Transfer Function Attributes dialog

Before setting the return path Transfer Function element attributes, we will use a different method to define a Parameter that will be used in the transfer function. Select Parameters from the Edit menu. This will open the Edit Parameters dialog, listing the parameters declared so far (only one – Gain, tagged with an * indicating that it is in use). Click the New button and define a Real parameter Tc with a value 0.1 (

Figure 9).

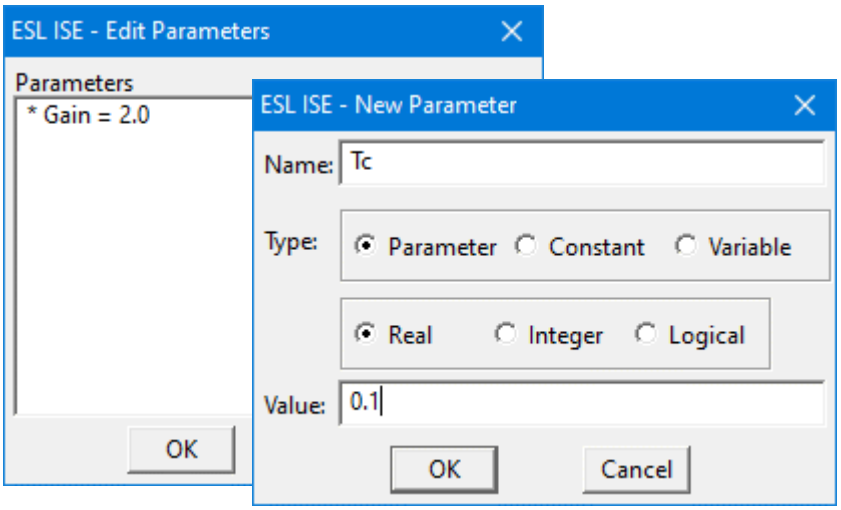

Figure 9 - Edit Parameters dialog

Note: *When defining a parameter from a simulation element Attributes dialog, the appropriate type (Real, Integer, Logical) is pre-set and determined by the simulation element attribute. Whereas from the Edit menu, parameters of any type can be defined but Real is selected as the default.*

Now you can set the return path Transfer Function element Attributes, as shown in Figure 10 and include the newly created parameter in the denominator.

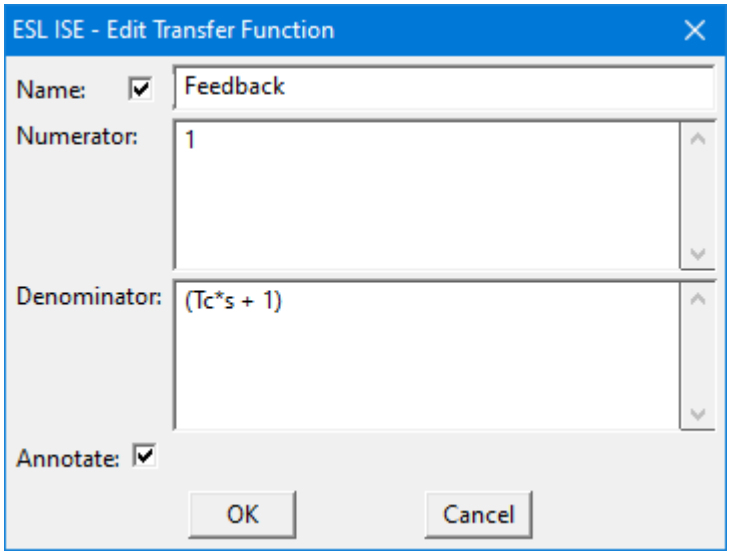

Figure 10 - Return Path Transfer Function Attributes dialog

If all the appropriate annotate check boxes have been clicked, the diagram should now have the appearance of Figure 11.

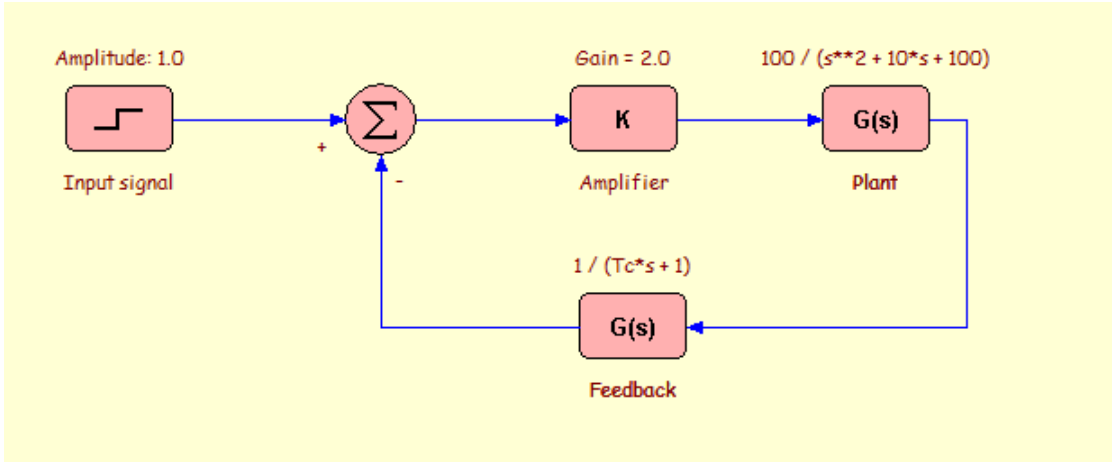

Figure 11 - Completed diagram

2.3 Naming Simulation Element Outputs

When ISE creates an ESL program from an ISE diagram, it generates ESL variables of the form O__nnnnn to represent the outputs of the simulation elements. In order to identify simulation element outputs at run time, you can give them more meaningful names. You do this with a right mouse click on the simulation element output termination and select Output Attributes. This opens the output variable dialog in which the name is entered (Figure 12). Clicking the annotation checkbox will display the names on the canvas. Figure 13 shows the diagram with appropriate names given to the simulation element outputs in the forward path.

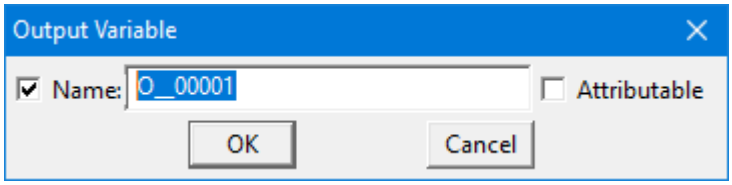

Figure 12 - Output Variable dialog

Note: *Checking the Attributable box allows an output to be used as an attribute in another simulation element.*

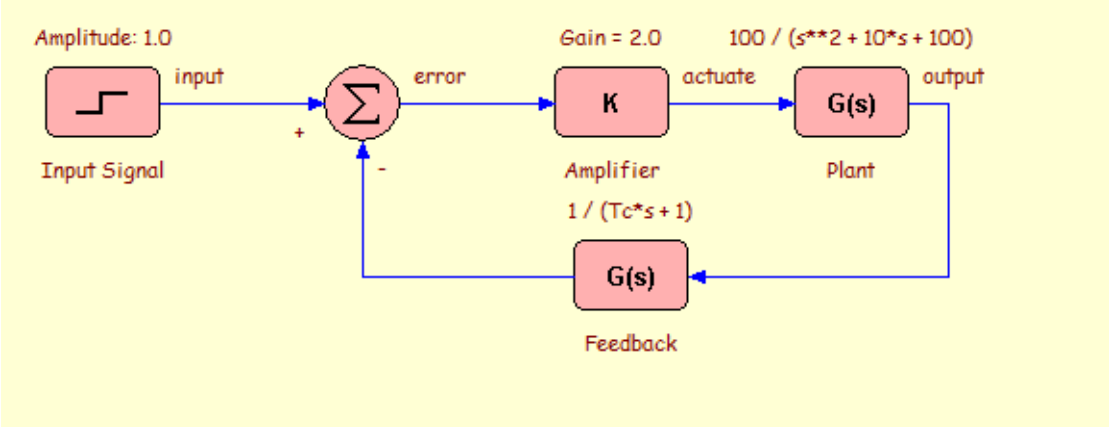

Figure 13 - Simulation Element output names added

2.4 Set Model Name

If you right-click on a blank area of the canvas and select Properties (or double left-click) the Module Properties dialog will be opened (Figure 14). Here you can select the type of ESL module to be created. The default is an ESL Model – which is what we want. The other types of module, Embedded and Remote Segments, are described in section 7.2. The default name is “default”, change this to “example”. It is not essential to do this – it will make subsequent dialogs and the generated ESL code more readable.

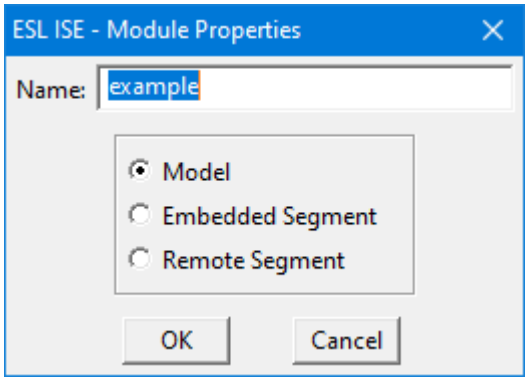

Figure 14 - Module Properties

2.5 Specifying Output

There are two ways of specifying output from an ESL program:

- by placing Display Icons on the canvas and connecting them to the outputs to be displayed
- using the Runtime Displays from the Simulation Execution Control panel (see section 2.7)

The main difference is that Display Icons are placed and connected during the editing phase and therefore become part of the block diagram, whereas the Runtime Displays lets you specify *and change* output specifications at run-time. Note that the Runtime Displays is the only way of specifying output when running an external ESL program (i.e. an ESL program created textually, running from ISE – see Chapter 5). We will use Display Icons.

Select the Extra panel on the palette and drag a Plot element and a Table element onto the canvas. Choose Connect from the Plot element short-cut menu and connect to the outputs of the Step Input and the Plant Transfer Function simulation elements. To do this, extend the instrumentation line that appears when Connect is selected to the required output termination node and left mouse click. Then extend a second line to the other node and left mouse click.

The connection operation is terminated by a right mouse click. Similarly, connect the Table element to the same nodes. The diagram should now have the appearance of Figure 15.

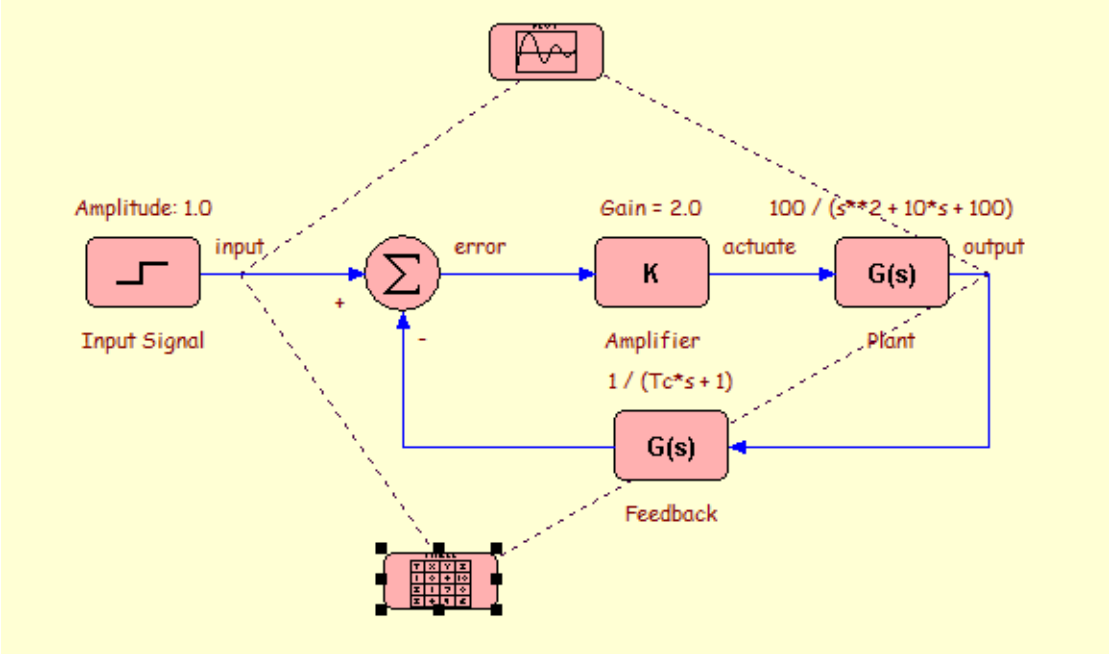

Figure 15 - Display Icons connected

Selecting Properties from a Display Icon short-cut menu allows Graphical or tabular output to be configured. (If you do not do this, default settings will be used.) Figure 16 shows the Plot Properties dialog with the Title and Subtitle properties completed and Display Grids selected for both the x- and y-axes. Note that in this case Auto Scaling has been selected for the X axis whereas the Y axis has been specified explicitly. A corresponding dialog may be opened for the Table display element. You should set the Plot and Table properties.

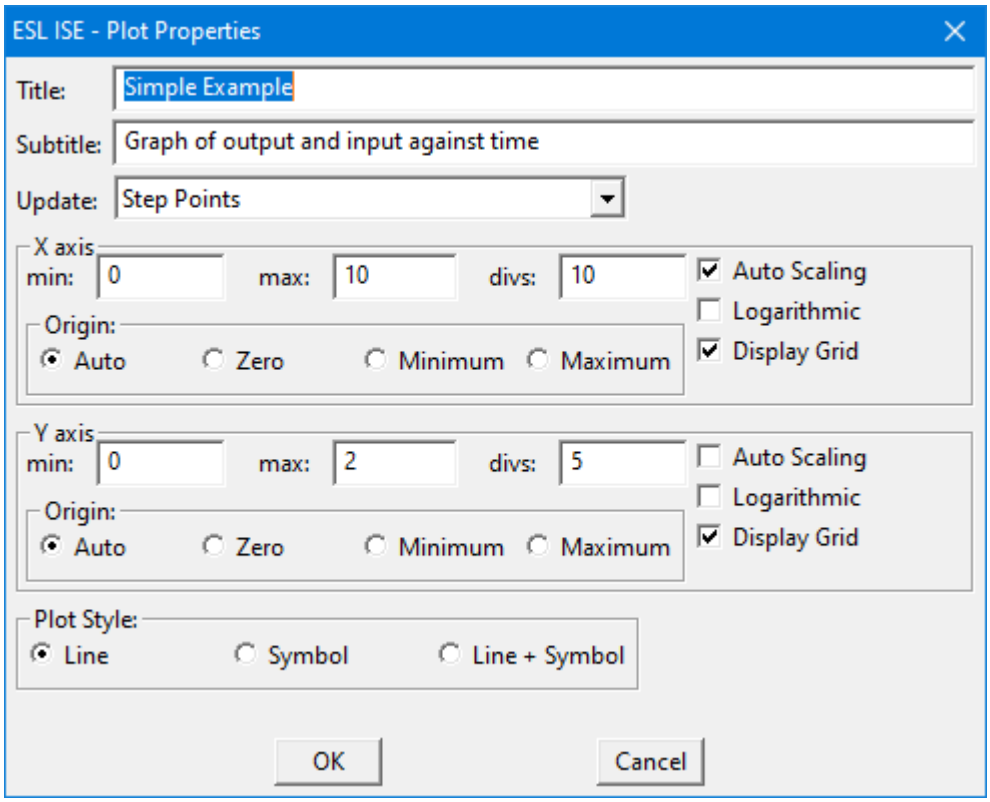

Figure 16 - Plot Display Properties

Note: *The Update property allows the frequency of output to be selected: integration Step Points; Communication Points or Communication Points and Discontinuities. Generally Step points are best for graphical output and Communication points for tabulated output.*

2.6 Saving the Application

At this point (or indeed at any point) you should save the application by selecting Save As from the File menu and specifying a suitable name e.g. *example* (the extension .ise will be automatically appended to the file name).

2.7 Running the Simulation

You have now created a block diagram representing the system to be simulated, entered all the relevant data and are in a position to run the simulation. This is initiated from the Simulate menu. First select the Setup option, which opens the dialog shown in Figure 17

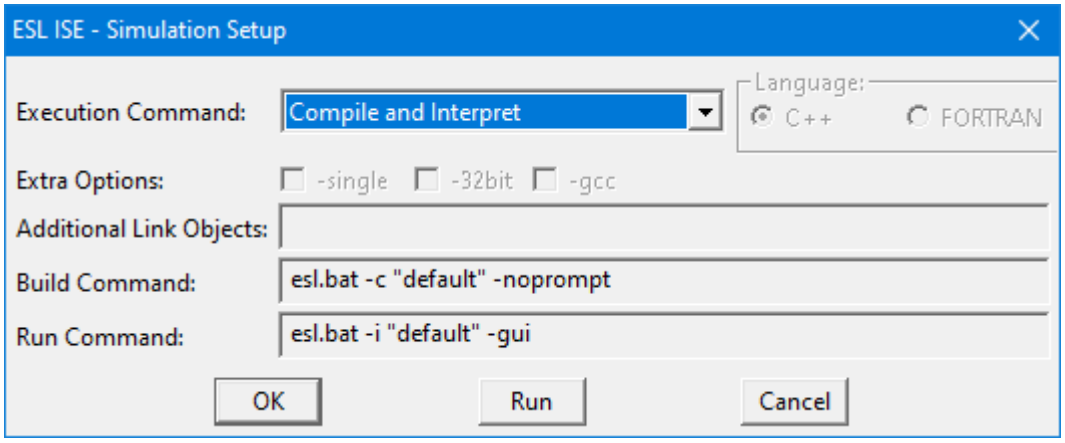

Figure 17 - Simulation Setup dialog

This dialog gives you the option to choose between the Interpret and Translate execution modes. Further, if you choose Translate, you can select C++ or Fortran as the target language.

Note: *When you run an ISE application, an ESL program is first created from the block diagram and compiled into an intermediate code known as h-code. The h-code may be interpreted directly or further translated into Fortran or C++ from which an executable program is created.*

Unless you have an appropriate C++ or Fortran compiler installed, leave the selection as the default interpret mode and click Run. (Assuming Simulation Setup is correctly configured, you can select Run directly from the Simulate menu.) ISE is now in the Run mode. This will open the Simulation Execution Control Panel (Figure 18) and also Plot and Table (Trend) windows (corresponding to the display elements on the block diagram).

Note: *When started from ISE, an ESL program runs as a separate process and communicates with ISE via the ESL-SEC program through a special Simulation Execution Control protocol.*

The simulation is now running and awaiting commands from the Control Panel. Notice the various Run options: to End of Run; to Next Communication Point; to Next (integration) Step Point; to Next Module (if the application has a model-submodel structure) and to a specified Time. There are also buttons for: Rerun; Restart; End Simulation and Finish. There are buttons for: Simulation Parameters; Variables; Runtime Displays and Advanced Simulation Options. These are discussed later.

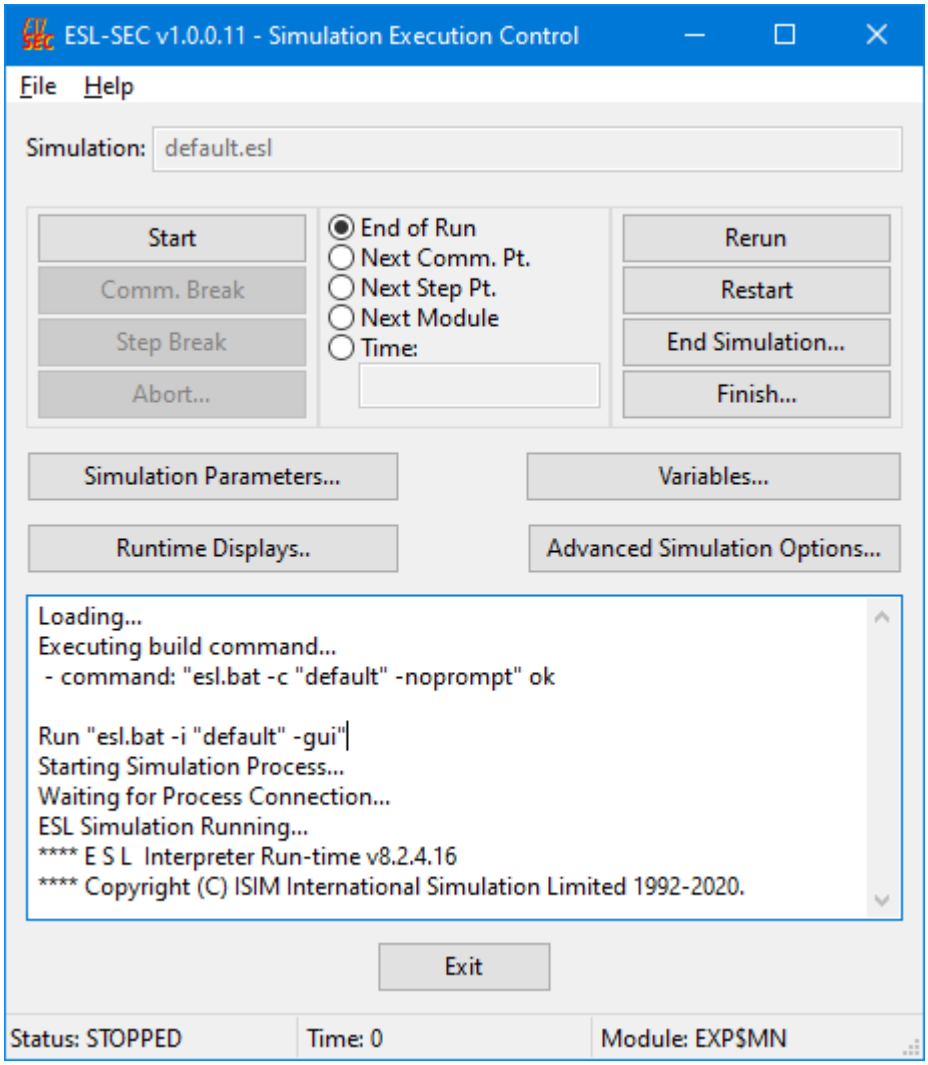

Figure 18 - Control Panel

First click the Simulation Parameters button. This opens a dialog giving you access to certain reserved variables used to control the running of the simulation - the Simulation Parameters (Figure 19).

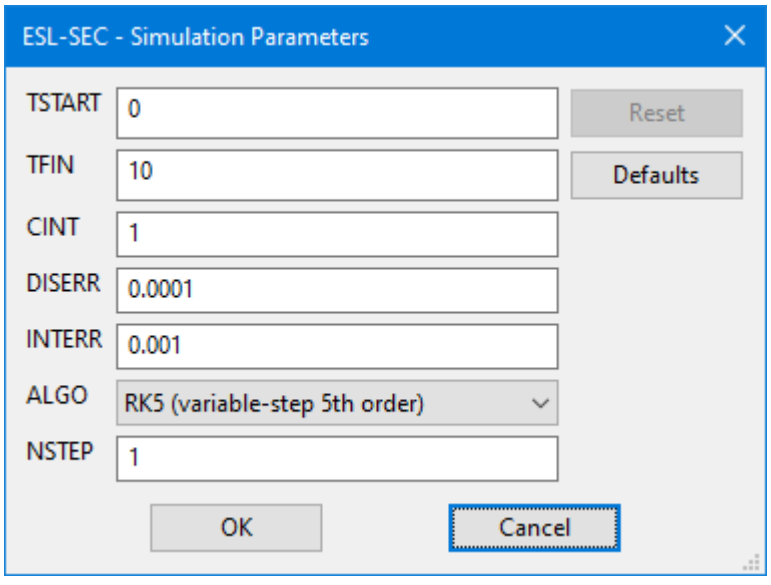

Figure 19 - Simulation Parameters dialog

The Simulation Parameters are:

- Tstart start time for a simulation run
- Tfin end or final time for a simulation run
- Cint communication interval for output at regular time intervals
- Diserr discontinuity detection error tolerance
- Interr integration algorithm error tolerance
- Algo choice of numerical integration algorithm
- Nstep minimum number of integration steps per Cint

The values shown are default values. The default RK5 integration algorithm is a variable step routine, which will automatically adjust the integration step-length and take additional integration steps during each communication interval when necessary to satisfy the error tolerance requirements. However, for this exercise, you should change Nstep to 10 in order to generate smoother graphical output by generating more points. Click OK to close the Simulation Parameters dialog.

Now run the model by clicking the Start button on the Control Panel. Figure 20 shows the output you should see.

Referring back to the original problem (Figure 1), we see that we have a proportional control system with a gain of 2.0. Since the steady-state characteristics of both the plant and feedback elements are unity, the system output should settle to a final value of 0.6667 for unity step input, as verified by the simulation.

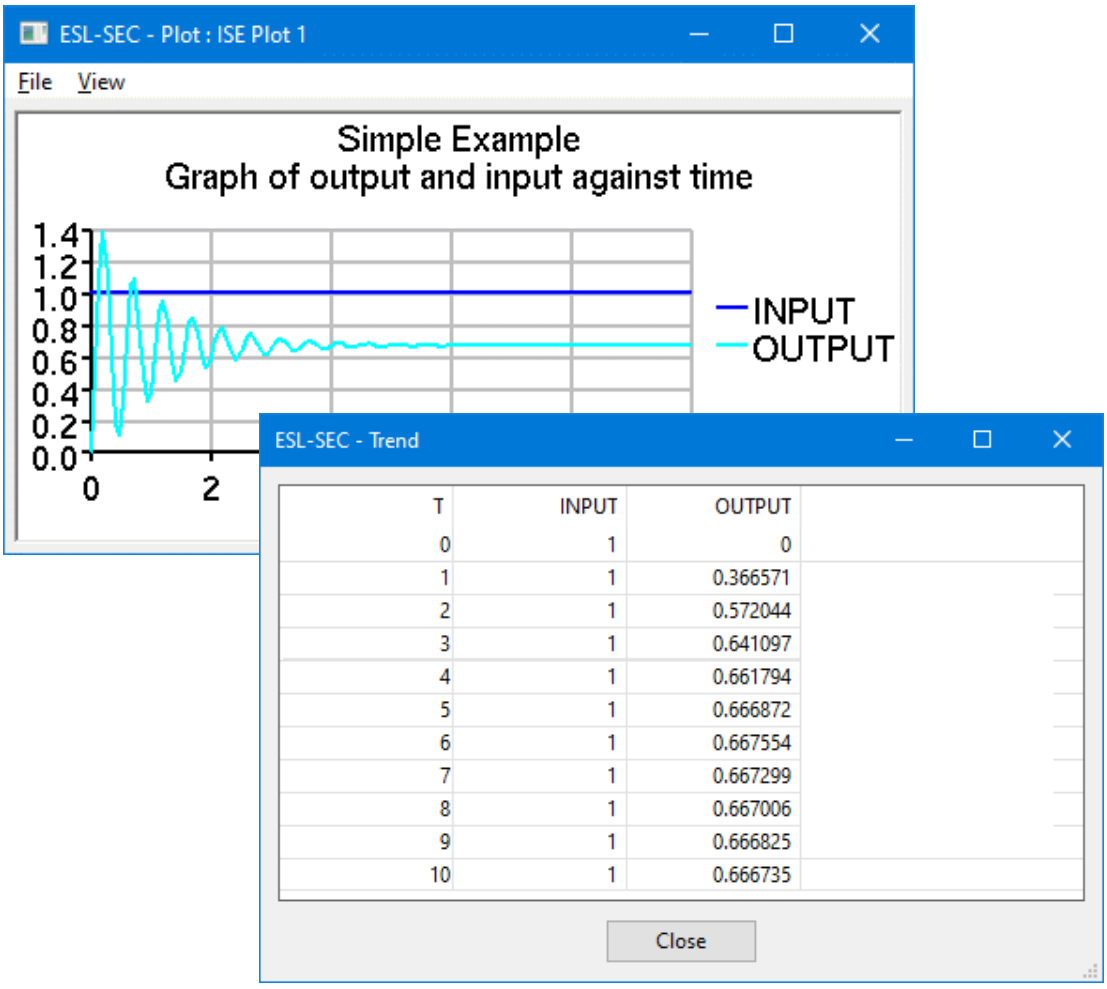

Figure 20 - Graphical and Tabular output for Simple Example

2.8 Varying Parameter Values

Having made one run of the simulation, you will typically want to investigate the effect of varying one or more parameter values. In ESL, any parameter created from the simulation element attributes dialogs or from the edit menu Parameters option may have its value changed at run-time. From the Control Panel, click the Variables button to open the Variables dialog (Figure 21). This gives you access to all parameters and variables in the modules contained in the application.

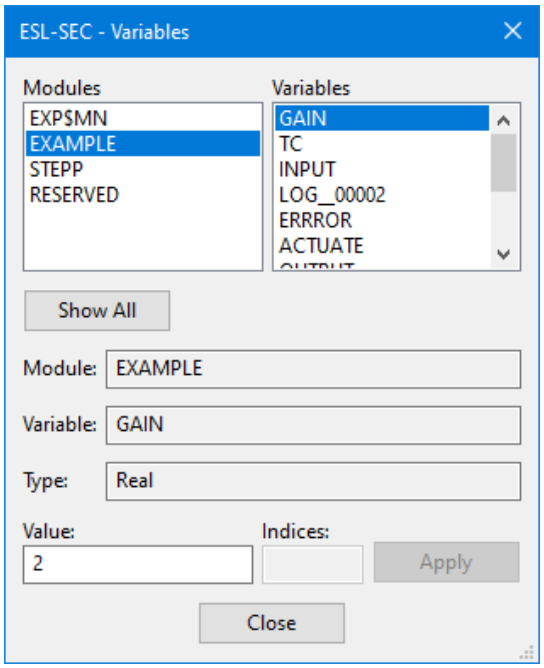

Figure 21 - Variables dialog

Select EXAMPLE (the name of the main model in this case) from the Modules list and GAIN from the Variables list. Note the details of GAIN displayed and change the value from 2 to 2.5 (say) and click Apply. To re-run the model, click Rerun followed by Continue on the Control Panel. A second graph will appear in the plot window as in Figure 22. Note the reduced steady-state error but increased oscillation caused by the increase in gain.

You should try varying the gain further or changing the value of the feedback time constant parameter Tc in a similar manner. Click Rerun and Continue on the Control Panel to initiate each new run.

On completion of running the model, exit the Run mode by clicking the Exit button on the Control Panel.

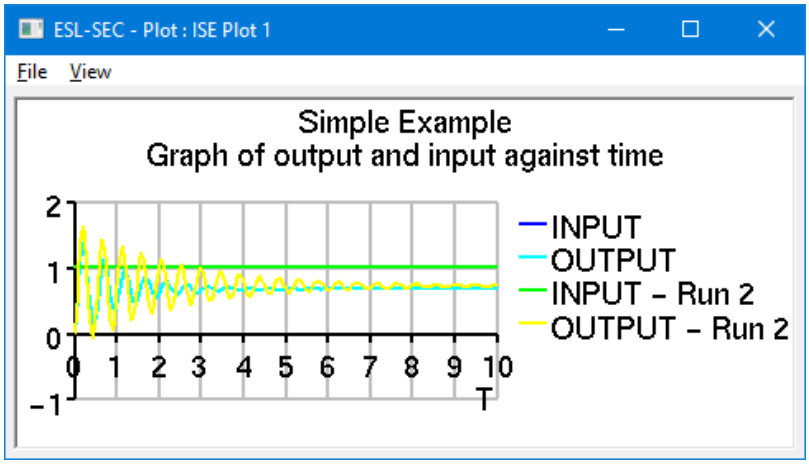

Figure 22 - Second Graph for Gain = 2.5

2.9 Using Runtime Displays

As stated at the start of section 2.5 , an alternative way of specifying output is through Runtime Displays, which can be opened from the Simulation Execution Control Panel. Run the previous example (or go to the Control Panel if the simulation is still open) and click the Runtime Displays button. The appearance should be as Figure 23. Under the Plot tab you will see the specification of ISE Plot 1 set from the corresponding Display Icon. Similarly, ISE Table 1 will be seen under the Table tab. To specify a new runtime plot, go back to the Plot tab and click New. Accept the default name, Plot 2, or set a different name. Select a Module from the left-hand panel (EXAMPLE is the main model in this case) and select a Variable from the right-hand panel, for example ERROR. Click Add (or double click the variable) to add it to the Contents panel. Note that the dependent variable is pre-set as (RESERVED) T, but this can be changed using the buttons on the right. Properties of the plot can be set from the Properties button (as with the Display Icon settings). Finally click Show Display to open a plot window. In a similar manner a new table can be specified under the Table tab. Close the Runtime Displays window and Start the simulation. The new displays should be as Figure 24.

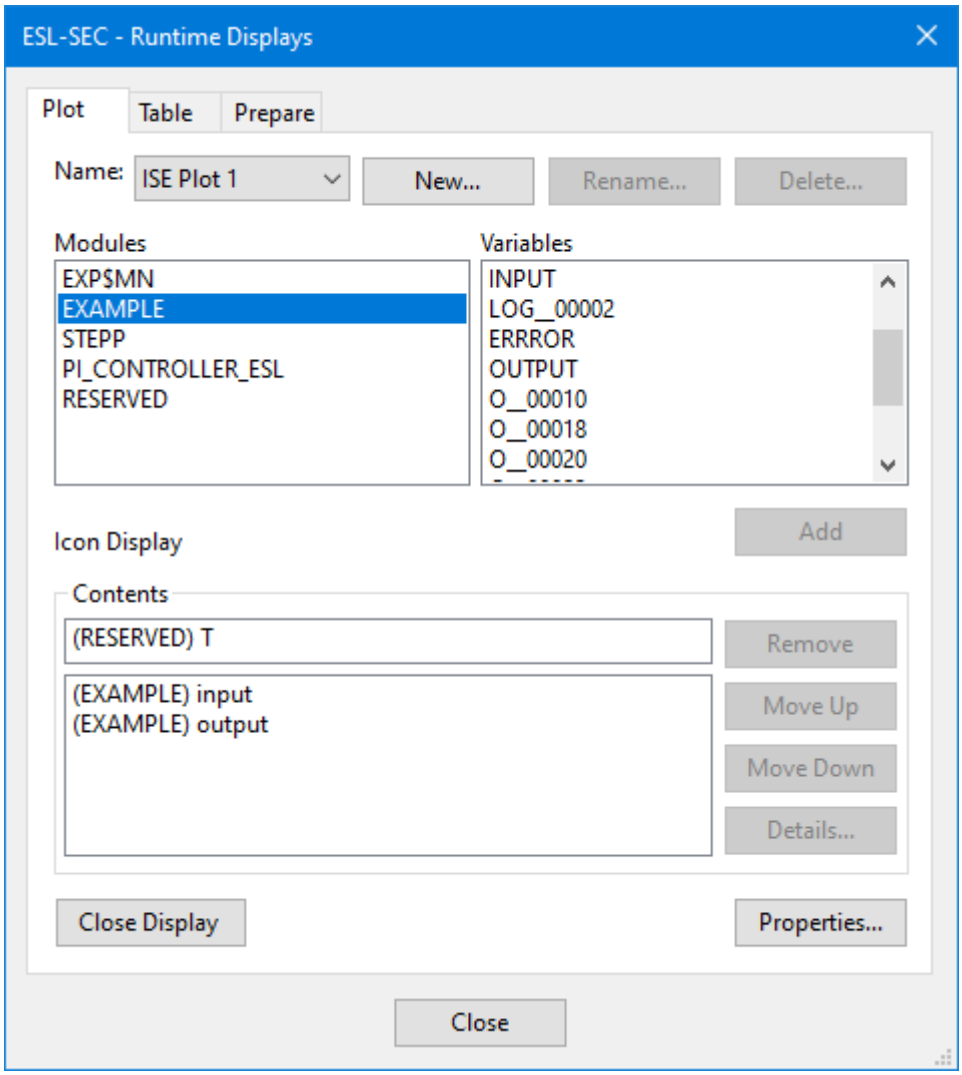

Figure 23 - Runtime Displays

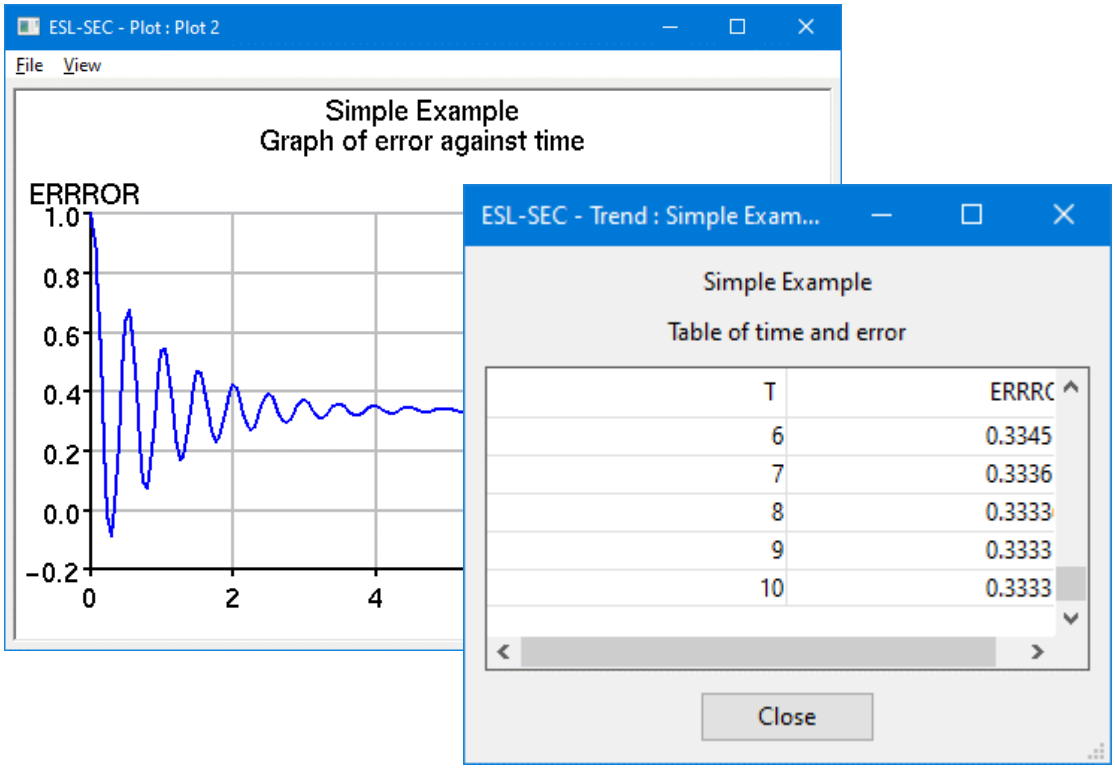

Figure 24 - Runtime Displays

2.10 Offline Display Analysis

The third tab in the Runtime Displays dialog is Prepare. This feature allows you to direct output to a prepare file (.dsp) for offline analysis using the ESL-Displays program. You set up a prepare file in a similar manner to Plots and Tables. Select the Prepare tab in Runtime Displays and choose the variables to be saved (Figure 25). From Properties you can set a title; subtitle; frequency of output and a filename (Figure 26). Click Create Display. When the simulation is run, the prepare file will be created.

Any prepare files that have been created can be accessed from Display Manager off the Window menu selection. This opens ESL-Displays (Figure 27).

Note: *ESL-Displays can also be started from a command prompt – esl_displays – useful for analysing data from outside the ISE environment.*

From the Load button you can load one or more prepare files. The file of interest is selected from the Display Files panel (example.dsp in this case) and variables for plotting from the Variables panel (in a similar manner to Runtime Displays). Variables may be selected from more than one file allowing, for example, comparisons to be made from different runs. The Export button allows a prepare file to be converted and saved as a readable Tab file. Properties are set as in Runtime Displays and clicking Plot generates the display (Figure 28).

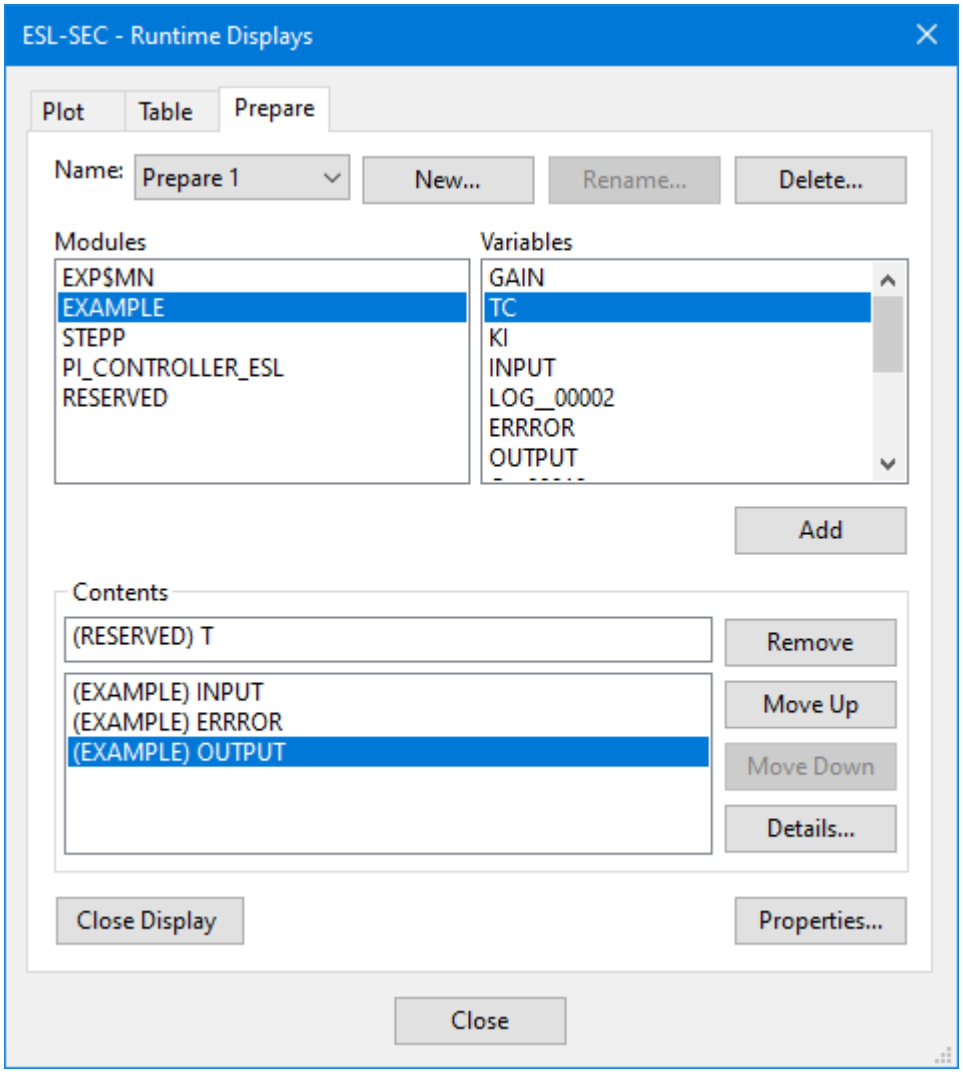

Figure 25 - Prepare file specification

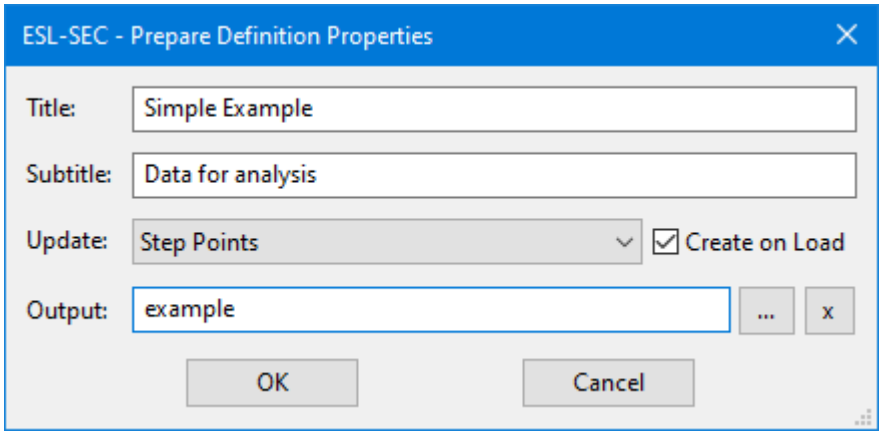

Figure 26 - Prepare properties

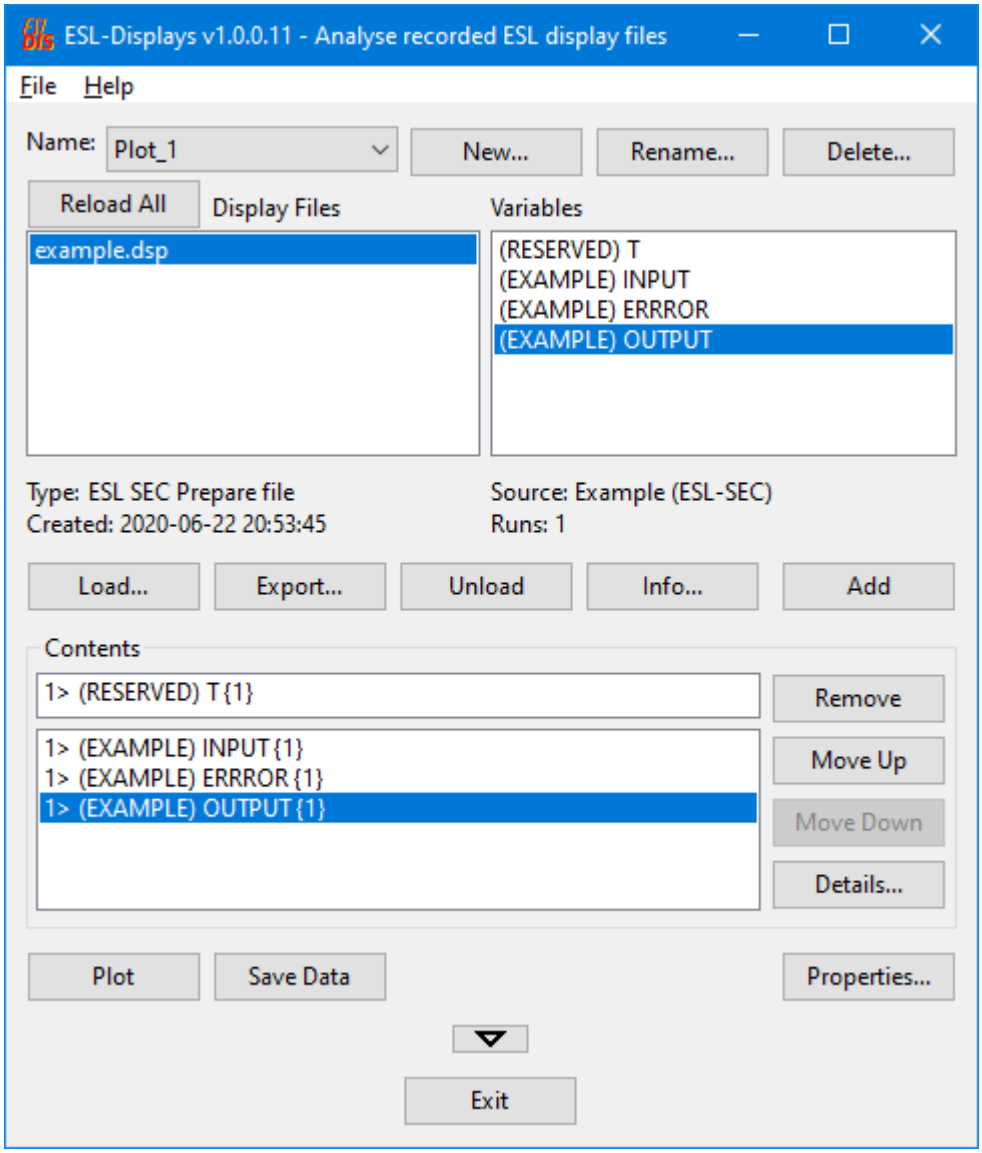

Figure 27 - ESL-Displays

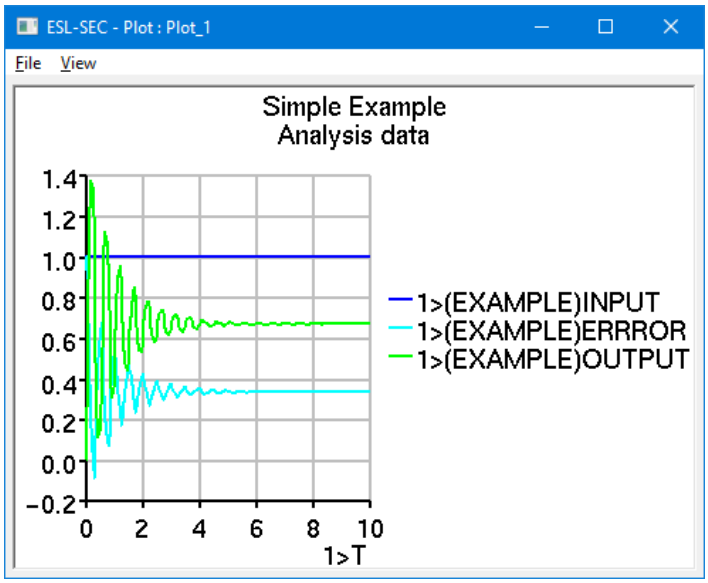

Figure 28 - Prepare file display

2.11 Further Exercises

Try replacing the Step Input simulation element with a Sinusoidal, Ramp or Square-Wave input element.

CHAPTER 3

Extending the Example - Submodels

In this chapter you will be shown how to extend the example that you created in the previous chapter. This will introduce:

- graphically defined submodels

The simple control system example introduced in Chapter 2 used, in effect, a "proportional" control law (represented by the Constant Multiplier element in the forward path. Suppose you wanted to use a "proportional plus integral" control law. This could be achieved by replacing the Constant Multiplier with a submodel.

Note: *There is in fact a standard PI control element to be found on the Library (Linear) panel on the palette. Building one from first principles is simply being used to illustrate the general procedure for creating submodels.*

3.1 Defining a Submodel

First of all, delete the Constant Multiplier (right mouse click the element and choose Delete from the short-cut menu). Then select the Extra panel of the palette and drag a submodel element onto the canvas in the space previously occupied by the Constant Multiplier. Right click the submodel element and select Set Definition>New Internal (Figure 29). Change the name in the New Submodel Definition dialog (Figure 30), leave the Graphical radio button selected and click the Edit button. This will open a new ISE submodel window, similar to the main window.

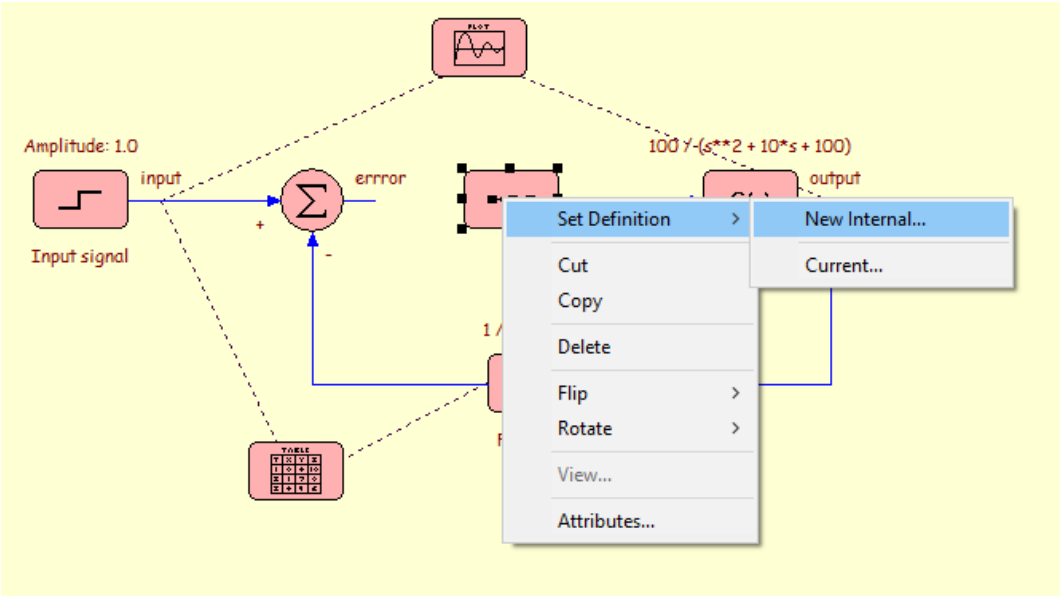

Figure 29 - Inserting a submodel

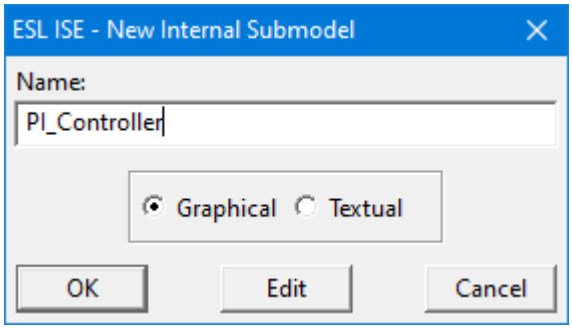

Figure 30 - New Submodel definition

3.2 Building the Graphical Submodel

The control law to be represented by the submodel is:

$$y = Gain\left(x + Ki \times \int x \, dt\right)$$

where *x* is the input error signal, *y* is the output actuation signal to the plant and *Gain* and *Ki* are parameters.

In the submodel window, drag three Real Input Argument elements and one Real Output Argument element from the Input/Output panel of the palette onto the canvas for the submodel inputs and output. Build the rest of the submodel using simulation elements from the Common Elements panel as shown in Figure 31.

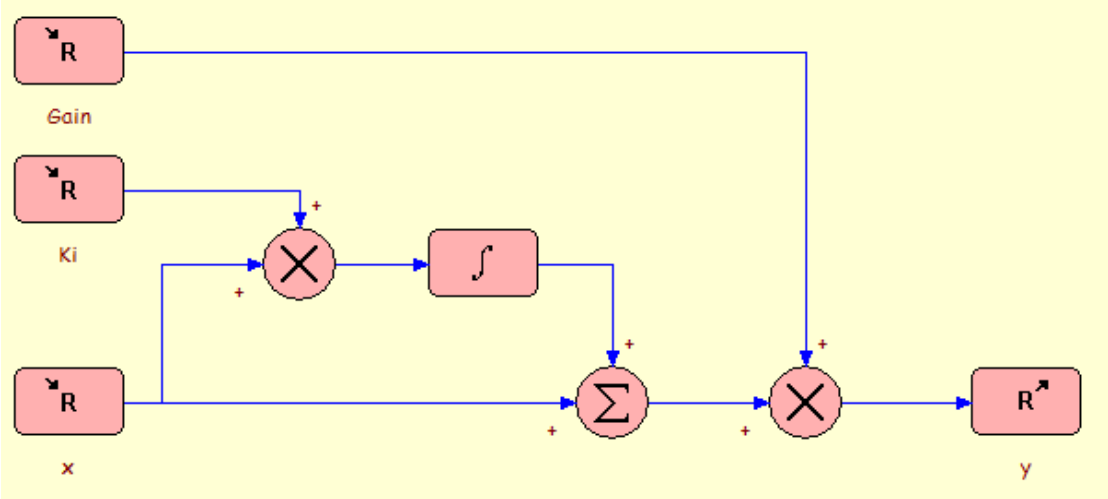

Figure 31 - Graphical PI Controller submodel

Note that the attributes of the Input and Output Argument elements are simply the names of the submodel arguments. You should double click the top termination of the Summer element to change its sign from - to +. The integrator initial condition attribute should be left as its default value of zero.

When you have completed the submodel diagram, click Close on the File menu. The Submodel element in the main window will now have input and output terminations, which you can connect up to complete the diagram as in Figure 32. Note that the Gain and Ki submodel inputs are provided by constant input elements with parameter attributes (the existing Gain parameter can be reused but a new parameter, Ki, has to be created). The values chosen for these parameters are such that there is initially no integral control action and the gain is the same as before. Annotation has been added above the Controller element using the Text element from the palette Extra panel.

You can display the submodel diagram at any time by a right mouse click on the submodel element and choosing Edit or by double clicking on the element.

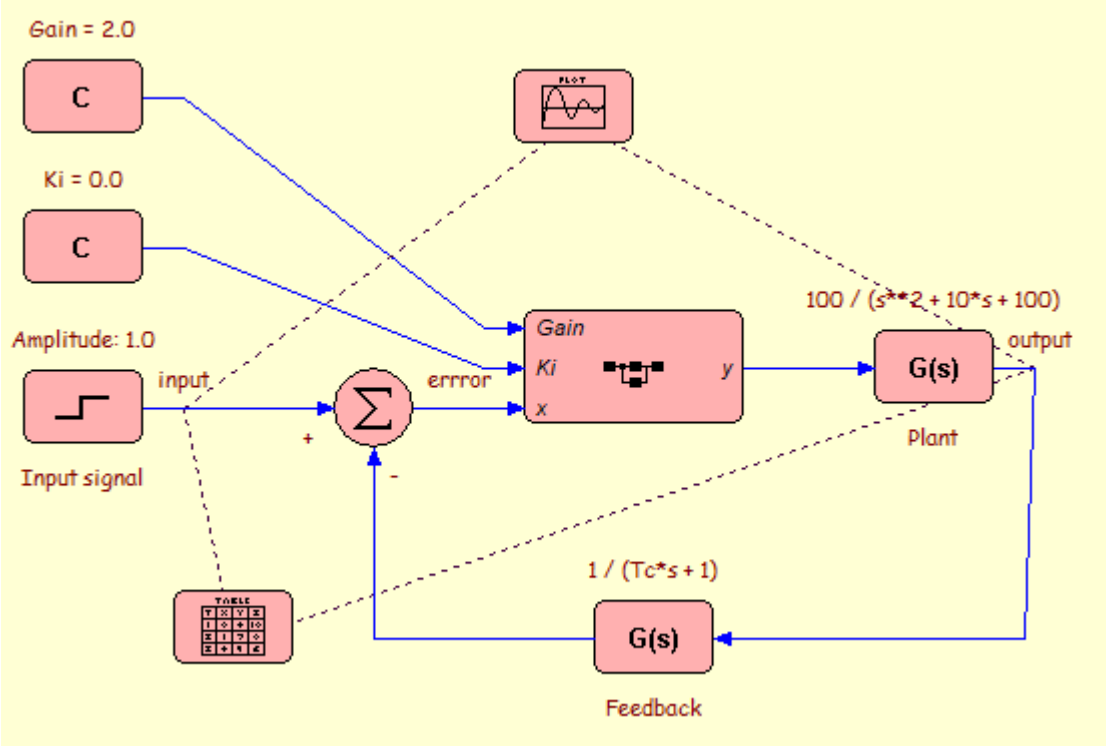

Figure 32 - Simple example with PI Controller

3.3 Running the Modified Model

Now run the model by clicking Run in the Simulate menu. Ensure that the parameter Tc has its original value of 0.1 by opening the Variables dialog and selecting TC from the EXAMPLE module. Start the simulation by clicking the Start button on the Control Panel. The graphical output should be identical to that obtained for the first run of the model in its original form (Figure 20).

Now try changing the values of the PI Controller submodel parameters and re-running the model. Change the parameter values from the Variables dialog by selecting the EXAMPLE module. Remember to click the Control Panel Rerun button followed by the Continue button to rerun the model. Suggested values, close to the optimum, are:

Ki = 5.0 (corresponding to an integral time constant of 0.2s)
Gain = 0.8

The above values should give the results shown in Figure 33 where the second graph corresponds to the new parameter values.

This example shows the effect of the integral control action in bringing the output of the control system quickly to the input demand value. Try experimenting with different parameter values.

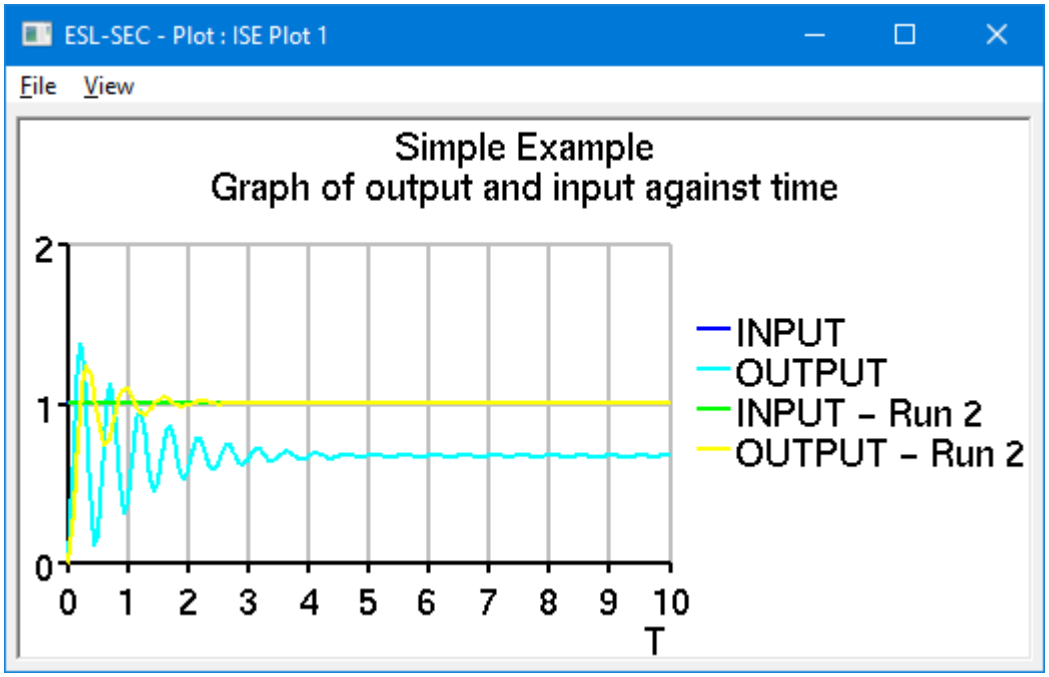

Figure 33 - Results using PI Controller

3.4 Submodel Definitions and Instances

It is important to understand that the PI_Controller submodel element on the main diagram represents an "instance" of the submodel not the submodel itself. That is, further Submodel elements can be dragged onto the diagram and have their definitions set to PI_Controller (by selecting Set Definition > Current from the Submodel element short-cut menu and selecting PI_Controller from the list of Internal submodels).

CHAPTER 4

A Textual Submodel

ESL supports both graphical and textual methods of system description. In this section you will replace the graphically defined controller with a textually defined submodel. This will introduce:

- use of the text editor
- an introduction to the ESL language

4.1 Inserting a Textual Submodel

To illustrate the use of textual submodels, you will replace the graphically defined PI Controller created in the previous section with the equivalent written directly in the ESL language.

First delete the instance of PI_CONTROLLER by a right mouse click and choosing Delete from the short-cut menu.

Note: *Note that the submodel itself remains available and can be assigned to a submodel element at any time. To remove the submodel altogether you should open the Submodel Manager off the Window menu and delete the submodel definition in there.*

Drag a Submodel element off the Extra panel of the palette and select Set Definition>New Internal. Give the definition a different name to the graphical version - PI_Controller_esl, say and click the Textual button. Click the Edit button and this will open a text editor (usually Microsoft Notepad) displaying the following skeletal code for an ESL submodel.

```
-- Embedded Text Submodel
SUBMODEL PI_Controller_esl ();
INITIAL

DYNAMIC

END PI_Controller_esl ;
```

Edit the code to create the controller submodel as below.

```
-- Embedded Text Submodel
SUBMODEL PI_Controller_esl (REAL:y := REAL:Gain,Ki,x);
  REAL:int_x;
INITIAL
  int_x := 0.0;
DYNAMIC
  int_x' := x;
  y := Gain*(Ki*int_x + x);
END PI_Controller_esl ;
```

The lines of code are explained below:

`-- Embedded Text Submodel`	a comment, all lines commencing with -- are treated as such
`SUBMODEL PI_Controller_esl (REAL:y := REAL:Gain,Ki,x);`	submodel definition statement defining inputs and outputs
`  REAL:int_x;`	declare a variable to represent the

```
                                    integral of x
INITIAL                             start of initial region
  int_x := 0.0;                     initialise integral variable
DYNAMIC                             start of dynamic region
  int_x' := x;                      differential equation to integrate x
  y := Gain*(Ki*int_x + x);         control law
END PI_Controller_esl ;             submodel end statement
```

This simple example illustrates the basic structure of a model or submodel. Following the definition statement, any local data are declared. In the initial region state variables are given initial values and any start-of-run calculations are executed. The dynamic region contains the differential equations and statements that form the mathematical model. Further detail of the ESL language is given in Chapter 5.

Note that while editing text, you cannot make further changes to the block diagram. Save the file and close the editor. Reconnect the submodel inputs and output and the model in the main ISE window should now have the same appearance as when the submodel was defined graphically (Figure 32). However, double clicking on the PI Controller submodel instance will now re-open the text editor. The program can be run in the same manner as before.

The advantage of being able to create textual submodels is that often some parts of a simulation are more easily described in terms of equations rather than by a graphical block diagram. It may be the model description has been provided in equation form and it makes sense to enter it as-is rather than convert it to a diagram. Also, parts of a system that are highly non-linear, particularly if they contain discontinuities, are more naturally described textually. ESL gives you the ability to combine graphical and textual system descriptions.

CHAPTER 5

The ESL Language

The heart of an ESL simulation is the actual ESL program. The ESL language is extensive and is described in detail in the sections of the on-line help. In this section you will be introduced to the main features of the language through examples.

5.1 Program Structure

A standard ESL program is termed a Study and contains several different kinds of program module, as shown below:

```
Study

  <packages>
  <procedures>
  <submodels>
  <model>
  <experiment>

End_study
```

A study normally contains a single model and experiment and, optionally, one or more packages, procedures and submodels. Packages, procedures and submodels can appear in any order, provided each is defined before it is referenced.

For more advanced use, a study may contain more than one model but only one can be executing at any time. Models and submodels may be omitted entirely, in which case the study becomes a purely procedural program.

5.1.1 Packages

Packages provide a way of sharing data (variables, constants and parameters) between program modules. Named packages are referenced from a program module through the *Use* statement. Packages are also used to identify externally accessible data for embedded ESL programs.

5.1.2 Procedures

Procedures are static program modules containing purely procedural code where inputs and outputs are passed through an argument list. An alternative form, which may be included in expressions, is a function version which returns a single data value.

5.1.3 Submodels

Submodels are dynamic program modules containing modelling code (including differential equations). Submodels allow a large system to be modelled in a hierarchical manner. A single generic submodel may be instantiated any number of times to represent specific components of the system. ESL provides a standard library of common submodels. Submodels are called from the *dynamic* region of the model or other submodels.

5.1.4 Model

The model is the top-level program module containing modelling code. A simple system may require a model only; a more complex system will include several layers of submodels below the model.

5.1.5 Experiment

The experiment contains procedural code that defines how the model is to be run. It may be very simple - just calling for a single run of the model, or more complicated, perhaps involving several runs of the model with different parameter values.

5.2 Model and Submodel Structure

An ESL Model is divided into a number of regions identified by a keyword. The general structure is shown below:

```
Model<name><argument list>;
  <declarations>
  Initial
    <initialisation code>
  Dynamic
    <modelling code>
    Step
      <integration step code, e.g. plotting>
    Communication
      <communication point code, e.g. tabulation>
  Terminal
    <end of run code>
End <name>;
```

Note: *Only the Dynamic region is mandatory (the Initial and Terminal regions and the Step and Communication sub-regions of the Dynamic region are optional).*

The structure of a Submodel is identical to the model except that there is no Terminal region.

The purpose of each of the regions is described in the following sections.

5.2.1 Model Statement

This statement declares the name of the model and may include an optional argument list.

Example:

```
My_model(Real:out1, out2 := Real:in1, In2);
```

The "`:=`" symbol separates the output arguments from the input arguments (output arguments appearing first). The input arguments (if present) are values that are passed to the model from the experiment once only *before* the model is executed; the output arguments (if present) are values passed back to the experiment at the *end,* when the model terminates.

5.2.2 Initial Region

This is where any calculations and assignments are carried out before a simulation run takes place. In particular, it is where state variables are normally initialised.

Example:

```
Par1 := Par2 + Par3;
x  := 0.0;
x' := 1.0;
```

Here *Par1* may be a parameter whose value depends on *Par2* and *Par3*; *x* and *x'* are state variables.

Note: *An alternative way of initializing state variables (or any variables) is in their declaration.*

5.2.3 Dynamic Region

The dynamic region is where the differential and algebraic equations that describe the dynamics of the system go. The main difference between the dynamic region of the model and other regions is that the dynamic region code is *declarative* whereas in other regions the code is *imperative*. Statements in the dynamic region describe dynamic relationships between model variables and are deemed to be executed concurrently or in parallel. Consequently the order in which such statements are presented in the dynamic region is immaterial – statements can be grouped logically, in the way which best describes the system being simulated. Of course, during execution, the dynamic region statements have to be executed in a particular order as the solution is advanced step by step. This is taken care of by the ESL compiler which automatically sorts the statements into an executable order. Because of the nature of the dynamic region, certain rules apply, for example, a model variable may be assigned a value at *one point* only – otherwise you would be trying to assign multiple values to the variable simultaneously. Similarly, all state variables must be correctly initialised. (The ESL compiler ensures that these rules, and others, are obeyed).

Note: *In some rare cases you may want to ensure that the dynamic region statements are executed in precisely the order in which you have presented them, e.g. for reasons of numerical accuracy. In such cases, the automatic sorting function can be overruled by the inclusion of a NOSORT statement in the code following the model statement.*

Examples of dynamic region statements:

```
X'' := -k*x' - x + 1;

Deriv := x1' + x2';

y := INTEG( 0.0, Eps*x + 3.2 );

z := TRANSFER( K(s+1)/(s**2 + 2*s +1) )*w;
```

The first statement is a natural way of writing a differential equation. Here it is a second order equation, but it could be first order or higher order – the limit is that the total length of the variable name plus primes (') must not exceed 28 characters (so you could define a 27th order differential equation in x – if you really wanted!) The second statement is just an algebraic assignment. The third statement is an integral equation using the library submodel INTEG. The fourth statement specifies a transfer function (see the on-line help for details of this).

5.2.4 Step Region

The step region code is executed at the end of every integration step. Typically Plot or Prepare statements would be placed here in order to maximize the output and produce smooth graphs. The integration step-size is determined by the reserved variables CINT and NSTEP. CINT specifies the communication interval (see next section); NSTEP specifies the *minimum* number of integration step to be taken in each communication interval. The *maximum* integration step-size is therefore given by CINT/NSTEP.

Note: *If you are using a variable-step integration algorithm such as RK5, the actual step-length will be determined by the algorithm to satisfy the error criteria. However, the step-size will not exceed CINT/NSTEP. For fixed-step integration algorithms such as RK2 and RK4, the step-size will normally be CINT/NSTEP. The exception to this is when the integration has to negotiate discontinuities (see Chapter 7).*

5.2.5 Communication Region

The communication region code is executed at regularly spaced communication intervals, as specified by the reserved variable CINT. This is a good place for numeric or tabulated output, as produced by the Tabulate statement.

5.2.6 Terminal Region

The terminal region contains code that is executed when the simulation run terminates, i.e. when *T >= Tfin* or some other terminate condition. It is intended for any calculations that have to be carried out at the end of a run. The Terminal region is only allowed in a model.

5.2.7 Simulation Parameters

The simulation parameters, which control a simulation run, are defined in a special *Reserved* package which is always visible in models, submodels and the experiment. If you need access to any simulation parameters a procedure, simply include a *Use Reserved* statement. The simulation parameters, with their default values are:

Tstart	(0.0)	- initial value of T at start of run
Tfin	(10.0)	- final value of T at end-of-run
Cint	(1.0)	- communication interval
Diserr	(0.0001)	- discontinuity detection error tolerance
Interr	(0.001)	- integration error tolerance
Algo	(1 or RK5)	- integration algorithm
Nstep	(1)	- number of integration steps in CINT

Algo can be specified by assigning one of the following numeric constants:

RK5	(1)	- fifth-order variable-step integration
RK4	(2)	- fourth-order Runge-Kutta integration
RK2	(3)	- second-order Runge-kutta integration
STIFF2	(4)	- second-order stiff integration
GEAR1	(5)	- Gear's variable-step stiff integration
GEAR2	(6)	- Gear's method with diagonal Jacobean
ADAMS	(7)	- Adams predictor-corrector integration
RK1	(8)	- Euler first order integration
LIN1	(21)	- Newton-Raphson Linearization routine
LIN2	(22)	- Simplex Linearization routine.

The last two constants *LIN1* and *LIN2* are used with the steady-state function *Trim*. There are in addition one or two special reserved parameters, described in the on-line help, providing information about the state of a run.

5.3 Program Example

The following example includes all of the program modules introduced above. It is essentially the same example that was used to illustrate graphical model construction. The code is explained in the following notes.

```
01 Study
02   Include "Integ";
03   Package SystemParameters;
04 -- Parameters of system
05     Real:A, B;
06   End SystemParameters;

07   Procedure ErrorSquared(Real:ActualValue,DemandValue)Return Real;
08 -- Function procedure - calculates square of error
09     Real: Value;
10     Value := (ActualValue - DemandValue)**2;
11    Return Value;
12   End ErrorSquared;

13   Submodel PIController(Real: y := Real: Gain, Ti, x);
14 -- Proportional plus integral controller submodel
15       Real: Intx;
```

```
16     Initial
17       Intx := 0.0;
18     Dynamic
19       Intx' := x;
20       y := Gain*(Intx/Ti + x);
21   End PIController;

22   Submodel System(Real: output := Real: Input);
23 -- Second order system submodel
24     Use SystemParameters;
25     Dynamic
26       output := Transfer(A/(s**2 + B*s + A))*Input;
27   End System;

28   Model ControlSystem(Real: Cost := Real: Gain, Ti);
29 -- Top-level model
30     Real:Demand, Response, Error, ActuationSignal, FeedbackSignal;
31     Initial
32       Demand := 1.0;
33     Dynamic
34       Error := Demand - FeedbackSignal;
35       ActuationSignal := PIController(Gain, Ti, Error);
36       Response := System(ActuationSignal);
37       FeedbackSignal := Transfer(1/(0.1*s + 1))*Response;
38       Cost := Integ(0.0, ErrorSquared(Response, Demand));
39       Step
40         Plot "Control System", t, Demand, [Response], 0,Tfin,0,2;
41      Prepare " ",t,Demand,Response,ActuationSignal,FeedbackSignal;
42   End ControlSystem;

43 -- Experiment
44   Use SystemParameters;
45   Real: Gain, Ti, Cost;
46 -- Set system parameters
47   A := 100.0;
48   B := 10.0;
49 -- Set simulation parameters
50   Tfin := 5.0;
51   Cint := 0.5;
52   Nstep := 5;
53 -- Call model from loop
54   Loop
55     Read Gain, Ti;
56     Terminate Gain = 0.0;
57     ControlSystem(Cost := Gain, Ti);
58     Print "Cost = ", Cost;
59   End_Loop;
60   Clear_Screen;
61 End_Study
```

- line 1 Study statement – start of ESL program
- line 2 include the ESL library submodel Integ
- lines 3-6 defines a package defining system parameters
- lines 7-12 defines a function procedure which returns a Real value
- lines 13-21 defines a submodel for a PI controller
- line 19 - example of a differential equation
- lines 22-27 defines a submodel for the system
- line 24 Use statement giving access to the system parameters

- line 26 Transfer statement describes the system transfer function
- lines 28-42 defines the model
- lines 35, 36 and 38 submodel calls
- line 40 Plot statement to generate a run-time plot
- line 41 Prepare statement to save data for post-run plotting
- lines 44-60 defines the experiment
- line 61 End_study statement – end of ESL program

The procedure ErrorSquared simply calculates the square of the difference between its two arguments and returns the value. This is an example of a function style procedure. The function appears in the expression in line 38.

Submodel PIController implements a simple proportional plus integral controller. Note the state variable intx defined by the differential equation in the Dynamic region (line 19) is initialised in the Initial region (line 17). All state variables must be properly initialised.

Submodel System models the system being controlled. In this case the dynamics of the system are specified as a transfer function in an ESL Transfer statement (line 26). The state variables implied by the transfer function are automatically initialised to zero. (See on-line help for how to initialise transfer function variables to non-zero values).

The Model ControlSystem is the high-level program module, which defines the interconnections between the submodels. Line 38 is a call to the standard library submodel Integ (specified by the include statement - line2), used to calculate the cost function. The step region includes statements to plot on-line and save data for post-run plotting. The significance of these statements being in the Step sub-region is that they are executed at every integration step. If they had appeared in the Communication sub-region, output would be generated at regular time intervals as defined by the reserved variable Cint.

The Experiment (which comprises all statements following the program module definitions) includes some local declarations (lines 44 and 45); statements to set the system parameters A and B (lines 47 and 48) and statements to set the simulation parameters Tfin, Cint and Nstep (lines 50 to 52). Tfin is the final time at which the simulation run will terminate. Time will run from Tstart (default value 0.0) to Tfin. Cint specifies that the Model and Submodel Communication sub-regions are executed at regular time intervals of 0.5 s and Nstep specifies that there will be a minimum of 5 integration steps in each communication interval. (There may be more steps if an adaptive integration algorithm is used where the step length may be reduced to satisfy the error criteria, or discontinuities occur). The main part of the experiment is a loop in which values are read for the variables Gain and Ti (the controller parameters) and the model is invoked. When the program experiment is run, the user is prompted to enter values for Gain & Ti from the console window or from a special input window (depending on how the model is invoked – from the command line or via the ESL ISE). Note that the Terminate statement stops the loop if a Gain of zero is entered. The Clear_screen statement closes the run-time plot.

5.3.1 Running the Program

There are three ways to run an ESL program: from a command prompt; from the ESL-SEC program or from ISE. First of all, type in or copy the code for the example program into a text file named *example.esl* (if you copy the code you need to delete the line numbers).

5.3.1.1 Running from a command prompt

Open a Command Prompt in the directory where you have saved *example.esl*.

The simplest way to run the program, using the interpreter option, is to type the command:

```
esl example
```

This will invoke the ESL compiler and, if there are no compilation errors, it will then invoke the ESL interpreter. You should get a response similar to that below:

```
c:\Temp\ESL examples>esl example
```

```
c:\Temp\ESL examples>esl example
**** E S L  Compiler v8.2.4.xx
**** Copyright (C) ISIM International Simulation Limited 1992-
20xx.
< INTEG          0  WARNINGS      0  ERRORS >
< SYSTEMPARAMETERS    0  WARNINGS      0  ERRORS >
< ERRORSQUARED    0  WARNINGS      0  ERRORS >
< PICONTROLLER    0  WARNINGS      0  ERRORS >
< SYSTEM         0  WARNINGS      0  ERRORS >
< CONTROLSYSTEM    0  WARNINGS      0  ERRORS >
< EXP$MN         0  WARNINGS      0  ERRORS >
**** E S L  Interpreter Run-time v8.2.4.xx
**** Copyright (C) ISIM International Simulation Limited 1992-
20xx.
```

```
Gain, Ti:
```

Enter values for the gain (Gain) and the integral control parameter (Ti), say 1.0 and 1.0. A run of the model will take place, a value should be printed for the cost function and an ESL plot generated i.e.

```
Gain, Ti: 1.0 1.0
Cost =   0.26006
Gain, Ti:
```

The ESL plot you should see is shown in Figure 34:

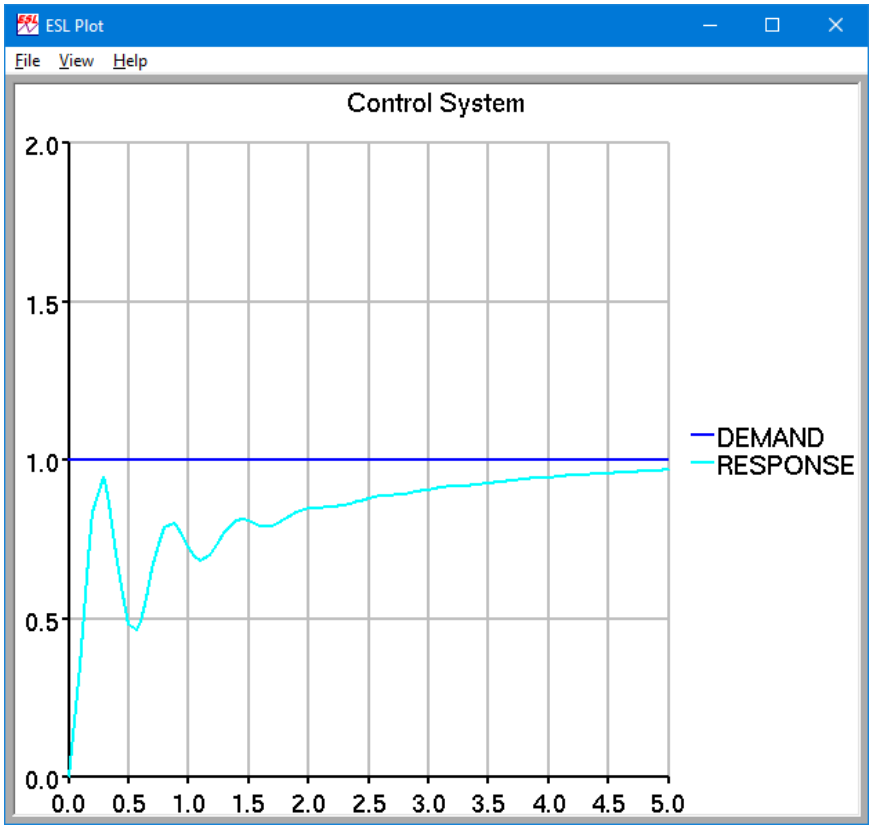

Figure 34 - ESL Plot from example.esl

Further values may now be entered for Gain and Ti giving corresponding cost function values and additional graphs on the same ESL Plot. Entering a value of zero for Gain (and any value for Ti) will terminate the experiment loop and the program.

You will find the full range of command line options in the on-line ESL help.

5.3.1.2 Running from ESL-SEC or from ISE

Chapters 2 and 3 showed you how to run a *graphically* defined simulation from the ESL-SEC Simulation Execution Control panel (this was the window that opened when you issued a Run command from ISE). From ESL-SEC you not only had control over the running of your simulation, you also had access to *Simulation Parameters* and all *Variables* and *Parameters* defined in your model. You were also able to specify *Runtime Displays* and were able to access other *Advanced Simulation Options*. ESL-SEC can be invoked from ISE or started from a Command Prompt to interactively run any *textually* defined ESL program, giving you all the interactive features available for graphical models. A detailed description of ESL-SEC will be found in the ESL-SEC User Guide in the ESL installation's 'doc' directory.

In the first instance we will simply run the ESL program *example* as it stands.

Either start ESL-SEC from a Command Prompt:

```
c:\Temp\ESL examples>esl_sec
```

or from the ESL ISE Window>Simulation Execution menu selection. Click **Setup** and navigate to example.esl in the *Simulation* text box (the function of the *Specification file* text box is explained later). The dialog should appear as in Figure 35. Note that *Execution Command* allows you to select from several command line options. If you select *Compile, Translate, Link and Execute*, you can select C++ or FORTRAN language options and specify additional link objects (external libraries etc). Advanced users can also select *Custom Run Command* and set their own Run command. In our case, leave the default Execution Command (Compile and Interpret (.esl)) and simply click the **Load** button. This will return to the Control Panel with the simulation ready to start. Clicking the **Start** button will open a User Input dialog corresponding to the Read statement in line 55 of the program (Figure 36). Enter values for Gain and Ti as before and click OK. This will produce the same ESL Plot you obtained when running the program from a command line. The value of *Cost* (Print statement in line 58) will appear in the message panel of the Control Panel. If you then click the Continue button on the Control Panel and OK the warning that “Continue may end the simulation”, you will get back to the User Input dialog and be able to enter further values for Gain and Ti (as before, entering a Gain of zero will terminate the program).

Note: *The option to translate and run an ESL program in FORTRAN or C++ requires the appropriate compiler to be installed on your computer. See the Development Guide for details.*

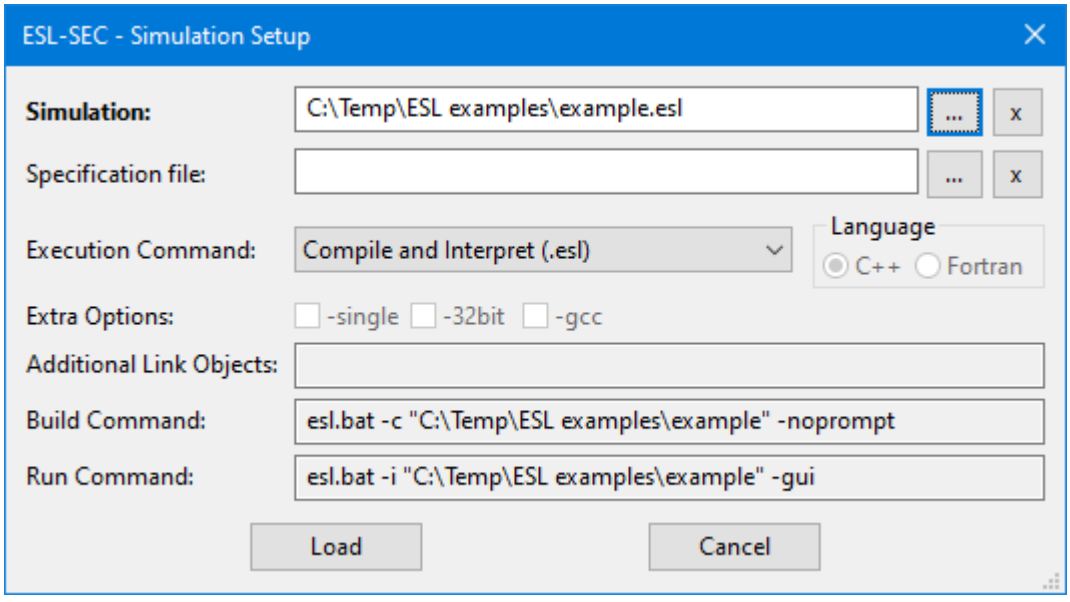

Figure 35 - Simulation Setup dialog

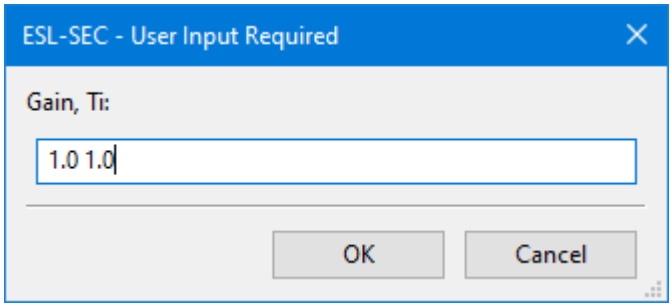

Figure 36 - User Input dialog for example.esl

Although the exercise described above demonstrated that any ESL program that can be run from a Command Prompt can also be run from ESL-SEC (from a Command Prompt or from ESL ISE), it does not make full use of the interaction offered by ESL-SEC. If you were intending to run your ESL program under ESL-SEC, you would not normally hard-code user input, output and plotting requirements in the program – these can all be specified interactively when running the program. This gives greater flexibility; for example you can easily change the graph plotting specification between runs, and change the values of *any* parameters from the Control Panel.

To illustrate this, edit the model and experiment of your program example.esl, as shown below, and save as example1.esl. Note that the model argument list has been removed; Gain, Ti and Cost have been re-declared as local parameters and a variable; the Plot and Prepare statements have been commented out; the Gain, Ti and Cost declaration in the experiment has been commented out; the loop has been replaced with a simple model call; and the Clear_Screen statement has been commented out.

```
   ..................
   ..................
  Model ControlSystem;
 -- Top-level model
    Real:Demand, Response, Error, ActuationSignal, FeedbackSignal;
    Parameter Real: Gain/1.0/, Ti/1.0/;
    Real: Cost;
    Initial
      Demand := 1.0;
    Dynamic
      Error := Demand - FeedbackSignal;
      ActuationSignal := PIController(Gain, Ti, Error);
      Response := System(ActuationSignal);
      FeedbackSignal := Transfer(1/(0.1*s + 1))*Response;
      Cost := Integ(0.0, ErrorSquared(Response, Demand));
      Step
--      Plot "Control System", t, Demand, [Response], 0,Tfin,0,2;
--      Prepare " ",t,Demand,Response,ActuationSignal,FeedbackSignal;
  End ControlSystem;

 -- Experiment
 -- Real: Gain, Ti, Cost;
  Use SystemParameters;
 -- Set system parameters
  A := 100.0;
  B := 10.0;
 -- Set simulation parameters
  Tfin := 5.0;
  Cint := 0.5;
  Nstep := 5;
  ControlSystem;
 --  Clear_Screen;
End_Study
```

Re-enter the Setup dialog and load example1.esl. Click *Runtime Displays* (Figure 37) to specify Plots, Tables and Prepare output.

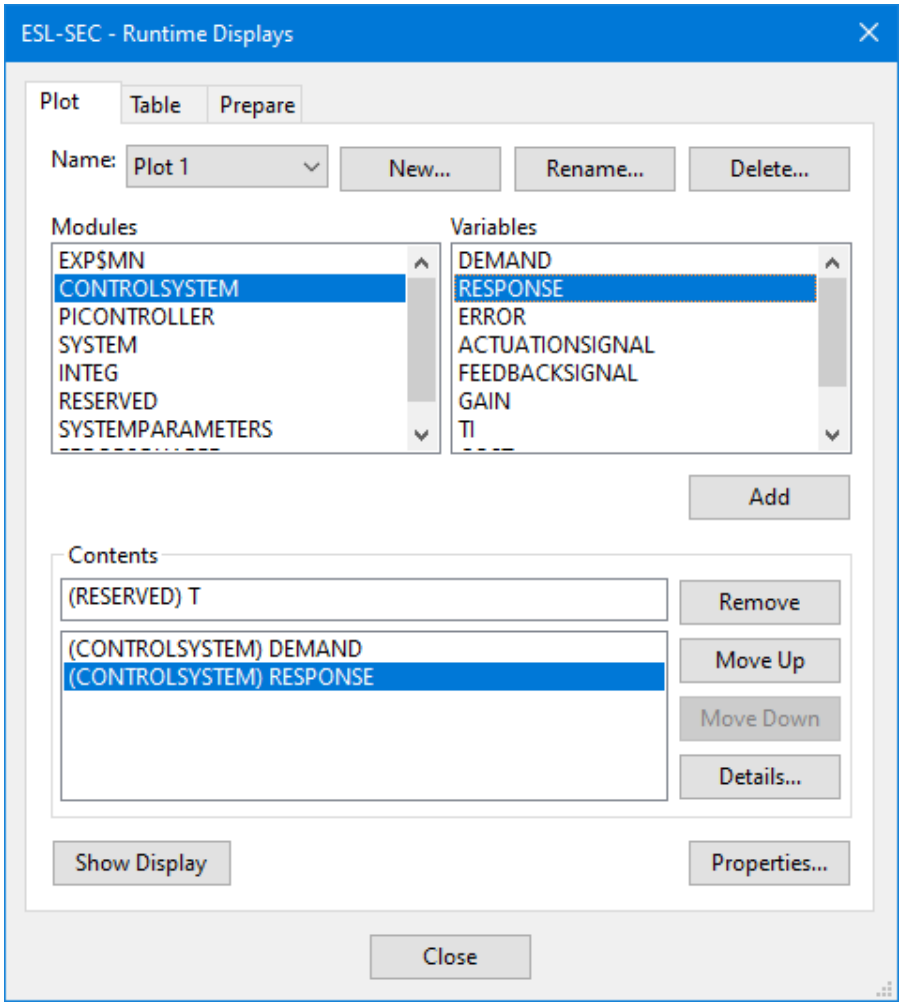

Figure 37 – Runtime Displays dialog

For example, to define a new Runtime Plot, select the *Plot tab*. The default name for the first plot is *Plot 1*, which you can change if preferred from the *Rename* button. Select the model (CONTROLSYSTEM) from the Modules panel and select the variables to be plotted from the Variables panel (either double click a variable or click the variable followed by the Add button) – *DEMAND and RESPONSE* in this case. The Remove, Move Up and Move Down buttons can be used to rearrange the list of plot variables. The Properties button allows you to refine the appearance of the plot, for example – specify a title and display grid. Finally click *Show Display* to open a plot window. Prepares can be specified in a similar manner from the Prepare tab.

You can set up a table, (from the Table tab) to show the value of the cost function, Cost. Under the table Properties, select Monitor from the Style options. This will display just the current value of T and Cost (rather than a full tabulated list) and will therefore show the final value at the end of each run (as was the case when running from a command line). Do not forget to click *Show Display* to open the table.

If you now click Start on the Control Panel, you should get one run of the model using the default values of Gain and Ti of 1.0 and 1.0 (specified in their declaration statement). The values of Gain and Ti can now be changed from the Variables dialog (click Variable on the Control Panel) and further runs made as described under Varying Parameter Values in Chapter 2.8. Don't forget to click Rerun and Continue to obtain each new run. Figure 38 shows a typical appearance after three runs of the program initiated from Simulation Execution in ESL ISE.

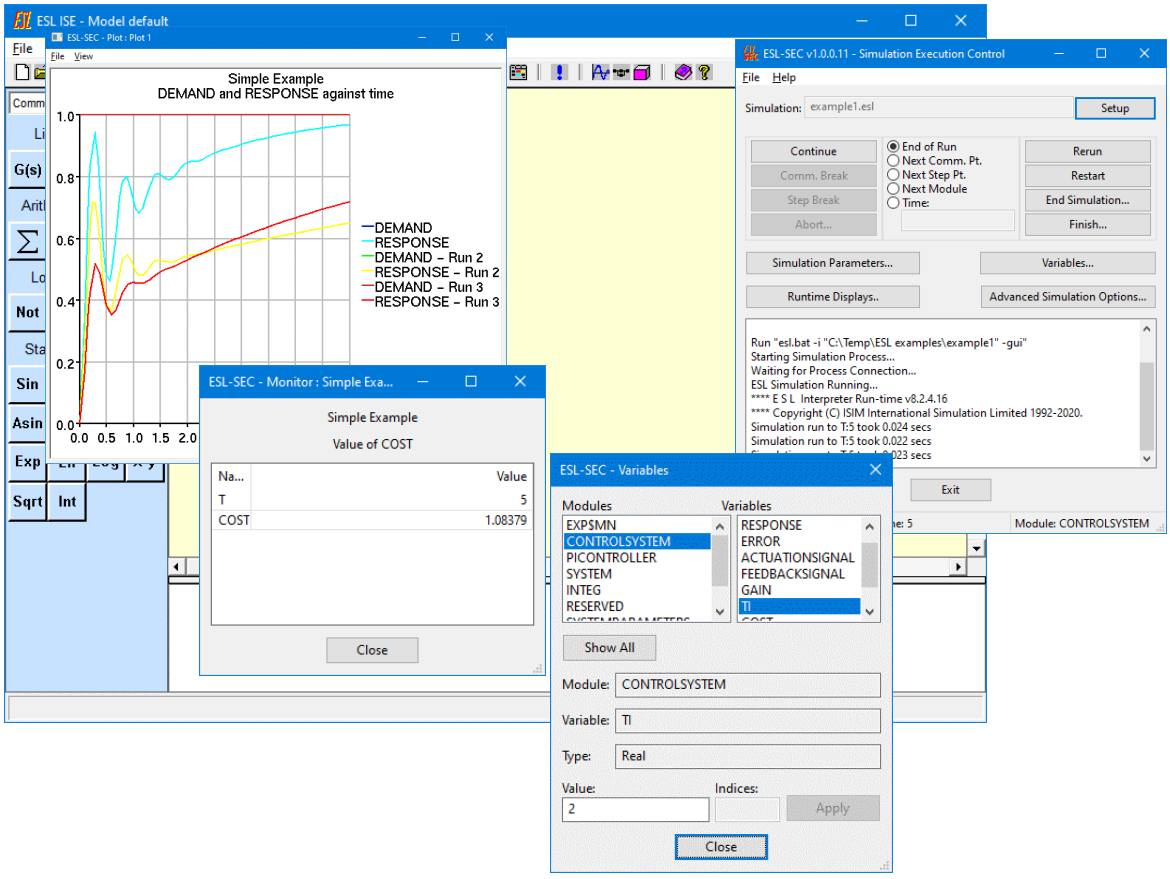

Figure 38 - Running example1.esl from ISE

Finally click *Exit* on the Control Panel to terminate the simulation. You will get a warning – "This will terminate the simulation. Do you wish to continue?" – click *Yes*. This will open a final window - *Specification Changed* (Figure 39). This gives you the option to save a specification file (.sec) which records all the interactive changes made to the program in the current session (Simulation Parameters, Runtime displays etc). You can view the changes and decide which, if any, to save. In this illustration the file name entered is the same as the original ESL program (example1.esl) and will create a file example1.sec.

For subsequent runs of the simulation, the specification file may be entered in the Simulation Execution Control setup dialog (Figure 40). This will cause the ESL simulation and all parameter and display specifications that were saved to be loaded.

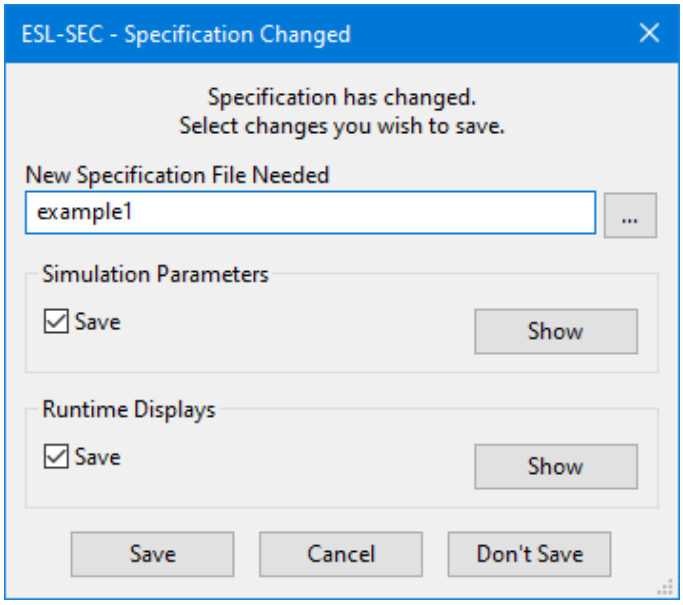

Figure 39 - Specification Changed dialog

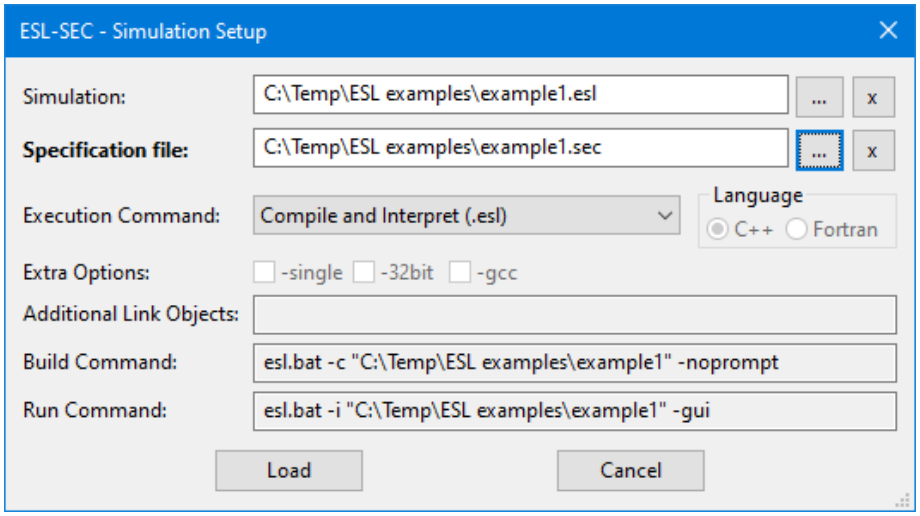

Figure 40 - Specification file in Setup

CHAPTER 6

External Submodels and Toolboxes

Once you can write submodels directly in the ESL language, you can incorporate these into ISE diagrams. There are two ways of doing this:

1. through the use of the Submodel Manager
2. by adding new icons onto the palette

6.1 The Submodel Manager

The Submodel Manager allows the management of all submodel definitions set in an application. It is invoked from the Window menu and its appearance is shown in Figure 41.

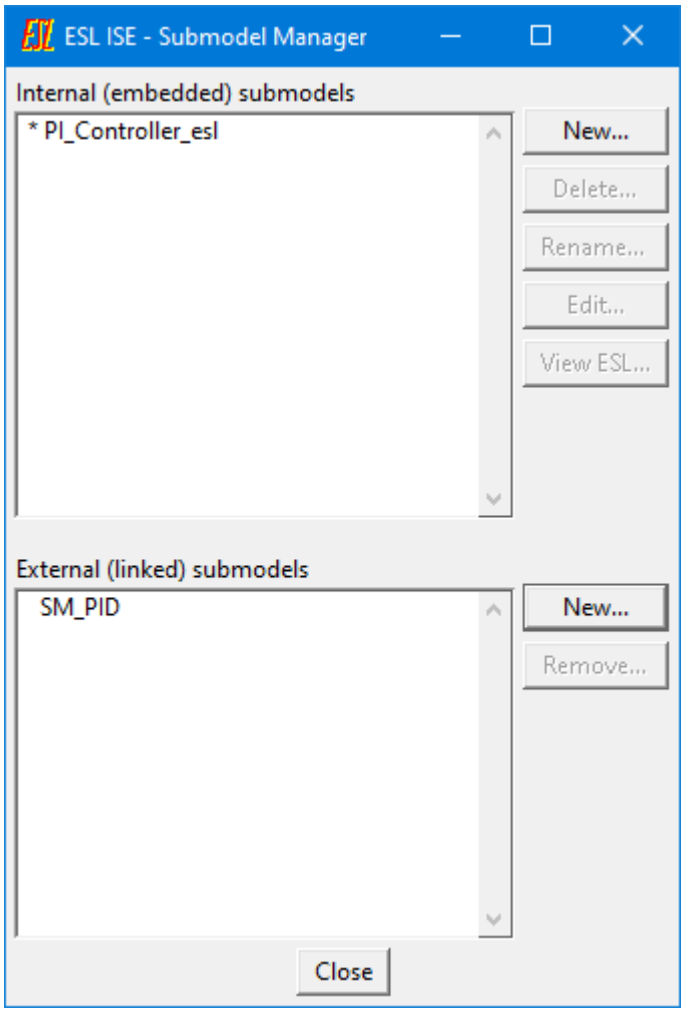

Figure 41 – Submodel Manager

There are two scrolling windows which list the Internal (embedded) and External (linked) submodel definitions, together with an indication of whether or not they are in use (an asterisk in the left column indicates current use). In the example above the embedded submodel Pi_Controller_esl is in use but the linked submodel SM_PID is not. The difference between embedded and linked submodels is: embedded submodels are those created, either

graphically or textually, *within* the application and saved with the application whereas linked submodels are defined in *external* ESL files.

You can rename internal definitions but only delete them if they are not in current use.

New internal definitions may be set by clicking the New button, to invoke the New Submodel Definition dialog.

External submodels, which are not in current use, may be removed. New ones may be added by clicking the New button and entering or browsing to an ESL file.

If an application contains submodel elements, the Submodel Manager contains a list of Internal (embedded) and External (linked) submodels. Embedded submodels may be viewed and edited by selecting the appropriate entry and clicking the Edit button. External submodels may be viewed once they have been included in a diagram by double clicking the element. Alternatively, they may be renamed or deleted by clicking the Rename or Delete button. If an application contains no submodel elements, or if there is no application, the dialog box is empty and a new submodel element may be defined. To do this: click New, assign the new submodel a name or accept the default, select the new entry and click Edit. (The option of defining a new submodel is, of course, also available when existing submodels exist.)

The View ESL button next to the Internal (embedded) submodels panel allows the ESL code that is generated from a graphical internal submodel to be viewed in the text editor. The code may be saved as an ESL file (extension esl) for reuse as an external submodel in other ISE applications. (Note the "<<< Viewing ESL - edits will be discarded. >>>" header must be deleted before saving.)

6.2 Toolboxes

When you open the ISE program, the palette is populated with a selection of commonly used simulation elements. The palette can be customised for specific application fields through the use of toolboxes.

You can create “Toolboxes” in ISE, to customise the palette. You can specify:

- The titles of the panels in the palette, and of the areas in each panel.
- The elements to be included in each area.
- Alternate icons for the standard elements.
- Different icons for the elements when they appear on the canvas.
- Properties (shape, layout of terminations, colour and size) of individual elements or elements within a panel or area.
- Submodels from ESL files, with their palette and canvas icons.

The idea is that you can customise the palette for your own application areas by modifying the standard palette and adding panels and areas to provide access to specialised ESL submodel libraries. Toolbox specifications can be saved to file allowing several different personalised palette configurations.

You create a toolbox by means of a wizard.

6.2.1 Creating a Toolbox

You access the Toolbox Wizard from the Options dialog, which may be opened from the View menu. In the General panel of the Options dialog (Figure 42), click the Edit button and enter a name for the toolbox in the File dialog. The Toolbox dialog opens displaying the Panels list (Figure 43).

You can populate the toolbox with the standard configuration of panels, areas and simulation elements by clicking the Standard button (as in Figure 43). Select a panel and click Edit (or double-click the panel) to see the areas assigned to it. Similarly, select an area and click Edit

to show the elements. Click Area on the Elements list to return to the areas and Panel on the Areas list to return to the panels.

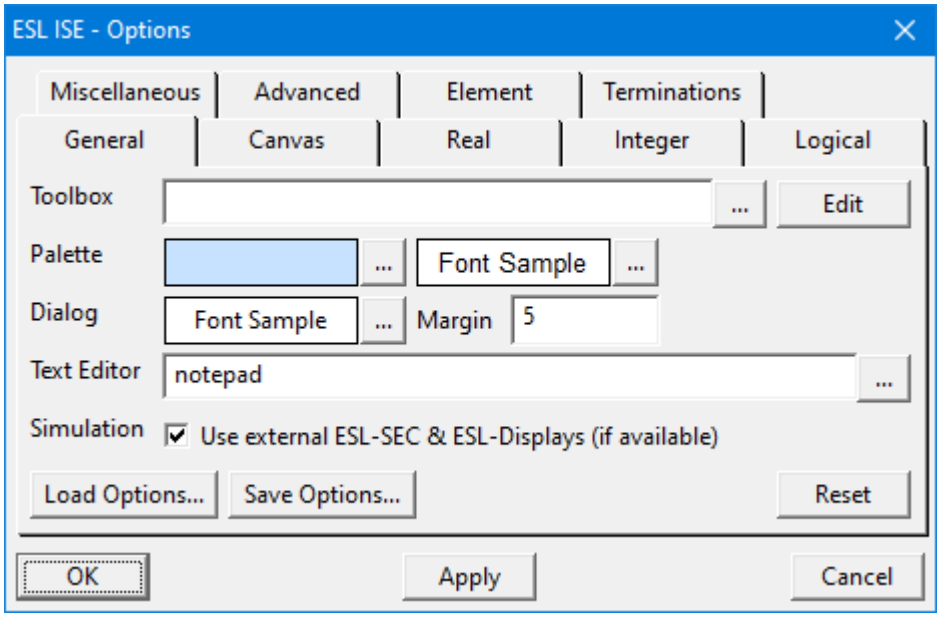

Figure 42 - ISE Options

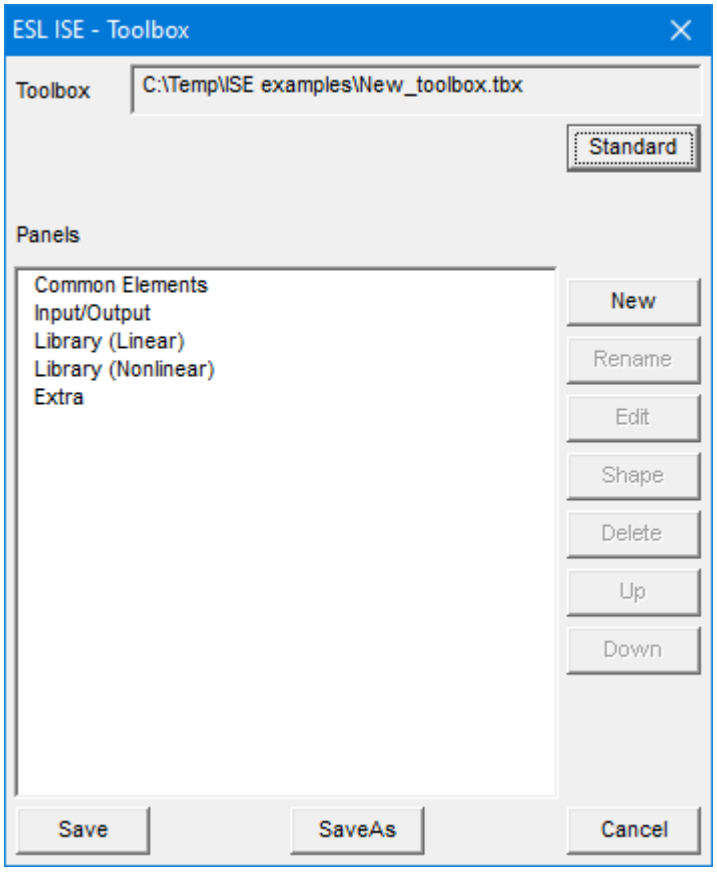

Figure 43 - Toolbox dialog

Use the Rename, Delete, Up and Down buttons to modify the content and layout of the lists of panels or areas and the New button to create a new panel or area. The Shape button allows you to globally define the following properties of elements within a panel or area. On an Elements list, the Shape button allows the properties of a single element to be defined.

- Shape of the elements – choose from a list of standard shapes.

- Layout of terminations – choose annotated or non-annotated. Circular arranges up to four non-annotated terminations around the element, for example, three input summer.
- Fill colour.
- Border colour.
- Size of element – width, height and corner radius.
- Circle diameter – if circle shape chosen.
- Orient – initial orientation of the element

A blank entry for Shape and Layout indicates that the defaults are selected. Clicking the Reset button for a panel selects the defaults. Clicking the Reset button for an area or an individual element selects the shape properties defined at the next higher level.

New elements or submodels can be added to an Elements list by clicking the Element or Submodel buttons. Element allows you to select from the complete set of simulation elements. Submodel allows you to enter or browse to an ESL file and select a submodel. On this list, the Edit button allows you to specify an alternate name for an element or submodel and choose a palette icon. You can also choose a different icon (or no icon) to be displayed when the element or submodel is placed on the canvas.

When you have finished defining a toolbox, click the Save or SaveAs buttons on the Panels list to save the definition to a toolbox file.

An example of a toolbox file is provided in the ise\examples directory.

6.2.2 Loading a Toolbox

In the General panel of the Options dialog, browse to (or enter) the toolbox file name and click the OK button.

6.2.3 Editing a Toolbox

In the General panel of the Options dialog, browse to (or enter) the toolbox file name and click the Edit button. Then proceed as for Creating a Toolbox.

6.2.4 Portability

When you use the Toolbox Wizard to browse and set file paths, the toolbox will normally be set with the absolute file path. This applies to not only to submodel (.esl) files, but also toolbox icon & canvas icons, and applies also if local help files are to be used.

If you create a set of submodels and associated files (e.g. a submodel library), and later wish to relocate them or allow someone to transfer them on another computer, the toolbox absolute file paths may no longer work.

ISE Toolboxes provide two features to support portability: using environment variables, and by installing under the ESL installation path.

If, in the Toolbox Wizard you enter (or edit) a file path to begin with an environment variable, then ISE will use that. For example, you may put a set of files for your submodel library under any directory and set an environment variable "MYSUBMODELLIB" to point to that directory. Then, in the Toolbox Wizard, you can use:

```
%MYSUBMODELLIB%\submodel1.esl - for a submodel file
%MYSUBMODELLIB%\submodel1.ico - for its toolbox and canvas icon
%MYSUBMODELLIB%\submodel1.pdf - for its local help file
```

If subsequently, the set of files are relocated, so long as the environment variable is updated, ISE will continue to find the files without the toolbox file needing to be changed. If the library is installed in a directory on another computer, the environment variable may be set to that directory, and the toolbox file will not need to be changed.

Note: *The environment variables should be system environment variables and should be set up prior to starting ISE. You may need administrator privileges to set up the environment variables.*

If, in the Toolbox Wizard you enter (or edit or browse to) a file path under the ESL top-level installation directory, ISE will show this as relative to a pseudo environment variable "ESLISEHOME". If a submodel library is installed in the corresponding directory (under its ESL installation) on another computer, the toolbox file will not need to be changed to reference the submodel library.

For example, you may place your submodel files on a directory

"C:\Program Files\ESL\esl-pro\mysubmodellib",

which would show, for example, submodel1 as

"%ESLISEHOME%\mysubmodellib\submodel1.esl",

for ESL installed on "C:\Program Files\ESL\esl-pro". If the library was transferred to a computer with ESL installed on (say) "D:\Program Files\ESL\esl-lite" to a subdirectory

"D:\Program Files\ESL\esl-lite\mysubmodellib",

then (again) the toolbox file would not need to be changed.

Note: *You may need administrator privileges to create directories and put files under the ESL installation directory.*

Note: *You should not set the "ESLISEHOME" environment variable explicitly, as it is set up internally by ISE when starting up.*

Finally, it is worth noting that the toolbox file (extension .tbx) is a normal text file which may edited in (say) Notepad, to change file paths if required.

CHAPTER 7

Advanced Features

In this chapter we introduce some of the more advanced features of ESL. As with the previous material in this User Guide, the aim is to give a broad overview of the topics. A more in-depth treatment will be found in the on-line Help.

7.1 Discontinuities

7.1.1 What are Discontinuities?

A discontinuity is an event which causes the algebraic or differential equations representing the system to suffer a *jump* or *step* change in one or more modelling variables. Such events are very common in real systems, for example limits, dead-space, hysteresis etc. Integration algorithms cannot integrate satisfactorily in the presence of discontinuities. In mathematical terms the function is *piece-wise continuous* with a discontinuity representing an abrupt change in a state variable, or its first or higher derivative. A discontinuity within an integration-step invalidates the Taylor series representation of the step, and consequently any of the integration algorithms used.

Although ESL protects integration from discontinuities, it is helpful to understand the consequences of an *unprotecte*d discontinuity on the integration process:

> *Fixed-step explicit* - causes erroneous results as the method is attempting to match Taylor series which is invalidated by the discontinuity. Small steps, giving longer execution times minimises this effect.
>
> *Variable-step explicit* - the method gives inaccurate results which are reflected in the error estimate. This causes the step mechanism to reduce the step which spans the discontinuity to a very small value at which the effect of the discontinuity is minimal. The final result usually has good accuracy but at the expense of excessive computation time.
>
> *Implicit methods* – are even more sensitive to discontinuities. The result is possibly an abortion, very slow execution and/or erroneous results.

7.1.2 Handling Discontinuities in ESL

ESL incorporates an integration-discontinuity control mechanism which accurately and efficiently detects and locates discontinuities. ESL does not allow a discontinuity to occur within an integration-step. It arranges for it to occur after the end of one step and before the beginning of the next i.e., between steps. This would normally lead to a gross time error, however at the end of each step a check is made to see if a discontinuity has occurred during the step. If this was the case, the step is repeated with a shorter step-length based on an interpolation of the discontinuity function (the relational expression describing the discontinuity). The interpolation process is repeated until the end of step occurs just after the point of discontinuity, but within a specified error bound. The change to a modelling parameter may then be made between steps, before proceeding with the simulation of the new state of the system. As the control mechanism does not allow any change to take effect during an integration-step, the integration routines are protected from the effects of a discontinuity occurring in mid-step.

The method is illustrated in Figure 44. Here a discontinuity occurs when the variable *A* becomes greater than or equal to the variable *B*. A discontinuity function is defined as $\theta = A - B$.

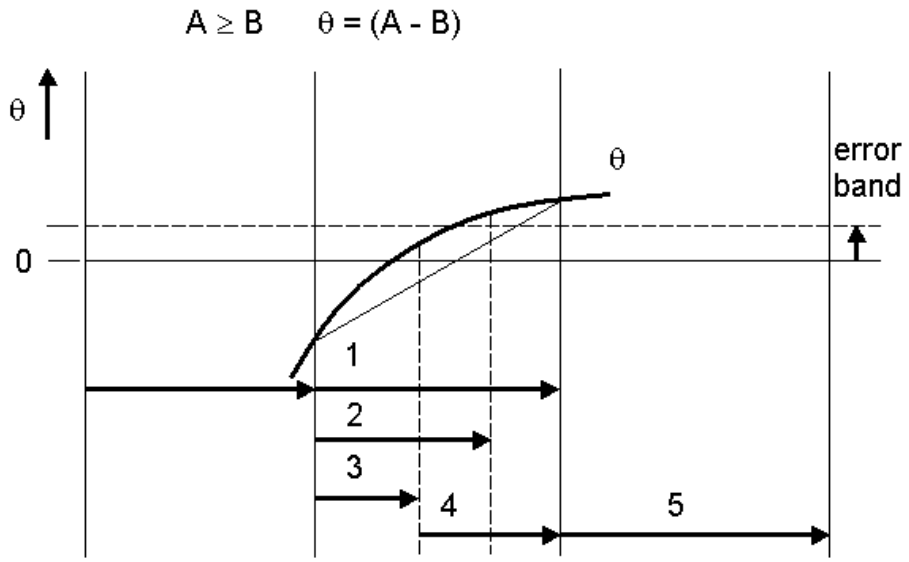

Figure 44 - Discontinuity detection

The sequence is:

- Step (1) has been computed by the integration algorithm, integration accuracy criteria have been satisfied.
- The discontinuity detection control, however, detects a discontinuity as θ has changed sign. It uses linear interpolation to suggest a step-length, step (2) that will be close to the point of discontinuity. Note that the linear interpolation *aims* for the centre of the error band.
- Step (2) is undertaken, but again it *overshoots* the discontinuity, and a further interpolation is used to refine the step-length i.e., step (3). This and any subsequent interpolation use quadratic, rather than linear, interpolation based on three values of θ which span the discontinuity.
- The result of step (3) is that θ now lies within the error-bound, and the discontinuity is regarded as being accurately detected.
- The result of the relational operation, A >= B, is now set to be true; during previous steps 1, 2 and 3, it had been maintained false.
- The recovery step, step (4), is computed using the new result of the relational operation. This step *aims* for the same point in time as the original step, step (1), in which the discontinuity was first encountered.
- Step (5) is a normal step following the discontinuity process.

7.1.3 Representation of Discontinuities in ESL

The ESL library contains submodels for dealing with commonly occurring discontinuities such as limiters, dead-space and hysteresis. However, two language constructs are available for modelling any non-standard discontinuous functions. These are the *If clause* and the *When statement*.

7.1.3.1 If clause

The If-clause is part of a modelling code assignment statement, and it may only appear in the dynamic region of a model or submodel. It acts as a two-way, or multiple-way, switch which assigns a single value to a variable, for example:

```
y:= If a > b Then x1 Else x2;
y:= If a > b Then x1 Else_If x< 0.0 Then x2 Else x3;
```

```
      y:= If a > b and c >= (2*threshold) Then x1 Else x2;
      y:= If a > b or c > b Then x1 Else x2;
```

The final *Else* is mandatory because an assignment must always be made to the variable. Additional *Else_If* clauses introduce further branches, or choices.

The value given to the variable *y* corresponds to the first logical expression which is true.

Note: *It is the logical expressions in the above examples that generate the discontinuity functions, i.e. expressions involving logical comparisons like* > < >= *etc..*

The following code shows the implementation of a limiter submodel using the If-clause:

```
SUBMODEL LIMIT(REAL:y := CONSTANT REAL:LL,UL; REAL:x);
------------------------------------------------------------------
--
-- A limiter sets lower and upper limits on the amplitude
-- of an input variable. The calling sequence is:
--
--    y:= LIMIT(LL,UL,x)
--
-- where:
--    LL is the lower limit;
--    UL is the upper limit;
--    x is the input variable.
--    y is given a value such that:
--          y = x, if LL < x < UL,
--          y = UL, if x >= UL,
--          y = LL, if x <= LL.
--
-- Note the inputs LL, UL must be UL > LL, and are assumed
-- constant throughout a run. The output is an algebraic
-- variable.
------------------------------------------------------------------
--
  REAL: range,xnorm;
INITIAL
  if LL >= UL then
    print "**** Error in LIMIT: Limits not consistent";
    STOP;
  end_if;
  range:= UL-LL;
DYNAMIC
  xnorm:= (x-LL)/range;
--
  y:= if xnorm > 1.0 then UL
      else_if xnorm < 0.0 then LL
           else x;
--
END LIMIT;
```

7.1.3.2 When statement

The *When* statement is a modelling code statement which may only appear in the dynamic region of a model or submodel. Its operation is fundamentally different from the If-clause. The If-clause is active on each execution of the dynamic region and causes an assignment to be made. The *When* body, however, is *only* executed at the instant when its logical expression become true. Consider:

```
When x >= ul Then
  Print "x >= ul has changed from FALSE to TRUE at time= ", T;
  trigger:= true;
End_When;
```

The body of the *When* statement is procedural, non-modelling code, which is only executed at the instant when the logical expression, *x* >= *ul*, changes from false to true. The *Print* statement accurately reflects the situation. Note in this example that if *trigger* is used elsewhere in the dynamic region, then it must have been initialised in the Initial region, or in its declaration. The above, however, will only set *trigger* when *x* becomes greater than or equal to *ul*, and *trigger* is never reset. The following addresses this situation:

```
When x >= ul Then
  trigger:= true;
When x < ul Then
  trigger:= false;
End_When;
```

Note: *Multiple When statements can be concatenated together with a single End_When.*

The following code shows the implementation of a sample and hold submodel using the When statement:

```
SUBMODEL SAMHLD(REAL:y := CONSTANT REAL:per; REAL:x);
------------------------------------------------------------------
--
-- Samples and holds the value of an input variable.
-- Samples are taken periodically and the output is the
-- value of the last sample taken. The calling sequence is:
--
--    y:= SAMHLD(per,x)
--
-- where:
--    per is the sampling period;
--    x is the input variable;
--    y is given a value such that:
--          y = x, initially,
--          y = x, at the last sampling period.
--
-- Note per is assumed constant throughout a simulation run.
-- The output is a memory variable.
------------------------------------------------------------------
--
  REAL: start;
INITIAL
  y:= x;
  start:= T;
DYNAMIC
  when T - start >= per then
    start:= start+per;
    y:= x;
  end_when;
--
END SAMHLD;
```

7.2 Segments

An important feature of ESL is its *segment* structures. Segments were originally included in ESL as a means of providing a parallel processing capability. The idea is that a large simulation can be broken down into self-contained segments that can be executed in parallel on different processors or networked computers. Communication takes place between segments at pre-determined communication points through a TCP IP protocol. We shall see that segments are useful even when they are not executed in a truly parallel manner and also

that segments provide the means of embedding ESL simulations in other programs. There are three types of segment in ESL:

- *Emulated segments* – these allow parallel operation to be emulated on a single computer – useful for testing parallel segments before assignment to separate processors and for implementing multi-rate simulations.
- *Remote segments* – these can be assigned to different processors for truly parallel operation.
- *Embedded segments* – used where an ESL model is to be integrated with another application.

7.2.1 Emulated Segments

A large simulation will typically include some parts which have fast dynamics (or small time constants) while other parts will have much slower dynamics (or long time constants). Consider, for example, an all-electric ship. The inverters and motor control circuitry will have very small time constants, perhaps sub-microsecond; the propulsion motors will have longer time constants, maybe of the order of milliseconds; while the dynamics of the ship itself would be characterised by time constants of seconds or larger. If the whole simulation is written as a single model-submodel structure, the integration step-length (and hence the time taken for the simulation to run) will be determined by the parts that have the fastest dynamics. Emulated segments allow different parts of the simulation to use the most appropriate step-length and integration algorithm, while still running the simulation on a single computer, and so achieving much shorter simulation times.

Emulated segments are defined within an ESL Study and called from the communication region of the model. The model may include some part of the simulation or may simply be the means of linking the individual segments. The general structure is shown below:

```
Study
  <packages, procedures and submodels>
  .........
  Segment Seg1(Real:out := Real:in);
  .........
  Initial
    CINT  := ...;
    NSTEP := ...;
    ALGO  := ...;
  .........
  Dynamic
    .........
  End seg1;
  <further segments>
  Model Mod1;
  .........
    Dynamic
    .........
      Communication
        Seg1(y := x);
        .........
  End Mod1;
  .........
  Mod1;
End_study
```

The structure of a segment is identical to that of a model. The simulation parameters to be used by the segment (CINT, NSTEP, ALGO) must be set in the segment initial region. CINT will normally be the same as that used by the model, but different values of NSTEP and ALGO may be set allowing a different integration step-length and/or integration algorithm to be used by the segment. An ESL Study may contain multiple segments – all called from the

model communication region. The segment in the example has only one input and one output – in general a segment may have multiple inputs and outputs.

An example of a program which uses an emulated segment *seg1.esl* will be found in the esl\examples directory and is described in some detail in the on-line help. It is suggested that as an introduction the ESL segments, you examine and run this example.

7.2.2 Remote Segments

Remote segments provide true parallel or distributed simulation over a network of computers using a client/server arrangement – the main simulation (model and experiment) being the client and the segments the servers.

The main difference between remote segments and emulated segments is that the remote segments must be converted into executable code (via the FORTRAN or C++ translation route) and copied to the computers on which they are to run. The general syntax for a remote segment is:

```
Remote
  <packages, procedures and submodels>
  .........
  Segment Seg1(Real:out := Real:in);
  .........
  Initial
    CINT  := ...;
    NSTEP := ...;
    ALGO  := ...;
  .........
  Dynamic
    .........
  End seg1;
```

Note that the code begins with the keyword *Remote* and contains one and only one segment plus associated packages, procedures and submodels. There is no model, experiment and no final *End_study* statement. The program structure has to be: ESL compiled; translated into FORTRAN or C++; compiled and linked to create an executable. The executable must then copied to the remote computer on which the segment is to be executed.

Note: *Different instances of the same segment may be run on different computers.*

The main simulation (the client), containing the model, must include external segment declaration statements (just the segment declaration statements from the remote structures followed by the keyword *External*), e.g.

```
Study
  <packages, procedures and submodels>
  .........
  Segment Seg1(Real:out := Real:in)External;
  .........
  <further external segment declarations>
  Model Mod1;
  .........
    Dynamic
    .........
      Communication
        Seg1(y := x);
        .........
  End Mod1;
  .........
  Mod1;
End_study
```

Before the distributed simulation can be run, a *segment location file* must be created on the local computer (where the main simulation is located). This file must have the same name as

the main simulation program but with extension ".rem" and is used to associate a segment name with a host and executable file. The segment location file has the general form:

```
Segment_Name<Spaces>Host Name<Spaces>Executable_Filename
```

For the above example, assuming there is just the one remote segment, *Seg1*, the segment location file might contain the line:

```
Seg1 PC001 remseg
```

where *Seg1* is the name of the segment; *PC001* the name of the remote computer and *remseg* the name of the remote segment executable.

A further consideration for running remote segments is that the user must be able to run a process on the remote machine using the "rsh" command. The system configuration to achieve this is beyond the scope of this document but may involve setting up a password entry and home directory on the remote machine.

The on-line help describes how the emulated segment example *seg1.esl* can be modified to run as a local model/remote segment arrangement.

7.2.3 Embedded Segments

The embedded segment provides a means of generating code that can be called from another non-ESL program – thus enabling the segment to be *embedded* in another program.

In an ESL embedded segment, all interface variables appear in ESL *Packages*. The code below is an example of an embedded segment for a simple linear model of a dc motor. Inputs to the model appear in the package Esl_inp; State outputs appear in the package Esl_state and algebraic outputs in package Esl_out. The package Esl_par contains parameters which should be accessible to the user and Esl_view contains viewables, i.e. any variables that may be plotted or are used to drive visualizations. The dynamic model itself is defined in the *Segment* structure. The choice of package name is entirely up to you, however package name beginning ESL_ are automatically exposed by the 'eslgen' command (see *options* below). You could declare all the interface variables in a single package – multiple packages have been used here to help distinguish the use of the variables.

```
EMBEDDED
Package Esl_inp;
  Real: va, tl;
End Esl_inp;
--
Package Esl_state;
  Real: ia, Wa;
End Esl_state;
--
Package Esl_out;
  Real: v_error;
End Esl_out;
--
Package Esl_par;
  Parameter Real:Kt/0.0275/, Kb/0.04/, Ra/9.0/,
    La/4.065e-03/, Ja/1.71e-06/, Ba/1.5e-04/;
End Esl_par;
--
Package Esl_view;
  Real: v_back, t_motor, t_avail;
End Esl_view;
--
```

```
Segment dc_motor;
  Use Esl_inp; Use Esl_state;
  Use Esl_out; Use Esl_par;
  Use Esl_view;
  Real: i, w, ve, vb, tm, ta;
  Dynamic
    ve:= va-vb;
    i:=Transfer(1/(La*s + Ra))*ve;
    tm:= Kt*i; ta:= tm-tl;
    w:=  Transfer(1/(Ja*s+Ba))*ta;
    vb:= Kb*w;
  Communication
    ia := i; wa := w;
    v_back  := vb; v_error := ve;
    t_motor := tm; t_avail := ta;
End dc_motor;
```

Using an ESL utility, *eslgen*, an embedded segment may be compiled into:

- a dynamic link library (DLL) providing a function interface which can be used in Microsoft Visual Basic or Visual C++ projects;
- a COM object, which can be used in Visual C++ projects (in an object oriented manner) and also other control/ActiveX hosts (such as Web Browsers);
- or a .NET assembly, which can be used in any .NET project such as C#.

The *eslgen* command has the following form:

```
eslgen -dll|-com|-comnr|-clr filename {io_packages}
```

The options are:

```
-dll   - create a DLL from an ESL embedded segment
       eslgen -dll file_no_ext {io_packages}
-com   - create a COM object from an ESL embedded segment and
register it
       eslgen -com file_no_ext {io_packages}
-comnr - create a COM object from an ESL embedded segment (but
do not register it)
       eslgen -comnr file_no_ext {io_packages}
-clr   - create a .NET (2+) assembly from an ESL embedded
segment
       eslgen -clr file_no_ext {io_packages}
The {io_packages} are the names of ESL packages that are to be
exposed. If none are specified, any beginning "Esl_" will be
exposed.
```

The generated embedded segment code (whether it be DLL, COM or .NET) provides a set of functions or methods for running the code. These are listed in Table 1, below. In addition to these functions, mechanisms are provided for accessing the interface variables (as declared in ESL packages). The detail of how to call the functions and access the variables depends on which type of code has been generated (DLL, COM or .NET) and is described in detail in the on-line help.

The idea is that, after calling ExStrt to initialise the code, any parameters (including simulation parameters) may be set or changed. ExInit is then called to initialise the segment (the Initial region is executed). ExSim is then called repeatedly in a loop to keep advancing the segment by the communication interval, CINT, on each call. Inputs are passed to the segment before each call of ExSim, and outputs retrieved after each call. For CLR operation, a special function, ExPrestep is provided to update segment outputs that depend directly on the inputs without advancing time. At any time the segment can be re-initialised by calling ExInit. When the simulation is complete the function ExFin is called to properly terminate the code.

Table 1 - Embedded segment functions

Name	***Meaning***
ExStrt	Prepare embedded code for use - must only be used once at program start.
ExInit	Initialise embedded segment for a single simulation run.
ExSim	Advance Simulation by one time-frame (specified by the simulation parameter CINT).
ExPrestep	Evaluate algebraic outputs without advancing the simulation (CLR only).
ExFin	Close down simulation - must only be used once at program termination.

Note: *Please refer to the on-line help or contact ISIM for further details on the use of embedded segments including directly producing FORTRAN or C++ code that may be used to invoke the simulation in an application.*

Embedded segments are a powerful feature of ESL allowing simulations to be easily incorporated various applications. Examples of the use of embedded segments include training simulators where, the graphical user interface has been provided by other software, a C++ program say, which calls upon an embedded ESL program to provide the underlying dynamic simulation.