https://ntrs.nasa.gov/search.jsp?R=20040110806 2018-02-26T02:44:32+00:00Z

Architecting a Simulation Framework for Model Rehosting

Michael M. Madden[*]
NASA Langley Research Center, Hampton, VA, 23681

The utility of vehicle math models extends beyond human-in-the-loop simulation. It is desirable to deploy a given model across a multitude of applications that target design, analysis, and research. However, the vehicle model alone represents an incomplete simulation. One must also replicate the environment models (e.g., atmosphere, gravity, terrain) to achieve identical vehicle behavior across all applications. Environment models are increasing in complexity and represent a substantial investment to re-engineer for a new application. A software component that can be rehosted in each application is one solution to the deployment problem. The component must encapsulate both the vehicle and environment models. The component must have a well-defined interface that abstracts the bulk of the logic to operate the models. This paper examines the characteristics of a rehostable modeling component from the perspective of a human-in-the-loop simulation framework. The Langley Standard Real-Time Simulation in C++ (LaSRS++) is used as an example. LaSRS++ was recently redesigned to transform its modeling package into a rehostable component.

Acronyms

DAVE	=	Dynamic Aerospace Vehicle model Exchange
EOM	=	equations of motion
FSSB	=	Flight Simulation and Software Branch
IDL	=	intermediate description language
KLOC	=	one thousand lines of code
LaSRS++	=	Langley Standard Real-Time Simulation in C++
XML	=	Extensible Markup Language

I. Introduction

PROPRIETARY simulation architectures remain the norm in the simulation community.[1] Exchange of dynamic models among different organizations continues to take the form of partial source code, data files, documentation, and limited check cases. The source code in a delivery may not be reusable. The source code may be coupled to other source files that were not delivered or the source code may not be portable. The effort to reuse the source code in the acquirer's[†] simulation architecture may be greater than re-engineering the model. Thus, re-engineering models is common. The Flight Simulation and Software Branch (FSSB) at NASA Langley Research Center (LaRC) has experienced the following problems with re-engineering and revalidating dynamics models from another source.

- The supplier may not have kept documentation current with their implementation, and documentation sometimes contains errors. Both problems lead to undue debugging activity required to convince the acquirer that their implementation matches the documentation even though it does not match delivered check cases. The debugging activity is also necessary for the acquirer to formulate the correct questions to ask the supplier in order to resolve the discrepancy, if support from the supplier is available.
- Check cases may have incomplete scenario descriptions or incomplete results. Incomplete check cases cause additional work if support from the supplier is not available. The developer must reverse engineer the missing scenario data from the results. When results are incomplete, the developer is left with insufficient

[*] Aerospace Technologist, Flight Simulation and Software Branch, MS 125B. Senior Member AIAA.
[†] The organization that provides the dynamics model will be called the "supplier". The organization receiving the dynamics model will be the "acquirer".

data to narrow the source of check case discrepancies; the developer must inspect a broad set of source code, possibly the whole model, for the defect.

- Check cases do not sufficiently cover the model code. The check cases do not exercise all of the code paths that one could expect to traverse during a normal flight (i.e., all of the nominal cases are not covered). Check cases rarely cover exceptional scenarios (i.e. scenarios at or beyond the flight envelope enclosed by aerodynamic and engine tables). The acquirer has no information to independently validate remaining code paths.
- The delivery usually comprises only the dynamic model of the vehicle. It may not include other models or algorithms that can affect check case results such as the equations of motion (EOM), trim algorithms, integration techniques (for derivatives), the atmosphere model, and the world model (gravity, shape, and rotation). These missing items add a perception of uncertainty to the supplier check case results that can mask defects in the acquirer implementation when discrepancies in check case results are small.

Even when support from the supplier is available, there can be a significant delay in obtaining an answer that solves an issue; and the communication expends effort for both organizations.

Organizations now face similar issues internally as the demand to deploy simulation models in design, analysis, and research applications grow. These applications can represent architectures that differ significantly from the simulation architecture. The organization is once again faced with the choice of reuse or re-engineering. However, the organization now has a greater economic incentive to favor reuse. Re-engineering and revalidating a dynamic model for each deployment represents a duplication of effort within the organization. That duplication extends into maintenance. Each team performs a duplicate effort to implement model changes and validate them. Re-engineering also introduces overhead in the form of additional communication. The supplier must provide technical support for acquirers. The supplier must mentor acquirers on the implementation details of the dynamics model and portions of the simulation architecture. The supplier must establish processes that ensure changes to the dynamics model are communicated to all acquirers. Identical behavior also gains greater importance for internal use. Even small differences in the behavior of the dynamics model between simulation, design, analysis, and research applications could lead projects in the wrong direction and may not be caught until a failure occurs. When a difference in behavior is found, the supplier may be asked to generate additional test cases to aid acquirers in debugging.

An organization can choose between two approaches to reuse, an intermediate description language (IDL) or source-code reuse. The IDL defines the dynamic model using high-level, code-neutral constructs. For example, an IDL file might describe the model as a hierarchy of parts with each part described using equations, data tables, and relationships (e.g. source of inputs, order of execution). A code translator generates code from the IDL for the target application. Customized code generation is the main advantage of IDLs. A well-constructed code translator will produce code that compiles and links into the target application without any manual code modifications. Moreover, the code can take full advantage of the target application's services. The main disadvantage to an IDL is the lack of mature standards and tools. In addition to the code translators, tools for creating and maintaining the IDL source are important. For effective reuse using IDLs, developers must make future model changes to the IDL source and regenerate the code. Because of the lack of standards and tools, the organization must invent the IDL and invest resources to develop tools, translators, documentation, and training. A long-term commitment to the IDL is required to see a return on this initial investment and its recurring maintenance costs. Recent developments aim to ease the adoption of IDLs. Jackson and Hildreth have proposed a standard IDL for dynamic models that is based on Extensible Markup Language (XML); this IDL is called Digital Aerospace Vehicle model Exchange (DAVE).[1] Graphical modeling tools have also matured to the point where they could double as IDLs and reduce start-up costs. For example, an organization could use Mathworks' Simulink®[‡] as an IDL; however, the organization may still need to invest resources in creating custom Simulink® blocks and custom Real-Time Workshop® code generators for their target applications.

Source-code reuse is the other approach an organization can choose. This paper only considers verbatim reuse (i.e. reuse without modifying the source).[2] Though source-code reuse does not require invention of a new language or creation of new tools, source-code reuse does have its own initial costs. Source code must be designed for reuse. A new reuse program may require extensive redesign of the existing simulation architecture; it can be more cost effective to design a new simulation architecture from scratch and abandon the old architecture. However, source-code reuse programs do not require the same level of investment as IDLs in developing new tools and training. The challenge for source-code reuse programs is producing a design that accommodates all of the target applications. The organization may have to compromise. The reusable design may provide the minimum useful interface for all

[‡] Visit http://www.mathworks.com for more information. Simulink® and Real-Time Workshop® are registered trademarks of Mathworks.

target applications. However, the reusable code may seamlessly integrate with a subset of the applications. Developers for the remaining applications will have to build an interface and translation layer to hook the reusable code into the target application and its services.

This paper examines the source-code reuse approach with an emphasis on rehosting the dynamic model. The author defines “rehosting” as a specialized form of reuse in which the component is a mostly self-operating whole that takes direction (i.e. simple commands) from its host. This paper explores design characteristics for a human-in-the-loop simulation architecture that allow the dynamic model to be re-hosted in other applications. The design must first target a level of abstraction for reuse. The paper compares three abstraction levels: universe (i.e. vehicle with environment models), vehicle, and vehicle subsystem (e.g. aerodynamics model). The universe level has the greatest potential to deliver low-cost integration and identical behavior across multiple applications. Once a level of abstraction is selected, the simulation architecture must isolate the rehostable model from other packages; architectural isolation involves data communication and interaction control with the rehostable model. Architectural isolation accomplishes physical separation of the model. The definitions of inputs and services of the rehostable model’s interface can still logically tie the model to the supplier’s environment. In a rehostable model, the definitions must be free from assumptions about the application environment. The paper focuses on definitions specific to vehicle simulation: the cockpit data definitions and the integration services (for derivatives). To be useful, the rehostable model must allow acquirers to define new scenarios (i.e. initial conditions). The scenario definition services influence how suppliers share scenarios for model validation. Rehostable models must also be portable. However, dynamic models do not have portability issues unique from other types of software and detailed treatments on portability are plentiful. This paper will not discuss portability.

The paper uses the Langley Standard Real-time Simulation in C++ (LaSRS++) in its examples.[3] LaSRS++ is the simulation framework used by FSSB to build simulation products for LaRC’s high-fidelity research simulators. Recent design changes enable rehosting of the LaSRS++ Model package[§]. The paper also uses LaSRS++ as a case study in updating an existing architecture for model rehosting. As a demonstration of a rehostable architecture from a production framework, the paper presents a simple main program that performs a pitch stick doublet on the high-fidelity Boeing 757 model from LaSRS++.

II. Level of Abstraction

Organizations can choose to design for reuse at any level of abstraction within the simulation architecture. This paper will compare and contrast three levels. These levels are depicted in Figure 1:

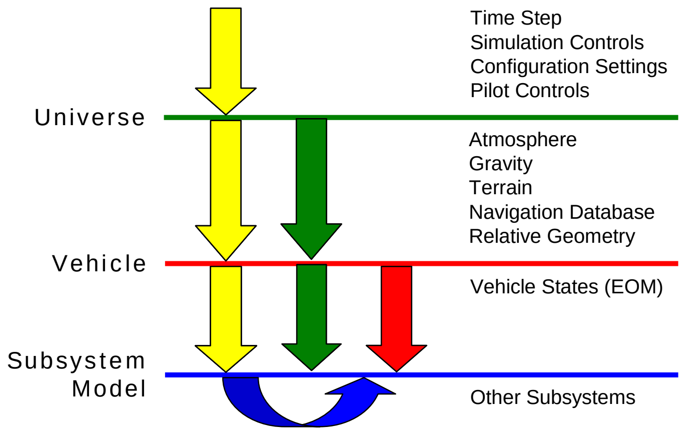

Figure 1 Input Interface for Different Levels of Abstraction

1. Universe – This level includes the entire modeling environment including the vehicle, world, and environment models plus any support computations such as relative geometry.
2. Vehicle – This level includes the code that defines the vehicle and its subsystems. This level is assumed to include the EOM and the trim algorithm.
3. Subsystem Model – This level includes only the code that defines vehicle subsystems such as the engine model. This level includes all of the vehicle’s subsystem models.

[§] This paper uses the Unified Modeling Language (UML) definition of “package”, “a general purpose mechanism for organizing elements into groups”. The LaSRS++ Model package is a group of classes logically associated with dynamic modeling.

Table 1 Commands at Each Level of Abstraction in LaSRS++

Universe	Vehicle	Subsystem Model
Universe()	Vehicle constructor	Subsystem Model constructor
~Universe()	Vehicle destructor	Subsystem Model destructor
resetPositionalModel()[a]	doResetCalc()	initialize()
doTrim()	doTrimCalc()	resetInitialConditions()
doHold()	doHoldCalc()	processInitializationFile()
doOperate()	doOperateCalc()	calculate()
propagatePositionalModel()	propagateSate()	
doPrint()	copyDataRamFileToPhysicalFile()	
generateLinearModel()	generateLinearModel()	
resetEnvironment()		
updateEnvironment()		
calcGeometry()		
addPositionalModel()		
delPositionalModel()		

[a] The top of the vehicle hierarchy in LaSRS++ is PositionalModel. A PositionalModel is any model that occupies space. The Vehicle class derives from PositionalModel and represents a PositionalModel that responds to forces and moments.

For a given level of abstraction, a rehostable design must define the minimum interface required to feed and operate the component. Figure 1 provides a simplified depiction of the flow of inputs in LaSRS++. Other simulation architectures may not have such a clear division between levels of abstraction. For example, a given simulation architecture may embed gravity and the word shape model (part of "terrain" inputs) in the vehicle's EOM.

The arrows in Figure 1 represent data that the host application must input into a given level of abstraction. (The arrow's color matches the source abstraction level with yellow representing inputs external to all models.) Universe is the level of abstraction with the fewest input requirements. Inputs become more numerous at the lower levels. At the Universe level, the inputs represent operator decisions; the inputs are not outputs from a dynamic model[¶]. At lower levels, the host application must re-engineer environment models (atmosphere, gravity, terrain, etc.) and other algorithms (e.g. EOM, trim, relative geometry) to generate some of the inputs. As a simulation architecture matures, the re-engineering of models and algorithms can represent a substantial development effort. For example, the LaSRS++ environment models (atmosphere, gravity, terrain, and navigation database) contain 30 KLOC and represent ~10% of the simulation framework. If an organization or project demands identical behavior from each model deployment, acquirers must perform additional testing to certify that re-engineered models and algorithms perform identically to the supplier's model or algorithm.

In LaSRS++, the difference in interface size between levels of abstraction is primarily input driven. The number of commands is small compared to the number of data items. Table 1 lists commands at each abstraction level. The listed commands constitute the minimal interface for operating a given abstraction level. The list shows the methods that LaSRS++ calls to operate that abstraction level. Each level has a richer set of commands that are accessible. However, internal interactions within a level of abstraction invoke the additional commands. Applications could use the additional commands to exercise fine control over the model. However, the economic goal of rehosting is replicating model behavior across multiple environments for low cost. A rehostable model assumes that the acquirer is willing to trade control for simplicity. The rehostable model aims to reduce model operation to a few semi-autonomous commands. In LaSRS++, the number of commands decreases as one progresses to lower abstraction levels because the scope of control also decreases. However, each increase in scope does not result in an equal increase in commands. The Universe level exerts control over environment models, vehicle models, and relative geometry calculations within a little more than double the number of commands provided by the Subsystem Model level. Higher levels of abstraction do not trade scope of control for interface complexity. The work to interface the additional commands at a higher-level of abstraction should require less effort than re-engineering the control logic and rehosting a lower-level abstraction.

The Universe level of abstraction has the greatest potential to achieve rehosting goals of model replication with the lowest effort. The Universe level has the smallest input interface. Universe inputs are not outputs from models or algorithms that acquirers must re-engineer and certify as identical to the supplier's model or algorithm. Relative to

[¶] The time step in LaSRS++ is fixed, and LaSRS++ treats it as a configuration setting.

its size, the command interface at the Universe-level exerts greater control on behalf of the acquirer than lower abstraction levels. The acquirer sees a net savings over re-engineering the control logic and reusing a lower-level abstraction. The remainder of the paper will focus on rehosting at the Universe level. However, some of the design characteristics of a rehostable Universe also apply to a rehostable Vehicle and Subsystem Model. The LaSRS++ architecture provides rehosting at both the Universe and Subsystem Model levels. However, LaSRS++ does not currently support rehosting at the Vehicle level.

III. Architecture Isolation

The level of abstraction establishes the context for the rehostable model's inputs and commands. The mechanisms for communicating input data and the external interactions of the model's commands must also be designed for rehosting. Communication mechanisms can introduce source code dependencies that prevent a given level of abstraction from being re-hosted in a new application. Similarly, a model can be implemented with control autonomy that extends outside of itself. Directing services from external packages or pushing outputs to external packages will cause the model to depend on the source files of the external packages. A model cannot be removed from an architecture and rehosted unless it is isolated from the surrounding architecture.

A. Communications

The top yellow arrow in Figure 1 represents interface data that originates outside of a rehostable Universe. Figure 1 also illustrates that this external data must flow down to lower levels. To make Universe rehostable, the communication mechanism that delivers the interface data must not bind Universe to the supplier's architecture. Such bindings can make Universe inseparable from the remaining supplier architecture, especially if such bindings are pervasive throughout the architecture. Even when the bindings have limited scope, acquirers must include or re-engineer parts of the supplier's architecture to rehost the model. The acquirers will require mentoring from the supplier to successfully integrate the model. Moreover, the integration is less resilient to changes in the supplier architecture.

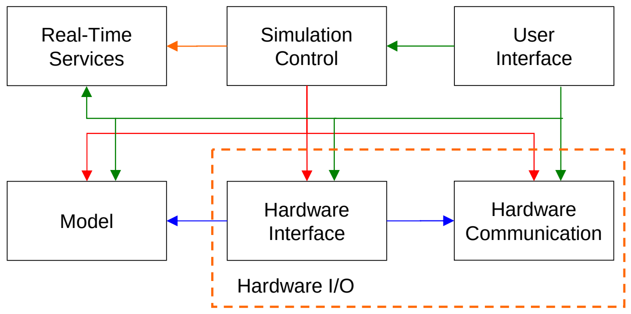

Figure 2 LaSRS++ High Level Architecture

To illustrate, Figure 2 depicts the high-level architecture of LaSRS++. LaSRS++ divides, into packages, services that are typical in many human-in-the-loop simulation architectures. These services are the source of the external input data to the Model package (i.e., the yellow arrow in Figure 1), which represents the Universe level of abstraction. Real-Time Services provides the time step. Simulation Control provides the simulation control data (e.g. mode) and configuration settings. The Hardware I/O subsystems (Hardware Interface and Hardware Communication) provide the pilot inputs from the cockpit. The arrows in Figure 2 depict the direction of dependencies in the LaSRS++ framework. LaSRS++ does not bind Model to its data sources; the binding occurs in the opposite direction. (Simulation Control has responsibility for communicating Real-Time Services data to the Model.) Isolating data communication requires a simple non-binding mechanism where external objects push inputs into the rehostable model and pull outputs from the rehostable model. Some LaSRS++ communication implements this push-pull mechanism directly using method calls. LaSRS++ uses two other communication designs that build upon the push-pull mechanism, the mediator pattern[5] and passing handles.

The Hardware I/O subsystems are an example of the mediator pattern. The Hardware Interface acts as a mediator for the Model and Hardware Communication. The Hardware Interface is responsible for reading inputs from the Hardware Communication and inserting the data in the Model. Likewise, the Hardware Interface actively reads outputs from the Model and inserts the data into Hardware Communication. A typical input operation in Hardware Interface would have the form:

```
        dynamic_model->putInput(hardware_communication->getOutput());
```

As implemented in LaSRS++, the mediator pattern is a design realization of the push-pull communication mechanism. Reference 6 provides more information about use of the mediator pattern for LaSRS++ hardware I/O.

LaSRS++ communicates the remaining "external" data to the Model using handles. These handles take the form of files or object references. LaSRS++ reads configuration settings from a file. All packages use the same file for configuration settings. Each package is responsible for parsing the package's configuration settings from the file.

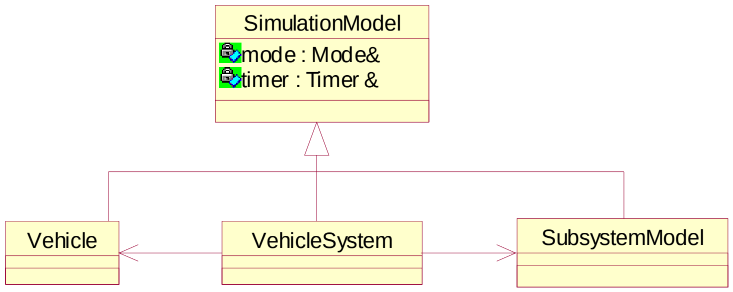

Figure 3 Vehicle-Subsystem Relationships

Thus, Simulation Control communicates configuration settings to the Model by pushing the file. Simulation Control uses object reference to communicate the time step and simulation control data. The time step and simulation control data are packaged into self-contained classes, i.e. classes with no external dependencies on any LaSRS++ package. The major classes used in this communication are Timer (time step) and Mode (simulation control). Simulation Control creates objects of these classes and pushes references to these objects into the Model. When Simulation Control updates the objects with new data, the new data is instantly accessible to the Model through the object reference. The key here is that the classes, which act as intermediaries for communication, are themselves isolated from the packages and do not reside in any package library. In LaSRS++, the Timer and Mode classes are compiled with other package neutral classes (such as integrators) into a “toolbox” library.

LaSRS++ uses the same communications mechanisms to provide architectural isolation to the subsystem models, enabling the subsystem models to be rehostable components. LaSRS++ uses the mediator pattern to isolate each subsystem model from its input sources. Each subsystem model has a companion mediator derived from the VehicleSystem class (see Figure 3). The VehicleSystem object retrieves the subsystems inputs from the appropriate sources. It then inserts the inputs into the subsystem model and operates the subsystem model. More details on the use of the mediator pattern for subsystem models are found in Refs. 7 and 8. The mediator pattern does not handle all the input sources. Time and simulation control data are communicated using object references. In fact, the object references that the Simulation Control package inserts into the Model package are distributed throughout the models in the package. Each higher-level abstraction passes these references to next level. LaSRS++ facilitates and enforces this reference passing by embedding these references as constructor arguments for the base class of all dynamic models, SimulationModel. When a dynamic model is created, it receives the references from its creator and, within its constructor, will pass those references onto models that it creates.

B. Control of External Interactions

Communications alone do not provide the architectural isolation required for rehostable models. The rehostable model must relinquish control of its external interactions. The external target of the interaction must assume control of the interaction or it must delegate control to an independent (i.e. reusable) proxy. In the latter case, the intermediary becomes part of the interface to the model when it is rehosted. Data communication as discussed in the previous section is one form of control. This section will focus on control of activities.

Vehicle destruction in LaSRS++ is an example of reassigning control to the target. The top-level class in the Model package is Universe. Universe maintains a list of vehicle objects. Removal from Universe’s list is the first step in deleting a vehicle object. In previous LaSRS++ designs, Universe was given responsibility for deleting the vehicle object. Deleting a vehicle object in LaSRS++ requires more than deleting the dynamic model. The Hardware Interface objects associated with the vehicle object must also be destroyed to reclaim memory and hardware connections (i.e. Hardware Communication objects) for future vehicle objects. Universe had to send a request to Simulation Control for deletion of the associated Hardware Interface objects. Universe could not be reused without inclusion or re-engineering of this Simulation Control interface. The recent LaSRS++ redesign corrected this issue by reassigning control of Vehicle deletion to the Simulation Control package. Simulation Control now requests that Universe remove the Vehicle from its list. Then, Simulation Control deletes the Vehicle and its associated Hardware Interface objects. Universe no longer depends on Simulation Control and is reusable without including or re-engineering the Simulation Control interface.

Simulation mode control in LaSRS++ is an example of delegating control to a proxy. The main simulation control datum in LaSRS++ is an enumeration of “modes.” The modes in LaSRS++ are RESET, TRIM, HOLD, OPERATE, PRINT, and LINEAR_MODEL. Some Model classes request mode changes. For example, the VehicleLimits hierarchy transitions the simulation to HOLD from OPERATE when a vehicle state exceeds a programmed limit. The limits can be used to define conditions that would exceed the structural limits of the real aircraft or that exceed safety limits for humans riding in a motion base. Originally, a VehicleLimits object would directly request the mode transition from Simulation Control. The VehicleLimits objects could not be reused without including or re-engineering the Simulation Control interface. The redesign corrected this by converting the mode enumeration into an independent class called Mode, which encapsulated the enumeration and all related services. As explained in the previous section, Simulation Control creates the Mode object and passes it as an object reference to the Model. The reference is then passed from parent model to child. The VehicleLimits object now requests the transition using the object reference. Simulation Control later queries the Mode object for a new transition request and executes the request. Reuse of the VehicleLimits object now requires the Mode class, but this dependency is not a problem because the Mode class is an independent, reusable class. The Mode class becomes part of the interface to model when it is rehosted.

IV. Data and Service Definitions

Architecture isolation enables the model to be physically rehosted in a new application. However, the definitions of the inputs and services can still logically tie the rehostable model to the supplier's environment and introduce additional effort or constraints for acquirers. This section examines two topics specific to vehicle models: definition of cockpit inputs and integration strategy (for derivatives).

A. Definition of Cockpit Inputs

Section III.A discussed the necessity of isolating the rehostable model from the source of its inputs. However, the definition of cockpit inputs should also be free from the supplier's environment. If the input definition reflects the raw hardware data in the supplier's environment, the input definition defeats the isolation provided by the communications mechanism. The rehostable model would be required to scale the inputs for the dynamic models and this scaling algorithm would be specific to one hardware cockpit. Rehosting the model would force acquirers to re-engineer the interface of the hardware cockpit. The interface may not make physical sense in a new environment. The acquirer may need to gain a depth of knowledge into the hardware, for which they would not otherwise have a use. Practical rehosting of the model requires a cockpit interface whose data content conveys general meaning to acquirers. Representing the data in the units expected by the vehicle model is one choice. The interface is tailored to the vehicle, and the interface has physical meaning to all acquirers. However, acquirers working with multiple vehicle models will have to tailor an interface to each model. Normalizing the cockpit inputs is an alternate design choice that allows acquirers to create a single generic interface to all models. LaSRS++ vehicles use a cockpit interface in which the cockpit controls are normalized between 0 and 1 (e.g. spoiler handle) or -1 and 1 (e.g. longitudinal stick) as appropriate. One disadvantage to normalized cockpit interfaces is an increase in computation. The application must normalize its inputs before feeding them to the rehostable model. The rehostable model must then scale the normalized values to the units and range expected by the vehicle model. It is a choice of reduced development cost and maintenance over increased computation. In the case of LaSRS++, the choice was not based on reuse considerations; LaSRS++ uses normalized cockpits to allow any vehicle model to run in any simulator operated by FSSB.[6] Another disadvantage of normalized cockpits is that the supplier must document the scaling of asymmetric controls. For example, if the longitudinal stick in a given model has a range of -6 in. to 12 in., the supplier must document whether:

- The model applies the scale factor of 12 in. to positive and negative stick values but treats all normalized values less than -0.5 as -6 inches.
- Or, the model applies a factor of 6 in. to negative stick values and a scale factor of 12 in. to positive stick values.

Choosing a different solution for each model reintroduce some tailoring of the interface; thus, organizations should choose the same scaling solution for asymmetric controls when possible.

B. Integration Strategy

The strategy that the rehostable model uses to integrate derivatives can impact its reuse in other applications. Integration strategy encompasses the integration algorithms, timing of integration, and how chains of integrators behave.[#] LaSRS++ and Simulink®/Real-Time Workshop® integration strategies are a good contrasting example.

[#] Fixed vs. variable time step is also a consideration. However, this paper will assume that target applications accommodate fixed time steps. The Model package in LaSRS++ was not designed to assume a fixed time step. Time step can be retrieved from a Timer object whenever it is used. However, the Simulation Control package assumes a fixed time step since FSSB runs only simulations using a fixed time step. FSSB has not validated the Model package using a variable time step. Thus, a vehicle implementation may contain assumptions that the time step is fixed.

Each product uses the integration strategy to achieve different goals. LaSRS++ uses an integration strategy that minimizes transport delay.[**] Simulink® uses a mathematically consistent strategy that produces uniform results for a broad array of analysis techniques. In its integration strategy, LaSRS++ uses a variety of integration algorithms. Integration algorithms in the EOM are selected to obtain the best approximation for a given state while minimizing transport delay. For example, the LaSRS++ integration algorithm for position is a truncated tailor series that uses both velocity and acceleration so that pilot actions, which result in acceleration changes, affect position in one frame. Typical LaSRS++ dynamic models perform integration inline, i.e. at the point in the model where the integrator appears. The effect of a derivative change (which may trace back to pilot actions) is immediately communicated downstream and will affect vehicle behavior within one frame. In a chain of integrators, the first derivative will influence the result of the last integrator in one frame. Simulink® uses the same integration algorithm for all integrators by default. Simulink® performs all integrations as a distinct stage within a frame. In a chain of integrators, the result of a predecessor integrator does not flow to the next integrator in the same frame. Thus, a change in a derivative at the start of a chain of three integrators will take two additional frames to affect the outcome of the last integrator. The different integration strategies give rise to special considerations and constraints when a component from either environment is rehosted in the other. To maintain minimum transport delay when a Simulink component is rehosted in LaSRS++, the LaSRS++ developer must run the component (or the whole vehicle model) at a integer multiple of the hardware I/O rate; the integer multiple is equal to the length of the longest chain of integrators in the Simulink model.[9] When a typical LaSRS++ component is rehosted in Simulink, the component does not provide centralized control of its integrators that acquirers could delegate to Simulink. The component will retain control of integrators and run them during the output computation phase in Simulink. The component can be used to run a simulation in Simulink®, but it will not produce correct results in analysis tools that require manipulation of the states (e.g. Simulink's linear modeler).

The ideal rehostable model would be capable morphing to different integration strategies. LaSRS++ has design characteristics that enable some morphing of its integration strategy. LaSRS++ integrators are part of a polymorphic class hierarchy whose base class is Integrator. The Integrator base contains the necessary interface for operating integrators; all modeling code operates the integrators through the base-class interface. The algorithm that the modeling code will invoke depends on the type of Integrator object that the model constructs. The model can allow the integration algorithm to change without changing the modeling code by providing methods that replace the Integrator object(s). The LaSRS++ EOM does provide this capability. However, subsystem models are not required to provide this capability, and none of the currently implemented subsystem models do. The models could easily add this capability at a later date if required. An acquirer could use the LaSRS++ Integrator design to create a new integrator type that delegates the integration to the application's integration engine. Using integrator replacement methods in the LaSRS++ EOM and subsystem models, the acquirer could replace the model's existing integrators with the application-specific integrator to delegate all integration control to the application. This mechanism could allow an application like Simulink® to control the integration of a LaSRS++ model. However, the acquirer would have to learn the model's decomposition and the location of all the integrators in the model. The resulting integration code would be tailored to the specific model. Creation of a generic interface requires a central access location for all of the model's integrators. The integrator list in the LaSRS++ EOM could be used for this purpose. The LaSRS++ EOM provides a method to add integrators to this list. The EOM operates all integrators on the list at a specific stage in the frame (after LaSRS++ computes vehicle accelerations and world dynamics). Because the LaSRS++ design guideline for subsystem models is inline integration, the subsystem models do not currently add their integrators to the list. It is possible to design a LaSRS++ vehicle to centralize access and control of its integration. Thus, a LaSRS++ vehicle could be designed to use the same integration strategy as Simulink. But, it is not currently possible to support both an inline and staged strategy in the same vehicle. This too could be changed with some design tweaking. A switch that can be set to inline or staged operation could be added to the base Integrator class. Subsystem models would then be required to both add their integrators to the EOM's integrator list and call the integrators inline. Whether the inline operation or the staged operation would run the integration algorithm would depend on the switch setting. LaSRS++ could emulate a vehicle-wide setting for the switch using the integration list in the EOM. User demand will determine whether future work implements this modification.

V. Defining Scenarios and Model Validation

The paper discusses defining scenarios and model validation together because defining scenarios is the first step in model validation. Though most scenario definition activity is intended for operational use, design decisions

[**] The delay between observer action and observation of the action's results.

should consider simplification of model validation. At a minimum, a rehostable model must allow acquirers to define new scenarios without code changes. The rehostable model cannot contain a finite set of hard-coded scenarios with the expectation that they will fulfill the needs of all users. The model could provide a programming interface that allows acquirers to code scenarios. The Model package in LaSRS++ does provide such an interface. Model classes provide methods for setting initial values, and the trim engine provides methods for selecting rules that describe the vehicle action in the scenario (e.g., a coordinated turn). Using a programming interface for scenario flexibility puts a burden on acquirers to write and test scenario code. Acquirers must also learn the internal structure of the rehostable model. The supplier must communicate this knowledge through training or documentation. Retraining and code modification will be necessary if the internal structure of the model changes. FSSB has had such experience rehosting Simulink®/Real-Time Workshop® components in LaSRS++, which uses the programming interface in the generated code for initialization. The internal structure of Real-Time Workshop® generated code has changed in the last three versions of the product. Each version has led to a cycle of re-learning, code modification, and validation. Cutting the additional cost for development and learning is desirable for internal deployment, and appreciated by external customers.

A programming interface also introduces problems for scenario sharing. Sharing scenarios is common among development teams on the same project. A supplier team must share scenarios as part of delivering the model; the supplier must bundle a set of scenarios with the model for validation testing in new applications. Scenario sharing is necessary between an acquirer and supplier when they collaborate on debugging of suspicious behavior. The scenario for a programming interface must be communicated as a description or a code fragment. A description cannot be tested for completeness and is therefore subject to omissions. Both communication forms require translation (e.g. code modification) into the target application; a translation step can introduce a defect. A better design for scenario definition is native support in the rehostable model for a script language. Acquirers need only learn the script language to define new scenarios. The internal structure of the rehostable model can change without impacting acquirers if the script language remains unchanged. Scenarios would be shared as scripts. The rehostable model executes the scripts directly; the acquirer performs no translation of the script that can insert a defect. The supplier can test the script for completeness before distributing it. Thus, the acquirer can be confident that a discrepancy in check case results is not the result of missing scenario information.

The LaSRS++ Model package provides a script language for defining the available scenarios of a vehicle object. An example script appears in the Appendix, Section A. The script language assigns a number to each scenario. The target application can then load a scenario by passing a number to vehicle object; the default case is zero if no case is passed. The LaSRS++ Model package provides a similar script language to configure the environment model for the scenario. This script is normally stored in a configuration file that also contains settings for the other LaSRS++ packages. The Model package reads the configuration file and ignores settings for the other packages. Both the configuration file and the vehicle's scenario file are necessary to define a complete scenario for the Model package.

The Introduction listed incomplete scenario definitions as one problem with supplied check cases. The scripted scenario definition discussed here solves this issue for static check cases (i.e. trim algorithm results) and dynamic check cases (i.e. time history results) without perturbations of cockpit controls. However, the scenario script does not describe cockpit perturbations for dynamic check cases. Section II established cockpit inputs as data external to a rehostable model. From this perspective, scripted cockpit perturbations are the responsibility of each target application. An organization can gain the same benefit as scripted scenarios by agreeing on a standard script language for cockpit perturbations. However, each acquirer must develop the execution engine for the script. A rehostable model with native support for cockpit perturbation scripting would alleviate the additional development. The feature would require an on-off switch in order to hand cockpit control to the target application. An alternative solution is to supply a rehostable model for cockpit perturbations.[††] This would be the preferred solution for LaSRS++. The LaSRS++ architecture strictly assigns responsibility for cockpit inputs to the hardware packages. Within the Hardware Interface package, LaSRS++ currently provides a DynamicCheckCockpit hierarchy for scripting cockpit perturbations. This hierarchy is not currently rehostable because it does not exhibit the necessary architectural isolation that would allow it to be physically separated from the remainder of LaSRS++ (outside of itself and the Model package). Future work will make the necessary design modifications.

The rehostable model reduces or eliminates the remaining problems with model validation that the Introduction lists: incomplete results, insufficient code coverage by check cases, and missing environment models and algorithms. The focus of model validation for a rehostable model is not the model's code but the acquirer's interface code. A re-engineering effort can get lost attempting to track down the source of a discrepancy with incomplete

[††] This component may not strictly be rehostable because it can depend on the rehostable model.

results. A rehosting effort knows that the likely cause is the interface code[‡‡], which should be less code to search than the model code. A rehostable model allows the supplier to easily eliminate the issue of incomplete results. The supplier can rehost the model in a simple program that runs the check cases and add the program to the delivery. If the acquirer finds the check case results to be incomplete, the acquirer can obtain the missing data by running the check-case program. Ideally, re-engineering efforts require check cases that cover a substantial amount of the model code to independently validate the re-engineered models. Rehosting efforts only require check-cases that cover all of the interfaces. If the supplier provides a check-case program, the acquirer can add any missing check cases to the program in order to obtain an independent result for validation. Finally, the rehostable model includes the supplier's environment models and algorithms so this is not a problem for rehostable model. A rehostable model bundled with a check-case program also eliminates the three check case problems for an acquirer that decides to re-engineer the model. The check-case program represents a small but complete executable copy of the supplier's model, which the acquirer is free to observe and to which the acquirer can add new check cases.

VI. Transforming an Existing Architecture for Rehosting

An organization that decides to peruse rehosting as a solution for model deployment has the choice between modifying their existing simulation architecture for rehosting or building a new architecture from scratch. LaSRS++ represents an example of how design choices in an existing architecture can enhance or detract from the transformation of an abstraction level into a rehostable component. Few of the design choices described earlier were made with the purpose of creating rehostable components. The original purpose of the mediator for Hardware I/O was to allow a vehicle model to run in any cockpit.[6] However, it also provides the added benefit of isolating the Model package from the hardware design. Designing packages to parse their settings from the configuration file was done to simplify the work of adding project-specific configuration settings to the file. The welcome side effect was reducing the interface for configuration settings to passing a single file. The VehicleSystem mediator was originally designed to allow unit testing of subsystem models and integration of reusable subsystem models from third parties.[7] However, its contribution to creating rehostable LaSRS++ subsystem models was recognized at the time of its design. Inserting time and simulation control data as object references was initially intended to prevent every dynamic model class from depending on the Simulation Control package and to allow some models to run with different time steps and simulation control data than their creator. However, this last goal left open the possibility for some dynamic models to remain bound to the Simulation Control package. In other words, not all simulation control data and services were packaged into independent classes initially. Reigning in the remaining simulation control data and services constituted the bulk of recent work to make the Model package in LaSRS++ a rehostable component.

The LaSRS++ architecture did not start with characteristics that would facilitate rehosting at the Universe level of abstraction. Each of the above design decisions represents an evolutionary step in the LaSRS++ architecture that was a large effort individually. Even the recent design tweaking has introduced small changes to 385 LaSRS++ classes. Different design choices could have caused the redesign of the Model Package as a rehostable model to be cost prohibitive. For example, LaSRS++ could have allowed the Model classes to retrieve timing and simulation control information directly from the Simulation Control package. Because Simulation Control stores this information in a class utility[§§], it would be easier for a developer to retrieve this data directly rather than code the passing of object references through multiple layers of abstraction. Always taking the easy coding path eventually leads to spaghetti code and the blurring of responsibilities. For example, the Hardware I/O code could have been designed to provide raw inputs. Vehicle developers would be responsible for scaling the inputs for the vehicle. The developers could choose to scale the inputs where they are used. This would spread the scaling code among multiple classes. These classes would be responsible for both scaling inputs and modeling part of the vehicle. Each class would require modification in order to create a rehostable component whose cockpit inputs were normalized. Although LaSRS++ was not originally architected for model rehosting, some luck and foresight in its evolution positioned the architecture for rehosting at the Universe level of abstraction.

The current LaSRS++ architecture, however, is not well positioned to transform the Vehicle level of abstraction into a rehostable component. LaSRS++ vehicles are bound to the Universe level above them. LaSRS++ vehicles directly access the world models for atmosphere, gravity, terrain, and navigation database information. Relative

[‡‡] The occasional compiler bug or previously undetected portability defect are remote possibilities.
[§§] A class utility only has static methods and data. For a class to access a class utility, the class simply includes the class utility header and makes scope-qualified method calls directly. It is similar in ease of use to making external calls to global functions.

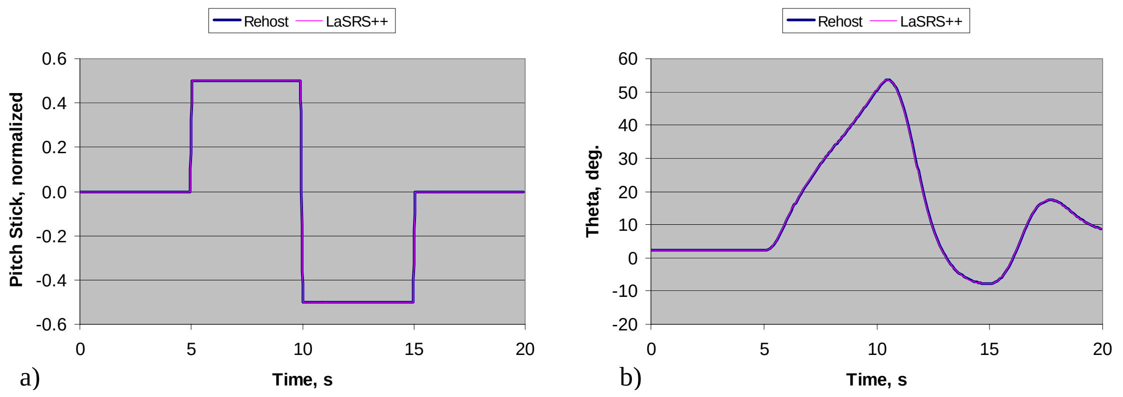

Figure 4 Comparison of Pitch Stick (a) and Theta (b) Between Rehosting Example and LaSRS++.

geometry is centrally computed and stored. Vehicles in multi-vehicle simulations (e.g., combat or traffic simulations) actively retrieve this information from this central store. Moreover, relative geometry may not provide all the necessary information that one vehicle needs to know about the other. LaSRS++ vehicles are free to use Universe level services to obtain references to other vehicles. Without a pre-existing requirement to make vehicles rehostable, LaSRS++ vehicles are given full control to obtain services and information from the Universe level because the LaSRS++ architects did not want to make assumptions about how future vehicle models may interact with their surroundings. If reuse at the Vehicle level of abstraction became a requirement, LaSRS++ would have to undergo its largest restructuring to date. To justify the cost, the need for Vehicle level reuse would have to recur with sufficient frequency. Otherwise, it is more cost effective to convince acquirers to reuse at the universe level or reconstitute the vehicle from the reusable subsystems. If such a restructuring did occur, it would rely on the architecture characteristics and mechanisms used to make the Universe and Subsystem Model level rehostable: mediators, handles, and reallocation of control. The latter will be the most problematic since reallocation of control would have to accommodate all current Vehicle-Universe interactions and possibly cover foreseen interactions. To prevent pollution of the universe-level with special cases, the interaction control mechanism must be extensible. Vehicle developers can use inheritance and polymorphism to add the special cases to the control mechanism when the vehicle is built. This mechanism could be similar to the existing Hardware Interface mediator which uses inheritance and polymorphism to control I/O for the specific needs of a particular vehicle or hardware device.

As LaSRS++ demonstrates, the ability to transform higher levels of abstraction for rehosting depends on the extent to which initial design decisions and later evolutions increased or decreased the coupling between dynamic model code and other simulation services. Although it may not be cost effective to completely transform a current architecture for rehosting, the supplier team can steer the architecture’s evolution towards that goal by applying the characteristics described here to the designs of future modifications. Eventually, the architecture may evolve to the point where a small leap is needed to transform a level of abstraction into a rehostable component.

VII. Rehosting the LaSRS++ Model Package – a C++ Example

Appendix Section B contains C++ code that rehosts the LaSRS++ Model Package in a simple program that executes a pitch stick doublet on a Boeing 757. This code was linked using the following object files and libraries:

- Object file built from the example code - b757_main.o
- The libraries for the Boeing 757 library:
  - libb757.a – primary B757 aircraft library
  - libb757aero.a – B757 aerodynamics functions/lookups
  - libb757autopilot.a – B757 autopilot functions
- The LaSRS++ Model package libraries
  - libdatarecording.a – data recording utilities
  - libenvironment.a – environment models
  - liblinearmodel.a – linear model classes
  - libmodels.a - vehicle base classes
  - libcockpits.a – normalized cockpit interface
  - libsystems.a – vehicle subsystem base classes
  - libnavigation.a – navigation database

    - libplayback.a – playback classes
    - libtrim.a – trim algorithms
- The LaSRS++ toolbox library – libtoolbox.a

The rehosted program size was 64MB. A LaSRS++ executable containing the same model is nearly four times as large, 244MB. Figure 4 compares the pitch stick and pitch (theta) values output by the sample program with the outputs from the full LaSRS++ simulation. As expected, the outputs are identical.

VIII. Conclusions

Model rehosting is verbatim reuse of a semi-autonomous model. Rehosting at the Universe level of abstraction provides the greatest potential for low-cost model deployment in an organization where identical model behavior across target applications is the main goal. Other levels of abstraction such as the Vehicle or Subsystem Model will require re-engineering and re-validation of environment models and algorithms (e.g. EOM and trim). Also, the Universe level has the smallest input interface and performs more control work per interface command than the other levels. However, the other abstraction levels may be more desirable in situations where identical model behavior is a lesser concern and finer control over a vehicle model or its parts is required. Cost will also play a factor. Depending on the original design and evolution of an existing simulation architecture, certain abstraction levels may require less effort to redesign for rehosting than others. In a few cases, creating a new architecture may be the most cost effective means of creating rehostable models.

A rehostable model must exhibit some key design characteristics. The rehostable model must be architecturally isolated from the supplier application. The communication mechanisms that the model employs must not establish dependencies external to the model. Mechanisms through which all inputs are pushed into the model and all outputs are pulled satisfy this criterion. These mechanisms can deploy intermediary objects and interfaces that have no external dependencies. These intermediaries become part of the interface of the model when it is rehosted. The rehostable model must relinquish control of its external interactions to the targets of those interactions. In some cases, this requires reformulating the interaction so that the external object requests a service from the model. In other cases, independent intermediaries can be used as proxies for services.

The definition of the model's data and services must refrain from retaining assumptions that logically tie the model to the supplier's environment. For a rehostable Universe, providing a normalized cockpit interface divorces the model from the supplier's hardware environment while allowing acquirers to build generic interfaces for a variety of vehicles. The model's use of integrators should be customizable to accommodate different integration strategies (e.g. algorithm replacement and inline vs. staged integration).

A rehostable Universe must give acquirers the means to define new scenarios. Native support for a scripting language fills this need while providing an error-free means of sharing scenarios among different teams. Scenario sharing is an important component of model validation. Model validation takes on a slightly different meaning in a rehosting effort of a Universe-level abstraction than in a re-engineering effort. In re-hosting, model validation focuses on finding defects in the interface code. The acquirer reuses the same model code as the supplier and it should need no re-validation. Thus, rehosting efforts suffer less from deficiencies in supplier's check cases. A rehostable Universe also provides a unique opportunity for the supplier to deliver an executable model in the form of a check-case program. The check-case program frees the acquirer to observe the complete set of model outputs and to create additional check cases. A check-case program makes a rehostable Universe a valuable asset to acquirers whether they choose to rehost or re-engineer.

Appendix

A. Scenario Script Example from LaSRS++

The text below is a selection from the scenarios script that FSSB supplies as a default for the Boeing 757 model. The script shows the case used in the code example in Section VII.

```
case: all
  clear recording set list.
  record time:                 true
  record positional model base: true
  record positional model body: true
  record vehicle body:         true
  record aero coefficients:    true
  record aircraft air:         true
  record aircraft air velocity: true
  record landing gear:          true
  record propulsion:            true
  record cockpit:               true
  trim maximum iterations:      4000
break;

case: 2
  runway as primary ref point: KDFW:RW18R
  geodetic coordinates: lat 32.194  deg, lon -98.662 deg, alt 24000
  flap handle position: 0.0
  gear handle: LANDING_GEAR_UP
  weight: 180000.0
  initial cg as percentage mean aerodynamic chord: 0.15
  moments of inertia relative to CG.
  Ixx: 1.219999e+06
  Iyy: 5.306344e+06
  Izz: 6.349998e+06
  Ixz: 1.52999e+05
  initial fuel weight: 30000.0
  euler angles: phi 0.0 deg, theta 0.0 deg, psi 92.0 deg
  straight flight on speed set.
    straight flight on speed vtotal: 280.0
    straight flight on speed type: INDICATED
break;
```

B. Source Code for Rehosting Example in C++

The code comments explain LaSRS++ interface usage not described elsewhere in this document.

```
#include "B757Base.hpp"
#include "Cockpit.hpp"
#include "ControlLoader.hpp"
#include "Mode.hpp"
#include "PositionalModelInitialInfo.hpp"
#include "Timer.hpp"
#include "Universe.hpp"

int main()
{
  Mode      mode(Mode::RESET); // Mode encapsulates operational state.
  Timer     timer(0.020);      // A 20 millisecond time step (50 Hz).

  // LaSRS++ supports M vehicles on N processors. A model must be assigned
  // to a processor group. In this code, all models are assigned to
  // processor group 0. Universe methods must be passed the processor group,
  // to which the action applies. Only one processor group is needed for
  // this single thread example.
  const int processor_group = 0;

  // Create the Universe (i.e. the Model package). "siminit" is the name of
  // the configuration file.
  Universe  universe(timer, mode, processor_group, "siminit");

  // LaSRS++ packages the construction data for a vehicle into a
  // PositionalModelInitialInfo object. In LaSRS++, the top of the vehicle
  // hierarchy is PositionalModel. The vehicle will obtain the configuration
  // file from universe. The "initialization filename" defined below
  // contains the scenario definitions which the file identifies by number.
  PositionalModelInitialInfo constructor_data(&universe);
  constructor_data.putCpuId(processor_group);
  constructor_data.putInitializationFileName("b757_init_file.ic");
  constructor_data.putInitialCaseNumber(2);

  // Construct a Boeing 757 object. LaSRS++ vehicles add themselves to
  // Universe's list of vehicles.
  B757Base b757(constructor_data);

  // RESET Operations (Scenario Definition)

  // Mode transition requires two steps. Requesting the transition and
  // executing the request. In LaSRS++, this design allows any object to
  // request a transition at any time in the frame, but LaSRS++ will only
  // executes requests at the end of the frame.
  mode.putRequestedState(Mode::RESET);
  mode.update();

  // Multiple passes are required for some vehicles to properly initialize
  // all variables.
  for(int i = 0; i < 50; i++)
  {
    universe.resetEnvironment(processor_group);
    universe.resetPositionalModel(processor_group);

    // Mode must be updated at the "end" of each frame because it tracks
    // past values.
    mode.update();
  }

  // TRIM Operations (Solve for Scenario Steady State)
  mode.putRequestedState(Mode::TRIM);
  mode.update();
  universe.doTrim(processor_group);

  // HOLD Operation (Freeze Model)
  // Some models need at least one frame in HOLD before entering OPERATE.
  mode.putRequestedState(Mode::HOLD);
  mode.update();
  universe.doHold(processor_group);

  mode.putRequestedState(Mode::OPERATE);
  mode.update();
  // Timer operations do not change the Timer attributes until run is called.
  universe.getTimer().run();

  for(int i = 0; i < 1000; i++)
  {
    // At five seconds, execute a doublet with pitch stick movement in each
    // direction for five seconds.
    if (i == 250) b757.getCockpit()->getControlLoader()->putPitchStick( 0.5);
    if (i == 500) b757.getCockpit()->getControlLoader()->putPitchStick(-0.5);
    if (i == 750) b757.getCockpit()->getControlLoader()->putPitchStick( 0.0);

    universe.doOperate(processor_group);
    universe.updateEnvironment(processor_group);
```

```
    universe.propagatePositionalModel(processor_group);

    universe.getTimer().increment(); // Increments elapsed time.
    mode.update();
  }

  // PRINT Operations (Write Data File)
  // The LaSRS++ Model package includes data recording. The data is
  // written to memory during OPERATE to maintain real-time execution.
  // The memory contents are dumped to file when PRINT mode is executed.
  mode.putRequestedState(Mode::PRINT);
  mode.update();
  universe.doPrint(processor_group);

  // Remove the Boeing 757 vehicle from the Universe list. The B757
  // object will be destroyed when the program exits and implicitly
  // calls the B757 destructor.
  universe.delPositionalModel(&b757);

  return 0;
}
```

References

[1]Jackson, E. B. and Hildreth, B. L., "Flight Model Exchange Using XML," *AIAA Modeling and Simulation Technologies Conference and Exhibit*, 5-8 August 2002, Monterey, California, AIAA-2002-4482.

[2]Madden, M. M., "Examining Reuse in LaSRS++-Based Projects," *AIAA Modeling and Simulation Technologies Conference and Exhibit*, 6-9 August 2001, Montreal, Canada, AIAA-2001-4119.

[3]R. Leslie, D. Geyer, K. Cunningham, M. Madden, P. Kenney, and P. Glaab, "LaSRS++: An Object-Oriented Framework for Real-Time Simulation of Aircraft," *AIAA Modeling and Simulation Technologies Conference and Exhibit*, August 1998, Boston, Massachusetts, AIAA-98-4529.

[4]Object Management Group, "OMG Unified Modeling Language Specification", Version 1.5, URL: http://www.uml.org/cgi-bin/doc?formal/03-03-01 [online database], March 2003.

[5]Gamma, E., Helm, R., Johnson, R., Vlissides, J, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Massachusetts. 1995, ISBN 0-201-63361-2, pp 273-282.

[6]P. Kenney, et. al, "Using Abstraction to Isolate Hardware in an Object-Oriented Simulation," *AIAA Modeling and Simulation Technologies Conference and Exhibit*, August 1998, Boston, Massachusetts, AIAA-98-4533.

[7]Cunningham, K., "Use of the Mediator Design Pattern in the LaSRS++ Framework," *AIAA Modeling and Simulation Technologies Conference and Exhibit*, August 1999, Portland, Oregon, AIAA-99-4336.

[8]Madden, M. M., "A Design for Composing and Extending Vehicle Models," *AIAA Modeling and Simulation Technologies Conference and Exhibit*, August 2003, Austin, Texas, AIAA-2003-5458.

[9]Madden, M. M., "An Object-Oriented Interface for Simulink Models," *AIAA Modeling and Simulation Technologies Conference and Exhibit*, August 2000, Denver, Colorado, AIAA-2000-4391.