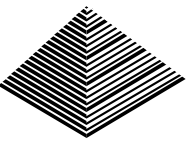
14



Flight Simulation

A Case Study in 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 vehicles and environment they are simulating. The creation and maintenance of these large systems present a substantial software development challenge: for performance, for integrability, and for modifiability. Scalability, a particular form of modifiability, is needed to grow these systems so that they can simulate more and more of the real world and further improve the fidelity of the simulation.

This chapter will discuss some of the challenges of flight simulation and introduce a new architectural style created to address them. The style is called 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
- Minimization of types of components
- A small number of systemwide coordination strategies
- Transparency of design

These principles result in an architectural style that, as we will see, features a high degree of component integrability as well as the other quality attributes necessary for flight simulation.

14.1 Relationship to the Architecture Business Cycle

The segment of the archtecture business cycle (ABC) that connects desired qualities to architecture is the subject of this case study. Figure 14.1 shows the ABC for structural-model-based flight simulators. Flight simulators are acquired by the U.S. Air Force. The end users of the systems are pilots and crews for the particular aircraft being simulated. The 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 of these simulators are intended for stand-alone use, but more and more they are intended to train multiple crews simultaneously for cooperative missions.

The flight simulators are constructed by a contractor selected as a result of a competitive bidding process. Sometimes the contractor is a prime contractor, and portions of the flight simulator are constructed by specialized subcontractors. The flight 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 as the real aircraft in various situations).

The beginning of the structural model design dates from 1987 when the U.S. Air Force began to investigate the application of object-oriented design techniques.

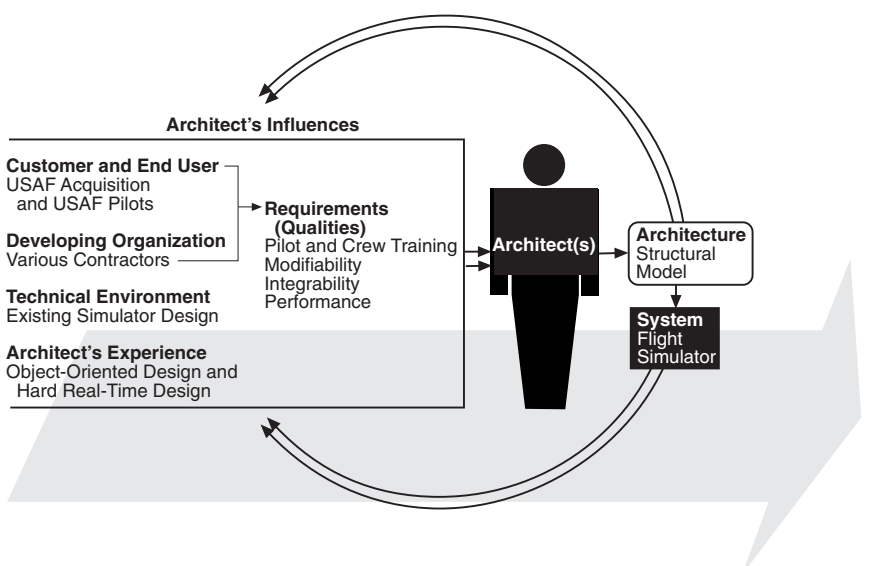


FIGURE 14.1 Initial stages of the ABC for the flight simulator.

Electronic flight simulators have been in existence since the 1960s and so exploration into new design techniques was motivated by the existing designs and the problems associated with them. The problems observed in existing flight simulators 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). Structural models have 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.

14.2 Requirements and Qualities

Flight simulation has always been an application that needed to be distributed in order to be computable at all. Although examples of single-processor flight simulators exist, they are typically of low fidelity, often games. True flight simulation has extremely high-fidelity demands: The virtual environment that the simulator creates must be as lifelike as possible in order to train the aircrew as effectively as possible.

Even in simulators created 30 years ago, distribution was used [Perry 66]:

> Linear computing operations, such as summation, integration and sign changing, are performed by d.c. operational amplifiers, consisting of high gain drift correct amplifiers with appropriate resistive or capacitive feedback networks. There are 150 such operational amplifiers. . . . The non-linear part of the computing equipment consists of servo-multipliers, electronic time-division multipliers, and diode function generators. In addition, forty-eight high gain amplifiers are available for special computing circuits which may be built up for each simulation. . . . The individual computing amplifiers, potentiometers, multipliers, etc. are connected together to form the overall computation network by means of a central patching panel, having over 2300 terminations.

Although the effects produced by such a simulator are primitive by today's standards, many of the principles behind the system's architecture have not changed. A single computer typically cannot provide the raw computation power to serve a flight simulator, or if it could, it would be horrendously expensive. This situation is not likely to change: As computer power increases, so do our demands and so does the complexity of the air vehicles being simulated. Thus, a number of computers were hooked together through a patch panel 30 years ago in order to create a flight simulator; today, they would typically be connected through a fiber-optic network.

Flight simulation is characterized by the following six properties:

1. *Real-time performance constraints.* Flight simulators must execute at fixed frame rates that are high enough to ensure fidelity. For instance, the simu-

lated aircraft must respond to control inputs as quickly as the real aircraft would. Different senses require different frame rates. Within a frequency class—say, 30 Hz or 60 Hz—all simulations must be able to execute to completion within the base time frame—one-thirtieth or one-sixtieth of a second.

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 Hz, 20 Hz, 15 Hz, 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 the change visually 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 for only one purpose: to train users when the equivalent training on the actual vehicle would be much more expensive or dangerous. In order to provide a realistic training experience, a flight simulator must be faithful to the actual air vehicle. However, air vehicles, whether civilian or military, are continually being modified and updated. The simulator software is, therefore, almost constantly modified and updated in order to maintain verisimilitude. Furthermore, the training for which the simulators are used is continually extended to encompass new types of problems (malfunctions) that might occur to the aircraft.

3. *Large size and high complexity.* Flight simulators typically comprise from tens of thousands of lines of code for the simplest training simulation to over a million lines of code for complex, multiperson 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. The technical reason is that 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. The political reason is that high-technology jobs such as simulator development are political plums, and so many politicians fight to have a piece of the work in their jurisdiction. 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.

5. *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 making the costs of testing, integrating, and modifying the software usually exceed the cost of development.

6. *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, by today's standards, 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 and this in turn moves the control surfaces, which affects the aerodynamics and causes the aircraft to turn. In the simulator, there is a model that 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 is based on examining the tasks that the pilot and crew perform, modeling the components that actually 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 causes problems with both modifiability and integration. If the module controlling turning is being integrated with the module controlling level flight and a problem is discovered in the data being provided to the turning module, that same data are probably being accessed by the level flight module, and so there were many coupling effects to be considered during integration and maintenance.

The growth in complexity (and its associated growth in cost) caused the U.S. Air Force organization that is responsible for the acquisition of flight simulators to begin to emphasize the software qualities of integrability and modifiability. This new emphasis was in addition to the requirement for high-fidelity performance.

By *integrability* we mean the ability to reconcile different portions of the system during the integration phase of initial development. One of the consequences of the growth in complexity was the growth of the cost of integration. With a new system under development that was projected to be 50 percent larger than the largest prior system (1.7 million lines of code), the integration phase alone would break the project's budgets for time and cost.

14.3 Architectural Approach

To manage the numerous challenges that flight simulation posed for its software designers, a new culture of software design for flight simulation needed to be created. This design framework, called *structural modeling,* will be discussed for the remainder of this chapter. In brief, the design framework consists of an object-

TABLE 14.1 How the Structural Modeling Paradigm Achieves Its Quality Goals

Goal	How achieved
Performance	Small number of systemwide coordination strategies and periodic scheduling strategy
Integrability	Separation of computation from coordination and indirect data relationships
Modifiability	Transparency of design, simplicity and similarity of substructures, and indirect data relationships

oriented design to model the subsystems and components of the air vehicle and real-time scheduling to control the execution order of the simulation's subsystems so that fidelity could be guaranteed. To expand slightly on what we said in the introduction, the main tenets of structural modeling are as follows:

- Simplicity and similarity of the architecture's substructures so that the architecture is easy to understand and, hence, modify.
- Small number of systemwide coordination strategies, which simplifies the complex performance and scheduling issues in making a high-fidelity real-time system.
- Indirect data relationships so that passing data between subsystems is done through intermediary components. In this way subsystems do not need to directly know of each others' existence or the details of where data come from and go to. This greatly simplifies the integrability and modifiability of the architecture.
- Separation of computation from decisions about how data and control are passed.
- Transparency of design (making the objects in the flight simulator directly analogous to physical parts of the air vehicle being modeled). This makes the flight simulator easier to design, modify, and maintain over time.

The driving design goals and the techniques that allowed these goals to be met are summarized in Table 14.1.

14.4 Architectural Solution

Logically, flight training simulators have three interactive roles. The first role is that of training the pilot and crew. They sit inside a motion platform, surrounded by instruments intended to replicate exactly the aircraft being simulated, and look at visuals that provide a representation of what would be seen outside of an actual aircraft. We are not going to describe the specifics of either the motion platform or the visual display generator. These 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 in the simulation. Otherwise, the pilot and crew are being trained incorrectly and the training may actually be counterproductive. Figure 14.2 shows a collection of modern flight simulators.

The second role associated with a flight simulator is that of the environment. Typically, the environment is a computer model rather than individuals operating in the environment, although with multiaircraft training exercises the environment can include individuals other than the pilot and crew we have been discussing. The environment includes 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 instructor for the simulation. Usually, a training session has a very specific purpose, and specific circumstances will occur during the training. During the training exercise, the instructor is responsible for monitoring the performance of the pilot and crew and for initiating training situations. Sometimes these situations are scripted

FIGURE 14.2 Modern flight simulators. Courtesy of the Boeing Company.

in advance, and other times the instructor introduces them. Typical situations include malfunctions of equipment (e.g., landing gear that does not deploy correctly during landing), attacks on the aircraft from foes, and weather conditions such as turbulence caused by thunderstorms. The instructor has a separate console that is used both to monitor the activities of the crew and to inject malfunctions into the aircraft and control the environment.

Figure 14.3 shows a reference model for a flight simulator. 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 on the instructor station.

The division between the instructor station and the other two portions is clear from a real-time execution perspective. 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 by using a periodic scheduling procedure. The environment model can be scheduled using either regime, depending on the complexity and richness of the environment being modeled. This requirement to coordinate and match two different scheduling paradigms will be revisited when we discuss the details of the structural model.

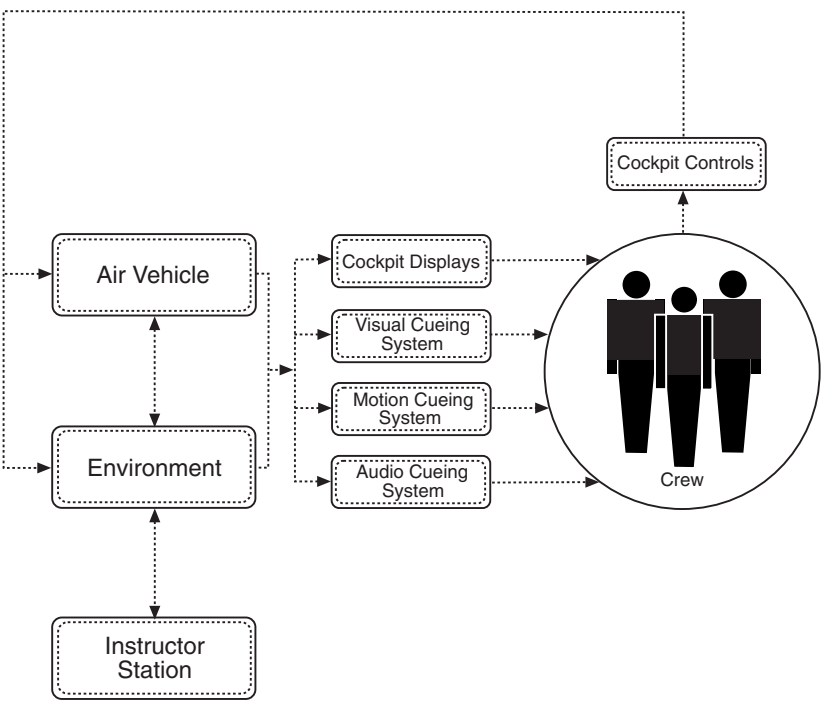


FIGURE 14.3 Reference model for a flight simulator.

The division between the instructor station and the other two portions is not as clean. For example, if an aircraft launches a weapon, it is logically a portion of the air vehicle until it leaves the air vehicle, at which point it becomes a portion of the environment. On firing, however, the aerodynamics of the weapon will be influenced initally by the proximity of the aircraft. Thus, any modeling of it must remain, at least initially, tightly coupled to the air vehicle. If the weapon is always considered a portion of the environment, the modeling of the weapon 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 scheduling regime to another. Even if the environment is scheduled using a periodic scheduling regime, it is likely to operate on a different granularity from that of the environment model.

The simulated behavior of the air vehicle is not calculated from first physical principles. Such simulations are used during the engineering process and are called *engineering simulations*, but they cannot yet be done within the constraints that real-time processing impose. Consequently, the simulations are performed using models supplemented with empirical data. For example, the thrust of an engine might be measured using the whole engine, but the impact of an individual rotor is not measured. Thus, if a rotor is replaced, keeping the simulation correct can be difficult.

For the rest of this section we focus primarily on the architecture of the air vehicle model. But before we can do that, we need a basic understanding of the ways in which time is treated in this domain.

TREATMENT OF TIME IN A FLIGHT SIMULATOR

Time and how it is managed is a critical consideration in the development of a flight simulator. Time is important because a flight simulator is supposed to reflect the real world, and it does this by creating time-based behaviors as they would occur in the real world. Thus, when the pilot in a simulator pushes on the flight 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 also 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.

Periodic Time Management. A periodic time-management scheme has a fixed (simulated) time quantum that is the basis of scheduling the system processes. It typically uses a nonpreemptive cyclic scheduling discipline. The scheduler proceeds by iterating through the following loop:

- Set initial simulated time.
- Iterate next two steps until the session is complete.

- Invoke each of the processes for a fixed (real) quantum. Each of the invoked processes calculates its internal state based on the current simulated time and reports its internal state based on the next period of simulated time. Each invoked process 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 the simulated time and the real time in synchronization as long as each individual process is able to advance its state to the next period within the time quantum that it has been allocated.

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 and then using multiple processors, if necessary, to ensure enough computation power to enable all of the processes to receive their quantum of computation.

A periodic simulator scheduling scheme is used when many activities are happening in parallel in simulated time. Thus, for example, when the simulation is based on the simultaneous solution of a collection of differential equations, a solution technique based on finite differences leads to a periodic scheduling strategy.

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

- Add initial simulated event to the event queue.
- While there are events remaining in the event queue:
  - Choose event in event queue with smallest (i.e., soonest) simulated time.
  - Set current simulated time to time of chosen event.
  - Invoke process that processes 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. There is no guarantee about the relationship of simulated time to real time and no guarantee about the amount of resources consumed by each process that is invoked.

Event-based simulations assume that the management of the event queue is not a major drain on resources. That is, the number of events occurring at a single instant of simulated time is small compared to the effort of managing those events.

Mixed-Time Systems. 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 in such a marriage.

A simple policy states that periodic processing occurs immediately after the synchronization interval and is completed before any aperiodic processing. Aperiodic processing then 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.

Given this understanding of the issues involved in managing time in a flight simulator, we can now present the architectural style that manages this complexity.

THE STRUCTURAL MODEL ARCHITECTURAL STYLE

The structural model is an architectural style as we defined it in section 2.2. That is, it consists of a collection of component configurations and a description of how these configurations coordinate at runtime. In this section, we present the structural model and discuss the considerations that led to its design. The structural model we present here is focused on the air vehicle model. One aspect to keep in mind is that the air vehicle model may itself be spread over several processors. Thus, the elements of the air vehicle structural model must coordinate internally as well as coordinate with the environment model and the instructor portions of the simulation.

A flight simulator can execute in several states including the following:

- *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 an environment in which they have been flying in a single-aircraft exercise, and then the instructor wishes to train them for a mid-air refueling exercise. 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. Replay is used, among other functions, to demonstrate to the crew what they have just done because the crew may get caught up in the operating of the aircraft and not reflect on their actions.

The structural modeling architectural style is divided, at the coarsest level, into two parts: *executive* and *application*.

The *executive* part 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* part handles the actual computation of the flight simulation: modeling the air vehicle. The application's functions are implemented by subsystems and their constituent components. First we will discuss the executive style in detail and then return to a discussion of the application style.

COMPONENT CONFIGURATIONS FOR AIR VEHICLE MODEL EXECUTIVE

Figure 14.4 shows the air vehicle structural model with the executive component configurations given in detail. These 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 is also responsible for coordinating time with other portions of the simulator. This can include other processors responsible for a portion of the air vehicle model, which also have their own timeline synchronizers. Finally, the timeline synchronizer implements a scheduling

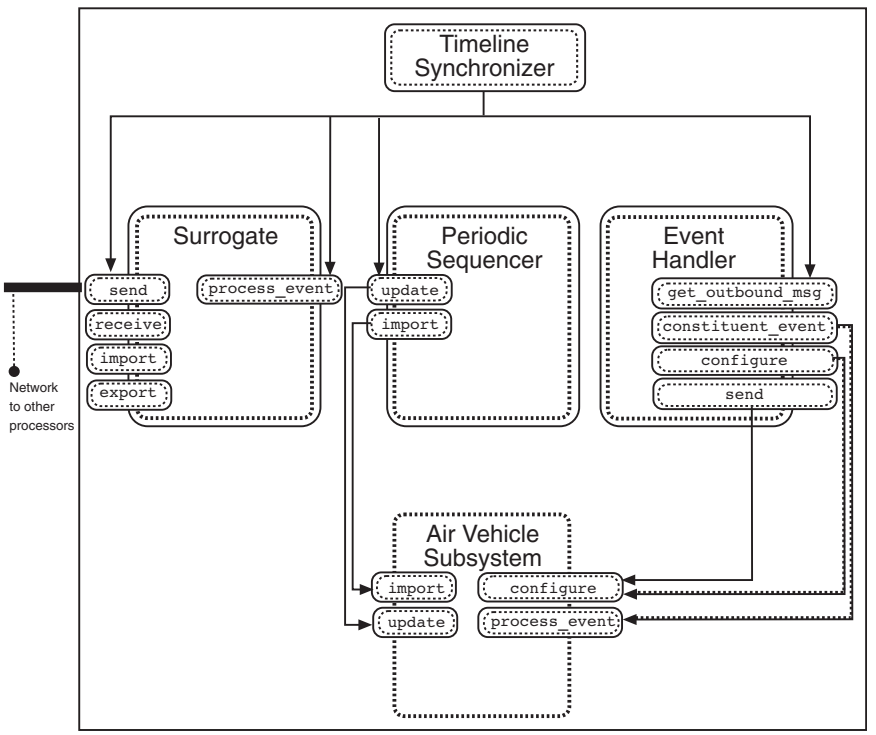


FIGURE 14.4 The structural model style of an air vehicle system processor with focus on the executive.

policy for coordinating both periodic and aperiodic processing. For the sake of continuity, precedence is given to the periodic processing.

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 is used to request the periodic sequencer to invoke subsystems to perform their `import` operation. The `update` operation requests that the periodic sequencer invoke the `update` operations of the subsystems.

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 at which changes are propagated 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 element is used to orchestrate all aperiodic processing performed by subsystems. This involves invoking the subsystems by aperiodic operations appropriate to the current system operating state.

The event handler provides four kinds of 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 component), `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 is a capability to determine which subsystem controller receives an event, using knowledge of a mapping between event identifiers and instances. The second is a capability to invoke the subsystems and to extract required parameters from events before invocation.

Surrogate. Surrogates 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 issues such as representation, communication protocol, and so forth.

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

COMPONENT CONFIGURATIONS FOR AIR VEHICLE MODEL APPLICATION

Figure 14.5 shows the component configurations that exist in the application subpart of the air vehicle srtuctural model. There are only two of them: the *subsystem controller* and its *components*. The term *component*, as used in the structural model, refers to a specific component type. This terminological ambiguity is restricted to this section, and we hope it will not cause confusion. Subsystem controllers pass data to and from other subsystem controller instances but only to their child components. Components will pass data only to and from their parents and not to any other component. They will also receive control only from their parents and return it only to their parents. These restrictions on data and control passing preclude a component from passing data or control even to a sibling component. The rationale for these restrictions is to assist in integration and modifiability by eliminating any coupling of a component instance with anything other than its parent. Any effect of modification or integration is mediated by the parent subsystem controller.

Subsystem Controller. Subsystem controllers are used to interconnect a set of functionally related components 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 components to satisfy trainer-specific functionality such as malfunctions and the setting of parameters.

A subsystem controller must provide the capability to make logical connections between its components and those of other subsystems. Inbound connections supply inputs produced outside of the subsystem that the subsystem's components 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.

Subsystem controllers must both order the `update` operations of their components and interconnect their inputs and outputs. This achieves the desired prop-

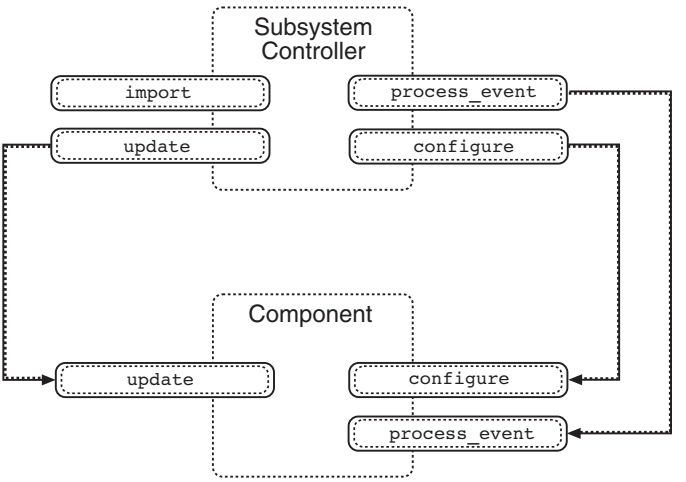


FIGURE 14.5 The application component configurations.

agation of changes through the components and satisfies the subsystem's outbound connections. The subsystem controller synthesizes, from its components, a simulation algorithm for the subsystem as a whole. In addition to updating components, the controller may be responsible for making intermediate computations as well as data transformations and conversions.

As we mentioned, a flight simulator can be in one of several states. This is translated through the executive to particular executive states. The executive then informs the subsystem controller of its current state. 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 informs 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 is provided to address issues of data consistency and time coherence.
- Stabilize the simulation algorithms of its components 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 components to a set of initial conditions in response to an event
- Route requests for malfunctions and the setting of simulation parameters to their components based on knowledge of component capabilities

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

There are two periodic operations provided by system controllers: `update` and `import`.

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 components by means of inbound connections, execute operations of its components in some logical order so that changes can be propagated through them, and retrieve their outputs for use in satisfying another's inputs or the subsystem's outbound connections. More than just a sequencer, this algorithm provides logical "glue" that cements the components 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 the subsystem controller to perform one iteration of its stabilization algorithm and 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 makes the assumption 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.

The import operation is used to request the subsystem controller to complete certain of its inbound connections by reading their values and locally store their values for use in a subsequent `update` operation. There are two aperiodic operations provided by subsystem controllers: process_event and configure.

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. A characteristic of this operation is that effects of servicing the event are reflected only in the subsystem's periodic outputs, which are not made available until the next *update* operation. 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.

The configure operation is used in system operating states, like *initialize*, in which the processing is predominantly aperiodic. The operation is used to establish a named set of conditions such as some training device configuration or training mission. The information that 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 components that cause the components to establish the conditions.

Component. Air vehicle model components can be simulations of real aircraft components, such as a hydraulic pump, an electrical relay, an engine rotor, a fuel

tank, and the like. They can support simulator-specific models such as forces and moments, weights and balances, and the equations of motion. They can be used to localize the details of cockpit equipment, such as gauges, switches, and displays. But no matter what specific functionality they are used to simulate, components are all considered to be architecturally equivalent.

In general, components are used to support the simulation of an individual part, or object, within some functional assembly. Each component provides a simulation algorithm that determines the state of the component based on the following:

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

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

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

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

The capabilities of updating and those of malfunctioning, parameter setting, and initializing differ in the incidence of their use by the subsystem controller. The component 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.

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

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

- Periodically propagating state changes through the components using their *update* operations

- Making logical connections among components using the input and output parameters on these operations
- Making logical connections among components and the rest of the simulation using the subsystem's inbound and outbound connections

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

Likewise, the presence and identities of simulation parameters are decided by the component designer based on the characteristics of the component's simulation algorithm. They are made known to the subsystem controller designer for use in realizing subsystem requests or for other purposes for which they are intended or are suitable to support.

Skeletal System and Pattern-Based Simplicity. What we have now described is a skeletal system, as defined in Section 13.2. We have a framework for a flight simulator, but none of the details—the functionality—have been filled in. In fact, this framework has also been used for helicopter simulation and even nuclear reactor simulation. The process of realizing a working simulation consists of fleshing out this skeleton with subsystems and components, as dictated by the functional partitioning process, which will be discussed next.

An entire flight simulator, which can easily exceed 1 million lines of code, can be described completely by six component types: components, subsystem controllers, timeline synchronizer, periodic sequencer, event handler, and surrogate. This is an example of pattern-based simplicity, as we discussed in Section 13.3, and it renders the architecture easier 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 these 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.

Both of these architectural approaches—skeletal system framework and pattern-based simplicity—have contributed significantly to the success of the structural modeling approach.

REFERENCE ARCHITECTURE

Now that we have described the architectural style with which the air vehicle model is built, we still need to discuss how operational functionality is allocated to instances of that style. That is, we need to define the reference architecture. We do this by defining instances of the subsystem controllers.

The reference architecture is the place where the specifics of the aircraft to be simulated are detailed. 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 reference architecture.

We begin with a desire to partition the functionality to components based on the underlying physical aircraft. To accomplish this we use an object-oriented decomposition approach. This 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. They are familiar with the aircraft (problem domain), and they can easily transfer this understanding to the simulator (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 the change in the simulator corresponds to analogous aircraft partitions. This 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 functional cohesion within partitions, placing the largest interfaces within partitions instead of across them.
- Localization of malfunctions is also achieved. Malfunctions 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. The effects of malfunctions 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.

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 the airframe, those things outside the airframe, and those things inside the airframe but ancillary to its operation. This results in the following specific groups:

- *Kinetics group.* Elements that deal with forces exerted on the airframe
- *Aircraft systems group.* Those parts concerned with common systems that provide the aircraft with various kinds of power or distribute energy within the airframe

- *Avionics group.* Those things that provide some sort of ancillary support to the aircraft but are not directly involved in the kinetics of the air vehicle model, the vehicle's control, or operation of the basic flight systems
- *Environment group.* Those things associated with the environment in which the air vehicle model operates

GROUP DECOMPOSITION

In this section we motivate the coarsest decomposition of the air vehicle model—the group decomposition. Groups decompose into systems which, in turn, decompose into subsystems. These subsystems provide the instances of the subsystem controllers. Groups and systems are not directly reflected in the architecture and exist to rationalize the functionality assigned to the various instances of subsystem controllers. For now, we begin with groups.

***N*-Square Charts.** One method of presenting information about the interfaces in a system is provided by *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 interfaces between the partitions, *n*-square charts are useful in evaluating the decisions. They are a good method for illustrating the overall view of interfaces and can be used to illustrate the abstractions utilized in various parts of the design.

An example *n*-square chart is shown in Figure 14.6. The boxes on the main diagonal represent the system partitions. Their inputs are found in the column in which the partition lies. The outputs from the partition are shown in the row in which the partition lies. Therefore the full set of inputs to a partition is 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.

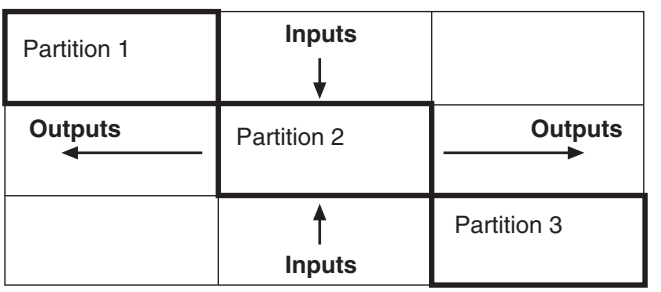


FIGURE 14.6 The *n*-square chart.

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 14.7 Air vehicle model domain *n*-square chart for groups.

Figure 14.7 shows an *n*-square chart depicting the interfaces between the groups we identified above. Interfaces external to the air vehicle model have been omitted to simplify the chart. The external 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 detail of the information presented increases accordingly. 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 and the structure finalized.

Not all of the air vehicle models will have a correspondence 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 will mean relying on the mathematical models and physical entities used to describe the vehicle's dynamics. Partitioning correctly based on mathematical models that affect the total aircraft is more difficult to do than partitioning based on the physical structure of the aircraft.

DECOMPOSING GROUPS INTO SYSTEMS

The next step is to refine the groups into systems. A system and a group can be units of integration: The functionality contained within the system is a relatively self-contained solution to a set of simulation problems. These units are a convenient focus for testing and validation. Partitions of groups 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 above. We will look briefly at the kinetics group systems.

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 motion of the vehicle and modeling the interaction of the vehicle and its control surfaces with the environment. The systems identified in this group are as follows:

- Airframe
- Propulsion
- Landing gear
- Flight controls

All of the subsystems in the propulsion system shown in Figure 14.8 deal with the model of the aircraft's engines. Multiple engines are handled by creating 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 the primary interface for the fuel system is to the engines. The fuel system calculates the forces acting on the airframe from movement of the fuel within the tanks as well as the gravitational effect of the fuel mass. At this point the reference architecture for the propulsion subsystem is complete. We have identified the division of functionality, its allocation to subsystems and subsystem controllers, and the connections among subsystems.

To complete the architecture, we would need to do the following:

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

This, however, concludes the presentation of the architecture in this case study.

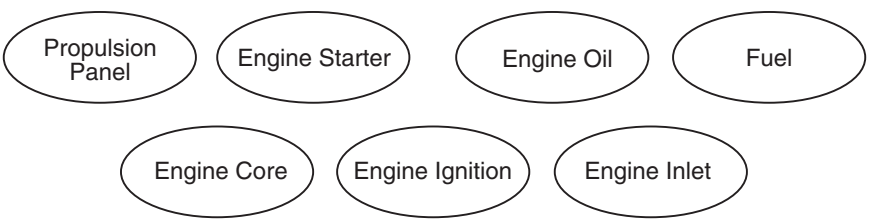


FIGURE 14.8 Propulsion subsystems.

14.5 Achievement of Goals

Structural modeling has been an unqualified success. Chastek and Brownsword describe some of the results achieved through the use of this architectural style [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 structural modeling: performance, integrability, and modifiability for operational requirements. In this section, we recap how the structural model achieves these quality goals.

PERFORMANCE

A key quality goal of the structural model is real-time performance. This goal is achieved primarily through the operation of the executive and the use of a periodic scheduling strategy. Each subsystem that is invoked by the executive has a time budget, and the hardware for the simulator is sized so that it can accommodate the sum of the time budgets of the subsystems. 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 style (the executive component configurations) and the functional decomposition (how the instances are invoked).

INTEGRABILITY

In Chapters 4 and 5, we discussed how qualities are embedded into an architecture. In this chapter, we see an example in which the architectural style is designed to minimize integration problems. This approach is to minimize the connections between the various component configurations used in the structural model.

In the structural model, both the data connections and the control connections between two subsystems are deliberately minimized. First, within a subsystem, the components can neither pass control nor data directly to any sibling component. Any data or control transfer occurs only through the mediation of the

subsystem controller. Thus, integrating another component into a subsystem involves ensuring that the data in the subsystem controller are internally consistent and that the data transferred between the subsystem controller and the component are correct. This is a much simpler process than if the new component communicated with other components because all of them would be involved in the integration. That is, the integration problem has been reduced to a problem that is linear, rather than exponential, in the number of components.

When integrating two subsystems, none of their components 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 components, this problem is limited in complexity.

In the structural model, therefore, integrability is simplified by deliberately restricting the number of possible connections. The cost for this restriction is that the subsystem controllers often act purely as data conduits for the various components, and this adds complexity. The benefits of this approach, however, far outweigh the cost in practice. Every project that has used structural modeling has reported easy, smooth integration.

MODIFIABILITY

Modifiability is simplified when there are few base component configurations for the designer and maintainer to understand and when functionality is localized so that there are fewer subsystem controllers or components involved in a particular modification. The technique of using *n*-square charts begins with the premise of reducing connections.

Furthermore, for those subsystems that are physically based, the decomposition follows the physical structure, and modifications also follow the physical structure. 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.

14.6 Summary

In this chapter we have described an architecture for flight simulators that was designed to achieve the qualities of performance, integrability, and modifiability. And projects were able to achieve these results with cost savings. For example, on-site installation teams were 50 percent of the size previously required because they could locate and correct faults more easily. The design achieved those qualities by restricting the number of component configurations in the structural model architectural style, by restricting communication among the components,

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.

14.7 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 paradigm has evolved over the past 10 years. Some of the early writings on this paradigm can be found in [Lee 88], [Rissman 90], and [Abowd 93*b*]. A more recent description 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].

14.8 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 style 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 structural modeling still be a reasonable approach? Why or why not? Under what circumstances would it or would it not be?

2. Structural modeling is a combination of several of the architectural styles described in Chapter 5. What are they and how are they used? Does this use conform to the advantages of those styles that we described in Chapter 5?

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