A DESIGN FOR COMPOSING AND EXTENDING VEHICLE MODELS

Michael M. Madden*
NASA Langley Research Center
Hampton, Virginia

Jason R. Neuhaus
Unisys Corporation
Hampton, Virginia

ABSTRACT

The Systems Development Branch (SDB) at NASA Langley Research Center (LaRC) creates simulation software products for research. Each product consists of an aircraft model with experiment extensions. SDB treats its aircraft models as reusable components, upon which experiments can be built. SDB has evolved its aircraft model design with the following goals:

1. Avoid polluting the aircraft model with experiment code.
2. Discourage the "copy and tailor" method of reuse.

The current evolution of that architecture accomplishes these goals by reducing experiment creation to "extend and compose". The architecture mechanizes the operational concerns of the model's subsystems and encapsulates them in an interface inherited by all subsystems. Generic operational code exercises the subsystems through the shared interface. An experiment is thus defined by the collection of subsystems that it creates ("compose"). Teams can modify the aircraft subsystems for the experiment using inheritance and polymorphism to create variants ("extend").

ACRONYMS

I/O	Input/Output
LaRC	Langley Research Center
LaSRS++	Langley Standard Real-Time Simulation in C++
LOC	Lines of Code
SDB	Systems Development Branch
UML	Unified Modeling Language

* Senior Member, AIAA

INTRODUCTION

SDB defines a project for each experiment (e.g. set of requirements) and assigns a team (e.g. a group of developers) to the project. "Project" is used frequently in place of "experiment" or "team" to differentiate from the requirements and the team for the aircraft model.

At a basic level, developing a simulation experiment involves adding, modifying, and/or subtracting behaviors to a known aircraft model. When LaRC used a procedural design paradigm to produce simulation products, projects employed the following methods to make some of the experiment modifications:

1. Adding conditional constructs to the "baseline"† aircraft code. These constructs can be control statements (e.g. if-then-else) or pre-processor guards.
2. Tailor the aircraft code for the experiment.

The problem with conditional constructs is that they pollute the aircraft model with experiment code. Each simulation experiment inherits code for other simulation experiments. Defects could be introduced that accidentally activate the code from another experiment. Code size and complexity also increases over time. This slows maintenance and heightens the opportunity for defects. The issue with "copy and tailor"‡ reuse is increased configuration management complexity. Devel-

† "Baseline aircraft code" is an approved version of the code, with which all projects start.

‡ Based on the configuration management tool that LaRC used at the time, a more accurate statement would be "retrieve and modify". The tool retrieved the baseline file and applied a modification file to it that resulted in the tailored file.

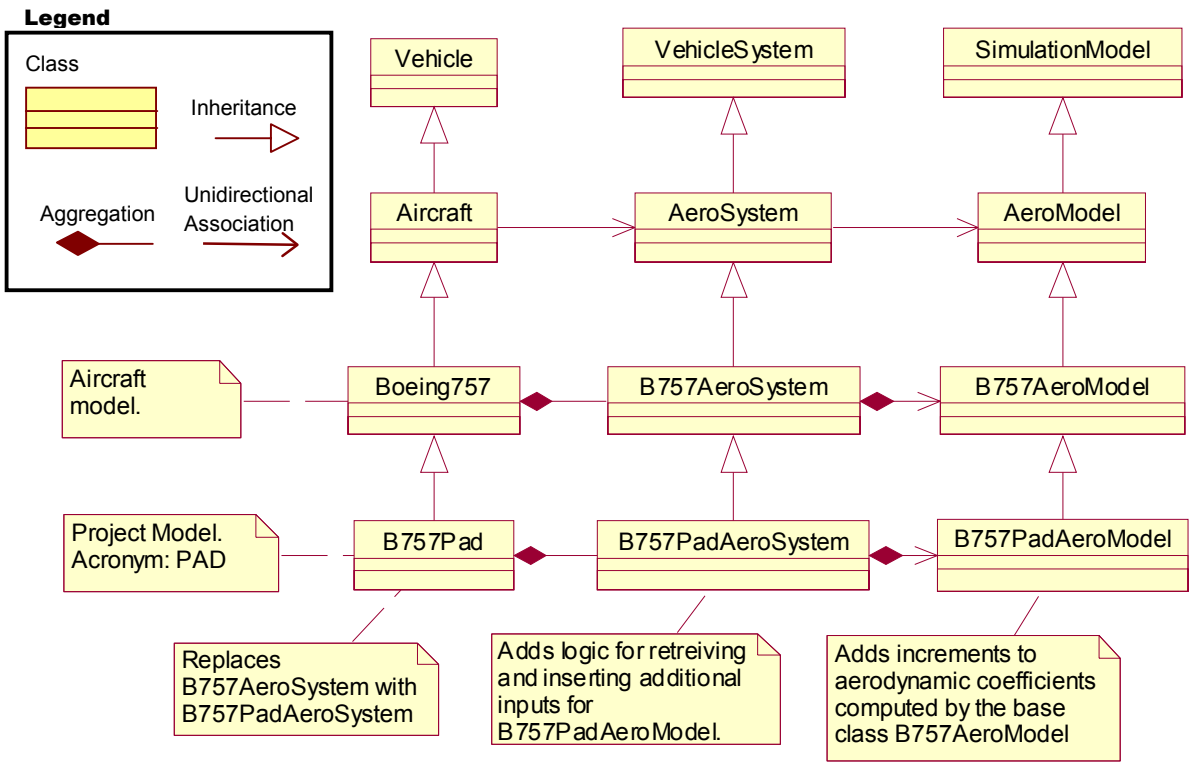

Figure 1 Aircraft Decomposition in LaSRS++

opers may have to reconcile conflicts between project changes and updates to the aircraft code. If a project creates a desirable feature (i.e., a feature that other projects want included in the AircraftModel), extracting that feature from the other project modifications could take considerable effort and testing.

When LaRC re-engineered the simulation framework to object-oriented technology, the architecture had a goal to avoid the need to copy aircraft code or to pollute it with project-specific items. This paper describes the current evolution of that architecture. The discussion employs the following conventions to keep text concise. "*ClassName* object" designates an object created from *ClassName* or its descendents. "*ClassName* derivative" indicates any class descended from *ClassName.* Method names that appear in *italics* are abstract interfaces that must be defined by descendent classes. Method names that appear in **bold** are concrete definitions of an abstract interface. The diagrams use the Unified Modeling Language (UML) notation.[1]

AIRCRAFT DECOMPOSITION

SDB builds all of its simulation products using an object-oriented framework called the Langley Standard Real-Time Simulation in C++ (LaSRS++).[2] All aircraft models derive from classes within LaSRS++. Figure 1 illustrates the structure of LaSRS++ aircraft[§]. An aircraft model is composed of three major parts:

1. Subsystem models descended from SimulationModel. Aerodynamic models and engine models are examples of subsystem models.
2. Mediator classes descended from VehicleSystem. VehicleSystems follow the mediator design pattern to decouple subsystem models from other parts of the aircraft.[3,4] VehicleSystems construct the subsystem model and handle I/O for the subsystem model. Cunningham covers the relationship between VehicleSystem derivatives and subsystem models in more detail.[4] In figure 1, B757Aero-

[§] The figure is a simplified representation of the architecture for the purposes of discussion. Some intermediate classes and some relationships are not shown.

System descends from VehicleSystem. It creates the B757AeroModel object. It retrieves, from various sources, the inputs to B757AeroModel. It feeds these inputs into the B757AeroModel object and executes the model. Lastly, the B757AeroSystem makes the model's outputs available to objects (e.g. other VehicleSystems).

3. The composite class descended from Vehicle. This paper will refer to the composite class as AircraftModel. The AircraftModel represents the entire aircraft. AircraftModel aggregates the VehicleSystem objects that, in combination, model the given aircraft. This paper covers the evolution of the relationship between Vehicle objects and VehicleSystem objects that enable verbatim reuse of AircraftModel while maintaining separation between aircraft code and experiment logic. In figure 1, Boeing757 is the AircraftModel.

When a project extends the aircraft model for an experiment, the project begins by deriving a class from the AircraftModel. This paper will refer to the derived class as the ProjectModel. The ProjectModel starts with the default composition of VehicleSystem objects defined by the AircraftModel. The ProjectModel can then modify this composition as required for the experiment. The ProjectModel has three basic options to change the composition: add, replace, or subtract VehicleSystem objects.

Figure 1 uses a fictitious project, the Porous Airfoil Demonstrator (PAD). PAD will evaluate the effects of adding passive porosity to a Boeing 757. PAD has modeled porosity effects as increments to the aerodynamic coefficients. The project team begins by deriving a B757PadAeroModel class from B757AeroModel. B757PadAeroModel inherits the aerodynamic properties of an unmodified Boeing 757. The class then adds the computations for modeling porosity and sums the increments with the basic Boeing 757 coefficients. The team next creates the B757Pad and B757PadAeroSystem classes. B757Pad represents the tailored aircraft model. It modifies the Boeing757 model by replacing the default B757AeroSystem with the B757PadAeroSystem. B757PadAeroSystem creates the B757PadAeroModel. B757PadAeroSystem derives from B757AeroSystem because the class reuses all of the logic from B757AeroSystem that feeds inputs into the base class portion of B757PadAeroModel, i.e. B757AeroModel. Yet, B757PadAeroSystem must provide any additional inputs for the passive porosity model. B757PadAeroSystem may provide additional outputs if the porosity increments to aerodynamic coefficients are required elsewhere[¶].

The example shows that the LaSRS++ architecture uses inheritance to reuse and extend behavior in all three major subdivisions of an aircraft model: the vehicle, the vehicle system, and the subsystem model. Inheritance acts as a substitute for conditional statements that were used in the procedural paradigm. Inheritance leaves the AircraftModel code unchanged. All code additions for the experiment are isolated in the derived classes. Experiment code is incorporated into the product only if the derived classes are linked into the product.

GENERALIZING AIRCRAFT BEHAVIOR

The aircraft decomposition in LaSRS++ is not sufficient to avoid copy and tailor. In earlier LaSRS++ designs, the AircraftModel and ProjectModel were responsible for operating their composition of VehicleSystem objects. The AircraftModel would contain the logic for operating the default composition of VehicleSystem objects. The ProjectModel, in changing the composition of VehicleSystem objects, would have to provide modified operational logic. In these earlier designs, projects overrode the AircraftModel logic with a tailored copy that was placed in the ProjectModel. Thus, the "copy and tailor" method of reuse crept into the normal process of creating ProjectModels.

Removing "copy and tailor" from the process of creating ProjectModels required architectural enhancements to the current decomposition. The early projects were examined for emerging patterns of VehicleSystem operation[#]. Five basic actions were identified:

[¶] Many LaSRS++ designers choose to make the outputs available by providing a constant reference to the underlying model. In this example, B757PadAeroSystem would provide a constant reference to B757Pad-AeroModel.

[#] Initialization actions were also identified and mechanized. Initialization is not covered due to lack of space

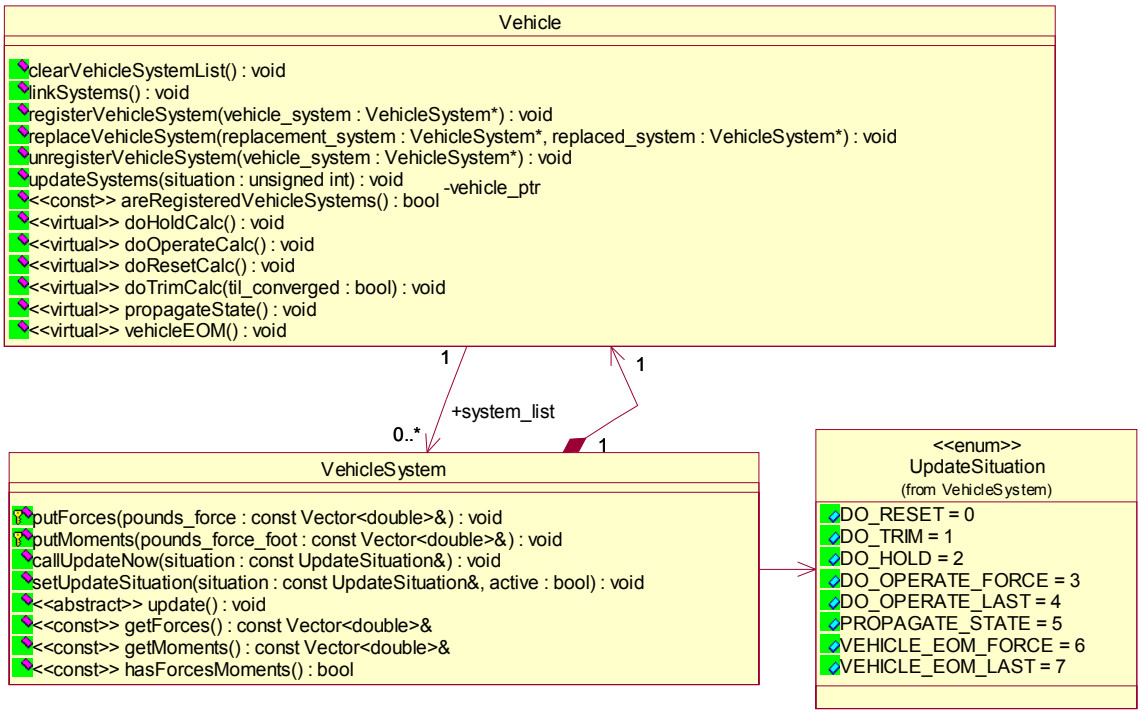

Figure 2 Detailed Vehicle-VehicleSystem Design

- The operational code must execute the VehicleSystem objects in the correct order.
- The operational code must call the update method defined by each VehicleSystem object.
- The operational code must execute each VehicleSystem object under the correct circumstances. LaSRS++ has different modes of execution (e.g. RESET, HOLD, OPERATE). It may be incorrect for a VehicleSystem to update in each mode. For example, does the object execute in RESET mode as well as OPERATE mode?
- The operational code must link VehicleSystem objects to the correct input sources. As stated earlier, VehicleSystem objects act as mediators for a companion subsystem model.
- The operational code must acquire VehicleSystem outputs that affect behavior of the vehicle.

The first three actions can be categorized as execution, the fourth deals with inputs, and the last deals with outputs. The design enhancements captured these actions as new abstractions, i.e. new attributes and operations in the VehicleSystem and Vehicle classes. The purpose of the new abstractions was the development of generic, reusable operation code that could be pushed into the LaSRS++ framework. Developers would no longer create operational code in the AircraftModel and ProjectModel classes. The primary responsibilities of AircraftModel and ProjectModel would reduce to defining the VehicleSystem composition. The new design would remove the need to "copy and tailor" while simplifying the work in creating new ProjectModels.

Figure 2 illustrates the new design. The Vehicle base class manages a list of the VehicleSystem objects that define the model. This list is named the system_list. Vehicle contains methods that manipulate its system_list and that execute behaviors for all items on the list. These methods are identified and described in the sections that follow. Vehicle also contains generic operational code for operating the VehicleSystems in the various simulation modes. These methods begin with the prefix ‘do’, e.g. doResetCalc(). VehicleSystem con-

---

and because the mechanisms are similar to those described for operation.

tains an abstract interface for the services that are required by the generic operational code in Vehicle. The next five sections look at how the design addresses each of operational actions.

Order of Execution (Execution)

The order in which VehicleSystem objects execute is important. Some VehicleSystem objects provide inputs to other VehicleSystem objects. For example, the B757AeroSystem requires control surface inputs for the B757AeroModel. The control surface inputs are computed when B757ControlSystem is executed. In LaSRS++, the B757ControlSystem must execute before the B757AeroModel. The Vehicle class equates order of execution with order of registration. When the Vehicle object constructs each VehicleSystem object, the Vehicle object registers the VehicleSystem object using Vehicle::registerVehicleSystem(). This method adds the VehicleSystem object to the end of system_list. The Vehicle object will execute the VehicleSystem objects in the order on the system_list.

The registerVehicleSystem() calls made by the AircraftModel class represent the default execution order for all variations of that aircraft. The Vehicle class provides methods that allow the ProjectModel to change the default registration order. If the ProjectModel wants to replace a VehicleSystem object created by the AircraftModel, it calls Vehicle::replaceVehicleSystem(). In the example, B757Pad would call Vehicle::replaceVehicleSystem() to replace the B757AeroSystem created by its parent, Boeing757, with B757PadAeroSystem. B757PadAeroSystem will now be executed at the same position in the system_list that was occupied by B757AeroSystem.

If the ProjectModel needs to reorder the list, then the ProjectModel can call Vehicle::clearVehicleSystemList() to empty the list and call Vehicle::registerVehicleSystem() to add VehicleSystem objects back onto the list in the desired order. Being a child of the AircraftModel, the ProjectModel has access to all of the VehicleSystem objects created by the AircraftModel. Therefore, it can combine the AircraftModel's VehicleSystem objects with its own VehicleSystem objects when it reorders the list.

Invoke the Correct Behavior (Execution)

Each VehicleSystem object has unique behaviors. But, in simple terms, they perform the same action: populate the subsystem model inputs and execute the subsystem model. Thus, the same meaningful name can be used to describe the action for all VehicleSystem objects. The design defines an abstract method for invoking the action, *VehicleSystem::update()*. Each VehicleSystem derivative defines that method to execute its own unique behaviors. This object-oriented mechanism is called polymorphism. When the client code calls the *update()* method, the actions that are actually taken depend on the object upon which the method is invoked. This allows generic code to invoke the same action on a collection of objects through its common ancestor but allows each object to uniquely respond. The Vehicle's generic code does not directly call this method, but it plays an important role in abstracting "Operational Situations" described next.

Operational Situations (Execution)

LaSRS++ has several operational modes. Within each of these modes, there may be different points, at which VehicleSystems are run. These points are called "operational situations" in this paper. The VehicleSystem design captures these operational situations in the enumeration VehicleSystem::UpdateSituation. The operational situations for LaSRS++ are defined in Table 1.

The VehicleSystem object defines the operational situations in which it runs by calling VehicleSystem::setUpdateSituation(). When each operational situation is encountered, the generic operational code in Vehicle calls the callUpdateNow() method on every registered VehicleSystem object. The code passes an UpdateSituation value as an argument to VehicleSystem::callUpdateNow(). Each VehicleSystem object compares the situation against those it has defined. If there is not a match, the VehicleSystem object returns without taking an action. If there is a match, the object runs the abstract *update()* method. This is where the polymorphic *VehicleSystem::update()* plays its role. VehicleSystem::callUpdateNow() is generic, non–polymorphic code because it relies on the abstract interface *VehicleSystem::update()* to execute the behavior appropriate to the object.

Table 1: Operational Situations	
Operational Situations	**Description**
DO_RESET	RESET mode after the scenario has been defined
DO_TRIM	TRIM mode after the equilibrium computation for the frame is complete
DO_HOLD	HOLD mode
DO_OPERATE_FORCE	The force computation in OPERATE mode
DO_OPERATE_LAST	After the acceleration computation in OPERATE mode
DO_PROPAGATE_STATE	After the integration of the Vehicle states while in OPERATE mode.
VEHICLE_EOM_FORCE	During the force computation when in the equilibrium computation of TRIM mode or the derivative computation of LINEAR_MODEL mode.
VEHICLE_EOM_LAST	After the acceleration computation when in the equilibrium computation of TRIM mode or the derivative computation of LINEAR_MODEL mode.

If a ProjectModel wants to add or subtract an operational situation for a VehicleSystem object, it calls VehicleSystem::setUpdateSituation(). This method takes two arguments, an UpdateSituation value and a Boolean that adds the situation when true and subtracts it when false.

In the example, B757AeroSystem sets the following operational situations on itself: DO_OPERATE_FORCE and VEHICLE_EOM_FORCE. When the simulation is running in OPERATE mode, Vehicle::doOperate() first computes the external forces and moments on the vehicle. It calls updateSystems(DO_OPERATE_FORCE). This method, in turn, calls callUpdateNow(DO_OPERATE_FORCE) on every VehicleSystem object. The B757AeroSystem will respond by executing B757AeroModel. Before Vehicle::doOperate() exits, it will call updateSystems(DO_OPERATE_LAST). B757AeroSystem will respond by performing no actions because it has not defined DO_OPERATE_LAST as an operational situation for itself.

One weakness to this design is that it assumes the same execution order holds for all operational situations. This has been true for all of the AircraftModels and ProjectModels that SDB has created. If an aircraft or a project must use a different execution orders for each operational situation, the operational methods (doReset(), doOperateCalc(), etc.) remain virtual so that their behaviors can be overridden by an AircraftModel or ProjectModel. The VehicleSystem design could be enhanced to add different execution orders for different operational situations. But, SDB decided that the additional complexity was unwarranted for a feature that might never be exercised.

Establishing Communications (Input)

The major role of the VehicleSystem object is as an agent for the SimulationModel object that handles the SimulationModel object's I/O. The VehicleSystem object knows the source of each input to the SimulationModel object. It feeds those inputs into the SimulationModel object and then executes the model. This keeps the SimulationModel class decoupled from other parts of the ProjectModel. The SimulationModel class can more easily be unit tested in isolation and can be reused in other ProjectModels.[4]

The VehicleSystem object must build connections to those input sources. The connection usually takes the form of a pointer or reference to the source. Such links could be established by passing the source reference as an argument to the VehicleSystem derivative's constructor. However, all sources of a VehicleSystem object must be constructed prior to the VehicleSystem object for the references to be valid. The result imposes a construction order based on association. It also assumes that there is a possible construction order, in which all input sources for each VehicleSystem object will be valid. This is not the case when VehicleSystem objects associate bidirectionally or when a circular association exists among a collection of VehicleSystem objects. For example, B757AeroSystem receives inputs from B757ControlSystem; and B757ControlSystem receives inputs from B757AeroSystem. To accommodate all possible association patterns, establishing com-

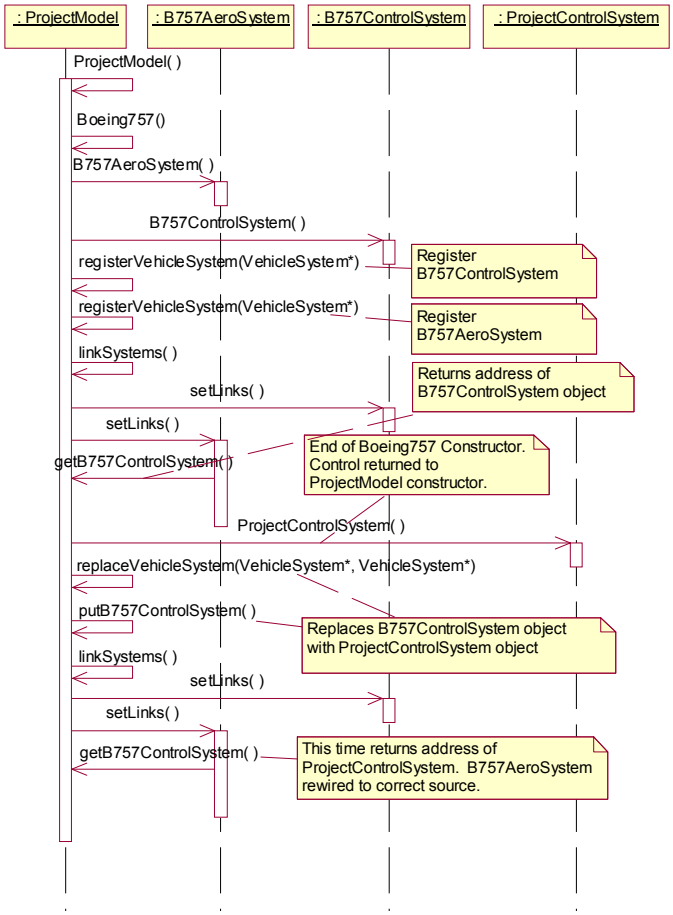

Figure 3 Establishing and Rewiring Connections

munication must be broken out as a distinct behavior from composition.

The VehicleSystem class provides the abstract method *setLinks()* for this purpose. The VehicleSystem derivative defines **setLinks()** to establish the connections to other VehicleSystem objects that provide the input data for its companion model. The AircraftModel and ProjectModel objects maintain accessors[**] to each VehicleSystem object they create. The concrete **setLinks()** calls these accessors to obtain the required sources. The Vehicle class provides the linkSystems() method to invoke *setLinks()* on each object that appears on the system_list.

[**] An accessor is a method that returns a class attribute. In this case, the accessor is returning a pointer or reference to a VehicleSystem object contained by the class.

For example, the B757AeroModel needs the control surface deflections to compute the aerodynamic forces. The control surface deflections are stored in B757ControlSystem. The Boeing757 constructs both the B757ControlSystem and B757AeroSystem. After it has registered the systems, it calls Vehicle::linkSystems(), which calls *setLinks()* for each system on its list. **B757AeroSystem::setLinks()** calls Boeing757::getControlSystem() to obtain and store a reference to the B757ControlSystem. B757AeroSystem will use this reference in its **update()** method to retrieve the control surface deflections and insert them into the B757Aero-Model object.

The design can also handle any necessary "rewiring" when the ProjectModel replaces a VehicleSystem object inherited from the AircraftModel. Figure 3 provides an example. Suppose a project creates an extended control system to replace the B757ControlSystem. This ProjectControlSystem would derive from B757ControlSystem. The Boeing757 class constructor will connect the B757AeroSystem to the B757ControlSystem. In replacing B757ControlSystem, the ProjectModel will want B757AeroSystem to receive its inputs from the ProjectControlSystem. Under early LaSRS++ designs, the ProjectModel would contain code that replaces the value of B757AeroSystem's pointer to the B757ControlSystem with the address of the ProjectControlSystem. Under the new design, AircraftModel allows the ProjectModel to replace the values of its VehicleSystem pointers with the address of project objects. The ProjectModel constructor would call Boeing757::putB757ControlSystem(&ProjectControlSystem). Then Boeing757::getB757ControlSystem() would return the address of ProjectControlSystem. How the implementation accomplishes the rewiring is more complex than this call. The ProjectModel constructor starts by calling the Boeing757 constructor. The Boeing757 constructor creates and registers the B757ControlSystem and B757AeroSystem objects. The constructor then calls Vehicle::linkSystems(), which invokes **B757AeroSystem::setLinks()**. **B757AeroSystem::setLinks()** calls Boeing757::getB757ControlSystem(), which returns the address of the

B757ControlSystem object. This is not the desired result. However, control is later returned to the ProjectModel constructor. The ProjectModel constructor creates the ProjectControlSystem object. It calls Vehicle::replaceVehicleSystem() to replace the B757ControlSystem with the ProjectControlSystem. It also calls Boeing757::putB757ControlSystem() to replace the address of the B757ControlSystem object with the address of the ProjectControlSystem object. Then, the constructor calls Vehicle::linkSystems(). **B757AeroSystem::setLinks()** will be re-invoked. When Boeing757::getB757ControlSystem() is called this time, the address of ProjectControlSystem is returned. The necessary rewiring is done. In other words, the calling the ProjectModel constructor results in two passes of Vehicle::linkSystems(). The first pass sets up links as defined by the AircraftModel. The second pass allows the project to redefine the links.

Influencing Vehicle Behavior (Output)

A large amount of data interaction can occur in an aircraft model. As explained in the prior section, the VehicleSystem handles the interactions between subsystems. Only the interaction between the subsystem model and the Vehicle object remains. The Vehicle class embodies the basic equations of motion. Thus, the only subsystem outputs that concern the Vehicle are inputs into the equations of motion. These are mass properties[††], forces, and moments.

LaSRS++ treats mass properties as a special subsystem that is required of all concrete descendents. The Vehicle class has, as an attribute, a pointer to a MassPropertiesSystem. It uses this pointer directly to retrieve mass properties when computing accelerations. Vehicle derivatives can set this pointer by calling Vehicle::putMassPropertiesSystem().

Forces and moments, on the other hand, can potentially be produced by any subsystem. Typically, more than one subsystem contributes. In early LaSRS++ designs, retrieving and summing the force and moment contributions from the subsystems was done as the definition of the abstract method *Vehicle::forcesMoments()*. Like the operational methods, explicit calls to the subsystem models' force and moment accessors were placed here. If the ProjectModel added or replaced subsystems that produced forces and moments, the ProjectModel had to copy **AircraftModel::forcesMoments()** and make modifications.

The updated design mechanizes the collection of forces and moments. The VehicleSystem class defines standard methods for derived classes to set a force and a moment. These are VehicleSystem::putForces() and VehicleSystem::putMoments(). The class also provides companion accessors that allow the Vehicle object to retrieve forces and moments. These are VehicleSystem::getForces() and VehicleSystem::getMoments(). In the example, the B757AeroSystem communicates the aerodynamic forces and moments by adding calls to VehicleSystem::putForces() and VehicleSystem::putMoments() in its **update()** method.

PUTTING IT ALL TOGETHER

The mechanization of each action has been detailed. But, how has the design improved overall aircraft modeling and project extensions? In earlier versions of LaSRS++, aircraft model construction would involve the following steps:

1. The subsystem developer creates the subsystem class and its VehicleSystem companion.
2. The vehicle integration developer creates the AircraftModel class with a shell constructor, destructor, operational methods [e.g. **doReset()**], and **forcesMoments()**.
3. In the AircraftModel constructor, add code for constructing the VehicleSystem object and for linking the VehicleSystem object to its sources.
4. In each appropriate AircraftModel operational method, add code to call the VehicleSystem object's execution method.
5. In **forcesMoments()**, add explicit calls to those VehicleSystem objects that produces forces and moments.
6. Combine the code changes for each of the VehicleSystem objects and verify placement and order of VehicleSystem execution calls in each AircraftModel operational method.

Because adding a VehicleSystem to the AircraftModel required changes to numerous AircraftModel methods,

[††] Mass, center of gravity, and moments of inertia.

creating the AircraftModel required significant collaboration among the subsystem developers and the integration developer. Either the subsystem developers made their changes directly to the AircraftModel and the integration developer merged them together or the integration developer added the code for one VehicleSystem object at a time in consultation with each subsystem developer.

Since each AircraftModel defined their operational methods, small variations in the operational methods did appear between AircraftModels[‡‡]. These variations sometimes caused confusion for developers as they moved from one AircraftModel to the next. This confusion sometimes manifested itself in code defects.

To extend the AircraftModel in early versions of LaSRS++, the Project Model would perform the following steps:

1. The subsystem developer creates the extended or additional subsystem and its VehicleSystem companion.
2. The vehicle integration developer creates a ProjectModel with a shell constructor and destructor.
3. The integration developer copies the AircraftModel's operational methods that will be extended. The integration developer also copies **forcesMoments()** if applicable.
4. In the ProjectModel constructor, add code to construct the new VehicleSystem objects and to link those new objects to their sources.
5. Add code to rewire source inputs for inherited VehicleSystem objects if necessary.
6. In each operational method, add/replace VehicleSystem execution calls as appropriate.
7. In **forcesMoments()**, add/replace accessors to VehicleSystem forces and moments as appropriate.
8. Combine the code changes for each of the VehicleSystem objects and verify placement and order of VehicleSystem execution calls in each ProjectModel operational method.

Whenever the operational method in the AircraftModel changes, the ProjectModel will have to replicate the change. Like the AircraftModel, ProjectModel creation required a significant collaborative effort among the subsystem developers and integration developer.

The updated design introduces generic operational code that can be moved into the Vehicle class. Figure 4 shows the code for Vehicle::doOperateCalc(). The aircraft and project teams are freed from writing operational code. This provides a very modest code savings of ~200 LOC for each aircraft and project. The greater benefits lie in the standardization of operation and in releasing the projects from duplicating the aircraft's operational code. Every aircraft and project now operates in the exact same manner. Defects rooted in confusion about operational differences disappear. Projects no longer copy the aircraft's operational code. Therefore, they no longer need to be concerned with migrating AircraftModel updates into the ProjectModel.

The updated design confines the work of defining the aircraft or project model to the constructor. Aircraft and project models are defined by the composition of the VehicleSystem objects that they create. AircraftModel construction now comprises the following steps:

1. The subsystem developer creates the subsystem class and its VehicleSystem companion. The developer codes the connections to input sources in the **setLinks()** method. The developer specifies the execution situations by making setUpdateSituation() calls in the constructor. If the VehicleSystem derivative produces forces, the developer adds putForces() and putMoments() calls in the **update()** method.
2. The vehicle integration developer creates the AircraftModel with a constructor and destructor.
3. In the constructor, the integration developer creates each VehicleSystem derivative. Then, the developer registers each one in the order of their execution by calling Vehicle::registerVehicleSystem(). Afterward, the developer adds a call to Vehicle::linkSystems(), which establishes the input connections for each VehicleSystem object.

Adding a VehicleSystem object to the AircraftModel now requires two lines, a creation line and a registration line in the constructor. The vehicle integration developer can do this alone. Only the order of registration needs to be verified. The updated design has more

[‡‡] This was particularly true of the *doReset()* method. Differences in how AircraftModels handled scenario definition and initialization were prevalent.

```
// This method updates all vehicle systems marked for update in the current situation.
void Vehicle::updateSystems(unsigned int situation)
{
  list<VehicleSystem*>::iterator iterator;
  for(iterator = system_list->begin(); iterator != system_list->end(); iterator++)
  {
    (*iterator)->callUpdateNow(static_cast<VehicleSystem::UpdateSituation>(situation));
  }
}

// The method immediately returns if the vehicle is operating off of a playback file.
// Otherwise, the method retrieves cockpit inputs.  It operates VehicleSystems marked for
// update prior to summing the forces and moments.  It computes the forces, moments, and
// accelerations.  Then, it operates VehicleSystems marked for update after the computation
// of accelerations.
void Vehicle::doOperateCalc()
{
  // If a playback file determines the vehicle's behavior, exit.
  if ( getPositionalModelPlayback()->isPlayback() ) return;


  // If a vehicle limit was hit in an earlier frame, the vehicle must return to
  // RESET mode before it can operate again.
  if (vehicle_limits && vehicle_limits->limitViolation()) return;

  processCockpitInputs();                          // Read cockpit inputs
  updateSystems(VehicleSystem::DO_OPERATE_FORCE); // Calculate forces and moments
  forcesMoments();                                 // Sum forces and moments.
  calcAccel();                                     // Calculate accelerations.

  // Update systems that must operate between acceleration calculation and integration.
  updateSystems(VehicleSystem::DO_OPERATE_LAST);
}
```

Figure 4 Example of Generic Operational Code

cleanly separated integration from local VehicleSystem derivative concerns. Order of execution is the only operational decision that cannot be made before integration because it depends on the set of VehicleSystem objects. All other operational decisions can be made during the design of the VehicleSystem derivative. In the updated design, the subsystem developer captures operational decisions in the VehicleSystem derivative without a need to coordinate work with other subsystem developers.

The updated design also simplifies ProjectModel creation:

1. The subsystem developer creates the subsystem class and its VehicleSystem companion. The developer codes the operational decisions as described in first step for the AircraftModel.
2. The vehicle integration developer creates the ProjectModel with a constructor and destructor.
3. In the constructor each VehicleSystem object is created. If the object is an addition to the AircraftModel, registerVehicleSystem() is called. If the object is intended as a replacement, replaceVehicleSystem() is called and the appropriate mutator is called to change the replaced VehicleSystem object's address with that of its replacement. Vehicle::linkSystems() is called to establish new connections and rewire old connections.

Except for the decision of whether the new VehicleSystem is an addition or replacement, creation of the ProjectModel differs little from creation of the AircraftModel. It is simpler than the old design. It also does not require code copying. ProjectModel construction also retains the clean separation of integration decisions from operational decisions that are local to the VehicleSystem derivative.

FUTURE WORK

The VehicleSystem design could be expanded further to add multi-rate and load balancing features. Multi-rate could encompass both super-rating (execute more than once per frame) and sub-rating (execute once per integral multiple of frames). The VehicleSystem could contain attributes that specify its frequency of operation. The generic code in the Vehicle class could use this information to determine whether and how many times

to run a VehicleSystem object at each execution event. The Vehicle class could also load-balance VehicleSystem objects across frames if desired. The load-balancing algorithm could be incorporated within registerVehicleSystem() or as a separate step.

CONCLUSIONS

Part of the evolution of LaSRS++ has been a focus on reducing the amount of duplication required to produce aircraft models and project models. LaSRS++ employs inheritance to reuse and extend aircraft models for experiments. Inheritance replaces the use of conditional statements or pre-processor directives that pollute the aircraft code with project-specific extensions. However, "copy and tailor" of operational code remained part of the extension method for projects in early LaSRS++ designs. The previous VehicleSystem design described by Cunningham has been expanded to mechanize the five operational decisions for subsystem models (i.e. the VehicleSystem objects): invoke the correct behavior, execute in the correct order, execute at the right event, establish communications, and influence vehicle behavior.[4] Defining mechanisms for these actions have produced two benefits.

1. The mechanisms allowed the creation of generic operational code in the Vehicle class that is applicable to the majority of aircraft and project models. Developers have been freed from writing operational code. Vehicle development has mostly been reduced to coding and composing VehicleSystem objects. Moreover, the generic code creates a standard for vehicle model operation.
2. Work is more cleanly divided between subsystem creation and integration. Of the five operational decisions, only order of execution must be established during integration. Subsystem developers can encode the other four decisions in the VehicleSystem derivative. Thus, a vehicle integration developer could compose the aircraft/project model from the VehicleSystem objects without knowing the details of these objects. The vehicle integration developer only needs to know the order of execution. Developers do not have to coordinate the coding of any single class, in particular the AircraftModel or ProjectModel class. This reduces the possibility of errors from conflicting code changes.

In its evolution, the LaSRS++ architecture has reduced project creation to "extend and compose". Projects first use inheritance and polymorphism to extend aircraft model classes with modified behaviors. Then, the projects define themselves through their composition of aircraft and project-extended components.

BIBLIOGRAPHY

1. Douglas, B. Real-Time UML. Addison-Wesley, Reading, Massachusetts. 1999. ISBN 0-201-65784-8.
2. Leslie, R.; et. al. LaSRS++: An Object-Oriented Framework for Real-Time Simulation of Aircraft. AIAA Modeling & Simulation Technologies Conference, Boston, August 1998, AIAA-98-4529.
3. 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.
4. Cunningham, K. Use of the Mediator Design Pattern in the LaSRS++ Framework. AIAA Modeling & Simulation Technologies Conference, Portland, August 1999, AIAA-99-4336.