Proceedings
of the 3rd International Modelica Conference,
Linköping, November 3-4, 2003,
Peter Fritzson (editor)

Mike Dempsey
Claytex Services Limited:

Automatic translation of Simulink models into Modelica using Simelica and the AdvancedBlocks library

pp. 115-124

Automatic translation of Simulink models into Modelica using Simelica and the AdvancedBlocks library

Mike Dempsey
Claytex Services Limited
5 Marston Close, Leamington Spa, UK
mike.dempsey@claytex.com
http://www.claytex.com/

Abstract

A new tool, Simelica™, is presented for converting Simulink® models into equivalent Modelica® models. The conversion is achieved while retaining the original structure of the Simulink model. The equivalent Modelica models are built from a new library of components, the AdvancedBlocks™ library.

The AdvancedBlocks library is designed to work with Simelica but also brings a new range of control system component models to the Modelica environment. The blocks are designed to enable the calculation method used to be varied in each particular instance that the block is required. For example, in the DiscreteIntegrator block you can choose from 3 different integration algorithms, whether to apply limits to the integrator or not, and how the initial condition is specified amongst many other options. The main focus is on delivering a user-friendly library to aid control system modelling.

Some example applications will be discussed to illustrate how effective the translation process can be.

1 Motivation

The use of system modeling and simulation is increasing in the automotive industry as we strive to reduce product development times whilst increasing the complexity and quality of the product. As the use of these simulation techniques increases so does the requirement to include more and more detail into the models and to ensure that the interaction between the different systems is being modeled adequately.

For many years Simulink has been the tool of choice for much of the automotive industry to develop both physical and control system models[1,2,3]. The main attraction of Simulink has been its flexibility and the range of toolboxes available to aid control system design, development and calibration. However, many users of Simulink are finding that as the physical system models increase in complexity, the task of developing these models further is becoming increasingly difficult and time consuming. Many are now looking at alternative systems and Modelica based tools are well placed in the market to meet these needs.

The adoption of the Modelica tools is currently limited to those departments within automotive manufacturers that are currently pushing forward the development of complex physical system models[4,5]. This is leading to problems within these companies where the control system engineers are still developing models in Simulink while the design engineers are developing physical system models using Modelica.

Currently tools such as Dymola™ provide methods to generate S-functions from the Modelica models[6] and this then enables the models to be simulated together in one environment. In our experience this method has not been completely successful. We have found that, with our more complex physical models, the Simulink solvers are unable to cope reliably with the generated S-function models. This has led to simulations effectively stalling where the time step becomes so small that the simulation is no longer making progress.

We then simply asked ourselves, why don't we make the process work the other way round? Why not convert the control system model into

Modelica and use that environment to simulate the interactions between control system and physical system. After all Modelica can support a block diagram modeling style and our physical models are working reliably in the Modelica environment.

2 Simelica

2.1 Overview

Simelica is a translation tool for converting Simulink models into equivalent Modelica models. It works as both a command line tool so that its use can be incorporated into scripts and also as a windows tool complete with graphical user interface (GUI).

The translation works by reading the Simulink .mdl file and interpreting this into a Modelica model based on the AdvancedBlocks library. Simelica is capable of dealing with all the modeling methods used in Simulink including:

- From-goto systems
- Signal Bus systems
- Muxed signals
- Data store read/write/memory systems

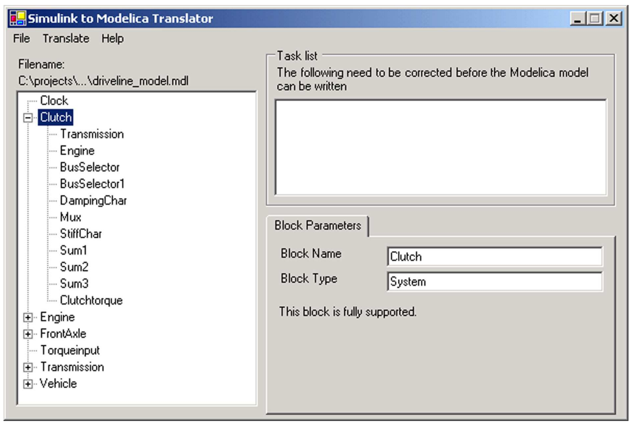
The majority of the standard Simulink library can be automatically translated into an equivalent Modelica block although there are some exceptions including the MatlabFcn, S-function and Stateflow® blocks.

2.2 Using Simelica

The command line version of Simelica provides simple functions to translate a single Simulink file or all the Simulink files contained in a specified directory. This version is useful for incorporation into scripts but it does not provide many of the features available through the GUI that most users will find useful, such as highlighting unsupported blocks. Figure 1 shows a screen shot of Simelica.

When running in GUI mode after the Simulink file is read into the tool the structure of the model is presented to the user. Any unsupported blocks are highlighted to the user at this point along with a brief explanation of what action the user must take either now, or after the Modelica file is generated to ensure that the translated model can be used.

Following translation, a log of the work done is produced. This will list any problem blocks encountered and include their full path in the model. The user can then easily see what, if any, parts of the translated model need further attention before it can be used.



Filename:
C:\projects\...\driveline_model.mdl

The following need to be corrected before the Modelica model can be written

Block Name: Clutch
Block Type: System

This block is fully supported.

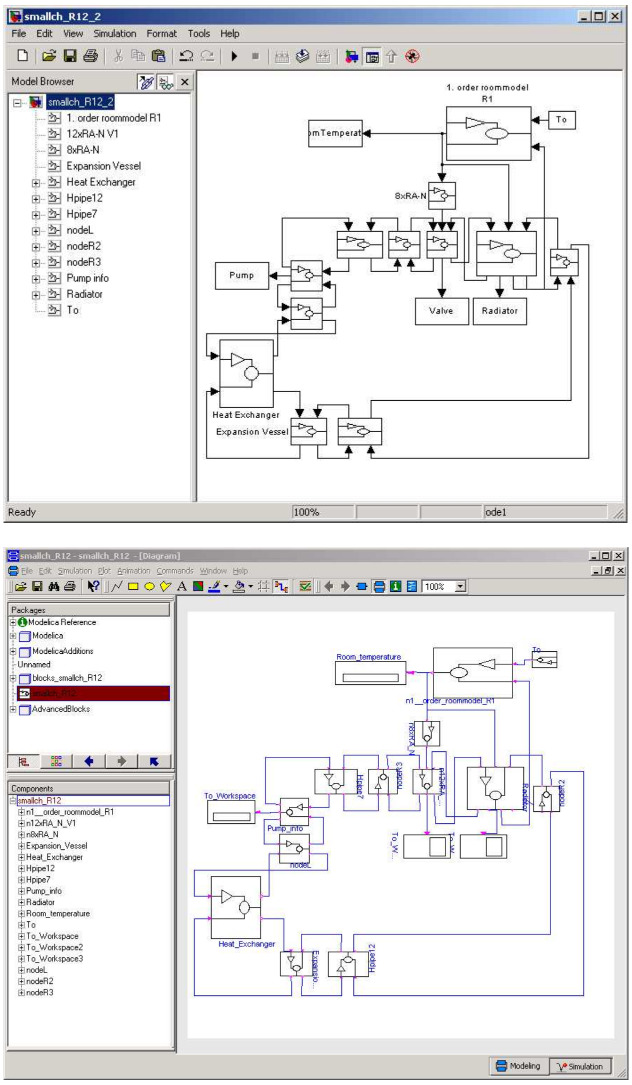
Figure 1: Screen shot of Simelica

As well as the need to translate a model it is also essential to translate the data from the Simulink environment into the Modelica environment. Data can be imported and incorporated into a translated model using Simelica. The model data has to be stored as a Matlab® binary file, which can then be read by Simelica and the data incorporated into the model through the use of a record that is available in every subsystem.

An additional consideration in the translation of data is that Simulink can load different data files into different points of the model through the use of masked subsystems. In Simelica, masked subsystems are identified and the user is given the option of incorporating a data file directly into each masked subsystem. In this case each masked subsystem gains its own unique workspace record to replicate the fact that Simulink defines local workspaces for masked subsystems.

The Modelica models generated by Simelica are based on the AdvancedBlocks library rather than interpreting the model into a flat model file. This ensures that the model appears similar and maintains the same structure as the original Simulink model. Figure 2 shows a comparison of a translated model in Simulink and Dymola. It shows that the model structure is preserved and the layout and connection of blocks in the Modelica version is similar to the original model.

Figure 2: Comparison of a translated model in Simulink (top) and Dymola



3 AdvancedBlocks Library

3.1 Overview

A new Modelica library of control blocks has been designed to provide equivalent blocks in Modelica to those in the Simulink standard library. The design of the library has focused on providing a user-friendly library that can be used effectively as a modeling library. The main focus has been on providing simple ways to select the different options available for each block, for example the integrator method to be used, the port data type to be used, etc. There are a number of areas of interest in the design of the latest version of the AdvancedBlocks library and these are described in the following sections.

3.2 Connector Definition

The first step in developing the new library was to define the connector for the blocks. A new connector was required for a number of reasons; firstly, Simulink supports the use of matrix, vector and scalar signals whilst the original Modelica.Blocks.Interface.InPort and OutPort connectors[7] only support vector signals. Therefore we needed to change the connector definition to support matrix signals. During the translation process Simulink scalar signals are converted into Modelica matrices with only one value and vector signals are converted into Modelica matrices with only 1 row.

A second consideration was that Simulink cascades sample times along the connections. This means that a block can inherit a sample time from its driving block. To achieve this in Modelica we needed to add an additional signal to our connector to carry the sample times from block to block. It is necessary for this sample time signal to be a matrix because when muxed signals are used in Simulink it is possible for each signal to be carrying with it a different sample time. To replicate this behavior in the AdvancedBlocks library we actually pass a sample trigger along the connections that tells the connected block at which point in time it should calculate its output.

The final consideration for designing the connector was that Simulink signals could be different data types. We therefore needed to find a way to define a connector in which we could easily change the data type. We also needed to find a structure that would allow the connectors to be replaced even though the basic data type of the signal might be changing. The syntax for replaceable classes[8] would specifically prohibit the simple swapping of connectors if the basic types are different. Fortunately it is possible to replace classes that extend from the same base class.

To overcome the constraints of the language and to meet the design requirements the connectors are defined in packages and are created in a two-stage process. Each connector package specifies either an input or output connector for a specific data type. All the connector packages are extended from an appropriate base package that defines a base connector and a base data type conversion function. There is one base package for input connectors, and one for output connectors. Figure

3 shows the base package definition for the output connectors.

Figure 3: Base Connector Package Definition

```
partial package Base
  partial connector Outport "Output signal"
    parameter Integer n=1 "Dimension 1 of signal matrix";
    parameter Integer m=1 "Dimension 2 of signal matrix";
    output Integer sampletrigger[n, m] "Sample trigger to be
passed between blocks";
  end Outport;
  partial function Convert
  end Convert;
end Base;
```

The data type conversion function is used to apply the correct data type to the output signal. The blocks within the AdvancedBlocks library all use variables of type Real internally to handle the calculations. To correctly convert the internal signal type to that required in the connector a function is used that changes the signal data type and applies any limits to the value that may be required for a specific data type.

A connector for each required data type is then defined within its own package. This package must include a connector and function definition that extends from those in the base package. Figure 4 shows how the output connector for the uint8 (unsigned 8 bit integer) data type is defined in the AdvancedBlocks library.

Figure 4: Definition of the uint8 connector

```
package uint8 "uint8 (unsigned 8 bit integer) output signal"
  connector Outport "uint8 output signal"
    extends Base.Outport;
    output Types.uint8 signal[n, m]  "Signal value";
  end Outport;
  function Convert
    extends Base.Convert;
    input Real u;
    output Types.uint8 y;
  algorithm
    y := integer(if u > 255 then 255 else if u < 0 then 0 else u);
  end Convert;
end uint8;
```

By declaring the different connectors within their own package it makes it possible to replace both the connector and conversion function using one redeclare statement. By ensuring that the connector and function names are the same in each package, the replaced package automatically changes the connector and conversion function to the chosen data type. In figure 5 the replaceable package Out1DataType is defined and then the Outport connector is instantiated from this replaceable package. The constraint on the replaceable package ensures that we will only ever be able to replace the connector package with another valid package.

This structure to the design of the connectors and data type conversion function means that each connector in a block in the AdvancedBlocks library can use a different data type and this is achieved by simply redeclaring the relevant package that defines that connector to match the desired data type.

Figure 5: Example use of a connector

```
block OutputExample
  replaceable package Out1DataType =
    AdvancedBlocks.Interface.Connectors.Outputs.uint8 extends
    AdvancedBlocks.Interface.Connectors.Outputs.Base;
  Out1DataType.Outport out1(n=1, m=1);
protected
  Real y1[1,1] "Result of internal calculation";
equation
  out1.signal = Out1DataType.Convert(y1);
end OutputExample;
```

Unfortunately this design cannot be implemented in the current version of Modelica because the data conversion function does not generate an event but integer values, such as those in the connector, are only allowed to change at events. This means that where we would like to use an Integer or Boolean data type in the connector we are unable to do so. The work around in the current version of the library is that all the connectors use a Real data type. The conversion functions also output a Real data type regardless of the actual data type desired but internally they apply the limits and round values as appropriate, i.e. round to the nearest integer if an integer data type is requested.

3.3 Continuous and Discrete time modes

A large proportion of the blocks in the Simulink standard library can run in different time-modes, i.e. either continuous or discrete time modes. In addition where blocks are able to run in discrete-time mode they can be defined to run at a set sample rate or they can inherit their sample time from their parent system or from their driving block.

To enable blocks within the AdvancedBlocks library to support running in these different time modes they have been defined so that they extend from a replaceable block that governs the calculation method used. Within each block that supports running in different time modes there is an encapsulated package that contains the different definitions required for operating in the different time modes. Figure 6 shows how the AdvancedBlocks.Math.Abs block is defined with the ability to switch between continuous and discrete time mode.

Figure 6: Block structure to support different time modes

```
block Abs "Abs block"
  extends TimeMode;
  replaceable block TimeMode = Options.Continuous extends
Options.Base;

  encapsulated package Options
    import AdvancedBlocks.Interface;

    partial block Base "Base class and calculation function"
      extends Interface.BaseBlock "Icon and common properties";
      extends Interface.IOLayers.SI.Inports "Input definition";
      extends Interface.IOLayers.SO.Outports "Output definition";
    protected
      Real y[nout[1, 1], nout[1, 2]] "Result of internal calculation";
    equation
      y = abs(u1);
    end Base;

    block Continuous "Continuous time mode"
      extends Base;
    equation
      y1 = y;
      y1st = -ones(nout[1, 1], nout[1, 2]); // Sample trigger to next
block
    end Continuous;

    block Triggered "Discrete time mode"
      extends Base;
    protected
      outer Boolean sampletrigger[1];
    equation
      y1st = if sampletrigger[1] then ones(nout[1, 1], nout[1, 2]) else
zeros(nout[1, 1], nout[1, 2]); // Sample trigger to next block
      when sampletrigger[1] then
        y1 = y;
      end when;
    end Triggered;
  end Options;
end Abs;
```

When the user then drags the Abs block into their model for use they can simply switch time modes by redeclaring the block TimeMode to be any of the versions contained in the Options package. This is made even easier in tools such as Dymola where the version of TimeMode to be used can be selected from a pull-down menu. In the Abs block shown in Figure 6 it is possible to choose between a Continuous time mode and a Triggered time mode. In the Triggered time mode the sample time is inherited from the parent system through the outer variable sampletrigger.

The structure of the Modelica code means that the actual equations defining the behaviour of the block are separate to the equations that force the block to act in a particular time-mode. This eases the maintenance of the library by not repeating blocks of code. This becomes a major consideration in the more complicated blocks.

3.4 Integrator Block

The continuous time integrator in Simulink is one of a number of blocks that can function in a variety of different ways depending on the choices made by the user each time the block is added to a model. The options for the integrator block include applying limits to the output, initialising with internal or external initial conditions, allowing for external reset signals, outputting state information and information on the limit condition[9]. To define all this in Modelica in a way that is easy to use has been achieved by extending the ideas described and used to change the time mode of the Abs block. This has led to the encapsulated package within the Integrator block becoming much more complex including several levels of hierarchy.

Each Integrator method is an extension of the same base class defined in the encapsulated package. The base block contains the definitions for the input and output connections and instantiates these from replaceable packages. This structure ensures that each integration option can redeclare the input and output layers to have the required number of connectors for this method. For example, if an external initial condition is required then two inputs are needed rather than one.

The result of this structure for the user is that they can easily choose what functionality they want within the integrator block in each instance. Figure 7 shows the dialog box produced by Dymola for the integrator block. Each option can be changed through the use of a pull-down menu showing the available options.

This same structure idea has also been used for many other blocks in the AdvancedBlocks library including the discrete integrator, math function block, trigonometric function block and many others.

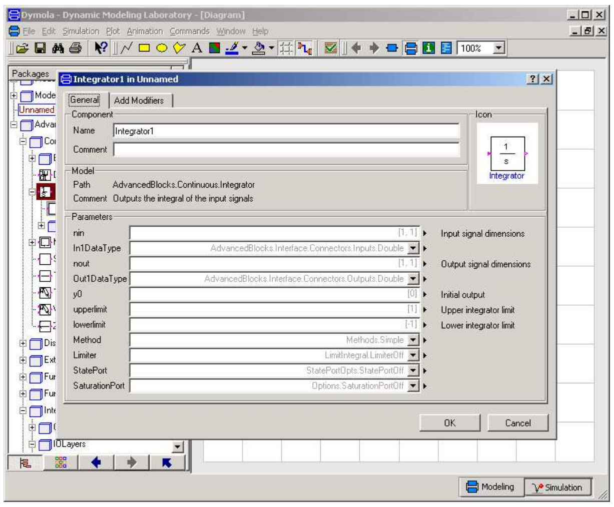


Figure 7: Dymola dialog box for the Integrator block

Dymola - Dynamic Modeling Laboratory - [Diagram]
File Edit Simulation Plot Animation Commands Window Help
100%
Packages
Integrator1 in Unnamed
General
Add Modifiers
OK
Cancel
Modeling
Simulation

Component
Name: Integrator1
Comment:

Model
Path: AdvancedBlocks.Continuous.Integrator
Comment: Outputs the integral of the input signals

Icon: Integrator

Parameters

nin	[1, 1]	Input signal dimensions
In1DataType	AdvancedBlocks.Interface.Connectors.Inputs.Double	
nout	[1, 1]	Output signal dimensions
Out1DataType	AdvancedBlocks.Interface.Connectors.Outputs.Double	
y0	[0]	Initial output
upperlimit	[1]	Upper integrator limit
lowerlimit	[-1]	Lower integrator limit
Method	Methods.Simple	
Limiter	LimitIntegral.LimiterOff	
StatePort	StatePortOpts.StatePortOff	
SaturationPort	Options.SaturationPortOff	

3.5 Iterator Systems

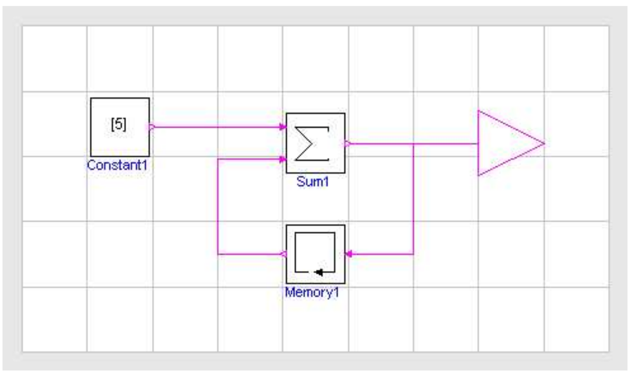
The latest version of Simulink includes for-iterator and while-iterator subsystems. In these subsystems the blocks are executed a number of times at each time step. The actual number of times that the sub-system interates at each time step can vary from time step to time step. The iterator subsystems have been introduced into Simulink to encourage its use as a control system software design and development tool. The key improvement for users in introducing these blocks is to facilitate the auto-coding of control system software. These subsystems along with the range of if-then-else and switch-case blocks make it much easier for controls engineers to design and develop the control system software.

Iterator subsystems can be translated into Modelica where a fixed number of iterations are specified such as in some instances of for-iterator subsystems. In these cases the blocks within the subsystem are instantiated into an array of blocks where the size of the array equals the number of iterations to be performed. For example, figure 8 shows how a simple subsystem would be defined if it was required to iterate 5 times at each time step. The output from this subsystem at the first time step would be 25, after the second time step it would be 50, etc.

In this example the constant, sum and memory blocks are declared as component arrays where the size of the array is equal to the number of iterations. Each block within the component array forms a different iteration of the for loop. The subsystem output connector is only connected to the Sum block in the final iteration of the for loop so that we get the full value of the loop passed out of this subsystem. The memory block is connected so that it effectively spans the iterations. The input to the memory block comes from the output of the Sum block in the current loop. The output from the memory block is connected to the input of the Sum block in the next iteration. In the final iteration of the loop the output from the memory block is connected to the input of the Sum block on the first loop.

To use this idea for while-iterator subsystems and for-iterator subsystems where the number of iterations can vary at each time step would require the component arrays to vary in size at each time step. It is not currently possible to implement this type of system in Modelica where the number of iterations varies at each time step.

Figure 8: Example Iterator subsystem



```
model ForIteratorSubsystem
   extends AdvancedBlocks.Interface.Subsystem;
public
  constant Integer NumIterations ={5} "Number of iterations";
  Sources.Constant[NumIterations] Constant(each k=[5]);
  Math.Sum[NumIterations] Sum;
  Continuous.Memory[NumIterations] Memory;
  Interface.Connectors.Outputs.Double.Outport out1;
equation
  for i in 1: NumIterations loop
    connect(Constant[i].out1, Sum[i].in1);
  end for;
  for i in 1: NumIterations  loop
    connect(Sum[i].out1, Memory[i].in1);
  end for;
  for i in 1: NumIterations - 1 loop
    connect(Memory[i].out1, Sum[i + 1].in2);
  end for;
  connect(Memory[NumIterations].out1, Sum[1].in2);
  connect(Sum[NumIterations].out1, out1);
end ForIteratorSubsystem;
```

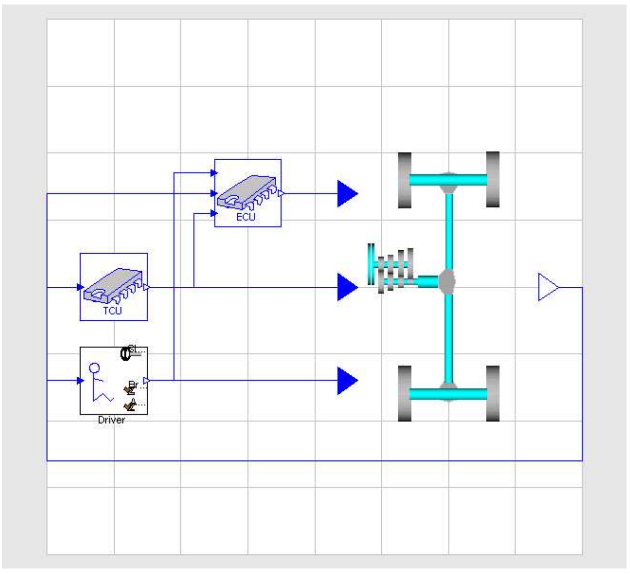
4 Example models

As well as a large number of relatively simple test cases a number of complex real-world examples have been translated. Two examples of translating real-world models are presented in the following sections and the simulation performance and results have been compared.

4.1 Cruise Control Simulation

In this example we have combined a detailed physical powertrain model with the actual cruise control function from an engine control system, see figure 9. The cruise control function is developed by the system supplier in Simulink and then used by both the customer and supplier to develop and calibrate the system into the end product. Ultimately the actual code downloaded into the engine control unit is generated automatically from the Simulink model and so the latest version of the cruise control strategy will always be available in Simulink.

Figure 9: Powertrain model and converted controller system model

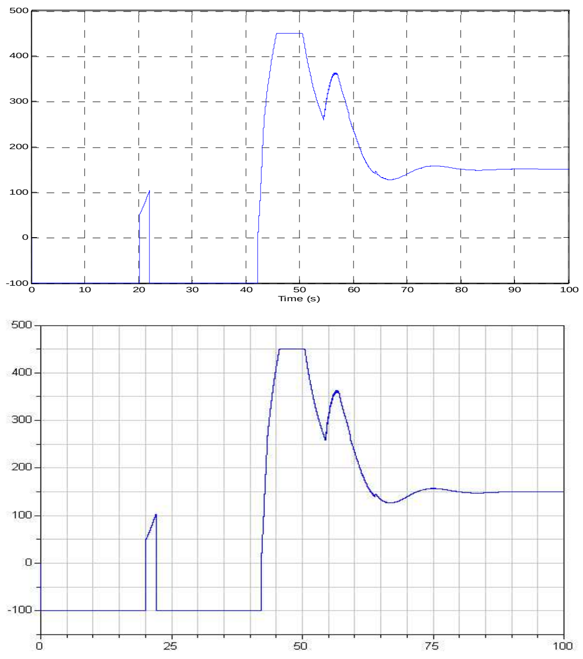


This cruise control function is designed to work as part of a torque structure engine management system. This means that the when the cruise control function is active it demands an engine torque and feeds this into the torque structure function which converts this torque demand into a throttle position, spark timing and amount of fuel to inject. These quantities are determined so that the engine will produce as close to the demanded torque value as is physically possible within the constraints of the calibration.

For this example we have chosen to convert just the cruise control function from Simulink into Modelica using Simelica. This is then coupled to a detailed powertrain model that does not include an engine model. The torque demand from the cruise control model is applied directly to the engine flywheel. In this way we can eliminate the need to calibrate the torque structure function on the assumption that this will be calibrated to translate the demanded torque into the actual engine torque produced at a later date.

The aim of this model was to enable the calibration of the cruise control function early in the development process. The task of calibrating the cruise control function traditionally requires a significant amount of test work to achieve good results. This is due to the difficulties involved in repeating each test exactly and the wide range of conditions that need to be tested. It is therefore an ideal candidate for applying simulation techniques which can reproduce the same test conditions repeatedly and help produce an initial calibration.

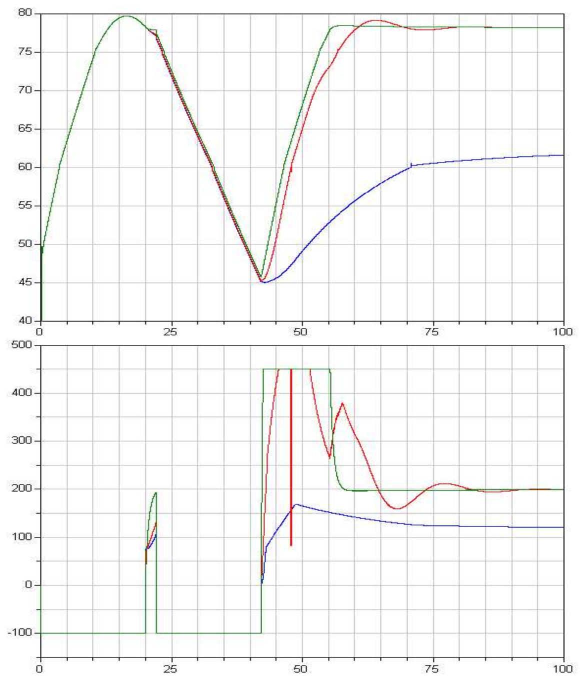
Figure 10: Comparison of Simulink (top) and Modelica Controller models



To translate the controller model and validate the generated Modelica model a Simulink model was generated that played measured data into the control system and recorded the outputs. This model, its parameter data and the measured data were then translated into Modelica using Simelica. Figure 10 compares the outputs from the controller function in both Simulink and Modelica. By ensuring that the Modelica controller model produces the same results as the original Simulink model we can be sure that the translated model is accurate.

Once we are satisfied that the translated controller model was behaving in the same way as the original Simulink model the new Modelica model could then be used to attempt to calibrate the control system. There are many parameters within the control system that need to be calibrated and by repeating the same test exactly the effect of altering these parameters can be assessed and a calibration can be defined. Figure 11 shows the effect of altering one of the gains in the control system on a given test.

Figure 11: Effect of controller gain on a Cruise Resume Event



In this test the driver puts the vehicle into cruise mode at 20 seconds but then presses the brake at 22 seconds forcing the vehicle out of cruise mode and into a gentle deceleration. At 42 seconds the driver presses the Resume button and the vehicle enters back into cruise mode and attempts to regain the speed it was travelling at when the driver first put the vehicle into cruise mode. The three results traces demonstrate the effect of altering one of the gains in the cruise control function on the vehicle response.

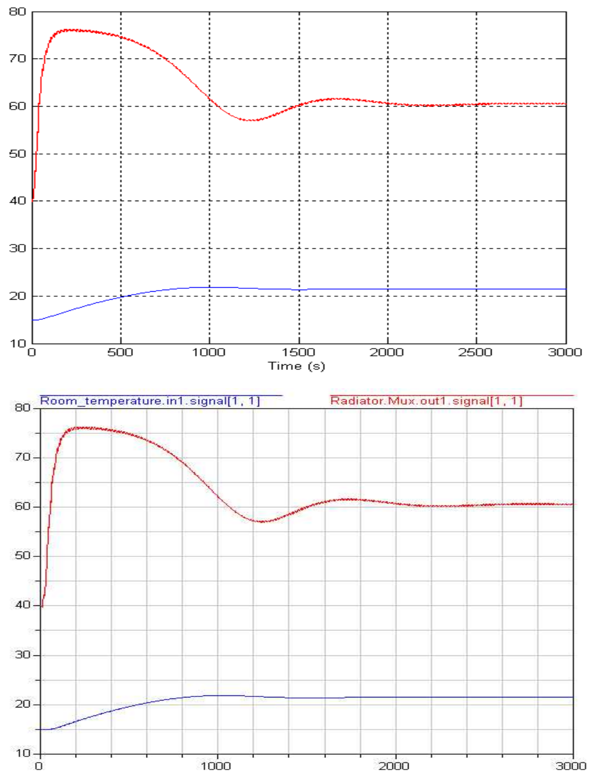
4.2 Central Heating System

The model shown in figure 8 was developed in Simulink to predict the performance of a small central heating system. The main motivator for attempting to translate this model into Modelica was to see if the simulation times would be improved. As Dymola generates efficient compiled models from the Modelica models and Simulink interprets the model at runtime it would provide an interesting comparison of simulation performance.

Using Simelica the model has been translated into Modelica and then compiled and simulated using Dymola 5.1a. Figure 2 shows this model in both Simulink and Modelica. It is clear from the diagrams that the same model structure and layout has been preseved during the translation process and any visible differences in the two diagrams are purely down to the way the two tools present the models graphically.

Figure 12 shows the results traces produced by both Dymola and Simulink versions of this model. It can be seen that although the model has been translated into Modelica the results obtained are the same. The time required to simulate a 24 hour period for the Modelica version of the model is 31 seconds but Simulink required just 9 seconds to carry out the same simulation on the same PC.

Figure 12: Comparison of Simulink (top) and Dymola simulation results



When comparing these simulation times it is also essential to consider that in Dymola 493 signals were stored but in the Simulink version only 12 signals were stored. In more complex systems the immediate availability of all this data would be very useful to help diagnose problems. To carry out the same investigation in the Simulink version of the model would require the user to manually add scopes to areas of the model that they suspect of causing the problem and then re-running the model. This process of adding scopes and re-running the model may have to be repeated several times before the problem can be correctly diagnosed.

5 Limitations

5.1 Limitations of Simelica

There are also some blocks available in Simulink that cannot be automatical translated into Modelica. These include blocks such as the MatlabFcn and S-function. The MatlabFcn block cannot be translated because it allows the user to use any Matlab script or command in the model, many of which do not have an equivalent in Modelica. The S-function block cannot be automatically translated because the c-code might need to be changed significantly to work as an external function in Modelica. It is possible to do this manually though. There are a number of other blocks that are currently unsupported but through the continual development of the tool the majority of these will be incorporated.

Another feature that cannot be automatically handled is the initialisation commands that can be fed into models and masked blocks. These cannot be supported because they allow any Matlab command to be used and executed during the model initialisation and many of these commands do not have an equivalent in Modelica. Rather than attempt to handle this and get it wrong, Simelica opts to simply copy all the commands from the initialisation layer into a comment in the block and then flag this to the user as a problem requiring attention.

The final limitation in the translation process currently is that matrix signals and signal data types are not supported. Although many of the features exist in the AdvancedBlocks library it is not yet possible for Simelica to correctly translate models that include these features. Where data types other than the Matlab data type double are used in the model the different data type will be ignored by the translator and the converted model will use the double data type. Models that contain matrix signals will have the signal dimensions incorrectly set. From the point-of-view of the AdvancedBlocks library and Simelica a matrix signal is any signal that has more than one row. Many of the blocks within the AdvancedBlocks library will not currently function correctly when matrix signals are used. These issues will be addressed in future versions of Simelica and the AdvancedBlocks library.

5.2 Limitations of the Modelica language

There are some key differences between the Modelica language and what is possible in Simulink. Modelica does not support the same flexibility in block naming as Simulink does. For example Simulink can use any special character in the block names; names can also start with numbers; names can contain white space characters. Some transformations therefore have to be made by Simelica to ensure that a block name conforms to the Modelica specification. The difficulty here can be that blocks that were named differently in Simulink purely because of the inclusion of a special character, or series of characters that are prohibited in Modelica could end up with the same name in the Modelica model leading to errors.

Although many of the modelling methodologies used in Simulink can be translated into a form for use in Modelica it is not always possible to provide an equivalent methodology in Modelica. For example, signal buses are translated into simple muxed signal systems where the bus selector is defined to extract particular signals by index rather than by name. In Simulink the names of the signals are passed along the connection include the heirarchy within the bus system. Signals can then be extracted by selecting a particular signal name. This feature is widely used in Simulink[10] as it provides a powerful way to pass large groups of signals around a model.

A large number of the blocks within the AdvancedBlocks library contain encapsulated packages that would ideally be hidden from the user. This could be achieved by declaring the package as protected but then all replaceable classes and parameters would not be visible in the GUI dialogs produced by Dymola. To get around this all the parameters and replaceable classes would then have to be declared in the block containing the encapsulated package but this would mean that the user is presented with options and parameters that might not be valid because of other selections they have already made. Another method of hiding these packages from the user whilst still making the parameters and replaceable classes visible in the tool dialogs is required. Ideally it would also be possible for the available options and required parameters to change as selections are made by the user.

6 Future

It is important to note that this paper refers to the current version of Simelica and the AdvancedBlocks library and that they will continue to evolve and support more features. They will both be continually developed to support the latest versions of Simulink and Modelica.

7 Acknowledgments

Many people have provided support during the design and development of Simelica and the AdvancedBlocks library and I would like to extend my thanks to them. Specifically Hilding Elmqvist, Hans Olsson, Sven Erik Mattsson and Dag Bruck from Dynasim, Martin Otter from DLR and Mike Tiller from Ford Motor Company.

8 References

1. S.R. Anderson, C.R. Ciesla, D.M. Carey, R. Shankar, “A powertrain simulation for engine control system development”, *1996 SAE International Truck and Bus Meeting and Exposition*, SAE 962171
2. P.M. Fussey, C.L. Goodfellow, K.K. Oversby, B.C. Porter, J.C. Wheals, ”Integrated Powertrain (IPT) Model – Stage 2: Systems Integration, Supervisory Control and Simulation of Emissions Control Technology”, *SAE 2001 World Congress*, SAE 2001-01-0928
3. J.A. MacBain, J.J. Conover, A.D. Brooker, “Full-vehicle simulation for series hybrid vehicles”, *Future Transportation Technology Conference*, SAE 2003-01-2301
4. M. Tiller, W.E. Tobler, and M. Kuang, “Evaluating Engine Contributions to HEV Driveline Vibrations”, *Proceedings of the 2nd International Modelica Conference*
5. S. Soejima, “Examples of usage and spread of Dymola within Toyota”, *Modelica Workshop 2000 Proceedings*
6. “Dymola 5.0 User’s Manual”, *Dynasim AB.*
7. “Modelica Standard Library 1.5”, *The Modelica Association* , 2002
8. “Modelica Language Specification, Version 2.0”, *The Modelica Association*, 2002.
9. “SIMULINK Release 13” (documentation), *The Mathworks Inc.*
10. C. Belton, P. Bennet,. P. Burchill, D. Copp, N. Darnton, K. Butts, J. Che, B. Hieb, M. Jennings and T. Mortimer, “A Vehicle Model Architecture for Vehicle System Control Design”, *SAE Congress 2003*, SAE 2003-01-0092