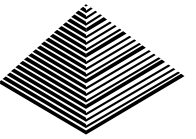
8



Flight Simulation

A Case Study in an Architecture for Integrability

The striking conclusion that one draws . . . is that the information processing capacity of [flight simulation computation] has been increasing approximately exponentially for nearly thirty years. There is at this time no clear indication that the trend is changing.
— Laurence Fogarty [Fogarty 67]

Modern flight simulators are among the most sophisticated software systems in existence. They are highly distributed, have rigorous timing requirements, and must be amenable to frequent updates to maintain high fidelity with the ever-changing vehicles and environment they are simulating. The creation and maintenance of these large systems presents a substantial software development challenge in designing for the following:

- Hard real-time performance
- Modifiability, to accommodate changes in requirements and to the simulated aircraft and their environments
- Scalability of function, a form of modifiability, needed to extend these systems so that they can simulate more and more of the real world and further improve the fidelity of the simulation

But, as the title of this chapter makes clear, an overriding concern was designing for *integrability*—a system quality attribute not covered in Chapter 4 but often arising as a driving concern in large systems, especially those developed by distributed teams or separate organizations. Integrability refers to the ease with which separately developed elements, including those developed by third parties, can be made to work together to fulfill the software's requirements. As with other quality attributes, architectural tactics can be brought to bear to achieve integrability (some of which are also aimed at modifiability). These tactics include keeping interfaces small, simple, and stable; adhering to defined protocols; loose

coupling or minimal dependencies between elements; using a component framework; and using "versioned" interfaces that allow extensions while permitting existing elements to work under the original constraints.

This chapter will discuss some of the challenges of flight simulation and discuss an architectural pattern created to address them. The pattern is a *Structural Model*, and it emphasizes the following:

- Simplicity and similarity of the system's substructures
- Decoupling of data- and control-passing strategies from computation
- Minimizing module types
- A small number of system-wide coordination strategies
- Transparency of design

These principles result in an architectural pattern that, as we will see, features a high degree of integrability as well as the other quality attributes necessary for flight simulation. The pattern itself is a composite of more primitive patterns.

8.1 Relationship to the Architecture Business Cycle

The segment of the Architecture Business Cycle (ABC) that connects desired qualities to architecture is the focus of this case study. Figure 8.1 shows the ABC for Structural-Model-based flight simulators. The simulators discussed in this chapter are acquired by the U.S. Air Force. Their end users are pilots and crews for the particular aircraft being simulated. Flight simulators are used for pilot training in the operation of the aircraft, for crew training in the operation of the various weapons systems on board, and for mission training for particular missions for the aircraft. Some simulators are intended for standalone use, but more and more are intended to train multiple crews simultaneously for cooperative missions.

The flight simulators are constructed by contractors selected as a result of a competitive bidding process. The simulator systems are large (some as large as 1.5 million lines of code), have long lifetimes (the aircraft being simulated often have lifetimes of 40 years or longer), and have stringent real-time and fidelity requirements (the simulated aircraft must behave exactly like the real aircraft in situations such as normal flight, emergency maneuvers, and equipment failures).

The beginning of the Structural Model pattern dates from 1987 when the Air Force began to investigate the application of object-oriented design techniques. Electronic flight simulators had been in existence since the 1960s, and so this investigation was motivated by problems associated with the existing designs. These included construction problems (the integration phase of development was increasing exponentially with the size and complexity of the systems) and life-cycle problems (the cost of some modifications was exceeding the cost of the original system).

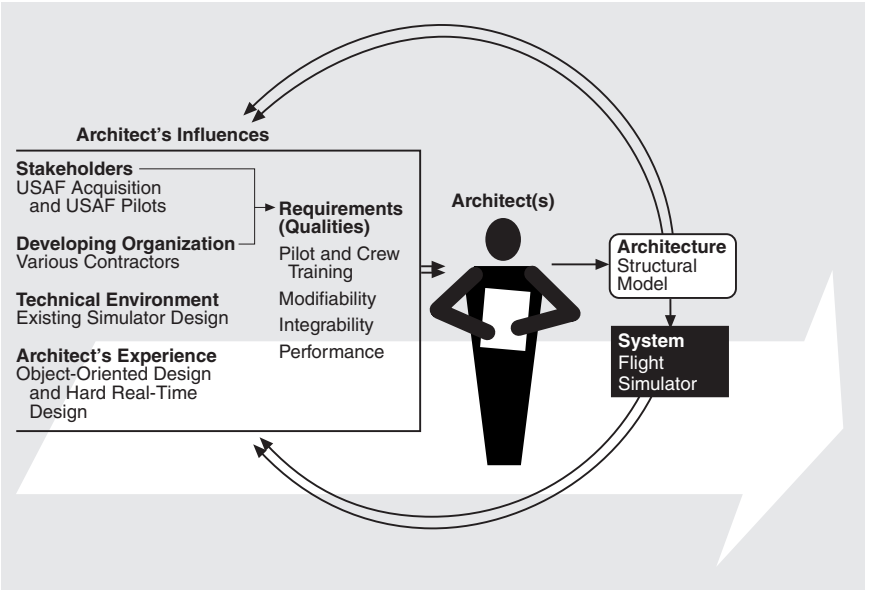


FIGURE 8.1 Initial stages of the ABC for the flight simulator

The Structural Model pattern was able to overcome these problems, as we will see. It has been used in the development of the B-2 Weapons System Trainer, the C-17 Aircrew Training System, and the Special Operations Forces family of trainers, among others.

8.2 Requirements and Qualities

There are three roles involved in a flight training simulator. The first is that of the crew being trained. They sit inside a motion platform surrounded by instruments intended to replicate exactly the aircraft being simulated, and look at visuals that represent what would be seen outside an actual aircraft. We are not going to describe the specifics of either the motion platform or the visual display generator in this chapter. They are driven by special-purpose processors and are outside the scope of the architecture we describe here. The purpose of a flight simulator is to instruct the pilot and crew in how to operate a particular aircraft, how to perform maneuvers such as mid-air refueling, and how to respond to situations such as an attack on the aircraft. The fidelity of the simulation is an important element in the training. For example, the feel of the controls when particular maneuvers are performed must be captured correctly. Otherwise, the pilot and crew are being trained incorrectly and the training may be counter-productive.

The second role associated with a flight simulator is that of the environment. Typically, the environment is a computer model, although with multi-aircraft training exercises it can include individuals other than the pilot and crew. It comprises the atmosphere, threats, weapons, and other aircraft. For example, if the purpose of the training is to practice refueling, the (simulated) refueling aircraft introduces turbulence into the (modeled) atmosphere.

The third role associated with a flight training simulator is that of the simulation instructor. Usually, a training exercise has a specific purpose and specific circumstances. During the exercise, the instructor is responsible for monitoring the performance of the pilot and crew and for initiating training situations. Sometimes these situations are scripted in advance, and other times the instructor introduces them. Typical situations include malfunctions of equipment (e.g., landing gear that does not deploy correctly), attacks on the aircraft from foes, and weather conditions such as turbulence caused by thunderstorms. The instructor has a separate console to monitor the activities of the crew, to inject malfunctions into the aircraft, and to control the environment. Figure 8.2 shows a typical collection of modern flight simulators.

FIGURE 8.2 Modern flight simulators. Courtesy of the Boeing Company.

USE OF MODELS

The models used in the aircraft and the environment are capable of being simulated to almost arbitrary fidelity. As an example of the range of fidelity, consider the modeling of the air pressure affecting the aircraft. A simple model is that the air pressure is affected only by the aircraft altitude. Somewhat more complicated is a model in which the air pressure is affected by altitude and local weather patterns. Modeling local weather patterns takes more computational power but allows consideration of updrafts and downdrafts. An even more complicated model with additional computational requirements is that the air pressure is affected by altitude, local weather patterns, and the behavior of nearby aircraft. One source of turbulence is an aircraft that has recently passed through the airspace of the aircraft being simulated.

A consequence of the capability to simulate the aircraft or the environment to almost arbitrary fidelity is that training simulators in the past always stressed the limits of computational power (and may always do so in the future). Since crew simulator training is an important portion of overall flight training, there are always strong arguments that slightly more fidelity will improve the training and hence the skill set of the crews being trained. Performance, therefore, is one of the important quality requirements for a flight simulator.

STATES OF EXECUTION

A flight simulator can execute in several states.

- *Operate* corresponds to the normal functioning of the simulator as a training tool.
- *Configure* is used when modifications must be made to a current training session. For example, suppose the crew has been training in a single-aircraft exercise and the instructor wishes to switch to mid-air refueling. The simulator is then placed into a configure state.
- *Halt* stops the current simulation.
- *Replay* uses a journal to move through the simulation without crew interaction. Among other functions, it is used to demonstrate to the crew what they have just done, because the crew may get caught up in operating the aircraft and not reflect on their actions. "Record/playback" was identified in Chapter 5 (Achieving Qualities) as an architectural tactic for testing. Here we find it used as a portion of the training process.

The simulators we discuss in this chapter are characterized by the following four properties:

1. *Real-time performance constraints.* Flight simulators must execute at fixed frame rates that are high enough to ensure fidelity. For those not familiar with frame rates, an analogy to motion pictures might be helpful. Each frame is a

snapshot in time. When a sufficient number of frames are taken sequentially within a time interval, the user sees or senses continuous motion. Different senses require different frame rates. Common simulator frame rates are 30 Hz or 60 Hz —one-thirtieth or one-sixtieth of a second. Within each frame rate, all computations must run to completion.

All portions of a simulator run at an integral factor of the base rate. If the base rate is 60 Hz, slower portions of the simulation may run at 30, 20, 15, or 12 Hz, and so on. They may not run at a nonintegral factor of the base rate, such as 25 Hz. One reason for this restriction is that the sensory inputs provided by a flight simulator for the crew being trained must be strictly coordinated. It would not do to have the pilot execute a turn but not begin to see or feel the change for even a small period of time (say, one-tenth of a second). Even for delays so small that they are not consciously detectable, a lack of coordination may be a problem. Such delays may result in a phenomenon known as *simulator sickness*, a purely physiological reaction to imperfectly coordinated sensory inputs.

2. *Continuous development and modification.* Simulators exist to train users when the equivalent training on the actual vehicle would be much more expensive or dangerous. To provide a realistic training experience, a flight simulator must be faithful to the actual air vehicle. However, whether civilian or military, air vehicles are continually being modified and updated. The simulator software is therefore almost constantly modified and updated to maintain verisimilitude. Furthermore, the training for which the simulators are used is continually extended to encompass new types of situations, including problems (malfunctions) that might occur with the aircraft and new environmental situations, such as a military helicopter being used in an urban setting.

3. *Large size and high complexity.* Flight simulators typically comprise tens of thousands of lines of code for the simplest training simulation and millions of lines of code for complex, multi-person trainers. Furthermore, the complexity of flight simulators, mapped over a 30-year period, has shown exponential growth.

4. *Developed in geographically distributed areas.* Military flight simulators are typically developed in a distributed fashion for two reasons, one technical and one political. Technically, different portions of the development require different expertise, and so it is common practice for the general contractor to subcontract portions of the work to specialists. Politically, high-technology jobs, such as simulator development, are political plums, so many politicians fight to have a piece of the work in their district. In either case, the integrability of the simulator—already problematic because of the size and complexity of the code—is made more difficult because the paths of communication are long.

In addition, two problems with flight simulators caused the U.S. Air Force to investigate new simulator designs.

1. *Very expensive debugging, testing, and modification.* The complexity of flight simulation software, its real-time nature, and its tendency to be modified regularly all contribute to the costs of testing, integrating, and modifying the software typically exceeding the cost of development. The growth in complexity (and its associated growth in cost) thus caused an emphasis for the architecture on integrability and modifiability.

One of the consequences of the growth in complexity was the increased cost of integration. For example, a large completed Air Force system (1.7 million lines of code) had greatly exceeded its budget for integration. Systems 50% larger were in concept, and they would have been prohibitively expensive. Hence, integrability emerged as a driving architectural concern.

2. *Unclear mapping between software structure and aircraft structure.* Flight simulators have traditionally been built with runtime efficiency as their primary quality goal. This is not surprising given their performance and fidelity requirements and given that simulators were initially built on platforms with extremely limited memory and processing power. Traditional design of flight simulator software was based on following control loops through a cycle. These, in turn, were motivated by the tasks that caused the loop to be activated. For example, suppose the pilot turns the aircraft left. The pilot moves the rudder and aileron controls, which in turn moves the control surfaces, which affects the aerodynamics and causes the aircraft to turn. In the simulator, a model reflects the relationship between the controls, the surfaces, the aerodynamics, and the orientation of the aircraft. In the original flight simulator architecture, this model was contained in a module that might be called Turn. There might be a similar module for level flight, another for takeoff and landing, and so forth. The basic decomposition strategy was based on examining the tasks that the pilot and crew perform, modeling the components that perform the task, and keeping all calculations as local as possible.

This strategy maximizes performance since any task is modeled in a single module (or a small collection of modules) and thus the data movement necessary to perform the calculations is minimized. The problem with this architecture is that the same physical component is represented in multiple models and hence in multiple modules. The extensive interactions among modules cause problems with both modifiability and integration. If the module that controls turning is integrated with the module that controls level flight, and a problem is discovered in the data being provided to the turning module, that same data is probably being accessed by the level flight module. For these reasons, there were many coupling effects to be considered during integration and maintenance.

The architectural pattern, called *Structural Modeling,* that resulted from the reconsideration of the problems of flight simulators will be discussed for the remainder of this chapter. In brief, the pattern includes an object-oriented design to model the subsystems and controller children of the air vehicle. It marries real-time scheduling to this object-oriented design as a means of controlling the execution order of the simulation's subsystems so that fidelity can be guaranteed.

8.3 Architectural Solution

Figure 8.3 shows a reference model for a flight simulator. The three roles we identified earlier (air vehicle, environment, and instructor) are shown interacting with the crew and the various cueing systems. Typically, the instructor is hosted on a different hardware platform from the air vehicle model. The environment model may be hosted either on a separate hardware platform or with the instructor station.

The logical division between the instructor station and the other two portions is clear. The instructor station supports the instructor's control and monitoring of the actions of the crew. The other two portions perform the simulation. The division between the air vehicle and the environment is not as clear. For example, if an aircraft launches a weapon, it is logically a portion of the air vehicle until it leaves the vehicle, at which point it becomes a portion of the environment. Upon firing, the aerodynamics of the weapon are influenced initially by the proximity of the aircraft. Thus, any modeling of the aerodynamics must remain, at least initially, tightly coupled to the air vehicle. If the weapon is always considered a portion of

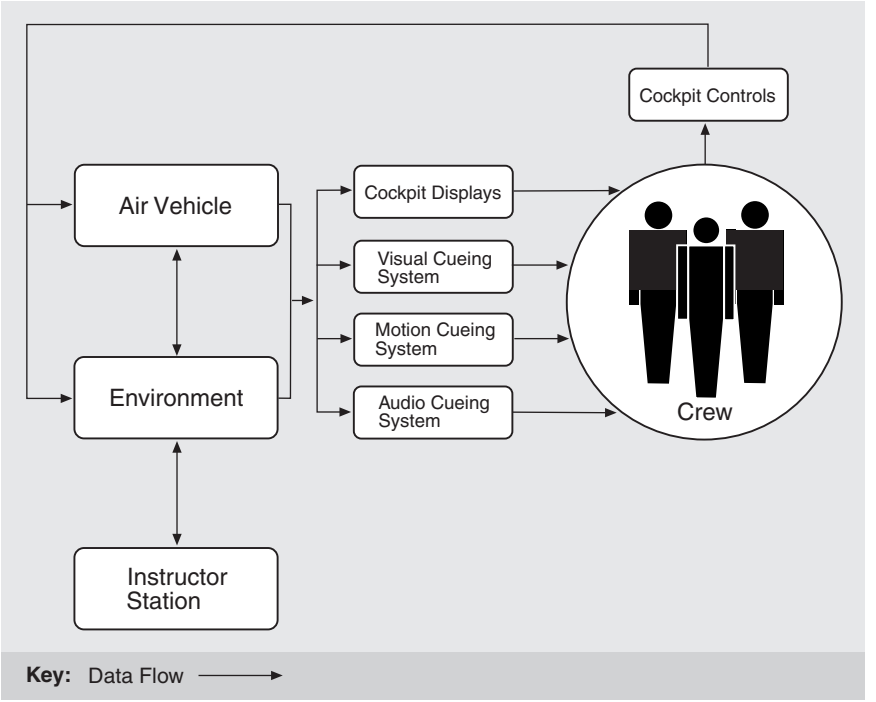


FIGURE 8.3 Reference model for flight simulator

the environment, its modeling involves tight coordination between the air vehicle and the environment. If it is modeled as a portion of the air vehicle and then handed off to the environment when fired, control of the weapon needs to be handed from one to the other.

TREATMENT OF TIME IN A FLIGHT SIMULATOR

Recall from Chapter 5 that resource management is a category of tactics to achieve performance goals. In a real-time simulator, the most important resource to manage is time itself. A flight simulator is supposed to reflect the real world, which it does by creating time-based real-world behaviors. Thus, when the pilot in a simulator activates a particular control, the simulator must provide the same response in the same time as the actual aircraft would. "In the same time" means both within an upper bound of duration after the event and within a lower bound of duration. Reacting too quickly is as bad for the quality of the simulation as reacting too slowly.

There are two fundamentally different ways of managing time in a flight simulator—periodic and event-based—and both of these are used. Periodic time management is used in portions that must maintain real-time performance (such as the air vehicle), and event-based time management is used in portions where real-time performance is not critical (such as the instructor station).

Periodic Time Management. A periodic time-management scheme has a fixed (simulated) time quantum based on the frame rate. That is the basis of scheduling the system processes. This scheme typically uses a non-pre-emptive cyclic scheduling discipline, which proceeds by iterating through the following loop:

- Set initial simulated time.
- Iterate the next two steps until the session is complete.
  - Invoke each of the processes for a fixed (real) quantum. Each process calculates its internal state based on the current simulated time and reports it based on the next period of simulated time. It guarantees to complete its computation within its real-time quantum.
  - Increment simulated time by quantum.

A simulation based on the periodic management of time will be able to keep simulated time and real time in synchronization as long as each process is able to advance its state to the next period within the time quantum allocated to it.

Typically, this is managed by adjusting the responsibilities of the individual processes so that they are small enough to be computed in the allocated quantum. It is the designer's responsibility to provide the number of processors needed to ensure sufficient computational power to enable all processes to receive their quantum of computation.

Event-Based Time Management. An event-based time-management scheme is similar to the interrupt-based scheduling used in many operating systems. The schedule proceeds by iterating through the following loop:

- Add a simulated event to the event queue.
- While there are events remaining in the event queue,
  - choose the event with the smallest (i.e., soonest) simulated time.
  - set the current simulated time to the time of the chosen event.
  - invoke a process for the chosen event. This process may add events to the event queue.

In this case, simulated time advances by the invoked processes placing events on the event queue and the scheduler choosing the next event to process. In pure event-based simulations, simulated time may progress much faster (as in a war game simulation) or much slower (as in an engineering simulation) than real time.

Mixed-Time Systems. Returning now to the scheduling of the three portions of the flight simulator, the instructor station is typically scheduled on an event basis—those events that emanate from the instructor's interactions—and the air vehicle model is scheduled on a periodic basis. The environment model can be scheduled using either regime. Thus, the coupling between the air vehicle and the environment may involve matching different time regimes.

Flight simulators must marry periodic time simulation (such as in the air vehicle model) with event-based simulation (such as in the environment model, in some cases) and with other event-based activities that are not predictable (such as an interaction with the instructor station or the pilot setting a switch). Many scheduling policies are possible from the perspective of each process involved in this marriage.

A simple policy for managing events within a periodically scheduled processor is that periodic processing must occur immediately after a synchronization step and complete before any aperiodic processing. Aperiodic processing proceeds within a bounded interval, during which as many messages as possible will be retrieved and processed. Those not processed during a given interval must be deferred to subsequent intervals, with the requirement that all messages be processed in the order received from a single source.

Communication from the portions of the system managed on an event basis to the portions managed using periodic scheduling appears as aperiodic and is scheduled as just discussed. Communication from the portions of the system managed using periodic schedule appears as events to the portions managed on an event basis.

Given this understanding of managing time in a flight simulator, we can now present the architectural pattern that handles this complexity. This pattern is for the air vehicle, and so the time management discussion is from the air vehicle's perspective.

THE STRUCTURAL MODEL ARCHITECTURAL PATTERN

Structural Model is an architectural pattern, as we defined it in Section 2.3. That is, it consists of a collection of element types and a configuration of their coordination at runtime. In this section, we present the Structural Model pattern and discuss the considerations that led to its design. Recall that the air vehicle model itself may be spread over several processors. Thus, the elements of the air vehicle structural model must coordinate internally across processors as well as with the environment model and the instructor portions of the simulation running on (potentially) different processors.

The constituents of the Structural Model architectural pattern are, at the coarsest level, the *executive* and the *application*.

- The *executive* portion handles coordination issues: real-time scheduling of subsystems, synchronization between processors, event management from the instructor–operator station, data sharing, and data integrity.
- The *application* portion handles the computation of the flight simulation: modeling the air vehicle. Its functions are implemented by subsystems and their children.

First we will discuss the air vehicle's executive modules in detail and then return to a discussion of its application modules.

MODULES OF THE AIR VEHICLE MODEL EXECUTIVE

Figure 8.4 shows the air vehicle structural model with the executive pattern given in detail. The modules in the executive are the *Timeline Synchronizer*, the *Periodic Sequencer*, the *Event Handler*, and the *Surrogates* for other portions of the simulator.

Timeline Synchronizer. The timeline synchronizer is the base scheduling mechanism for the air vehicle model. It also maintains the simulation's internal notion of time. The other three elements of the executive—the periodic sequencer, the event handler, and the surrogates—all must be allocated processor resources. The timeline synchronizer also maintains the current state of the simulation.

The timeline synchronizer passes both data and control to the other three elements and receives data and control from them. It also coordinates time with other portions of the simulator. This can include other processors responsible for a portion of the air vehicle model which have their own timeline synchronizers. Finally, the timeline synchronizer implements a scheduling policy for coordinating both periodic and aperiodic processing. For the sake of continuity, precedence is given to the periodic processing.

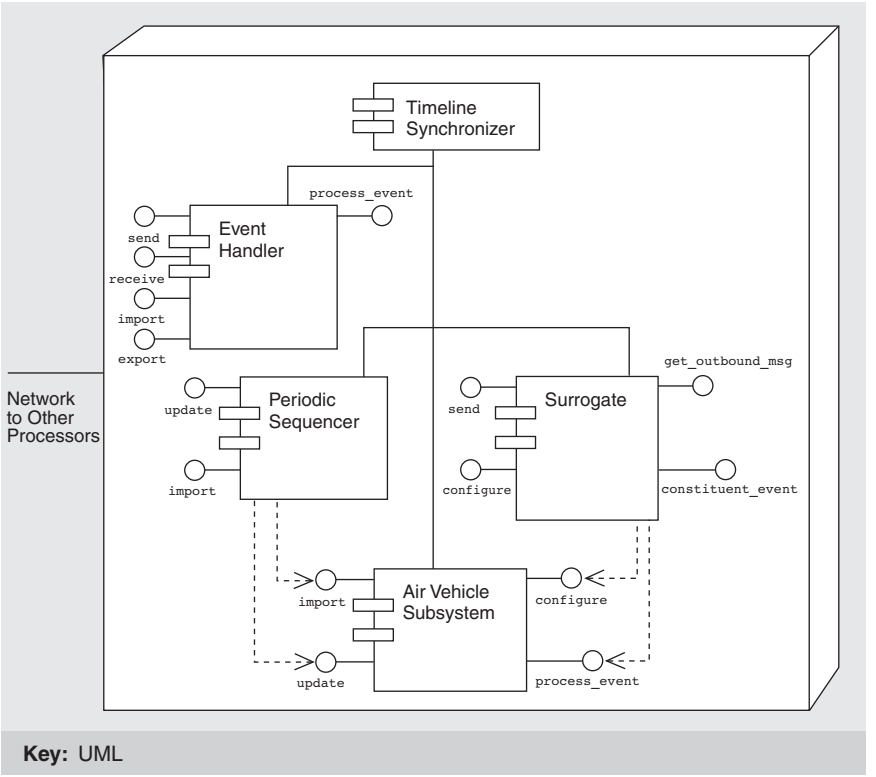


FIGURE 8.4 The Structural Modeling pattern of an air vehicle system processor with focus on the executive

Periodic Sequencer. The periodic sequencer is used to conduct all periodic processing performed by the simulation's subsystems. This involves invoking the subsystems to perform periodic operations according to fixed schedules.

The periodic sequencer provides two operations to the timeline synchronizer. The `import` operation requests that the periodic sequencer invoke subsystems to perform their `import` operation. The `update` operation requests that the periodic sequencer invoke subsystems' `update` operations.

To conduct its processing, the periodic sequencer requires two capabilities. The first is to organize knowledge of a schedule. By *schedule* we mean the patterns of constituent invocations that represent the orders and rates of change propagation through the simulation algorithms realized by the constituents. The enactment of these patterns essentially represents the passage of time within the air vehicle simulation in its various operating states. The second capability is to actually invoke the subsystems through their periodic operations by means of some dispatching mechanism.

Event Handler. The event handler module is used to orchestrate all aperiodic processing performed by subsystems. This involves invoking their aperiodic operations.

The event handler provides four operations to the timeline synchronizer: `configure` (used to start a new training mission, for example), `constituent_event` (used when an event is targeted for a particular instance of a module), `get_outbound_msg` (used by the timeline synchronizer to conduct aperiodic processing while in system operating states, such as `operate`, that are predominantly periodic), and `send` (used by subsystem controllers to send events to other subsystem controllers and messages to other systems).

To perform its processing, the event handler requires two capabilities. The first capability is to determine which subsystem controller receives an event, using knowledge of a mapping between event identifiers and subsystem instances. The second capability is to invoke the subsystems and to extract required parameters from events before invocation.

Surrogate. Surrogates are an application of the "use an intermediary" tactic and are responsible for system-to-system communication between the air vehicle model and the environment model or the instructor station. Surrogates are aware of the physical details of the system with which they communicate and are responsible for representation, communication protocol, and so forth.

For example, the instructor station monitors state data from the air vehicle model and displays it to the instructor. The surrogate gathers the correct data when it gets control of the processor and sends it to the instructor station. In the other direction, the instructor may wish to set a particular state for the crew. This is an event received by the surrogate and passed to the event processor for dispatching to the appropriate subsystems.

This use of surrogates means that both the periodic scheduler and the event handler can be kept ignorant of the details of the instructor station or the platform on which the environment model is operating. All of the system-specific knowledge is embedded in the surrogate. Any change to these platforms will not propagate further than the surrogate in the air vehicle model system.

MODULES OF THE AIR VEHICLE MODEL APPLICATION

Figure 8.5 shows the module types that exist in the application subpart of the air vehicle structural model. There are only two: the *Subsystem Controller* and the *Controller Child*. Subsystem controllers pass data to and from other subsystem controller instances and to their children. Controller children pass data only to and from their parents, not to any other controller children. They also receive control only from their parents and return it only to their parents. These restrictions on data and control passing preclude a controller child from passing data or control even to a sibling. The rationale for this is to assist integration and modifiability by eliminating coupling of a child instance with anything other than its parent.

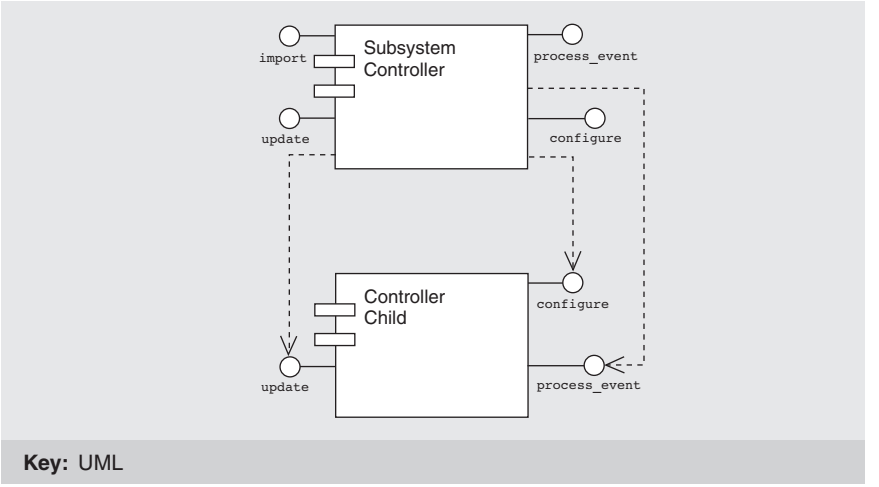


FIGURE 8.5 The application module types

Any effect of modification or integration is mediated by the parent subsystem controller. This is an example of the use of the "restrict communication" tactic.

Subsystem Controller. Subsystem controllers are used to interconnect a set of functionally related children to do the following:

- Achieve the simulation of a subsystem as a whole.
- Mediate control and aperiodic communication between the system and subsystems.

They are also responsible for determining how to use the capabilities of their children to satisfy trainer-specific functionality such as malfunctions and the setting of parameters.

Because the Structural Model pattern restricts communication among controller children, a subsystem controller must provide the capability to make logical connections between its children and those of other subsystems. Inbound connections supply inputs produced outside of the subsystem that the subsystem's children need for their simulation algorithms. Outbound connections satisfy similar needs of other subsystems and of surrogates. These connections appear as sets of names by which a subsystem controller internally refers to data considered to be outside of itself. When such a name is read or written, the appropriate connections are assumed to be made. How the connections are actually made is determined later in the detailed design and is a variation point of the pattern (see Chapter 14, Product Lines, for a discussion of variation points). In addition to making connections between its children and those of other subsystems, the subsystem controller also acts as an intermediary among its own children

since restricting communication means that they are not allowed to directly communicate among themselves.

As we mentioned, a flight simulator can be in one of several states. This is translated through the executive to a particular executive state. The executive then reports its current state to the subsystem controller. The two states that are relevant here are *operate* and *stabilize*. The operate state instructs the subsystem controller to perform its normal computations relevant to advancing the state of the simulation. The stabilize state tells the subsystem controller to terminate its current computation in a controlled fashion (to prevent the motion platform from harming the crew through uncontrolled motion) as follows:

- Retrieve and locally store the values of inbound connections under the direct control of an executive. Such a capability addresses issues of data consistency and time coherence.
- Stabilize the simulation algorithms of its children under the control of executive instances and report whether it considers the subsystem as a whole to be currently stable.

Subsystem controllers *must* be able to do the following:

- Initialize themselves and each of their children to a set of initial conditions in response to an event.
- Route requests for malfunctions and the setting of simulation parameters to their children based on knowledge of child capabilities.

Finally, subsystem controllers may support the reconfiguration of mission parameters such as armaments, cargo loads, and the starting location of a training mission. Subsystem controllers realize these capabilities through periodic and aperiodic operations made available to the periodic sequencer and event handler, respectively.

Subsystem controllers must support the two periodic operations—`update` and `import`—and may support two others (which are aperiodic)—`process_event` and `configure`.

Update – The `update` operation causes the subsystem controller to perform periodic processing appropriate to the current system operating state, which is provided as an input parameter. In the *operate* state, the `update` operation causes the subsystem controller to retrieve inputs needed by its children by means of inbound connections, to execute operations of its children in some logical order so that changes can be propagated through them, and to retrieve their outputs for use in satisfying another's inputs or the subsystem's outbound connections. More than just a sequencer, this algorithm provides a logical "glue" that cements the children into some coherent, aggregate simulation. This glue may include computations as well as data transformations and conversions.

In the *stabilize* state, the `update` operation is used to request that the subsystem controller perform one iteration of its stabilization algorithm, and to determine whether locally defined stability criteria are satisfied. The update operation provides

one output parameter, indicating whether the subsystem controller considers the subsystem to be currently stable. This assumes that such a determination can be made locally, which may not be valid in all circumstances.

Subsystem controllers *may* provide the capability to do the following tasks.

Import – The import operation is used to request that the subsystem controller complete certain of its inbound connections by reading their values and to locally store their values for use in a subsequent `update` operation.

There are two aperiodic operations provided by subsystem controllers: `process_event` and `configure`.

Process_event – The `process_event` operation is used in operating states that are predominantly periodic, such as operate, to ask the subsystem controller to respond to an event. The event is provided by an input parameter to the operation. Several events from the instructor–operator station fall into this category, such as `process_malfunction`, `set_parameter`, and `hold_parameter`.

Configure – The `configure` operation is used in system operating states, like *initialize*, in which the processing is predominantly aperiodic. This operation is used to establish a named set of conditions such as some training device configuration or training mission. The information the subsystem controller needs to establish the condition may be provided as an input parameter on the operation, as a location in a memory on secondary storage, or in a database where the information has been stored for retrieval. To complete the operation, the subsystem controller invokes operations of its children that cause the children to establish the conditions.

Controller Children. Air vehicle model controller children may be simulations of real aircraft components, such as a hydraulic pump, an electrical relay, or a fuel tank. They can support simulator-specific models such as forces and moments, weights and balances, and the equations of motion. They can localize the details of cockpit equipment, such as gauges, switches, and displays. No matter what specific functionality they simulate, controller children are all considered to be of the same module type.

In general, controller children support the simulation of an individual part, or object, within some functional assembly. Each child provides a simulation algorithm that determines its own state based on the following:

- Its former state
- Inputs that represent its connections with logically adjacent children
- Some elapsed time interval

A child makes this determination as often as it is requested to do so by its subsystem controller, which provides the required inputs and receives the child's outputs. This capability is called *updating*.

A child can support the capability of producing abnormal outputs, reflecting a malfunction condition. In addition to potentially modeling changes in normal operating conditions, such as wear and tear, which can result in malfunctions over time, children can be told to start and stop malfunctioning by their subsystem controller.

A controller child can also support the setting of a simulation parameter to a particular value. Simulation parameters are external names for performance parameters and decision criteria used in the controller child's simulation algorithm. Each child can initialize itself to some known condition. Like other child capabilities, parameter setting and initialization must be requested by the subsystem controller.

The updating, malfunctioning, parameter setting, and initializing capabilities differ in the incidence of their use by the subsystem controller. The child is requested to update on a periodic basis, effecting the passage of time within the simulation. Requests for the other capabilities are made only sporadically.

Controller children support these capabilities through a set of periodic and aperiodic operations made available to the subsystem controller. `update` is the single periodic operation and is used to control the periodic execution of the simulation algorithm. The child receives external inputs and returns its outputs through parameters on the operation. Two aperiodic operations are provided by the children: `process_event` and `configure`.

All logical interactions among children are mediated by the subsystem controller, which is encoded with knowledge of how to use the child operations to achieve the simulation requirements allocated to the subsystem as a whole. This includes the following:

- Periodically propagating state changes through the children using their `update` operations
- Making logical connections among children using the input and output parameters on these operations
- Making logical connections among children and the rest of the simulation using the subsystem's inbound and outbound connections

Controller child malfunctions are assumed to be associated with abnormal operating conditions of the real-world components being modeled. Therefore, the presence and identities of these malfunctions are decided by the child's designer and made known to the subsystem controller's designer for use in realizing subsystem malfunction requests. Subsystem malfunctions need not correspond directly to those supported by the children, and certain of them can be realized as some aggregation of more primitive failures supported by children. It is the subsystem controller's responsibility to map between low-level failures and subsystem-level malfunctions.

Likewise, the presence and identities of simulation parameters are decided by the controller child's designer based on the characteristics of the child's simulation algorithm. They are made known to the subsystem controller's designer for

use in realizing subsystem requests or for other purposes for which they are intended or are suitable to support.

SKELETAL SYSTEM

What we have thus far described is the basis for a skeletal system, as defined in Chapter 7. We have a structural framework for a flight simulator, but none of the details—the actual simulator functionality—have been filled in. This is a general simulation framework that can be used for helicopter and even nuclear reactor simulation. The process of making a working simulation consists of fleshing out this skeleton with subsystems and controller children appropriate to the task at hand. This fleshing out is dictated by the functional partitioning process, which we will discuss next.

It is rather striking that an entire flight simulator, which can easily comprise millions of lines of code, can be completely described by only six module types: controller children, subsystem controllers, timeline synchronizer, periodic sequencer, event handler, and surrogate. This makes the architecture (comparatively) simple to build, understand, integrate, grow, and otherwise modify.

Equally important, with a standard set of fundamental patterns one can create specification forms, code templates, and exemplars that describe those patterns. This allows for consistent analysis. When the patterns are mandated, an architect can insist that a designer use *only* the provided building blocks. While this may sound draconian, a small number of fundamental building blocks can, in fact, free a designer to concentrate on the functionality—the reason that the system is being built in the first place.

ALLOCATING FUNCTIONALITY TO CONTROLLER CHILDREN

Now that we have described the architectural pattern with which the air vehicle model is built, we still need to discuss how operational functionality is allocated to instances of the modules in that pattern. We do this by defining instances of the subsystem controllers, to detail the specifics of the aircraft to be simulated. The actual partitioning depends on the systems on the aircraft, the complexity of the aircraft, and the types of training for which the simulator is designed.

In this section, we sketch a sample partitioning. We begin with a desire to partition the functionality to controller children based on the underlying physical aircraft. To accomplish this we use an object-oriented decomposition approach, which has a number of virtues, as follows:

- It maintains a close correspondence between the aircraft partitions and the simulator, and this provides us with a set of conceptual models that map closely to the real world. Our understanding of how the parts interact in the aircraft helps us understand how the parts interact in the simulator. It also makes it easier for users and reviewers to understand the simulator because

they are familiar with the aircraft (the problem domain) and can easily transfer this familiarity to it (i. e., the solution domain).

- Experience with past flight simulators has taught us that a change in the aircraft is easily identifiable with aircraft partitions. Thus, the locus of change in the simulator corresponds to analogous aircraft partitions, which tends to keep the simulator changes localized and well defined. It also makes it easier to understand how changes in the aircraft affect the simulator, therefore making it easier to assess the cost and time required for changes to be implemented.
- The number and size of the simulator interfaces are reduced. This derives from a strong semantic cohesion within partitions, placing the largest interfaces within partitions instead of across them.
- Localization of malfunctions is also achieved as they are associated with specific pieces of aircraft equipment. It is easier to analyze the effects of malfunctions when dealing with this physical mapping, and the resulting implementations exhibit good locality. Malfunction effects are readily propagated in a natural fashion by the data that the malfunctioning partition produces. Higher-order effects are handled the same as first-order effects. For example, a leak in a hydraulic connection is a first-order effect and is directly modeled by a controller child. The manifestation of this leak as the inability to manipulate a flight control is a higher-order effect but it happens naturally as a result of the propagation of simulation data from child to subsystem controller and from one subsystem to another.

In breaking down the air vehicle modeling problem into more manageable units, the airframe becomes the focus of attention. Groups exist for the airframe, the forces on it, the things outside it, and the things inside it but ancillary to its operation. This typically results in the following specific groups:

- *Kinetics.* Elements that deal with forces exerted on the airframe
- *Aircraft systems.* Parts concerned with common systems that provide the aircraft with various kinds of power or that distribute energy within the airframe
- *Avionics.* Things that provide some sort of ancillary support to the aircraft but that are not directly involved in the kinetics of the air vehicle model, the vehicle's control, or operation of the basic flight systems (e.g., radios)
- *Environment.* Things associated with the environment in which the air vehicle model operates

GROUP DECOMPOSITION

The coarsest decomposition of the air vehicle model is the group. Groups decompose into systems, which in turn decompose into subsystems. Subsystems provide the instances of the subsystem controllers. Groups and systems are not directly reflected in the architecture— there is no group controller—and exist to

organize the functionality assigned to the various instances of subsystem controllers. This decomposition is managed via a process using *n-square charts*.

***n*-Square Charts.** One method of presenting information about the interfaces in a system is *n*-square charts. We will make use of this presentation method to illustrate how the partitions we selected relate to each other. Because some of the factors we consider in making partitioning decisions are based on the partition interfaces, *n*-square charts are useful in evaluating those decisions. They are a good method for capturing the input and output of a module and can illustrate the abstractions used in various parts of the design.

An example of an *n*-square chart is shown in Figure 8.6. The boxes on the main diagonal represent the system partitions. Their inputs are found in the column in which the partition lies; their outputs are shown in the corresponding row. The full set of inputs to a partition is thus the union of all the cell contents of the partition's column. Conversely, the full set of outputs is the union of all the cell contents in the row in which the partition resides. The flow of data from one partition to another is to the right, then down, to the left, and then up.

Figure 8.7 shows an *n*-square chart depicting the interfaces between the groups identified above. Interfaces external to the air vehicle model have been omitted for simplicity. These interfaces terminate in interface subsystems. The data elements shown on this chart are aggregate collections of data to simplify the presentation. The interfaces are not named here; nor are they typed. As we investigate partitions, looking at more limited sets of elements, the information presented becomes more detailed. Systems engineers can use this approach to the point where all of the primitive data objects in the interfaces are shown. During detailed design, the interface types and names will be determined.

Not all of the air vehicle models will correspond to aircraft structure. The aerodynamics models are expressions of the underlying physics of the vehicle's interaction with the environment. There are few direct analogs to aircraft parts. Partitioning this area means relying on the mathematical models and physical

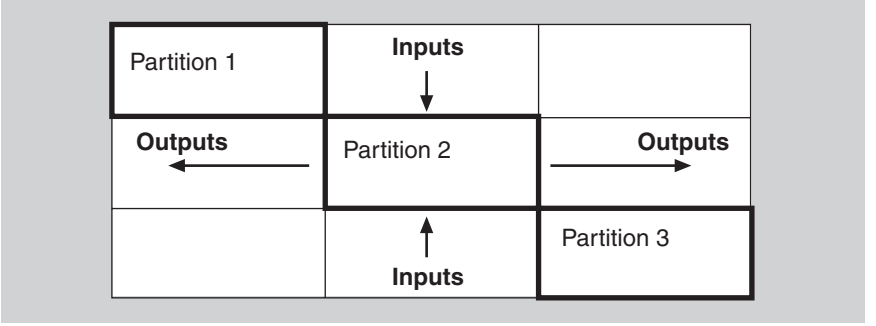


FIGURE 8.6 The *n*-square chart

Kinetics Group	Loads	Vehicle State Vector	Vehicle Position
Power	**Aircraft Systems Group**	Power	
Inertial State	Loads	**Avionics Group**	Ownship Emissions
Atmosphere, Terrain, and Weather Data		Environment Emitter Data	**Environment Group**

FIGURE 8.7 Air vehicle model domain *n*-square for groups

entities that describe the vehicle's dynamics. Partitioning correctly based on mathematical models that affect the total aircraft is more difficult than partitioning based on the aircraft's physical structure.

DECOMPOSING GROUPS INTO SYSTEMS

The next step is to refine groups into systems. A system and a group can be units of integration: The functionality of a system is a relatively self-contained solution to a set of simulation problems. These units are a convenient focus for testing and validation. Group partitions exist as collections of code modules implemented by one engineer or a small group of engineers. We can identify systems within the groups we have defined. We will look briefly at the kinetics group systems as an example.

Systems in the Kinetics Group. These systems consist of elements concerned with the kinetics of the vehicle. Included in this group are elements directly involved in controlling the vehicle's motion and modeling the interaction of the vehicle and its control surfaces with the environment. The systems identified in this group are:

- Airframe
- Propulsion
- Landing gear
- Flight controls

All of the subsystems in the propulsion system shown in Figure 8.8 deal with the model of the aircraft's engines. Multiple engines are handled by creating

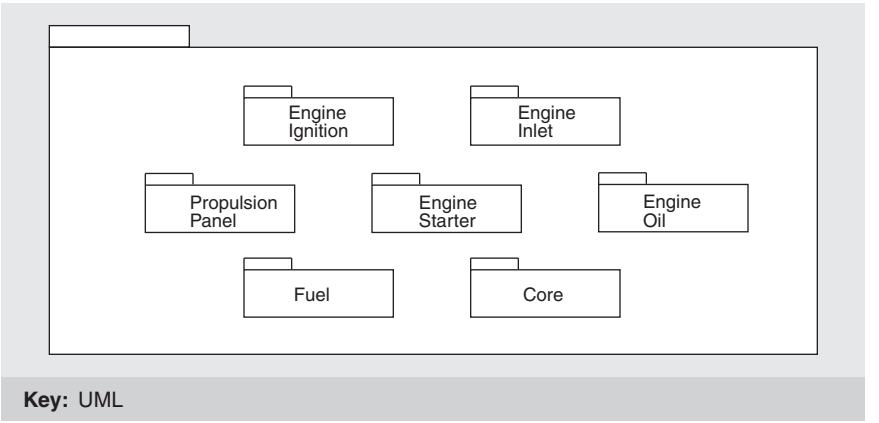


FIGURE 8.8 A propulsion subsystem

multiple sets of state variables and duplicate instances of objects, where appropriate. This system's principal purpose is to calculate engine thrust, moments caused by rotation of engine parts, and the forces and moments caused by mass distribution of fuel.

The aircraft's fuel system is grouped here because its primary interface is to the engines. It calculates the forces acting on the airframe from the movement of the fuel within the tanks as well as the gravitational effect of the fuel mass.

At this point we have identified the division of functionality, its allocation to subsystems and subsystem controllers, and the connections among subsystems. To complete the architecture, we need to do the following:

- Identify the controller children instances for the propulsion subsystem.
- Similarly decompose the other groups, their systems, and their subsystems.

To summarize, we decomposed the air vehicle into four groups: kinetics, aircraft systems, avionics, and environment. We then decomposed the kinetics group into four systems: airframe, propulsion, landing gear, and flight controls. Finally, we presented a decomposition of the propulsion system into a collection of subsystems.

8.4 Summary

In this chapter, we described an architecture for flight simulators that was designed to achieve the quality attributes of performance, integrability, and modifiability. And projects were able to achieve these results with cost savings. For

TABLE 8.1 How the Structural Modeling Pattern Achieves Its Goals

Goal	How Achieved	Tactics Used
Performance	Periodic scheduling strategy using time budgets	Static scheduling
Integrability	Separation of computation from coordination Indirect data and control connections	Restrict communication Use intermediary
Modifiability	Few module types Physically based decomposition	Restrict communication Semantic coherence Interface stability

example, onsite installation teams were 50% of the size previously required because they could locate and correct faults more easily. The design achieves those qualities by restricting the number of module type configurations in the Structural Model architectural pattern, by restricting communication among the module types, and by decomposing the functionality according to anticipated changes in the underlying aircraft.

The improvements in these simulators have principally accrued from a better understanding of, and adherence to, a well-analyzed and well-documented software architecture. Chastek and Brownsword describe some of the results achieved through the use of this pattern [Chastek 96, 28]:

> In a previous data-driven simulator of comparable size (the B-52), 2000–3000 test descriptions (test problems) were identified during factory acceptance testing. With their structural modeling project, 600–700 test descriptions were reported. They found the problems easier to correct; many resulted from misunderstandings with the documentation. . . . Staff typically could isolate a reported problem off-line rather than going to a site. . . . Since the use of structural modeling, defect rates for one project are half that found on previous data-driven simulators.

At the start of this chapter we identified three quality goals of the Structural Model pattern: performance, integrability, and modifiability for operational requirements. Here, we recap how the pattern achieves these goals. Table 8.1 summarizes this information.

PERFORMANCE

A key quality goal of the Structural Model pattern is real-time performance. This is achieved primarily through operation of the executive and use of a periodic scheduling strategy. Each subsystem invoked by the executive has a time budget, and the hardware for the simulator is sized so that it can accommodate the sum of all time budgets. Sometimes this involves a single processor; other times, multiple processors. Given this scheduling strategy, the achievement of real-time performance comes from requiring the sum of the times allocated to the subsystems

involved in the control loops to be within one period of the simulator. Thus, real-time performance is guaranteed by a combination of architectural patterns (the executive module configurations) and the functional decomposition (how the instances are invoked).

INTEGRABILITY

In the Structural Model pattern, both the data connections and the control connections between two subsystems are deliberately minimized. First, within a subsystem the controller children can pass neither control nor data directly to any sibling. All data and control transfers occur only through mediation by the subsystem controller. Thus, integrating another controller child into a subsystem requires that the data in the subsystem controller be internally consistent and that the data transferred between the subsystem controller and the controller children be correct. This is a much simpler process than if a new child communicated with other children because all of them would be involved in the integration. That is, achieving integration has been reduced to a problem that is linear, rather than exponential, in the number of children.

When integrating two subsystems, none of their children interact directly and so the problem is again reduced to ensuring that the two subsystems pass data consistently. It is possible that the addition of a new subsystem will affect several other subsystems, but because the number of subsystems is substantially less than the number of controller children, this problem is limited in complexity.

In the Structural Model, therefore, integrability is simplified by deliberately restricting the number of possible connections. The cost of this restriction is that the subsystem controllers often act purely as data conduits for the various controller children, and this adds complexity and performance overhead. In practice, however, the benefits far outweigh the cost. These benefits include the creation of a skeletal system that allows incremental development and easier integration. Every project that has used structural modeling has reported easy, smooth integration.

MODIFIABILITY

Modifiability is simplified when there are few base module configurations for the designer and maintainer to understand and when functionality is localized so that there are fewer subsystem controllers or controller children involved in a particular modification. Using *n*-square charts helps to reduce connections.

Furthermore, for subsystems that are physically based, the decomposition follows the physical structure, as do modifications. Those subsystems that are not physically based, such as the equations of motion, are less likely to be changed. Users of structural modeling reported that side effects encountered during modifications were rare.

8.5 For Further Reading

For an historical introduction to the computation and engineering involved in creating flight simulators, see [Fogarty 67], [Marsman 85], and [Perry 66].

The Structural Modeling pattern has evolved since 1987. Some of the early writings on this pattern can be found in [Lee 88], [Rissman 90], and [Abowd 93]. A report on results of using the pattern can be found in [Chastek 96].

The reader interested in more details about the functional decomposition used in example flight simulators is referred to [ASCYW 94].

8.6 Discussion Questions

1. The strong relationship between the structure of the system being simulated and the structure of the simulating software is one of the things that makes the Structural Modeling pattern so flexible with respect to mirroring the modeled system in the event of change, extension, or contraction. Suppose the application domain were something other than simulation. Would the Structural Modeling pattern still be a reasonable approach? Why or why not? Under what circumstances would it or would it not be?

2. The data and control flow constraints on subsystem controllers and controller children are very stringent. As a designer and implementor, do you think you would welcome these constraints or find them too restrictive?

3. How does the use of a skeletal system restrict the designer? How is this beneficial and how is it detrimental?