

# AN OBJECT-ORIENTED INTERFACE FOR SIMULINK MODELS

Michael M. Madden\*

NASA Langley Research Center  
MS 125B  
Hampton, VA 23681

## Abstract

Visual programs for modeling dynamic systems have become popular because they simplify the construction and maintenance of math models. These products also offer an appealing variety of analytical tools that aid the evaluation of model performance. With the addition of code generators, these tools can convert their visual diagrams into source code that can run standalone or be integrated within an existing software product. The simulation industry has experienced a growing trend of integrating generated code with existing, simulation frameworks. Because the simulation framework and the auto-code generator are based on independent designs, developers must add software layers that overcome incompatibilities. Ideally, the developer strives to design a reusable interface for the visual-modeling product. This frees future projects from handling auto-code integration issues. However, achieving a reusable interface may require customization of the visual modeling product and restrictive guidelines on its use.

Many simulation customers at NASA Langley Research Center have selected Mathworks' Simulink® for visual modeling. Mathworks supplies a code generator for Simulink called the Real-Time Workshop® (RTW). The simulation software group at Langley created a

reusable, object-oriented interface that integrates RTW code with the Langley Standard Real-Time Simulation in C++ (LaSRS++). LaSRS++ is an object-oriented framework for creating the real-time simulations that run in Langley's simulator facilities.<sup>1</sup>

## Introduction

The design of the LaSRS++/RTW interface has two main goals. First, LaSRS++ should treat all vehicles equally, whether or not they contain RTW code. Second, the design should avoid changes to the auto-code and to the user's process for generating the auto-code. The latter protects the interface against obsolescence with future RTW releases and minimizes additional procedures for the Simulink user. These goals result in the following requirements:

1. The interface shall support execution of auto-code in the simulation modes: RESET (initialization and scenario definition), TRIM (solve for steady state at initial condition), HOLD (freeze the simulation), and OPERATE (propagate through time).
2. The interface shall support the execution of auto-code through multiple cycles of RESET-TRIM-HOLD-OPERATE. In other words, a simulation with auto-code must be able to run several scenarios or the same scenario repeatedly without the need to shutdown and re-execute the simulation.
3. The interface shall allow multiple, heterogeneous Simulink models and multiple copies of the same model to operate in the same simulation. LaSRS++ supports multiple, heterogeneous vehicles in a simulation. Failure to support multiple, heterogeneous Simulink models limits vehicle models that contain auto-code to a single instance in a simulation. It also prevents a vehicle model from con-

---

\* Senior Member, AIAA

Copyright © 2000 by the American Institute of Aeronautics and Astronautics, Inc. No copyright is asserted in the United States under Title 17, U.S. Code. The U.S. Government has a royalty free license to exercise all rights under the copyright claimed herein for Government Purposes. All other rights are reserved by the copyright owner.

taining more than one auto-coded Simulink model. Failure to support multiple copies of the same model precludes reuse of auto-code within a simulation. For example, a user could not build an actuator model in Simulink, generate auto-code from it, and use an instance of the auto-code for each control surface.

4. The interface shall use the auto-code as generated. To add new features to the auto-code, developers must customize RTW code generation by modifying the Target Language Compiler (TLC) scripts.

Applying these basic requirements reveals a host of problems. Many are rooted in the design of RTW auto-code, which focuses on real-world execution rather than execution in a simulation. The biggest issues are:

1. RTW does not support a “trim” mode; the auto-code initializes to a defined state then operates.
2. RTW auto-code is designed to run once, i.e. in a single initialize-operate loop-terminate cycle.
3. Until version 3.0, RTW did not directly support multiple models or copies of the same model in the same program. This support is provided in the newly introduced Generic Real-Time Malloc (GRTM) Target. Thus, the LaSRS++/RTW interface will not work with prior versions of RTW.

In addition, the RTW auto-code design conflicts with two major LaSRS++ design goals:

1. Minimizing transport delay<sup>†</sup> is a design goal for LaSRS++. RTW’s design for resolving continuous states tends to increase transport delay.
2. LaSRS++ simulations are designed to run at any fixed frame rate. The frame rate for a simulation is established at runtime. RTW generates auto-code with a pre-determined frame rate that is taken from the simulation parameters defined within Simulink.

All of the above issues are overcome with a combination of interface design choices, user guidelines on Simulink model construction, modification of TLC files, and procedural workarounds.

---

<sup>†</sup> Transport delay is time between the generation of input and its visible affect on vehicle dynamics. Frequently, the term is used to refer to the time between pilot input and the response of the visual or motion system to that input.

### **Overview of the RTW code design**

RTW uses the Target Language Compiler (TLC) to generate code from a model. The code-generation rules are packaged in a set of scripts processed by the TLC. A package of TLC scripts is called a *target*. RTW ships with ready-to-use targets for a variety of runtime environments. The Generic Real-Time Malloc (GRTM) target that is required for the LaSRS++/RTW interface follows the basic design described below.

All model data is connected directly or indirectly to a central structure of type Simstruct. Access to the model data is coordinated through a Simstruct instance, of which there is only one instance per model. Because the Simstruct type is subject to change in future versions, the Simstruct instance is not intended for direct access. Instead, RTW supplies functions for manipulating the Simstruct data. In this manner, it encourages object-like interaction with the Simstruct data.

The generated code consists of three classes of functions: initialization, operation, and termination. The initialization functions include the model registration function which is named after the model and the MdlStart() function. The model registration function constructs, populates and initializes the Simstruct instance. The MdlStart() function initializes all states in the model and performs other one-time initialization tasks. The operation functions include MdlOutputs(), MdlUpdate(), and MdlDerivatives(). MdlOutputs() computes the output of each block in the model. MdlUpdate() updates discrete states. MdlDerivatives() calculates the derivatives for continuous states in each block. The termination function is called MdlTerminate(); it executes shutdown code for each block.

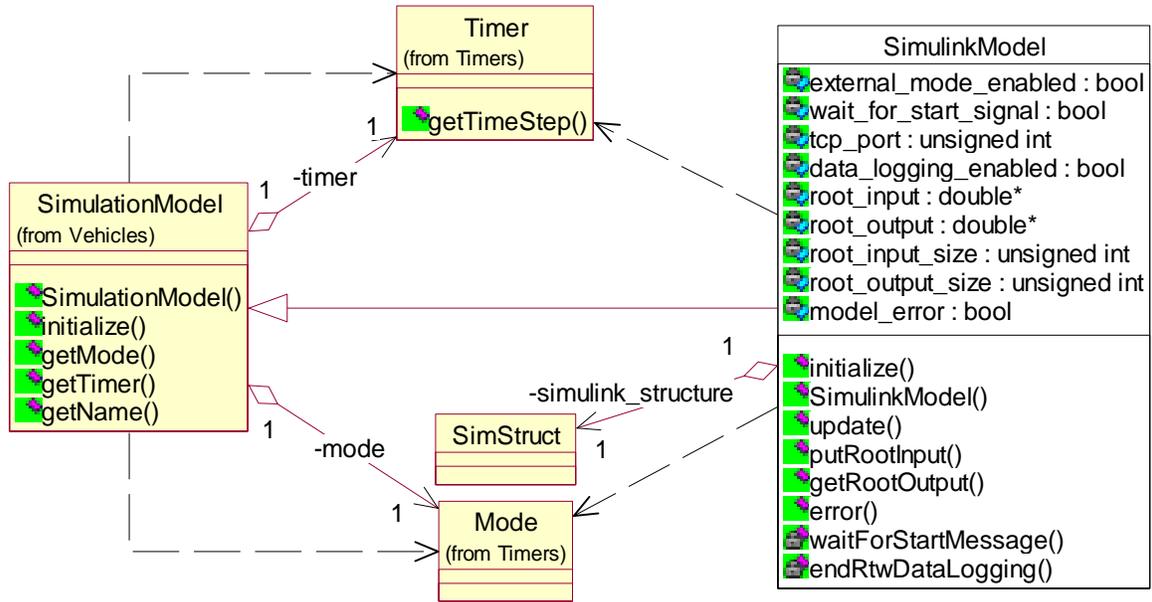
RTW includes a code library that is reusable across all models. This library provides fixed-step integration functions, a data logging function, and external mode<sup>‡</sup> interface functions for the generated code.

### **LaSRS++ Interface Design**

The LaSRS++/RTW interface is encapsulated in a single class, SimulinkModel. Figure 1 illustrates the de-

---

<sup>‡</sup> External mode is a feature that allows the Simulink block diagram to communicate with an external process executing the RTW code.<sup>6</sup>



**Figure 1 UML Class Diagram of Simulink-RTW Interface**

sign of SimulinkModel in Unified Modeling Language (UML) notation.<sup>2</sup> Figure 2 displays a simplified representation of the SimulinkModel header file. SimulinkModel is a C++ class wrapper for the auto-code. The class encapsulates the details of operating the auto-code. Direct access to the Simstruct instance and RTW functions is not provided. Instead, the class replaces the complexity of operating the auto-code with a very simple interface.

SimulinkModel presents the auto-code as a system with single-dimensional input and output vectors. Inputs are defined using the putRootInput() method. Outputs are extracted using the getRootOutput() method. Both methods use an integer index argument to identify the data being manipulated. The SimulinkModel defines just four behaviors for the model: construction, destruction, initialize(), and update(). The constructor builds and initializes the Simstruct instance using the model's registration function. The constructor also performs one-time initializations on the model. The registration function is an argument to the constructor. Each SimulinkModel object can represent a unique Simulink model. The registration function passed to the constructor determines which model a given Simulink-

Model object will contain. The initialize() method resets the model to an elapsed time of zero. The simulation calls initialize() in RESET mode. The update() method operates the model; as will be discussed later, it is called in both TRIM and OPERATE modes of the simulation. The destructor frees all of the memory used by the auto-code and terminates its operation. SimulinkModel also provides an error monitoring method, error(). This method returns true if the auto-code reports a non-fatal error since the last time error() was called. SimulinkModel handles all error reporting. The client code decides whether it will continue past a non-fatal error.

SimulinkModel derives from SimulationModel. SimulationModel is the base class for all math models in LaSRS++. Thus, LaSRS++ treats the SimulinkModel as a specialized math model. SimulinkModel must support the basic behaviors of all SimulationModels. SimulationModels contain references to a Timer, which encapsulates the time step for the SimulationModel, and to a Mode, which contains the current simulation mode. SimulationModels must behave appropriately based on the information in Timer and Mode.

```

#include "SimulationModel.hpp"

extern "C" {
#if !defined(RT) // Mathwork's simstruc.h requires that RT be defined for RTW code.
#define RT
#endif
#include "simstruc.h" // SimStruct cannot be forward declared; it is a typedef.
typedef SimStruct* (*SimulinkModelConstructor)(void); // Model registration function.
}

class SimulinkModel: public SimulationModel {
public:
    SimulinkModel(const Mode&          simulation_mode,          // Simulation mode.
                  const Timer&        simulation_timer,        // Time step for model.
                  SimulinkModelConstructor model_constructor,  // Registration function.
                  bool                 use_external_mode,      // Enable external mode.
                  unsigned int         external_mode_tcp_port, // Port # for external mode.
                  bool                 wait_for_start_message, // External mode option.
                  bool                 data_logging);          // Enable data logging.

    virtual ~SimulinkModel();
    virtual void initialize(); // Redefine SimulationModel::initialize().
    virtual void update();    // Operate the Simulink model.
    void putRootInput(double input, unsigned int index) {root_input[index] = input; };
    double getRootOutput(unsigned int index) const      {return root_output[index]; };
    bool error() const {
        bool return_error = model_error; model_error = false; return return_error; };

private:
    SimStruct*   simulink_structure; // Model data and pointers to model functions.
    mutable bool model_error;       // Set to true when model reports an error.

    // External mode and data logging option variables.
    bool     external_mode_enabled; // Enables external mode if true.
    bool     wait_for_start_signal; // Sim pauses for simulink start message.
    unsigned int tcp_port;         // TCP port number of external mode connection.
    bool     data_logging_enabled; // Enables RTW data logging if true.

    // The simulink model I/O is represented as single dimensional arrays.
    double* root_input;
    double* root_output;
    unsigned int root_input_size;
    unsigned int root_output_size;
};

```

**Figure 2 Simplified Header File for SimulinkModel**

Implementation of the SimulinkModel class borrows heavily from the `grt_malloc_main.c` file that ships with RTW<sup>§</sup>. As a result, SimulinkModel can call RTW library functions for data logging and external mode<sup>¶</sup>. Both features are optional. Arguments to the SimulinkModel constructor enable and configure these options. RTW data logging can be enabled for more than one model in the same program. Only one SimulinkModel object can enable external mode because the external mode code uses global data.

RTW data logging allows the customer to select, through the Simulink product, variables internal to the

Simulink model for recording. However, since RTW code is designed to run in a single initialize-operate-terminate cycle, the RTW data logging is only guaranteed to function for the first run of the simulation. The data logging functions may return errors on subsequent runs. These errors are not fatal and do not affect the operation of the model, but the resulting data log may corrupt. The presence of ToWorkspace blocks in a Simulink model require that data logging be turned on because RTW adds their input signals to the data log. Disabling data logging for a model with ToWorkspace blocks causes a segmentation violation during operation. By default, RTW data logging bases its memory requirements on the start and end times defined for the model. The SimulinkModel modifies the Simstruct so

<sup>§</sup> RTW supplies this file for creating a standalone executable from the GRTM generated code.

<sup>¶</sup> External mode operation has not been tested to date.

that it will run for infinite time<sup>#</sup>. Obviously, RTW cannot allocate memory for an infinite-time run. RTW will issue a warning and use a default buffer size of 1024. To remove the warning and use a different buffer size, the Simulink model developer must set **Simulation -> Parameters -> Workspace I/O -> "limit rows to last"**, prior to generating code.

Though it significantly reduces the complexity of adding RTW code to LaSRS++, SimulinkModel is not a complete solution. The SimulinkModel class does not know how the indices of its input and output vectors map to other variables in the LaSRS++ simulation. The software engineer must still create client code that directs input data from the LaSRS++ simulation into SimulinkModel and directs output data from SimulinkModel into the simulation. The client code must also monitor SimulinkModel for non-fatal errors and trigger appropriate actions. The class must also exercise the update() method during the appropriate LaSRS++ modes as discussed in the section "Simulation Mode Support".

### Solving Interface Issues

The simplicity of the LaSRS++/RTW interface design masks the implementation choices and procedural workarounds necessary to meet the requirements stated in the introduction. This section details these solutions.

#### Multiple Simulink Model Support

Only one target supports multiple Simulink models in the same program, the Generic Real-Time Malloc (GRTM) target. The LaSRS++/RTW interface will operate only with auto-code generated by the GRTM target. GRTM first appeared in RTW 3.0. Thus, the LaSRS++/RTW interface requires RTW 3.0 or later. Furthermore, the LaSRS++/RTW interface only works with code generated for single-task operation.

GRTM differs from the other pre-packaged targets in two fundamental ways to enable multi-model support, as illustrated by comparison against its older sibling, the generic real-time (GRT) target. First, GRT statically

---

<sup>#</sup> This is accomplished by calling `ssSetTFinal()` to set the final time to zero. Simulink interprets a final time of zero as equivalent to an infinite final time.

allocates all persistent data with a mixture of internal and external linkage; moreover, GRT uses the same variable names for the major data structures in all auto-code that it generates. This prevents multiple Simulink models from running in a program. For each common name with external linkage, the linker would assign the same memory location. All Simulink models in a program would then share the same memory locations, leading to data corruption. On the other hand, GRTM dynamically allocates all data structures and effectively binds them to a Simstruct pointer returned by the model's registration function. Thus, each instance of a model receives its own copy of persistent data, and the generated code contains no common variable name with external linkage.

Second, GRT uses the same names for functions that operate the model<sup>\*\*</sup> and gives these functions external linkage. Thus, distinct Simulink models cannot exist in the same program because the function names would conflict at link time. GRTM avoids this conflict by declaring the functions with internal linkage. Only the registration function has external linkage and its name is the same as the model. Thus, all distinct models must have unique names to coexist in the same program. To make the operation functions available externally, GRTM places function pointers in the Simstruct structure and assigns the addresses of the generated functions to the pointers. The functions for a Simstruct instance are executed by exercising the function pointers that it contains; i.e., each Simstruct structure is bound to the functions that act on it. This technique is similar to the virtual function mechanism in C++.<sup>3</sup>

Unfortunately, the integration algorithm is not among the functions stored as a pointer in the Simstruct instance. Thus, all models in a program must use the same integration algorithm. In fact, the Simulink user's selection of integration algorithm in the Simulation Parameters dialog has no meaning to the LaSRS++/RTW interface. The setting does not influence the generated code. It only influences the template makefile that RTW generates with the code; it adds the file containing the selected algorithm to the list of files to be compiled

---

<sup>\*\*</sup> These functions are `MdlStart()`, `MdlOutput()`, `MdlUpdate()`, `MdlDerivatives()`, and `MdlTerminate()`.

and linked. LaSRS++ simulations use their own makefiles and ignore the generated makefile. The LaSRS++ makefiles allow the program creator to select any of the RTW integration algorithms at build time. The simulation will apply the algorithm, with which it is linked.

The library code shipped with RTW is reused across many targets. This code uses pre-processor variables to customize itself for a specific target. The variable `RT` must be defined when compiling the code for all RTW targets. The variable `RT_MALLOC` must be defined for the GRTM target. Thus, the LaSRS++/RTW interface and all RTW library code must be compiled with the variables `RT` and `RT_MALLOC` defined. In addition, on UNIX systems, the variable `UNIX` must be defined. Unfortunately, the RTW library code relies too heavily on pre-processor variables to also define the number of continuous states in a model (`NCSTATES`) and the number of sample times in a model (`NUMST`). Obviously, code utilizing these pre-processor variables will be compiled with only one value for each variable and those values will be applicable to only one model. All other models would not function properly, defeating the ability to have multiple models in the same program. However, the `Simstruct` structure contains the data represented by these variables; and functions<sup>††</sup> are supplied to extract this data. `NCSTATES` can be replaced with a call to `ssGetNumContStates()`. The function `ssGetNumSampleTime()` can be substituted for `NUMST`. By examination of the RTW library source, it is apparent that some attempt had been made to remove these pre-processor variables from code exposed when `RT_MALLOC` is defined; however, the task was left incomplete. The integration algorithms still contain a reference to `NCSTATES`. To repair this defect, a copy of the code was made for the LaSRS++/RTW interface and `NCSTATES` was replaced with `ssGetNumContStates()`. Likewise, the code for external mode references `NUMST`. Once again, a copy of the code was made; and `NUMST` was replaced with a call to `ssGetNumSampleTime()`<sup>‡‡</sup>. Also, `grt_malloc_main.c`

---

<sup>††</sup> Most `Simstruct` functions are actually implemented as pre-processor macros.

<sup>‡‡</sup> A large number of function signatures in the external mode code also had to be modified to make the switch.

uses `NCSTATES` and `NUMST` heavily. When sections of this code were copied into the methods of `Simulink-Model`, `NCSTATES` and `NUMST` were replaced with their functional equivalents.

#### Transport Delay

The LaSRS++ simulation framework minimizes transport delay as a design goal. As one means of achieving this goal, LaSRS++ simulations resolve continuous states (i.e., integrates derivatives) as they occur in the execution path, and LaSRS++ propagates the results immediately. `Simulink`, on the other hand, collects the computed derivatives and simultaneously integrates the derivatives as the final step of operation. At best, the output of each integrator will not affect the behavior of the vehicle until the following frame. If several states are chained together in the execution path, the inputs influencing the first derivative may not affect vehicle behavior for several frames. For single-pass integration algorithms, the input's influence on the model's output will be delayed one frame for each state on the chain. Simulations commonly use single-pass integration algorithms because they are computationally efficient; they are susceptible to this delay. Systems that use a large number of transfer functions are also vulnerable to this problem. Mathworks' design is correct for a continuous model. But, its discrete representation introduces an artifact in the form of transport delay. At the frame rates normally run for a simulation, the resulting delay can lead to vehicle response that the pilot finds unrealistic or that causes the pilot to adapt differently than in the actual aircraft.

The solutions are limited. One can write a new TLC that reflects the modeling philosophy of LaSRS++, but this solution requires a large resource commitment. The next two solutions require additional computation, which the computer may not be able to support. One could use a multi-pass integration algorithm. For a model with  $n$  states in its longest chain, the algorithm must run  $n$  passes. The input will now influence the last state in the chain with a single integration. But, the results are still delayed one frame. Also, a multi-pass algorithm mixes passes where the input does not influence the last state with passes where the input is influential. Passes where the input is influential remain until the  $n^{\text{th}}$  frame after introduction. Thus, the input's total

influence is spread over  $n$  frames. Alternatively, one can run the RTW code at a higher rate; ideally, the rate is the product of the simulation rate and  $n$ . This approach resolves the total influence of the input in one simulation frame. Depending on the purpose of the Simulink model, the vehicle plant<sup>§§</sup> may have to run at the higher rate to maintain realistic behavior. Thus, this approach is frequently more computationally expensive than the multi-pass integration algorithm. The processing power of the host computer may prevent the RTW code from running at the “ideal” rate, forcing the project to experiment with different simulation rates to balance computing resources and vehicle response.

Using discrete blocks instead of continuous blocks may not solve the problem. Like continuous states, discrete states are not updated by calling `MdlUpdate()` until after the block outputs are calculated by calling `MdlOutput()`. Some algorithms, like the Euler algorithms in the discrete integrator, produce pure delays. However, some discrete algorithms are spread over both the `MdlUpdate()` and `MdlOutput()` functions to take advantage of currently computed inputs, eliminating the delay. But, as discussed in the section “Simulation Mode Support”, some of these blocks may not behave correctly for TRIM mode and cannot be used. Thus, the Simulink user must be aware of the limitations of each discrete block to effectively use them for simulation.

#### Variable Time Step

LaSRS++ allows the user to define the time step during simulation start-up. The `Timer` member in the `SimulationModel` base class communicates the simulation time step (or a multiple or integral divisor of the simulation time step for multi-rate simulations) to the math model. Hand-coded models are designed for a variable time-step. It is necessary for components that are reused with a variety of simulations. However, the default RTW code generator does not provide an interface for changing the time step of the auto-code. When the code is generated, RTW imposes the time step defined for the model in Simulink’s “Simulation Parameters” dialog.

---

<sup>§§</sup> The vehicle plant models the physical aspects of the vehicle. Usually, it consists of the aerodynamics model, the engine model, the gear model, and the control surface models.

The TLC could be modified to provide a means of changing the time step. But, the need does not yet justify the effort at LaRC since a workaround exists. When a new time step is required, the RTW auto-code is regenerated with the new time step, compiled, and linked into the simulation. Although the simulation projects at LaRC use a variety of time steps, an individual simulation project tends to settle on one time-step. Thus, the workaround, though laborious, is infrequently used. To protect against accidental mismatch of time steps in the auto-code and LaSRS++ simulation, the `SimulinkModel` class compares the time step that it inherits from `SimulationModel` with the time step stored in the auto-code. If the time steps differ, `SimulinkModel` will issue a warning to the user.

#### Simulation Mode Support

RTW does not support the simulation modes in LaSRS++. To operate within a LaSRS++ simulation, the auto-code must behave properly in the four major modes RESET, TRIM, HOLD, and OPERATE. Furthermore, the auto-code must behave properly through multiple runs (i.e., cycles of RESET-TRIM-HOLD-OPERATE) without the need to shutdown the simulation and restart it.

In RESET, the simulation resets the elapsed time to zero and defines the scenario for the next run. During RESET, the simulation calls the `initialize()` method of all `SimulationModel` objects. `SimulinkModel`, which derives from `SimulationModel`, resets the elapsed time in the auto-code by calling `sfcnInitializeSampleTimes()` in its `initialize()` method. Any signals in the model that require initialization must be connected to a root-level “inport” block. The `SimulinkModel` client is responsible for passing the initial value to the `SimulinkModel`.

In aircraft simulation, it is undesirable to have transient behavior in the first few seconds of operation; in fact, such transients can prevent safe use of a motion system. The aircraft simulation must be initialized to a steady state before operating. The simulation contains algorithms that solve for the steady-state values of the simulation states. In LaSRS++, these algorithms are exercised in TRIM mode. LaSRS++ uses a simple trim algorithm of the form:<sup>4</sup>

$$x_{n+1}^i = x_n^i + g_n^i y_n^i \quad (i = 1, m)$$

where  $x$  is an independent state,  $g$  is a gain,  $y$  is a dependent value, and  $m$  is the number of equations required to trim the aircraft for the given scenario. The equation calculates a new state with the purpose of reducing the dependent values to zero; thus, the dependent values are frequently referred to as the error. The gains set the proper direction and scale to make the algorithm work. The algorithm operates in a loop. For each pass, the simulation operates the math models with the new states; then, the algorithm updates the states. The process repeats until the magnitude of the error vector becomes smaller than a defined tolerance.

To successfully and efficiently “trim” in LaSRS++, all math models modify their behavior in two fundamental ways. First, transfer functions calculate the steady-state output for their input in one pass<sup>¶</sup> and initialize their states to the steady-state condition. Second, integrators do not integrate. Their output is set to a value, possibly controlled by the trim algorithm. In many cases, the integrator’s input (i.e., the derivative) must be reduced to zero; i.e., the derivative becomes an error value in the trim algorithm. It is not unusual for the derivative’s corresponding trim state (i.e. the state with the greatest influence on its value) to be the integrator output.

The RTW auto-code has no feature similar to the TRIM mode in LaSRS++. TRIM behaviors must be designed into the Simulink model. The Simulink user must assign a root inport block to be the TRIM flag.<sup>5</sup> Currently, the SimulinkModel design does not force the first inport block to be the TRIM flag. Since SimulinkModel cannot know which signal is the TRIM flag, the LaSRS++ client code must set the flag appropriately. When the flag is high, the Simulink model will activate TRIM mode behaviors; otherwise, the model will behave normally.

All integrators in the model must be reset integrators.<sup>5</sup> If the integrator output is a state in the trim algorithm, then the initial condition signal must be connected to a root-level inport block. If the derivative is an error in the trim calculation, the input signal to the integrator

---

<sup>¶</sup> In other words, the transfer function must supply the output that results if the input is applied for an infinite amount of time.

must be connected to a root output block. The reset switch must be connected to the TRIM signal. The reset integrator will work as expected if it outputs the reset input when the TRIM signal is high. This was the reset integrator’s behavior in the Simulink 1.0. However, Mathwork’s decided to change the behavior starting with Simulink 2.0. Now, reset integrators use the reset input when the reset switch changes value. The TLC for the GRTM target was changed to restore the original behavior of the reset integrator.

Transfer function blocks (discrete or continuous) cannot be used; the blocks have no ability to reset themselves or compute their steady-state output in one pass.<sup>5</sup> Instead, transfer functions must be re-implemented as a system block containing reset integrators, and they must be designed to return the steady state output when the elapsed time is zero. The research community at LaRC has created a library of these transfer function replacements. Other behavior modifications for TRIM mode must be linked to a switch whose decision input is the TRIM signal.

In TRIM, the client code is responsible for communicating states and errors to the trim algorithm. The client code also retrieves newly calculated states from the trim algorithm and communicates them to the Simulink-Model object. The client sets the TRIM input high and calls SimulinkModel::update() to compute new error values from the new states. SimulinkModel::update() has logic that prevents integration, incrementing time, and data logging while not in OPERATE mode. Integration and incrementing time would interfere with the TRIM algorithm. Data logging is undesirable in TRIM; the purpose of data logging is to record the model’s behavior during operation. SimulinkModel provides these behavior modifications for TRIM mode without the need for intervention from the client code.

In HOLD, the simulation does not update its states; and the simulation time does not increment. The vehicle is frozen. This is best achieved by not exercising the math models. However, there are special cases where a math model needs to run in HOLD. Thus, SimulinkModel allows its update() method to be called in HOLD mode. However, as stated previously, SimulinkModel will not perform integrations, increment time for the Simulink

model, or log data except in OPERATE mode. These operations would violate the intent of HOLD mode.

In OPERATE, the client communicates inputs to the Simulink model by calling `putRootInput()`. Then, it calls `update()` for the `SimulinkModel` object. When `update` returns, the client extracts the `SimulinkModel`'s outputs by calling `getRootOutput()` and communicates the outputs to the simulation. Optionally, the client will also call `error()` and take appropriate steps if an error is reported. The Simulink model requires no special considerations to behave properly in OPERATE; it is designed to operate.

The root issue that prevents RTW auto-code from performing correctly through multiple runs is that RTW does not provide an interface for resetting the model once it begins operation. However, the changes that enable correct behavior in RESET and TRIM mode also establish this missing interface. A Simulink model built using the guidelines above will safely restart without the need to completely terminate it and recreate it.

### Example

This section provides an example of adding the `SimulinkModel` class to a LaSRS++ simulation. A customer has built a Simulink model called `simple_control_law` illustrated in Figure 3. This model contains an integrator in the longitudinal law that must be added to the trim algorithm. The derivative is the error and its output is the derivative's companion state. As required for the LaSRS++/RTW interface, the integrator's reset value is attached to inport block #2 and the derivative is exported through outport block #4. Also, a trim flag is installed as inport block #1. Though not visible, all transfer functions have been re-implemented with reset integrators. To illustrate how the root-level "port" blocks map to `SimulinkModel`'s I/O vectors, the "Accels\_g" block has a length of three.

### Code Generation

To generate code, the Simulink user selects the Generic Real-Time Target and configures it for "code generation only" as specified in the *Realtime Workshop User's*

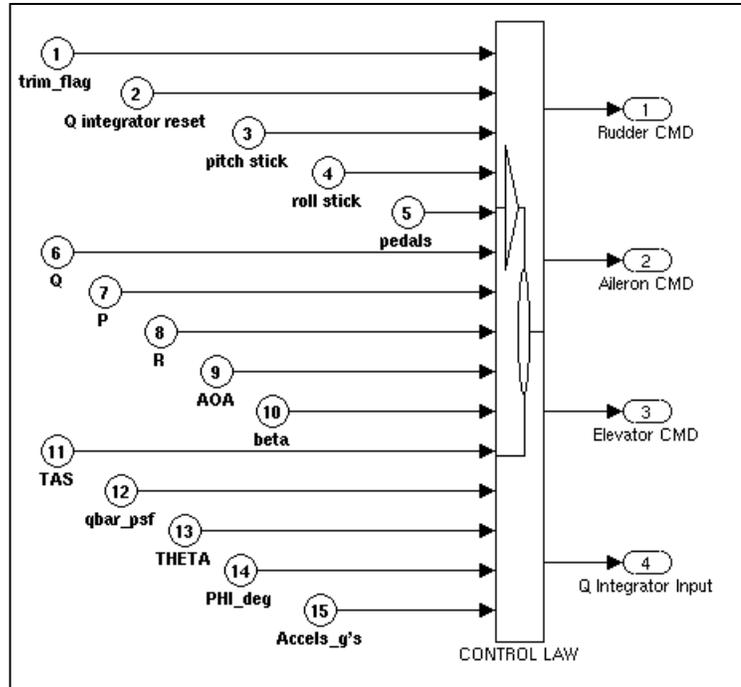


Figure 3 Example Simulink Model

*Guide*.<sup>6</sup> In this example, the user does not identify data for RTW data logging and the model does not contain `ToWorkspace` blocks. Thus, the user does not need to change settings for data logging. The user launches the Simulation Parameters dialog (**Simulation->Parameters->Solver**) and sets the solver to fixed-step and single-tasking mode. The user also sets the step size to the simulation step size. The user then generates the code by selecting **Tools->RTW Build**.

### Client Code

Figure 4 and 5 provides the client code that integrates the model into the LaSRS++ simulation<sup>##</sup>. In this example, the client code is a class derived from the `SimulinkModel` class. Figure 4 shows the header file. Enumerations are used to name the indices into `SimulinkModel`'s I/O vectors. The enumeration simplifies code changes if later revisions of the model reorder the root-level "port" blocks; only the value of the enumeration must be changed to correct the client code. The index for `putRootInput()` starts with zero, following C++ con-

<sup>##</sup> For brevity, the code shown is simplified for the discussion. It does not accurately reflect the code, as it would appear in a LaSRS++ simulation. But it does represent the level of work required for the client code.

```

#include "SimulinkModel.hpp"

class Aircraft; // Base class for all aircraft models.

class ExampleModel: public SimulinkModel {
public:
    enum InputMap {TRIM_FLAG = 0, QI_RESET= 1, PITCH_STICK= 2,
                  ROLL_STICK= 3, PEDALS = 4, PITCH_RATE = 5,
                  ROLL_RATE = 6, YAW_RATE= 7, AOA = 8,
                  BETA = 9, TAS = 10, QBAR = 11,
                  THETA = 12, PHI = 13, ACCEL_X = 14,
                  ACCEL_Y = 15, ACCEL_Z = 16};
    enum OutputMap {RUDDER_CMD = 0, AILERON_CMD= 1,
                   ELEVATOR_CMD= 2, QI_INPUT = 3}

    ExampleModel(Aircraft* parent_aircraft);
    virtual void update();
    double getRudderCmd() const {getRootOutput(RUDDER_CMD)};
    double getAileronCmd() const {getRootOutput(AILERON_CMD)};
    double getElevatorCmd() const {getRootOutput(ELEVATOR_CMD)};
    void putPitchStick(double x) {putRootInput(PITCH_STICK,x)};
    void putRollStick(double x) {putRootInput(ROLL_STICK,x)};
    void putPedals(double x) {putRootInput(PEDALS,x)};

private:
    Aircraft* aircraft; // Pointer to parent aircraft.
    double qi_reset; // Reset value for pitch law integrator.
}

```

**Figure 4 Header for ExampleModel**

vention; it does not start with one which is Simulink convention. The last three elements of the input vector are the three acceleration components represented by the “Accels\_g” block. Thus the example illustrates that the indices of the SimulinkModel vectors do not normally match the root-level port numbers in the Simulink model but the order is maintained. The class provides mutator methods for the pilot commands that directly inject the values into SimulinkModel’s input vector. Likewise, the class includes accessor methods for the control surface commands that directly extract data from SimulinkModel’s output vector.

Figure 5 shows the body file for ExampleModel class. The body file includes C-style external declaration of the model registration function, `simple_control_law`. The ExampleModel constructor passes the registration function to the constructor for SimulinkModel. It also configures SimulinkModel to obtain its Mode and Timer information from the parent aircraft. It disables external mode and data logging. ExampleModel redefines the `update()` behavior that it inherits from SimulinkModel. `ExampleModel::update()` sets the TRIM

flag appropriately. It also injects the currently computed trim value for the pitch law integrator. It retrieves data from the parent aircraft and copies it to SimulinkModel’s input vector. For brevity, the code does not show the complete list of `putRootInput()` calls. Then, the method calls `SimulinkModel::update()` to run the RTW code. If in TRIM mode, the method calculates a new trim value for the pitch law integrator; the derivative input to the integrator is the error in the algorithm. The method concludes by checking whether the auto-code reported any errors and terminates the simulation if any were found. The only step now remaining is to add the ExampleModel code and RTW auto-code to the LaSRS++ project

makefile and build the simulation.

### Conclusions

The resultant LaSRS++/RTW integration is simple to reuse. The basic interface consists of a single class called SimulinkModel. SimulinkModel reduces the auto-code to a system with one input vector and one output vector. SimulinkModel has only four behaviors: construction, initialization, update, and destruction. The registration function that is passed as a constructor argument determines the Simulink model that the SimulinkModel object encapsulates. The SimulinkModel class also provides limited support for RTW’s data logging and external mode features. The design was realized with minor modification to RTW; in fact, only one TLC modification was made to force reset integrators to use the reset input when the reset switch was set to high. Also, the RTW library source had to be modified to accommodate multiple Simulink models in a simulation. Establishing guidelines for Simulink model construction and auto-code generation solved most of the integration issues. The SimulinkModel class will work with any Simulink model that follows the guidelines. The Simu-

```

#include "ExampleModel.hpp"
#include "Aircraft.hpp"

extern "C" { // Registration function prototype.
    extern SimStruct* simple_control_law(void);
}

ExampleModel::ExampleModel(Aircraft* parent_aircraft):
    SimulinkModel(parent_aircraft->getMode(),
                  parent_aircraft->getTimer(),
                  simple_control_law, false, 0, false, false),
    aircraft(parent_aircraft),
    qi_reset(0.0)
{}

void ExampleModel::update() {
    // For brevity, not all putRootInput calls are shown.
    putRootInput(TRIM_FLAG , getMode() == TRIM);
    putRootInput(QI_RESET  , qi_reset);
    putRootInput(PITCH_RATE, aircraft->getQ());
    putRootInput(AOA       , aircraft->getAlpha());

    SimulinkModel::update(); // Executes the model

    // Trim algorithm for illustration.
    if (getMode() == TRIM) {
        qi_reset += -0.1 * getRootOutput(QI_INPUT);
    } else {
        qi_reset = 0.0;
    }
    if (error()) exit(-1);
}

```

**Figure 5 ExampleModel Body File**

link user must generate auto-code using the “Generic Real-Time Malloc” target but no special steps were added to the code generation process. The simulation developer writes only a small amount of code to integrate the SimulationModel object. This code sets construction arguments, calls operations at the appropriate time, populates the input vector, extracts data from the output vector, and performs error checking.

### **Bibliography**

- <sup>1</sup> Leslie, R.; Geyer, D.; Cunningham, K.; Madden, M.; Kenney, P.; Glaab, P. *LaSRS++: An Object-Oriented Framework for Real-Time Simulation of Aircraft*. AIAA-98-4529, Modeling and Simulation Technology Conference, Boston, MA, August 1998.
- <sup>2</sup> Quatrani, Terry. *Visual Modeling With Rational Rose and UML*. Addison-Wesley. Reading, MA, 1998. ISBN 0-201-31016-3.

<sup>3</sup> Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, MA, 1997. ISBN 0-201-88954-4.

<sup>4</sup> Neuhaus, J. *An Object-Oriented Design For Trim*. AIAA-2000-4388, Modeling and Simulation Technology Conference, Denver, CO, August 2000.

<sup>5</sup> Hunt, G. *Real-time Constraints for Simulink 2*. Internal memo, NASA LaRC, March 1999.

<sup>6</sup> *Real-Time Workshop® User's Guide*, The Mathworks, Inc., January 1999.

<sup>7</sup> Booch, Grady. *Object-Oriented Analysis and Design With Applications*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1994. ISBN 0-8053-5340-2.

<sup>8</sup> Dellinger, W.; Salada, M.; Shapiro, H. *Application of Matlab®/Simulink® to Guidance and Control Flight-Code Design*. AAS 99-062, Proceedings of the 22<sup>nd</sup> Annual AAS Rocky Mountain Guidance and Control Conference, Breckenridge, CO, February 1999.