AIAA Modeling and Simulation Technologies Conference and Exhibit
2-5 Aug 2010, Sheraton Centre Toronto, Toronto, Ontario, Canada AIAA-2010-7776

Integrating Simulink with other Simulation Environments

Mark McBroom[1]
MathWorks Inc., Novi, MI
Tom Erkkinen[2]
MathWorks Inc., Novi, MI
Matt Behr[3]
MathWorks Inc., Natick, MA

Modeling, simulation, and code generation are vital technologies for developing aerospace systems. The ability to integrate models from commercial off-the-shelf (COTS) tools into large software programs used as corporate or program-wide simulation environments is becoming increasingly important. Techniques for generating code from Simulink® models and integrating that code into corporate simulation environments are described herein. The techniques are based on actual aerospace use cases, and examples are shown using MathWorks products [1].

I. Introduction

Simulink® software is a robust modeling platform capable of simulating systems containing continuous time dynamics, multiple discrete sample rates, and asynchronous events. It can model a subsystem, a system, or systems of systems. There are, however, situations in which legacy systems mandate or modeling requirements necessitate the use of other simulation environments or frameworks in addition to Simulink.

This paper examines methods for integrating Simulink generated code into a code-based simulation environment, which is typically written in a traditional programming language such as C, C++, or FORTRAN. While the topics discussed also apply to integration with COTS simulation tools other than Simulink, this is not the paper's focus. Similarly, while architectures for distributed computer simulation systems such as High Level Architecture (HLA) are mentioned, such architectures are also not part of the paper's focus.

We examine the various forms simulation environments can take, technical challenges in getting multiple environments to work together, and features of Simulink that enable integration. This paper is based on experiences drawn from working with many aerospace companies that have successfully integrated Simulink into their corporate simulators.

II. Factors Affecting the of Choice of Simulation Environments

The primary reasons for requiring the use of multiple simulation environments are history, mandate, and modeling requirements.

Due to the complexity of the systems being developed and the difficulty in testing them, simulation has been a key part of aerospace systems development for decades. Thus, organizations often have simulation environments at their disposal from previous programs. Legacy simulation environments offer several benefits: they have been proven on real programs, people within the organization are familiar with them, and they have capabilities that support the organization's specific development needs and processes.

However, legacy systems also have drawbacks. First, they are often costly to maintain. Second, as senior engineers retire, the organizational knowledge required to operate legacy systems can dissipate. Third, organizations are often reluctant to update these tried-and-true systems to take advantage of more modern technology. For

[1] Manager, Pilot Engineering
[2] Manager, Code Generation and Certification Technical Marketing
[3] Manager, Aerospace and Defense Industry Marketing

example, many corporate simulators do not fully leverage multi-core computers and high performance computing clusters.

Another factor driving the use of multiple simulation environments is mandate. Government agencies, for example, are now relying more on simulation in the acquisition process. In these cases, contractors choose their own internal development tools and processes but they also must deliver code to their customer that integrates into the customer's simulation framework. Other organizations are required to deliver code that integrates into theatre-level or system-of-systems simulations. Rather than create additional simulations, such organizations can reuse their design simulations and integrate them with system-of-systems simulations.

Finally, there are cases in which the primary simulation environment, in this case Simulink, is not the most appropriate modeling tool. Examples include scene generators that model specific physical phenomenon such as compressible fluid flow in high fidelity.

III. Integrated Simulation Environments

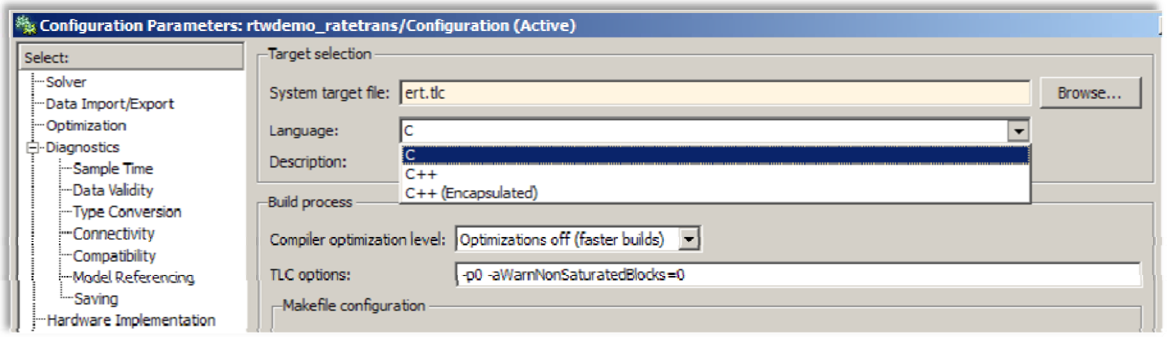
There are several ways to integrate simulations including HLA, which was developed specifically for distributed simulation systems, and standard inter-process communication techniques and protocols including shared memory, TCPIP/ UDP, or COM.

This paper focuses on code-based, static integration in which a component from Simulink (such as a digital filter or actuator model) is designed, coded, compiled, linked, and invoked as defined by the *master* environment. An example of this type of integration is described in the 2005 AIAA Paper "Enabling Interoperability of Native Engineering Toolsets with System Simulations and Flight Software" [2].

One other approach is to integrate code from a simulation environment into Simulink, with Simulink serving as the master, but this approach is well documented [3] and not discussed herein.

A. C vs. C++

The first thing to consider when integrating generated code is the language and structure of the custom simulation environment. As shown in Fig. 1, the Real-Time Workshop® Embedded Coder™ product has three code language options for generating code from Simulink: C, C++, and C++ (Encapsulated). The language and architecture of the target simulation environment is the primary factor in selecting which option to use.



System target file: ert.tlc
Language: C
Description:
Compiler optimization level: Optimizations off (faster builds)
TLC options: -p0 -aWarnNonSaturatedBlocks=0

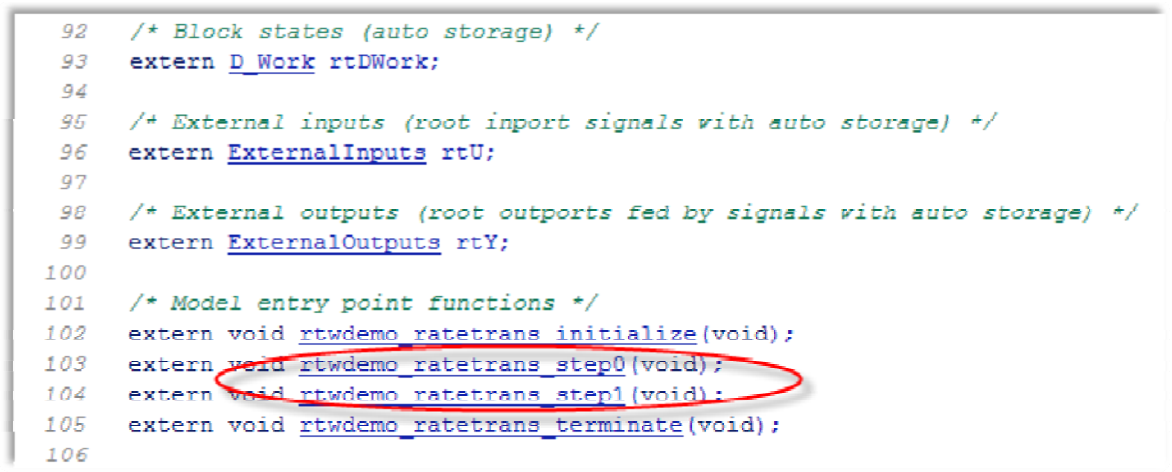
Figure 1. Target language selection.

C language should be selected if the simulation environment is written in C, FORTRAN, or another procedural language. The C++ option should be selected if the simulation environment is written in C++ but does not have strong object-oriented architecture. The C++ option generates code similar to the C option but ensures that the generated code is compatible with an ISO compliant C++ compiler. The C++ (Encapsulated) option generates a C++ class for the Simulink model. This option should be used when the simulation environment is written in C++ and utilizes object-oriented architecture.

B. Tasking Execution Modes

There are two options for controlling the tasks generated from a Simulink model that will be called from the target simulation environment. The most common is to generate code from a full Simulink model with an initialize function, a terminate function, and one or more step functions, plus scheduling code.

Scheduling code is generated when the Simulink model contains blocks with different sample rates. When the Tasking Mode is defined as Single-Tasking, a single step function will be created with scheduling code to manage the execution of each sample rate. The simulation environment must be configured to execute the step function at the same sample rate specified in the Simulink model. When the Tasking Mode is defined as Multi-Tasking, a step function will be created for each sample rate in the Simulink model. Scheduling code will be generated to handle the exchange of data between the step functions. The simulation environment must be configured to execute each of the step functions at the sample rates specified in the Simulink model. The code shown in Fig. 2 contains the function prototypes for a Simulink model with two sampling rates.



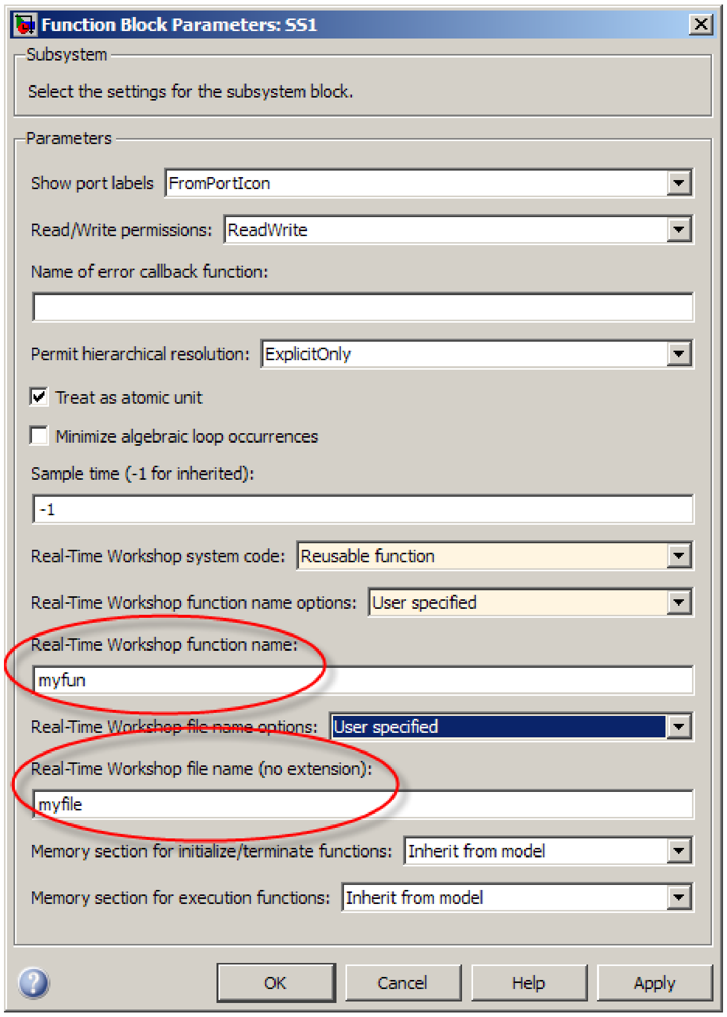
```
/* Block states (auto storage) */
extern D_Work rtDWork;

/* External inputs (root inport signals with auto storage) */
extern ExternalInputs rtU;

/* External outputs (root outports fed by signals with auto storage) */
extern ExternalOutputs rtY;

/* Model entry point functions */
extern void rtwdemo_ratetrans_initialize(void);
extern void rtwdemo_ratetrans_step0(void);
extern void rtwdemo_ratetrans_step1(void);
extern void rtwdemo_ratetrans_terminate(void);
```

Figure 2. Function interface for a multi-rate Simulink model.

The second, less-common option for controlling tasks is to generate code for individual subsystems of a larger Simulink model without scheduling code. For this option, the user is responsible not only for configuring the simulation environment to execute the generated code at the proper rate(s), but also for writing additional code to manage any data that is shared between different Simulink subsystems.

C. Function and File Partitioning

By default, Real-Time Workshop Embedded Coder will minimize the number of generated functions and source files because function boundaries inhibit some code optimizations. However, users can control how the function and source files are created and specify their names, as shown in Fig. 3. Users can also choose between an interface that is based on global data and one that is based on function arguments.


Function Block Parameters: SS1

Subsystem

Select the settings for the subsystem block.

Parameters

Show port labels: FromPortIcon

Read/Write permissions: ReadWrite

Name of error callback function:

Permit hierarchical resolution: ExplicitOnly

[✓] Treat as atomic unit

[ ] Minimize algebraic loop occurrences

Sample time (-1 for inherited): -1

Real-Time Workshop system code: Reusable function

Real-Time Workshop function name options: User specified

Real-Time Workshop function name: myfun

Real-Time Workshop file name options: User specified

Real-Time Workshop file name (no extension): myfile

Memory section for initialize/terminate functions: Inherit from model

Memory section for execution functions: Inherit from model

Figure 3. Function and file partitioning.

D. Function Signature Control

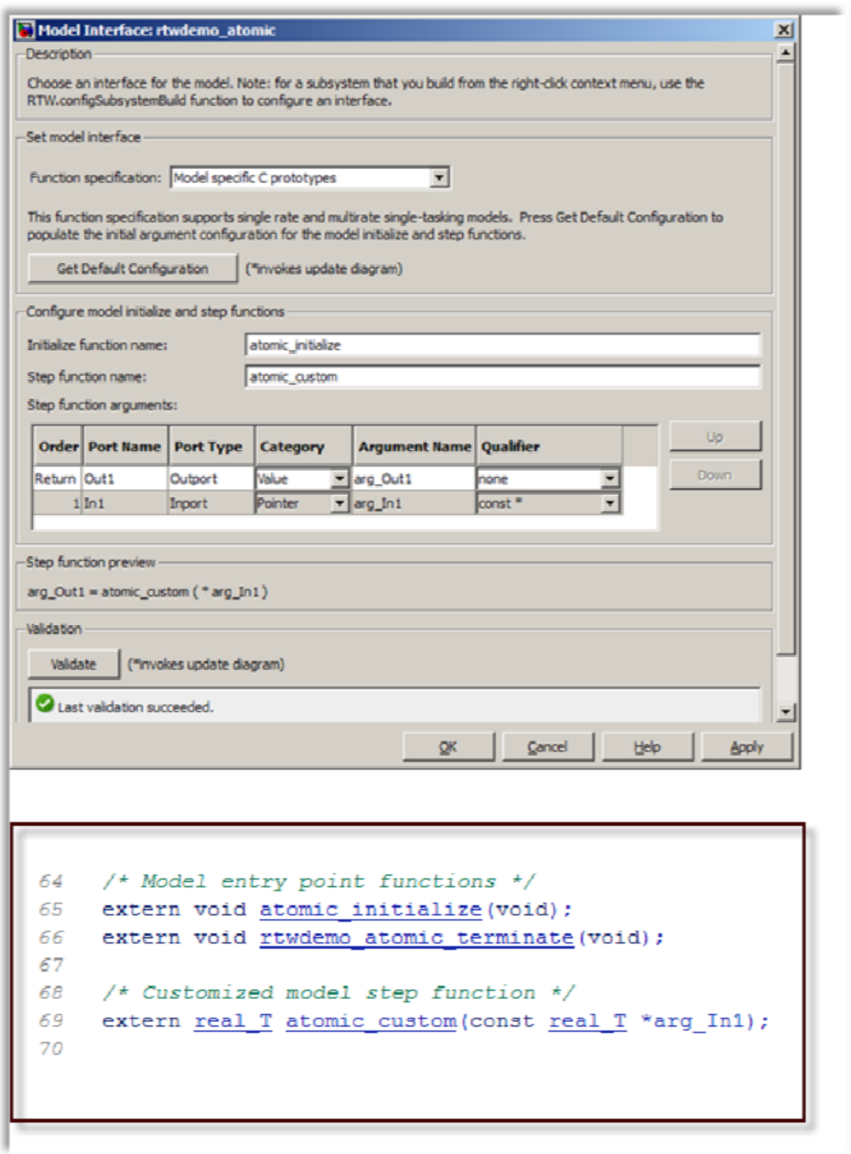
Target simulation environments usually have a documented application programming interface (API) for integrating user code. To facilitate integration using such APIs, Real-Time Workshop Embedded Coder provides a number of options for controlling the function signatures in the generated code.

C and C++

By default, Real-Time Workshop Embedded Coder generates the initialize, step, and terminate functions as void-void functions with global data. Separate data structures are created for each category of data in a Simulink model. The generated code allocates storage for each of these separate data structures, including variables created for Simulink inports, outports, parameters, and states.

The user can override this default behavior and pass pointers to each of the structures as arguments to the initialize, step, and terminate functions. In this case, the generated code is better suited for reuse because it does not rely on global data; however, the user is responsible for declaring storage for each of the data structures.

The user can explicitly control the signature, or prototype of the initialize, step, and terminate functions by specifying the function name, the argument names, passing mechanism (by reference or by value), and qualifiers. The Real-Time Workshop Embedded Coder dialog box for controlling a function prototype is shown in Fig. 4. In this example, the Simulink model has one inport and one outport. Instead of using the default void-void prototype, the user has configured the function prototype to pass the input as a constant pointer (const *) and use the Simulink outport as the return value.



Model Interface: rtwdemo_atomic

Description

Choose an interface for the model. Note: for a subsystem that you build from the right-click context menu, use the RTW.configSubsystemBuild function to configure an interface.

Set model interface

Function specification: Model specific C prototypes

This function specification supports single rate and multirate single-tasking models. Press Get Default Configuration to populate the initial argument configuration for the model initialize and step functions.

Get Default Configuration (*invokes update diagram)

Configure model initialize and step functions

Initialize function name: atomic_initialize

Step function name: atomic_custom

Step function arguments:

Order	Port Name	Port Type	Category	Argument Name	Qualifier
Return	Out1	Outport	Value	arg_Out1	none
1	In1	Inport	Pointer	arg_In1	const *

Step function preview

arg_Out1 = atomic_custom (* arg_In1)

Validation

Validate (*invokes update diagram)

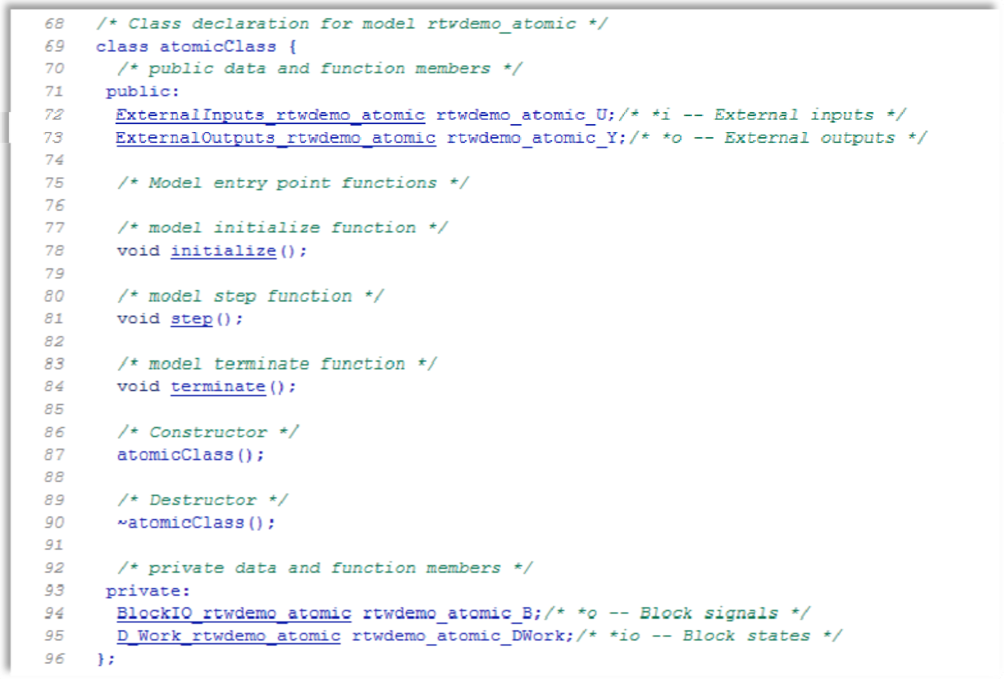
Last validation succeeded.

```
/* Model entry point functions */
extern void atomic_initialize(void);
extern void rtwdemo_atomic_terminate(void);

/* Customized model step function */
extern real_T atomic_custom(const real_T *arg_In1);
```

Figure 4. Function prototype control and associated code.

C++ (Encapsulated)

Similar capabilities for function signature control are available when generating C++ (Encapsulated) code. By default, void-void methods are generated for the initialize, step, and terminate functions. The data structures for inports, outports, parameters, and states are properties of the C++ class as shown in Fig. 5. Storage for the data structures is created when the class is instantiated by the simulation environment.



```
/* Class declaration for model rtwdemo_atomic */
class atomicClass {
  /* public data and function members */
 public:
  ExternalInputs_rtwdemo_atomic rtwdemo_atomic_U;/* *i -- External inputs */
  ExternalOutputs_rtwdemo_atomic rtwdemo_atomic_Y;/* *o -- External outputs */

  /* Model entry point functions */

  /* model initialize function */
  void initialize();

  /* model step function */
  void step();

  /* model terminate function */
  void terminate();

  /* Constructor */
  atomicClass();

  /* Destructor */
  ~atomicClass();

  /* private data and function members */
 private:
  BlockIO_rtwdemo_atomic rtwdemo_atomic_B;/* *o -- Block signals */
  D_Work_rtwdemo_atomic rtwdemo_atomic_DWork;/* *io -- Block states */
};
```

Figure 5. C++ class declaration.

The user can override this default behavior and explicitly control the class name and prototype for the step method. The step method name, argument names, passing mechanism (by reference or by value), and qualifiers are fully configurable.

E. Data Interface Control

Users can configure the data interface based on the needs of the target simulation environment. For the C and C++ code generation options, data can be scoped at the global, file, or function level. When scoped globally, predefined storage classes are available, including const, volatile, exported global, imported global, #define, structure, and bit-field. Get and set access methods, user-specific definitions, and C++ definitions are also available.

For the C++ (Encapsulated) option, all data are properties of the C++ class generated for the Simulink model. The properties can be scoped as either private or protected. Access methods can be generated to provide a controlled interface to the properties of the class.

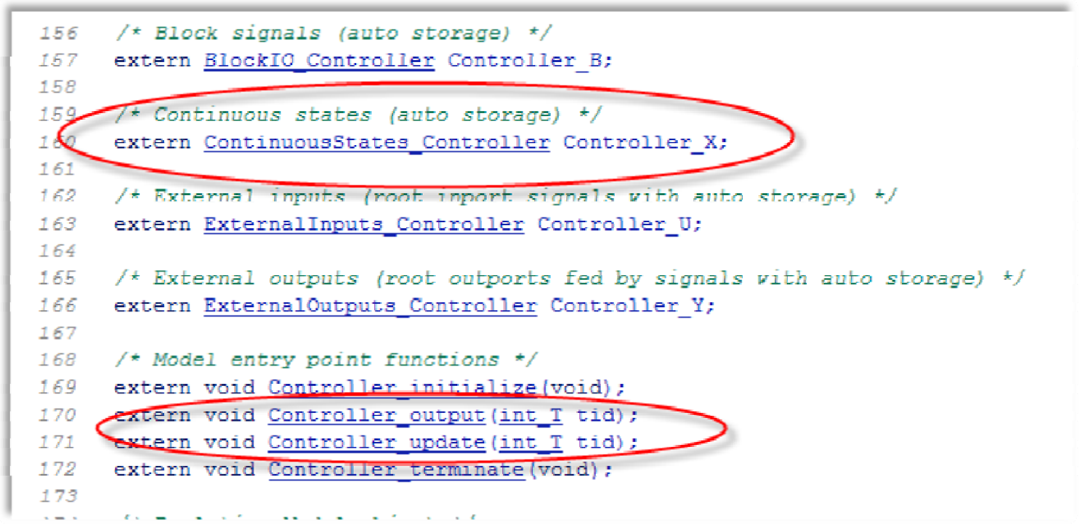
F. Shared Library vs. Source Code Export

One way to integrate generated code into a target simulation environment is to compile and link the generated source code with the target simulation environment. This technique involves configuring an Integrated Development Environment or build script with the proper settings, such as include file paths, for example. If the simulation environment is to be built on a different computer, then the files generated by Real-Time Workshop Embedded Coder must be collected and copied there. The packNGo utility can be used to automate this step.

Alternatively, the user can generate the source code and then compile and link it into a shared library (for example, a DLL or SO file). This option simplifies integration into the target simulation environment by reducing the number of files that need to be collected and copied from system to system.

G. States and Continuous Time

Models containing no states or only discrete states are the simplest case. If the model is fully discrete, no solver is needed. The discrete states can be fully contained within the generated code, or can be exposed if the target simulation environment needs access to them for analysis. If the model includes continuous states, there are three options: the model can be discretized, the solver can be generated and contained fully within the generated code, or the states can be exposed and integrated by the solver from the target simulation environment. Figure 6 shows the code generated from a Simulink model with a fixed step, continuous solver. A separate data structure holds continuous state information, while the step function has been separated into two functions. The update function computes the continuous states for the next time step, and the output function computes the output values based on the continuous state.



```
/* Block signals (auto storage) */
extern BlockIO_Controller Controller_B;

/* Continuous states (auto storage) */
extern ContinuousStates_Controller Controller_X;

/* External inputs (root inport signals with auto storage) */
extern ExternalInputs_Controller Controller_U;

/* External outputs (root outports fed by signals with auto storage) */
extern ExternalOutputs_Controller Controller_Y;

/* Model entry point functions */
extern void Controller_initialize(void);
extern void Controller_output(int_T tid);
extern void Controller_update(int_T tid);
extern void Controller_terminate(void);
```

Figure 6. States and continuous time.

IV. Conclusions

This paper presented an overview of system simulation environments and explained how history, mandate, and requirements factor into the decision to integrate multiple simulation environments--including commercially available and proprietary simulation tools. Several approaches for integrating algorithms and models from different modeling domains were discussed. Finally, a detailed discussion on integrating code generated from a COTS tool into the corporate simulation environment was presented. The capabilities and features covered were specific to Simulink and its C/C++ code generation technologies, but the requirements, such as control of function prototypes, data declaration, and tasking information, are generic and apply to any simulation software integration challenge.

Readers are encouraged to contact the authors for additional information or to share code generation and integration experiences.

References

1. MathWorks, Inc. website, www.mathworks.com
2. Melde P., Collins B., and Campbell T., "Enabling Interoperability of Native Engineering Toolsets with System Simulations and Flight Software," AIAA Modeling and Simulation Technologies Conference and Exhibit, August 15-18 2005, San Francisco, CA.
3. “Integrating Existing C Functions into Simulink Models with the Legacy Code Tool,” Simulink product documentation, MathWorks Inc., Release R2010a dated March 2010.