


# Six-Degree-of-Freedom Digital Simulations for Missile Guidance, Navigation, and Control

Patricia A. Hawley and Ross A. Blauwkamp



*This article presents a brief history of missile simulations and a discussion of the programming languages and paradigms used for developing them. Evolving language and programming paradigms elicit requirements for new simulation architectures. Within this execution framework, engineering-level guidance, navigation, and control simulations must include certain functional modules to capture the performance characteristics of the missile system. The level of model sophistication required depends on the particular engineering question to be answered. Six-degree-of-freedom simulations are effective tools for cost and risk reduction during the development and deployment of missile systems.*

## INTRODUCTION

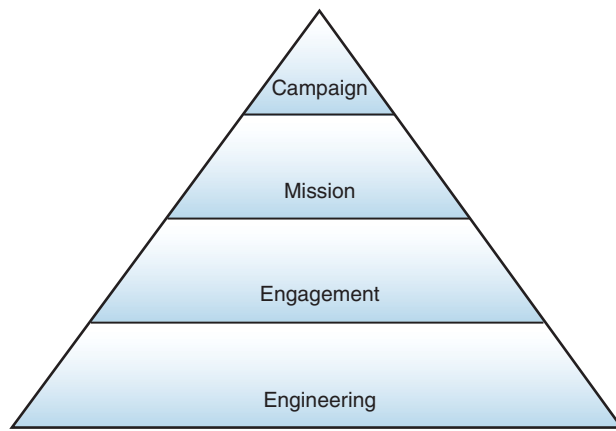
The DoD definitions<sup>1</sup> of the terms model and simulation are as follows:

- Model: A physical, mathematical, or otherwise logical representation of a system, entity, phenomenon, or process.
- Simulation: A method for implementing a model over time.

Models and simulations are further classified by the DoD into four levels: campaign, mission, engagement, and engineering. These four levels are shown and defined in Fig. 1. A campaign-level simulation includes such a large number of model elements that using engineering-level models on a single computer would probably take years of execution time. Typically for guidance,

navigation, and control (GNC) analysis, the number of assets is more limited, and the simulation is at the engineering level—although the level of sophistication of the models used varies with the system questions to be answered.

This article discusses the engineering questions, model implementations, and simulation architectures used in a GNC simulation. We start with a brief historical review of GNC simulations and their uses, and then we examine the requirements of a digital simulation independent of the models and outline current simulation designs. Finally, we characterize the essential models for a GNC simulation and the different levels of detail for these models. Additionally, we consider some



**Figure 1.** These four levels of simulation reflect the level of detail in the simulations and the scope of the questions being asked. Starting at the most detailed level, an engineering simulation models a missile system's components and their interactions to the highest fidelity possible. Next, an engagement simulation omits some of the detail of the engineering simulation but includes models for launch platforms and threats so that the system's effectiveness at neutralizing the threat can be ascertained. A mission-level simulation omits more details and aims to address the tactical effectiveness of the missile system to perform a specific mission (e.g., air defense). Finally, a campaign-level simulation seeks to determine the best capability mix of "blue" forces against "red" forces by focusing on order of battle and probability of kill. Ideally, all available engineering details would be included at all levels of simulation, but this is generally not feasible.

engineering questions that the simulation may answer based on the level of model fidelity.

## HISTORY

Orville and Wilbur Wright did not simulate airframes; they prototyped them. As pilots, they acted as the guidance and navigation subsystems, and they solved any unstable control systems problems with the airframe during flight testing by improvising attitude commands and after landing by modifying the control surfaces to achieve, after an iterative process, safe and stable performance. Their successes triggered the development of airplanes around the world, but their methodology was costly both in terms of material and in the health and safety of the pilot. It also was impractical for unmanned airframes such as missiles.

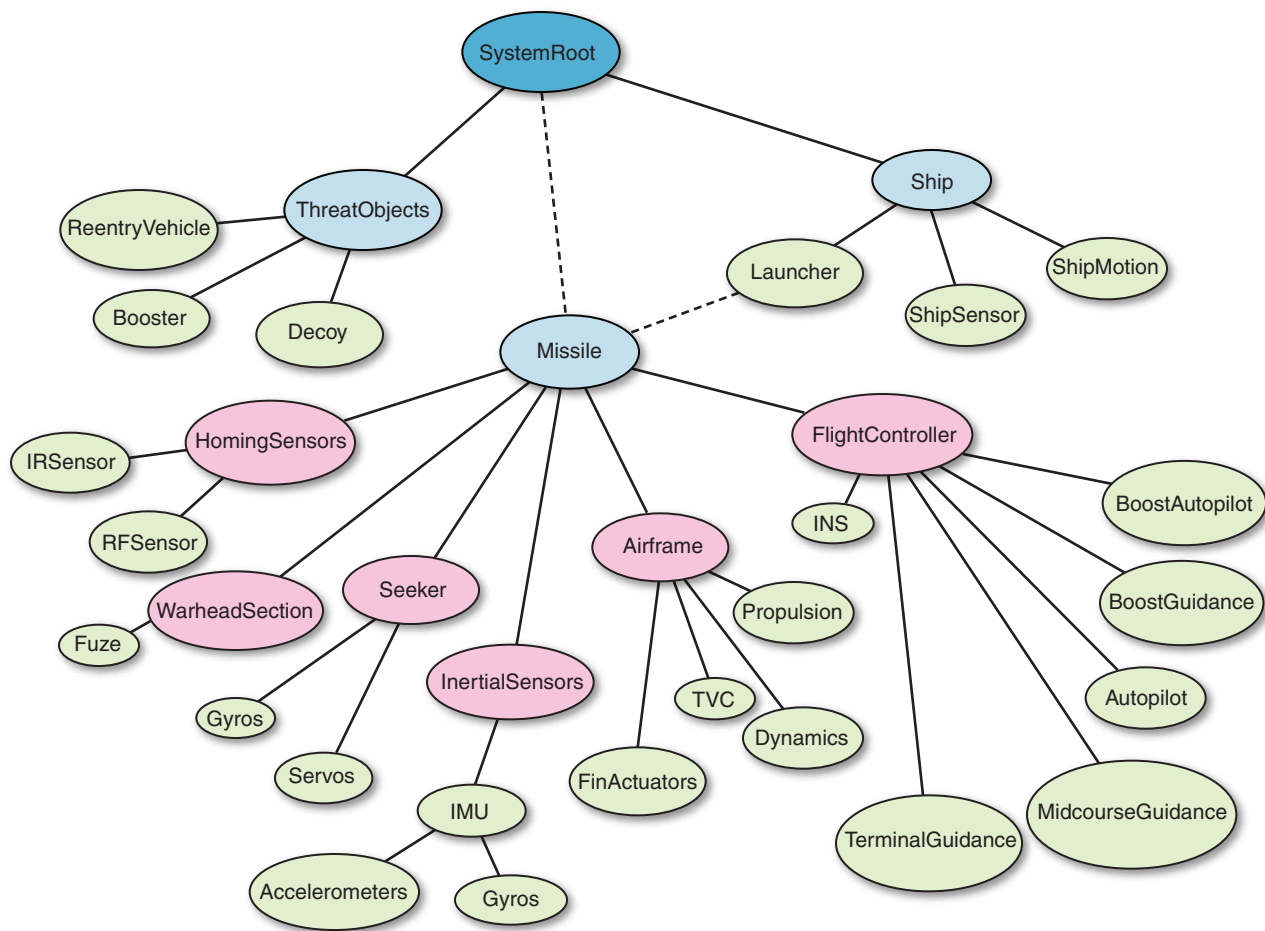
The first missiles were the Greek and Roman ballistae, whose motion gave us the term ballistic trajectory. Their designers and users determined the performance characteristics of these weapons empirically and incrementally modified and improved them over the years. It wasn't until Sir Isaac Newton supplied the mathematical and physical language to describe this motion that engineers could more accurately predict the performance of

ballistic missiles. During World War II, German engineers improved on the launch mechanism of their ballistae by adding rocket propulsion and a simple azimuth control for rudimentary guidance. The performance of these first guided missiles was poor, but it was sufficient to inspire an entire segment of today's defense industry.

The first "simulation" of a missile consisted of a rocket engine burn time and the ballistic equation of motion to determine the missile's achievable range, as well as a heading to determine its approximate impact point.<sup>2</sup> Current missile systems are described by nonlinear differential equations, partial differential equations, and/or discrete-time equations. These models may encompass high-fidelity aerodynamics involving tables of wind tunnel measurements,<sup>3</sup> time-varying propulsion characteristics, digital autopilots, one or more homing sensors, inertial sensors, communication links, and one or more guidance laws. The complexity of these missiles is reflected in the costs and the capabilities of such systems. Instead of simply hitting a target as large as a London neighborhood (the goal of the German missiles of World War II), current interceptors are expected to impact within centimeters of the aimpoint. Given the expense of testing such complex systems, and the difficulty in fully evaluating all components to their full range of capability, simulations are an effective means of cost and risk reduction.

## PROGRAMMING LANGUAGES AND PARADIGMS

The first digital programs were written in assembly language, and the combination of hardware and language limited their scope. Fortunately high-level programming languages provided engineers with more sophisticated tools for building programs. One of the first high-level programming languages was FORTRAN, the IBM Mathematical FORMula TRANslation System. FORTRAN allowed engineers to write mathematically sophisticated equations to model missile systems. The challenge then was to write equations that were succinct enough to execute on the slow memory-limited early digital computers—without any particular software architecture. To achieve real-time performance with hardware in the loop, engineers used analog computers rather than slower digital ones. An analog computer uses the voltages and currents of electrical components as surrogates for the state variables in differential equations and, therefore, could represent the operating condition for a missile during testing of subsystems such as tail actuators or seeker heads. Because analog computers require special-purpose hardware and configurations and are limited by noise, nonlinearities, and parasitic effects, they have been replaced by digital computers as the speed and memory capacity of the digital computers have improved.<sup>4</sup> Invariably, with each increase in hardware performance, engineers increase the complexity of

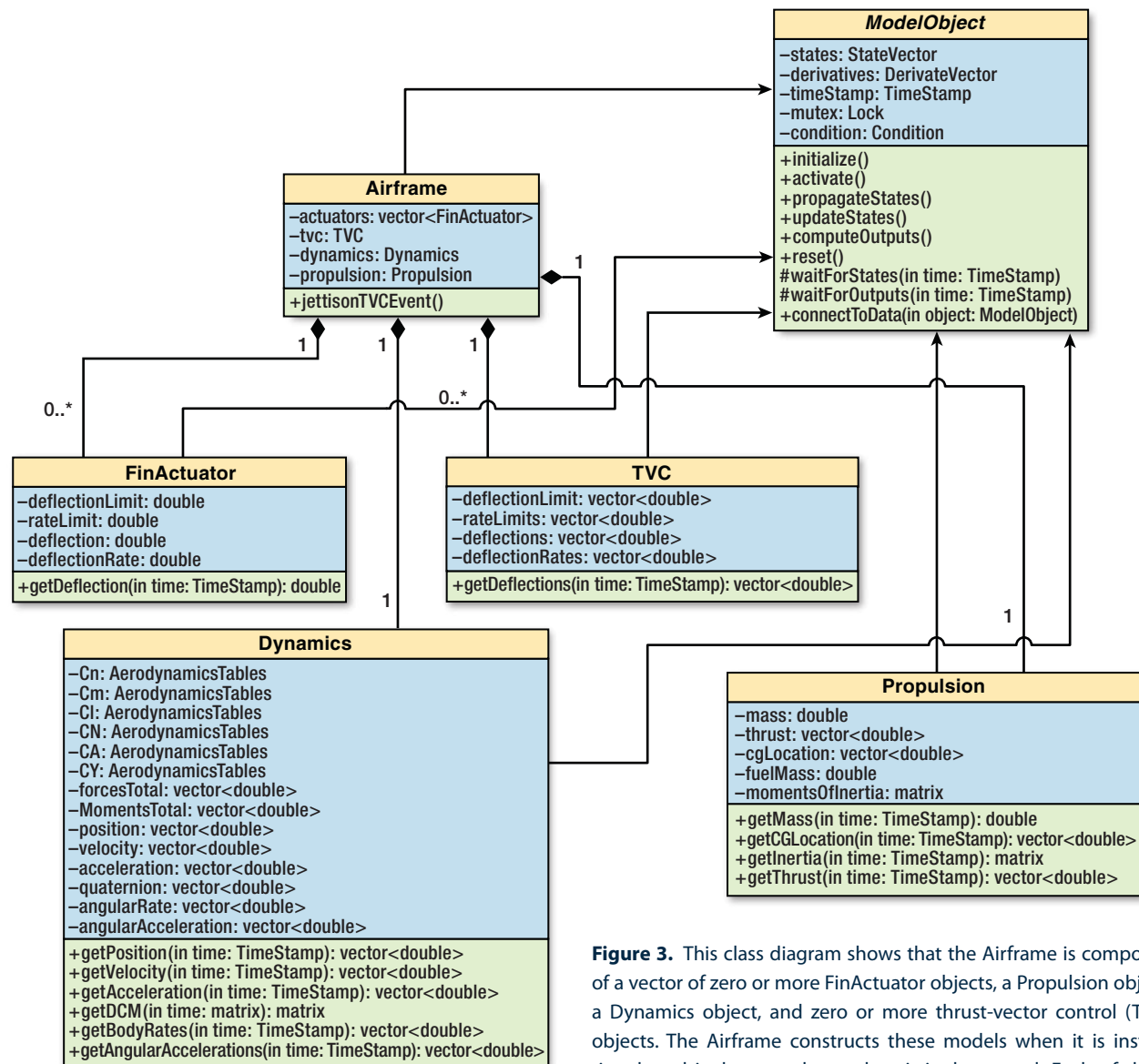


**Figure 2.** This tree shows a hierarchical model tree for a missile system. The SystemRoot in cyan connects the tree to the simulation executive. The top nodes—Missile, ThreatObjects, and Ship (in blue)—correspond to the top-level functional descriptions of objects that would appear in a mission or engagement simulation. The pink nodes are higher-level models composed of the more detailed models that are the leaves of the tree (in green). These leaves correspond to the engineering models that would appear in a 6-degree-of-freedom (6DOF) simulation.

the models that they describe in computer code. Logically, there should be a corollary to Moore’s Law (which states that the power of digital hardware doubles approximately every 2 years) to indicate that the models coded on these rapidly advancing computing platforms double in complexity every 2 years.

FORTRAN is a high-level procedural programming language with high-quality mathematical libraries for numerical computations, but initially there were few data structures—only scalars, arrays, and COMMON blocks—and few control constructs—IF, GOTO, and DO; so, as the size of the code blocks and the size of the code development teams increased, the maintenance and reliability of the programs became problematic.<sup>5,6</sup> “Spaghetti code” proliferated and undermined the effectiveness of engineering models for testing the performance of increasingly sophisticated missile systems. The first attempt to address this problem was the develop-

ment of structured programming: a top-down software-development methodology that imposed a disciplined breakdown of the data flow in a simulation. Operations on the data were partitioned into modules or procedures and executed sequentially, and the system states often were represented by an appropriate set of data structures. This methodology exposed the control flows that produce spaghetti code, namely the infamous GOTO statement and the equally nefarious FORTRAN COMMON block, but did not eliminate the problems associated with global scoping of variables in a simulation. Data flowing through a simulation built by using structured programming may suffer unintended consequences as a result of a small change in the internal workings of a particular code module,<sup>7</sup> and the engineer maintaining the simulation may have a difficult time finding the source of the problem if the change was made by another team member. Despite these drawbacks, there are millions of



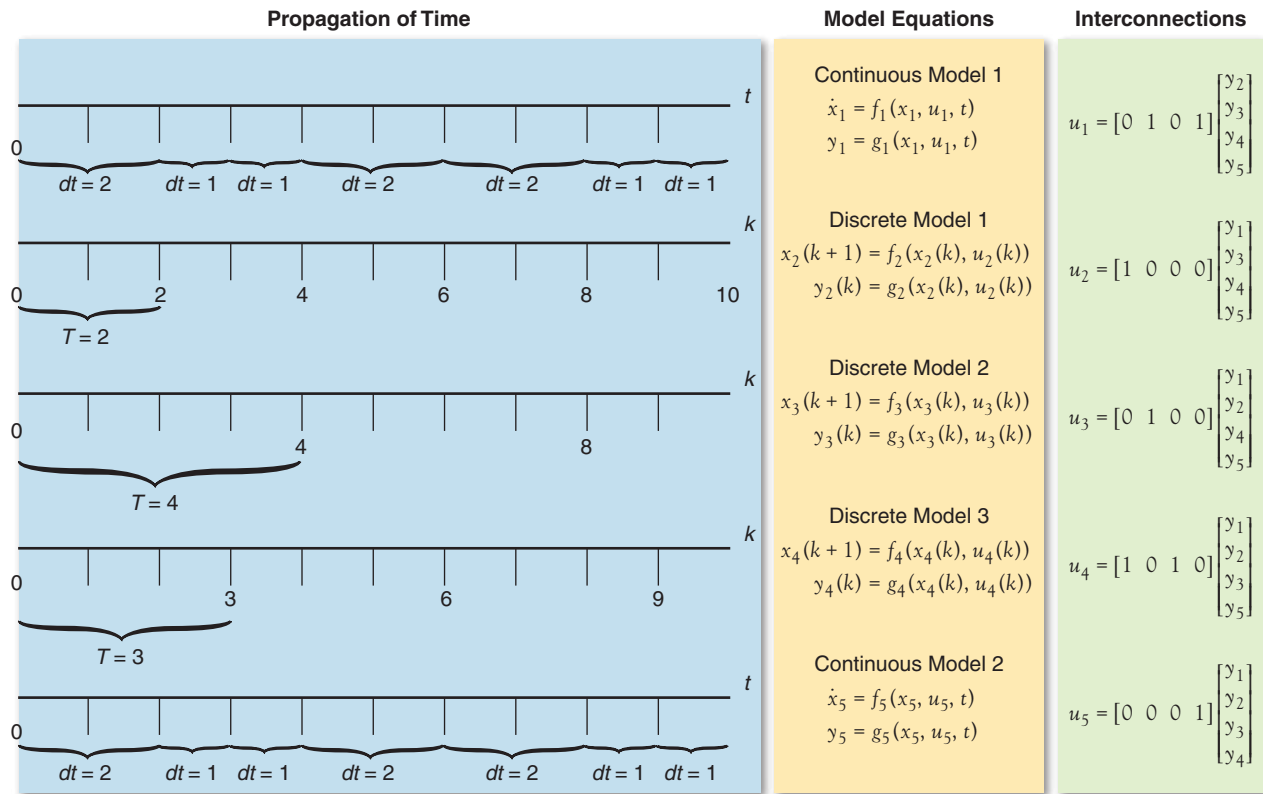
**Figure 3.** This class diagram shows that the Airframe is composed of a vector of zero or more FinActuator objects, a Propulsion object, a Dynamics object, and zero or more thrust-vector control (TVC) objects. The Airframe constructs these models when it is instantiated, and it destroys them when it is destroyed. Each of these classes inherits methods from the ModelObject that encapsulate the

functionality required by the simulation executive. The get functions (in green) allow other models to access the outputs of these objects. A model can generate an event as indicated by the jettisonTVCEvent method in the Airframe model. The variables in blue are the private data of the models. Notice that the ModelObject does not maintain a list of subscribers for the Observer pattern because the objects do not push data to their subscribers. Instead, the objects pull data from publishing objects with the correct time stamps and may wait for the data to be ready. The connectToData method finds the object supplying the required data by searching the model tree. These objects do not propagate time, but they do depend on it.

lines of structured-programming FORTRAN code still in use. Existing FORTRAN numerical libraries often are linked into the C++ programs to supply efficient mathematical utilities, and it is possible to wrap legacy FORTRAN code in a C-style interface for use in a C++ simulation.

To foster greater maintainability of programs, a more effective separation of concerns was needed. The next paradigm for high-level programming was object-oriented (OO) programming. In an OO design, data and

actions are bound together in objects to separate one model's functionality from another model's functionality and to separate time and other simulation services (e.g., random numbers and I/O) from the models. This methodology specifically addressed the *scoping*<sup>8</sup> of data and/or state variables in the simulation; namely, a model has state variables, and these states are *private*<sup>9</sup> data of the class. An instance of a class is called an object, and other objects cannot directly affect the states of the model; they can only request access to publicly available

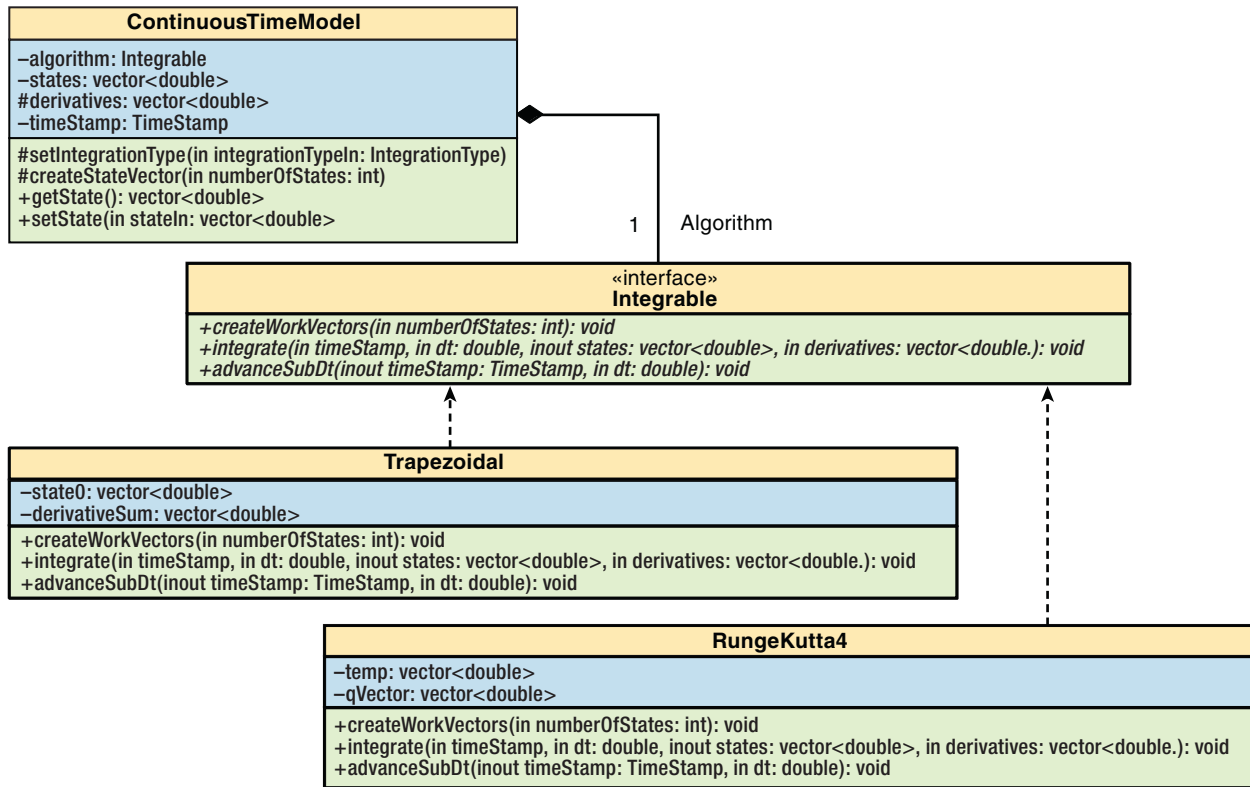


**Figure 4.** The blocks highlighted in yellow are the model equations that correspond to the leaves in a hierarchical tree that are either discrete or continuous. The equations to the right (green) indicate how the blocks are interconnected. The time axes on the left (blue) indicate each model's expected update times independent of the other models; however, to be mathematically correct, the differential equations must be propagated such that whole time steps ( $n * dt$ ) align with the update times for the discrete models that provide inputs to the continuous models. This requirement forces the time steps for the numerical integration to change to smaller values if necessary to align with the discrete time propagation as shown by the variability of  $dt$  on the axes. These three behaviors separate into three components in the 6DOF simulation architecture: time propagation, model equations, and model interconnections/communication.

information provided by the model. Multiple instances of a class can be present in a simulation, but because the states are *encapsulated*,<sup>10</sup> these instances are independent. A particular missile subsystem may be composed of multiple modules forming a hierarchical tree of nested models. To simplify code development and reuse, classes may *inherit*<sup>11</sup> states and/or behaviors from base classes; for example, a digital autopilot model class would inherit from the DiscreteModel base class. Inheritance supports the concept of *polymorphism*,<sup>12</sup> wherein a model can be treated as a plug-and-play component in the simulation—a higher-fidelity seeker model can be swapped into the simulation for the terminal homing phase after dynamically removing a lower-fidelity seeker model used for the midcourse phase. These programming concepts require a high-level language such as Smalltalk, C++, or Java. Often, C++ is chosen because it is backward-compatible with the C programming language used for low-level hardware coding and provides OO classes with inheritance and templates for generic programming. C++ is statically typed and allows for user-created types

with operator overloading; i.e., the programmer may create a type (a class) and explicitly overload the binary + operator to perform a syntactically appropriate combination of two objects of that type. Attempting to use the overloaded operator with another type generates a compiler error. Catching usage errors at compile time versus run time typically improves simulation execution times by eliminating conditional tests from the final code.

Grouping data or state variables into objects that dictate the operations that may be performed on these states illuminated the various ways that engineers use data and operations while writing code. The authors of *Design Patterns: Elements of Reusable Object-Oriented Software*<sup>13</sup> catalogued many of the most commonly seen pairings of data and operations and grouped them into categories: creational, structural, and behavioral. A pattern describes a programming idiom, i.e., the partitioning of functions and responsibilities into particular classes or objects to achieve a given task or algorithm. An engineer equipped with a set of design patterns or idioms is equipped with a set of tools for producing effective



**Figure 5.** This class diagram highlights the way the Strategy pattern is used to choose from among multiple integration algorithms for the differential equations in a continuous-time model. Calls are made by the ContinuousObject on its field myIntegrable, whose type is the abstract type, Integrable. Calls on the abstract type are dispatched to the specific concrete instance this field references, an object of class Trapezoidal or RungeKutta4.

tive maintainable code. In the context of missile simulations, these patterns illustrate the separation of state models from the mechanisms that propagate the equations over time and the mechanisms that allow models to communicate with each other and with the execution architecture. As an example, consider the hierarchical tree of model objects in Fig. 2; its related classes conform to the structural Composite pattern. The class diagram in Fig. 3 illustrates the Airframe branch of the tree in Fig. 2; the ModelObject base class implements the functions that support the simulation executive and defines empty (abstract) methods for the derived classes to implement, describing the differential and/or difference equations for the models. This is the Template pattern. A critical factor for missile simulations is the propagation of the state equations over time for continuous models (consider Fig. 4). Flexibility of the numerical integration algorithms used can be achieved by the Strategy pattern (see Fig. 5). A field of the model is defined as an abstract algorithm type, and methods defined by this type (e.g., integrate) are called on this field in a generic way by the enclosing model. Each concrete algorithm implements the methods of the abstract type in its own specific

way, for example, trapezoidal or fourth-order Runge-Kutta integration (see Box 1). Communication between models can be encapsulated in the Mediator pattern or more commonly in the Observer pattern. The Observer pattern, when extended to a distributed programming environment, is called the Broker<sup>14</sup> pattern. Thus, the major advantage of design patterns in OO programming is a common vocabulary and building blocks for the engineers developing a missile simulation.

## ARCHITECTURE REQUIREMENTS

Programming paradigms and languages are the hammers and nails of the simulation engineer but not the blueprints. To get to the blueprints, the engineering team must decide on a set of software requirements. First, recall that the separation of the model implementation from the time-propagation algorithm occurs quite naturally (as illustrated in Fig. 4), but the interaction of the time-stepping and synchronization with the calculation of equations in models must be mathematically correct to achieve the correct propagation of the state space equations. This independence of the layers

**BOX 1. RUNGE-KUTTA INTEGRATION FORMULA**

$$x(t_{n+1}) = x(t_n) + \frac{dt}{6}(k_1 + 2k_2 + 2k_3 + k_4), \text{ where } t_{n+1} = t_n + dt$$

$$k_1 = \dot{x}(t_n, y_n) \equiv \text{state derivatives at } t_n$$

$$k_2 = \dot{x}(t_n + \frac{dt}{2}, x_n + \frac{dt}{2}k_1) \equiv \text{state derivatives at midpoint of interval using states advanced to the midpoint using Euler's method and } k_1 \text{ as the slope}$$

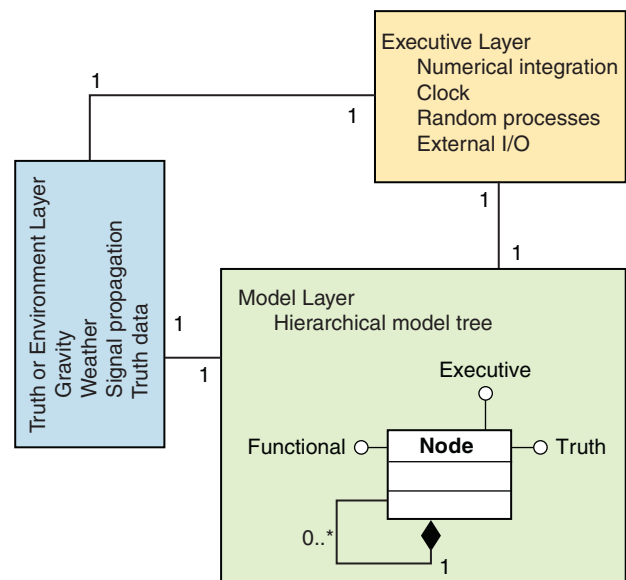
$$k_3 = \dot{x}(t_n + \frac{dt}{2}, x_n + \frac{dt}{2}k_2) \equiv \text{state derivatives at midpoint of interval using } k_2 \text{ as the slope}$$

$$k_4 = \dot{x}(t_n + dt, x_n + dt * k_3) \equiv \text{state derivatives at endpoint of interval using } k_3 \text{ as the slope}$$

The commonly used fourth-order Runge-Kutta integration algorithm breaks each major integration interval into four minor integration steps. Each step involves the calculation of the derivatives with an updated state. Note that time does not step uniformly through the minor steps. The advantage of using a higher-order algorithm, such as this one, is that the error shrinks by  $dt^4$  rather than by  $dt$ , as it does in a first-order algorithm, such as the Euler method.

is illustrated in Fig. 6 by the separation of the executive utilities and environment from the models. Clearly, the executive elements do not depend on the models, but they do supply functionality that the models may use. Second, the model-state equations (Fig. 4 center) often depend on other models' states and/or outputs (Fig. 4 right), so communication between these models must be not only transparent but also synchronous at internal time steps in the numerical integration (Fig. 4 left). Third, as model complexity rises and run-time execution slows, multithreading or distributed processing becomes a possible solution to the requirement for real-time execution while maintaining model fidelity. In summary, the architecture executive requirements are as follows:

1. Synchronous evaluation of continuous state equations at time-step bounds for both major and minor time steps in the numerical integration algorithm
2. Synchronous evaluation of continuous and discrete equations at the time-step bounds for the discrete-time models
3. Asynchronous evaluation of aperiodic events (e.g., separation of missile stages)
4. High-quality, random number generation for noise processes
5. Reproducible results
6. Start and restart of simulation runs at an arbitrary time
7. Flexible configuration of the executive and model parameters at run time
8. Transparent logging of outputs (usually by creating a hierarchical file structure that mimics the model tree structure)
9. Real-time execution for hardware-in-the-loop applications
10. Multithreading to enhance run-time performance<sup>15</sup>



**Figure 6.** This diagram illustrates the layers of a 6DOF simulation where the time management, I/O, and random processes are encapsulated in the Executive Layer; the models for gravity, weather, and signal propagation are encapsulated in the Environment Layer; and the hierarchical model tree is encapsulated in the Application Layer (as in Fig. 2). Note that the nodes of the hierarchical tree have interfaces for utilizing the Executive and Truth Layers as well as the functional interfaces for the model elements. Often other layers are illustrated for a simulation architecture: for example, the User Interface Layer, which often is a graphical user interface (GUI), and the Operating System Layer for platform-specific code. These layers are optional for a missile 6DOF and may be implemented as part of the three layers shown.

11. Distributed processing to enhance run-time performance and/or operation in a High-Level Architecture (HLA) environment<sup>16</sup>
12. Portability and maintainability
13. Scalability

The first three requirements are necessary for mathematical correctness. The fourth requirement refers to statistically valid Monte Carlo results that require independent random processes, and the fifth requirement refers to reproducing a single run in the presence of random processes by logging the seed(s) used in the random number generator(s). The sixth requirement simply allows a user to (re)run the simulation from an arbitrary time so, for instance, the terminal phase of flight could be explored in more detail without running from  $t = 0$ . The seventh and eighth requirements address inputs to and outputs from the simulation for configuration and post-processing, respectively. The next three requirements may or may not be part of the final specification for the architecture, but they should be considered for future reuse of the code. Portability and maintainability refer to executing the code on different hardware and software platforms (e.g., PCs or workstations running Windows or Linux) and adopting good programming practices and configuration management so that a team can efficiently develop code and transition that code to new users and developers. Scalability refers to using this architecture with as many models as the user chooses without degrading the performance. This last item would, of course, impact the run-time performance, but it should never affect the accuracy of the simulation results.

Another layer of the architecture handles the communication between models and the external “world” (gravity, signal propagation, weather, etc.). This layer of utilities has the following requirements:

- Loose coupling between models supports dynamic creation and destruction of models during a simulation run.
- Communication must be transparent between models.
  - Models request data from a particular class, and the communication mechanism finds the nearest matching object and connects the subscriber to the publisher (Observer pattern).
  - As models are created and/or deleted from the simulation, the communication mechanism adjusts the subscriptions.

All of these requirements are independent of the system to be modeled and, therefore, the architecture that matches these requirements promotes reusability.

The following are some examples of current APL simulations within the Air and Missile Defense Department that are using OO architectures with model hierarchies:

- OO Simulation Architecture (OSA):
  - Evolved Sea Sparrow Missile launch-to-intercept simulation is a high-fidelity, 6DOF implementation.

- C++
- Single threaded
- Synchronous execution
- Composite pattern for model tree
- Mediator pattern for model communication
- Java Event-Driven Implementation (JEDI):
  - SwarmSim is the Swarm Simulation for Small Boat Defense.
  - MIDAS is the Missile Defense Analysis Simulation.
  - Both are launch-to-intercept 6DOF simulations for ongoing system design and feasibility studies.
    - Java
    - Multithreaded
    - Synchronous execution (rendezvous between threads)
    - Composite pattern for model tree
    - Observer pattern for model communication
- Open Architecture Simulation Interface Specification (OASIS):
  - OASIS is a specification for 6DOF simulation development in either C++ or Java.
  - The Multiple Kill Vehicle End-To-End Simulation is built with Simitar, an OASIS-compliant simulation framework.
    - C++ (Simitar is the C++ implementation of the OASIS; it has had several engineering releases and is being actively developed.)
    - Single threaded for the first few releases
    - Asynchronous execution
    - Composite pattern for model tree
    - Observer pattern for model communication

## THE COMMON MISSILE MODEL

Up to this point, the discussion of a missile simulation has indicated only that the subsystems comprise smaller subsystems, etc., and thus are naturally represented by a hierarchical tree (as in Fig. 2). What has not been discussed is what makes up a missile system. To a certain extent, the mathematical model chosen depends on the question to be answered.

As mentioned earlier, a minimal set of equations for the V-2 rocket would treat the airframe as a point mass without aerodynamics but with a thrust equation and just the ballistic equations of motion after motor burnout with an azimuth constraint. These equations are sufficient to obtain rough estimates of the impact point. Variations in wind conditions and motor burn as well as heading and attitude control errors would affect actual performance. Adding simple trim aerodynamics<sup>17</sup> with a transfer function representation of the autopilot and a proportional navigation guidance

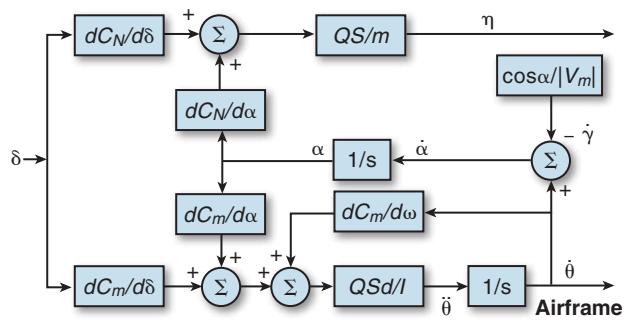


law<sup>18</sup> produces a 3DOF simulation model that often is used to estimate a missile’s operational footprint. This approach is valid during the initial phases of a new missile’s development or to model the general performance of a system whose specific design details are not known (adversaries). A variation on this approach would be to successively modify the guidance law over some set of guidance law options to assess which law yields the best performance in the scenario context. Again, these results would be optimistic because they ignore so many other factors in the system, but the engineer can use the demonstrated trends to direct the next phase of model development.

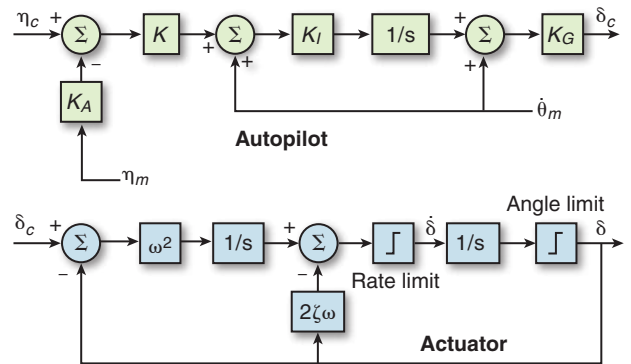
Another level of fidelity is shown in the models of Figs. 7–9. Figure 7 shows the “tennis court” model for an axisymmetric airframe linearized about a flight condition (e.g., the pitch channel of the airframe). Figure 8 is the classic “three-loop” autopilot for the pitch channel and a second-order transfer function model for an actuator with angle and rate limiting. Figure 9 illustrates a simple inertial measurement unit (IMU) and the proportional navigation guidance law. Assuming a roll-stabilized missile, the engineer can use these models to perform 5DOF analysis of the system at the selected flight condition. This level of modeling can be useful for investigating observed instabilities during a test flight or computing values for a gain-scheduled autopilot during initial design studies (e.g., airframe control design and trade studies).

Subsequently, a full 6DOF simulation coded from the proposed missile system specifications can be used to verify the predicted performance of the system once the full aerodynamic characteristics, functional algorithms, and expected noise sources are included. A 6DOF simulation can be used to design initial flight tests to exercise various missile subsystems at particular operating conditions. Similarly, after flight testing of the system has begun, the 6DOF simulation parameters can be validated against the measured telemetry. The interactions of the missile with supporting engagement systems, such as surface-based radar and weapons control, can be explored and overall system performance can be evaluated. The simulation can be used to assess the risks associated with modifications to the missile or to assess its performance against a new threat. For an operational system, a simulation identifies and illuminates key sensitivities in the existing hardware. The effect on system responsiveness and lethality of potential hardware or software changes can be characterized. Understanding this system behavior allows the engineer to revise the missile specifications, if necessary, for future deployments. Finally, it can be used to generate operational guidelines for deployment and firing protocols.

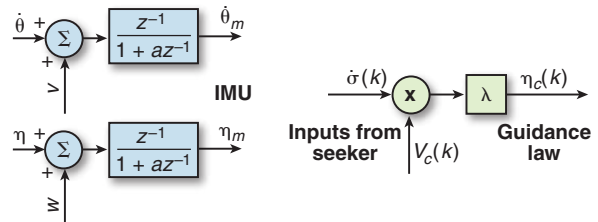
Although the degree of sophistication of the models varies from the simplest 3DOF to the full-up 6DOF, in



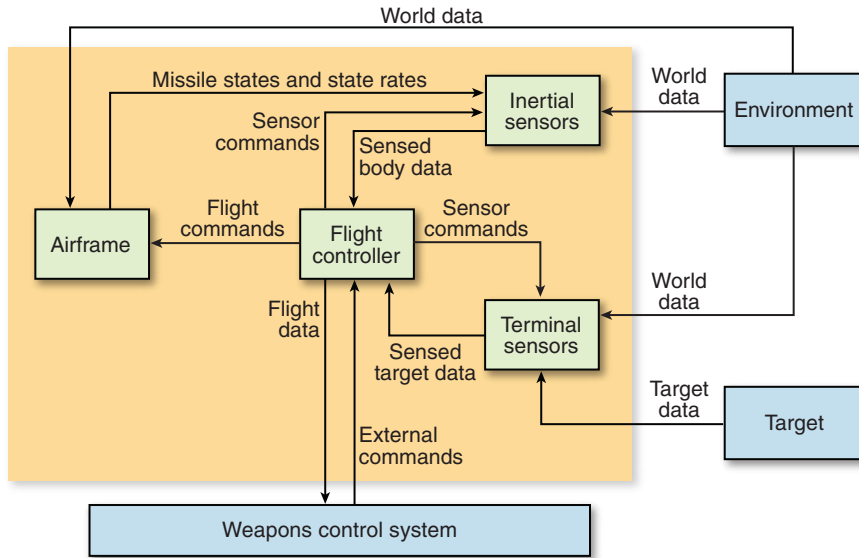
**Figure 7.** Aerodynamics airframe model where  $\delta$  is the actuator deflection,  $Q$  is the dynamic pressure,  $S$  is the reference area,  $d$  is the reference diameter,  $I$  is the moment of inertia,  $m$  is the mass,  $\alpha$  is the angle of attack,  $\eta$  is the acceleration,  $\gamma$  is the flight-path angle, and  $\theta$  is the Euler angle. The partial derivatives are the aerodynamic coefficients linearized at a selected flight condition.



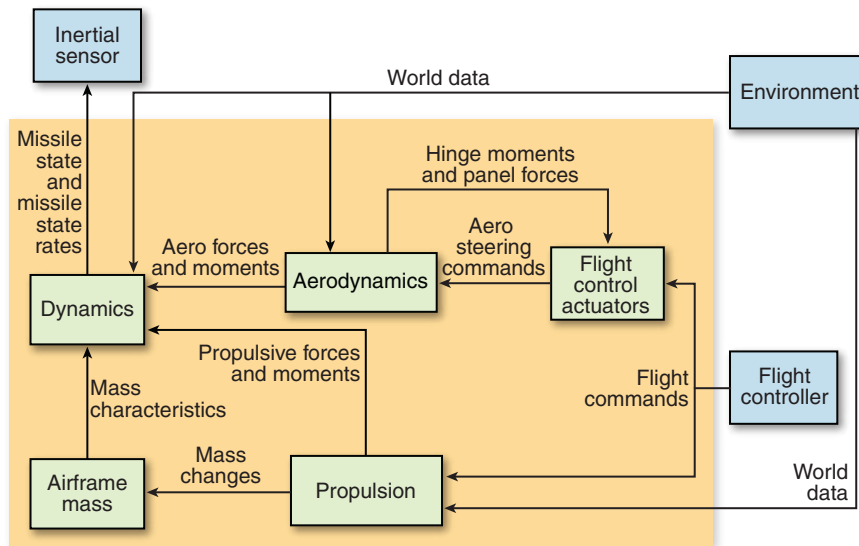
**Figure 8.** A “three-loop” autopilot and simple transfer function model for the actuator. The autopilot uses the acceleration command from the guidance law and the measured acceleration and body rate (see Fig. 9) as inputs to obtain the actuator command. The gains in the autopilot are scheduled as a function of flight condition to achieve missile stability and command following. The actuator command passes through a second-order transfer function with angle and rate limiters.



**Figure 9.** The IMU adds noise to the truth values for the acceleration and body rate and passes the measurement through a discrete transfer function. The noise includes nonlinearities, bias, random-walk noise, and white noise. The guidance law is proportional navigation where the navigation constant,  $\lambda$ , typically is set to four, and the inputs are the line-of-sight rate and closing velocity to the target computed at discrete times.



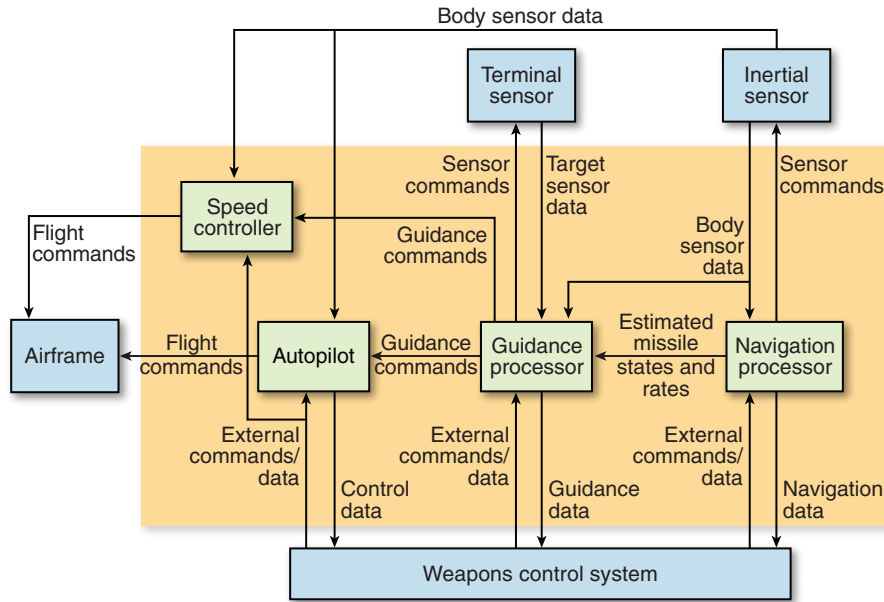
**Figure 10.** Common missile model. The blocks shown illustrate the subsystems required for a generic missile for GNC system studies. The subsystems outside the orange block are external to the missile body, and the ones inside the orange block are onboard the missile. The inertial sensors include any IMU, inertial reference unit, and/or GPS unit that measure and/or estimate the missile’s states. Terminal sensors may include RF or IR seekers that may be strap-down or gimballed and measure the target’s states for guidance. Environment encompasses any physical processes that affect the missile’s operation: gravity, atmospheric pressure, weather, multipath, etc. Target includes the target’s motion as well as any observables, e.g., radar cross-section and irradiance. Weapons control system accounts for initial conditions at launch as well as any uplinked information after launch.



**Figure 11.** Airframe model. The airframe model includes the subsystems that produce the dynamic equations of motion. These subsystems can include missile staging, control surface actuators, TVC vanes, and engine throttling as well as aerodynamics. The equations can be simple—a point mass with trim aerodynamics and perfect actuators and propulsion—or complex—an extended body with flexible bending modes, a center-of-mass offset from the center of control, thrust misalignments, and actuator models with hysteresis, friction, misalignments, etc.

each case, the model tree hierarchy has some basic required components. During the original design of the OSA simulation, separating out the executive portion from the model portion produced a set of diagrams for the critical elements of a generic missile simulation. These elements were dubbed the Common Missile Model<sup>19</sup> and appear in Figs. 10–12. A missile is a physical entity whose motion and orientation are controlled to intercept a target entity with a specified accuracy. The basic elements of a missile include an airframe, inertial sensors to stabilize the attitude and assist in guidance, one or more terminal sensors and/or command uplinks, a flight controller, and models external to the missile such as the launch platform, the threat, and any environmental models (weather, gravity, etc).

An airframe model computes missile states from forces and moments. Missile states are defined as those attributes that define missile translational and rotational motion: typically, they are the Cartesian position, velocity, the quaternion (for orientation), and the angular velocity. The airframe receives steering commands from the flight controller that then are realized as forces and moments by the actuators. The airframe model has several components: dynamics, airframe mass, aerodynamics, propulsion, and flight control actuators (see Fig. 11). Dynamics are the continuous-time missile states (position, velocity, orientation, and angular rates) and state derivatives (velocity, acceleration, angular rates, and angular accelerations) given the forces, moments, and mass characteristics, e.g.,



**Figure 12.** Flight controller. The flight controller encompasses the models for GNC. These models may be simple conceptual equations with no noise, latency, or other error sources, or they may be detailed models of Kalman filters and guidance laws with mode logic for various stages in flight and a high-fidelity autopilot or wrapped flight code. A speed controller is shown for the case when the propulsion may be throttled; for solid propellants, this model may be omitted.

$$\frac{d\vec{R}}{dt} = \vec{V}, \text{ where } \vec{R} \equiv \text{position and } \vec{V} \equiv \text{velocity}$$

$$\frac{d\vec{V}}{dt} = \vec{A}, \text{ where } \vec{A} \equiv \text{acceleration}$$

$$m\vec{A} = \sum \vec{F} = \vec{F}_{\text{gravity}} + \vec{F}_{\text{aerodynamics}} + \vec{F}_{\text{propulsion}},$$

where  $m \equiv \text{mass}$  and  $\vec{F} \equiv \text{force}$

$$\frac{d\vec{Q}}{dt} = f(\vec{Q}, \vec{\omega})$$

$$\frac{d\vec{\omega}}{dt} = \sum \vec{M} = \vec{M}_{\text{gravity}} + \vec{M}_{\text{aerodynamics}} + \vec{M}_{\text{propulsion}},$$

where  $\vec{Q} \equiv \text{quaternion}$ ,  $\vec{\omega} \equiv \text{angular velocity}$ , and  $\vec{M} \equiv \text{moment}$

Usually, the moments act about the center of mass of the rigid body so there is no gravity moment (in a uniform gravitational field). Airframe mass calculates the missile characteristics (e.g., center of mass and moment of inertia) as inputs to the dynamics. Aerodynamics computes the aerodynamic forces and moments as functions of the flight conditions and control surface deflections, i.e., the forces and moments are obtained via lookups and interpolation in multidimensional tables:

$$\vec{F}_{\text{aerodynamics}} = Dynpsr S \begin{bmatrix} C_A(\text{mach}, \alpha, \phi, \delta, \text{altitude}) \\ C_Y(\text{mach}, \alpha, \phi, \delta, \text{altitude}) \\ C_N(\text{mach}, \alpha, \phi, \delta, \text{altitude}) \end{bmatrix},$$

$$\vec{M}_{\text{aerodynamics}} = Dynpsr SD \begin{bmatrix} C_l(\text{mach}, \alpha, \phi, \delta, \text{altitude}, \vec{\omega}) \\ C_m(\text{mach}, \alpha, \phi, \delta, \text{altitude}, \vec{\omega}) \\ C_n(\text{mach}, \alpha, \phi, \delta, \text{altitude}, \vec{\omega}) \end{bmatrix},$$

where  $Dynpsr \equiv \text{dynamic pressure}$ ,  $S \equiv \text{reference area}$ ,  $D \equiv \text{reference diameter}$ ,  $\alpha \equiv \text{angle of attack}$ ,  $\phi \equiv \text{aerodynamic roll angle}$ , and  $\delta \equiv \text{control surface deflections}$ .

Propulsion provides the missile thrust forces, moments, and mass variations. Flight control actuators are those mechanisms that physically realize the control commands, e.g., control surface deflections. For an endo-atmospheric system, the control surfaces typically are tails, ailerons, or canards, but other flight control systems may include thrust-vector control vanes or pulse-width modulated thrusters. The airframe provides truth information to the inertial sensors (typically, accelerometers and rate gyros) and interacts with the physical world through environmental objects such as gravity and atmosphere (Mach, speed of sound, temperature, wind, weather, etc.).

The inertial sensors measure the airframe's motion and feed it back to the flight controller.

Terminal sensors are a collection of objects that provide measurements of the target-relative states (e.g., position and velocity and/or line of sight and line-of-sight rate) to the flight controller. Two major types of terminal sensors are RF and IR receivers. These sensors interact with an environmental object to account for various physical-world sources of measurement noise (thermal noise, multipath, clutter, etc). Various levels of fidelity may be employed in the sensor models used in a GNC simulation, but final performance predictions usually are made with high-fidelity models, particularly for the terminal mode of the engagement.

The flight controller consists of the navigation processor, the guidance processor, the autopilot, and the speed controller (see Fig. 12). Typically, the navigation processor consists of an inertial navigation system (INS). The INS provides estimates of the missile states to the guidance processor based on measurements obtained from rate gyros and accelerometers (the inertial sensors) but may include processing of communication uplinks or possibly GPS measurements. The guidance processor consists of the guidance law(s) and the guidance filter(s). The guidance filter processes the measurements from the terminal sensor(s) and inertial sensors to produce well-behaved estimates of the target states. The guidance law(s) combines these with the estimated missile

states to produce steering commands. Note that there may be a different guidance law and filter for each phase of flight. The autopilot transforms the steering commands to control surface commands for the airframe. In addition, the autopilot may have multiple control options based on the phase in flight. If the missile motor uses solid propellant, the speed controller is omitted; otherwise, it typically throttles the engine to achieve the required speed. All of these flight control functions are, of course, critical elements in a GNC simulation, but they cannot act alone, so the other subsystems and external models are implemented to simulate the actual system's operation.

Many of these state equations are nonlinear and time-varying. In the past, GNC designs have relied on linearization techniques to simplify designs. Within the framework of linear state space systems, the separation principle states that the model observer (INS and guidance filters) does not affect the eigenvalues of the controller (airframe and autopilot), so the development and modeling of these subsystems could be treated independently (see Box 2). However, to achieve greater performance, more modern designs use nonlinear models and control design techniques, and the separation principle breaks down.<sup>20</sup> This more exacting design regime requires greater model fidelity and stricter requirements on the simulation execution, and this performance is provided by a well-designed 6DOF digital simulation.

## CONCLUSIONS

A digital simulation may serve many purposes. For a new concept, a 3DOF simulation explores system performance within the constraints of the proposed model specifications. If the proposed models do not achieve the design objectives, the specifications may be revised or the concept may be scrapped altogether—without the high cost of hardware prototyping and flight testing. For a system under development, a 6DOF simulation verifies expected performance given the design specifications, provides an operational platform during hardware prototyping, and is validated during flight testing. It is an essential tool for designing operational flight tests that exercise the various subsystems at selected points in the performance envelope. It also is essential for identifying and illuminating key sensitivities in the existing hardware of an operational system.

Simulation architecture requirements do not vary with the level of model complexity. In all cases, the simulation must provide synchronous propagation of state equations with support of asynchronous events and repeatable random processes for correct mathematical modeling. For continued utility throughout the development process, the simulation architecture should additionally support the following:

### BOX 2. SEPARATION PRINCIPLE

Consider the linear system:

$$\begin{aligned}\dot{x} &= Ax + Bu \\ y &= Cx,\end{aligned}$$

where  $x$  is the state vector,  $u$  is input vector, and  $y$  is the output vector.

Then an observer for this system is of the form:

$$\begin{aligned}\dot{\hat{x}} &= A\hat{x} + L(y - \hat{y}) + Bu, \\ \hat{y} &= C\hat{x}, \\ \text{or} \\ \dot{\hat{x}} &= (A - LC)\hat{x} + Bu + Ly,\end{aligned}$$

where  $\hat{x}$  is the estimated state. Because the feedback control will use the estimated state, then

$$u = -K\hat{x}.$$

If the error vector is defined such that

$$e = x - \hat{x},$$

then

$$\begin{aligned}\dot{e} &= (A - LC)e \\ \text{and} \\ u &= -K(x - e).\end{aligned}$$

Finally, the system equations become

$$\begin{bmatrix} \dot{x} \\ \dot{e} \end{bmatrix} = \begin{bmatrix} A - BK & BK \\ 0 & A - LC \end{bmatrix} \begin{bmatrix} x \\ e \end{bmatrix}.$$

These equations illustrate the separation principle of designing a controller and an observer independently for linear systems. The last matrix equation is block-triangular so the eigenvalues for  $x$  and the eigenvalues for  $e$  are independent with  $K$  influencing  $x$  and  $L$  influencing  $e$ . These conditions fail for nonlinear control laws and observers.

- Real-time execution for hardware evaluation via hardware-in-the-loop and/or computer-in-the-loop
- Multithreaded execution to help achieve real-time performance
- Distributed processing both to achieve real-time performance and to operate in an HLA environment
- Portability and maintainability
- Scalability

For GNC studies, the missile 6DOF simulation must implement the subsystems shown in Figs. 10–12. The sophistication of the models depends largely on

the information that is available for the system—new concepts may not have any information besides general specifications—and the engineering question that is being asked of the system. In some cases, a simple Matlab or Simulink model may be an appropriate tool for investigating a concept; however, as more model information becomes available and/or hosting multithreaded flight code becomes a requirement, transitioning to a 6DOF implementation makes sense. Aerodynamic tables can be wrapped in a “mex” file for use in Matlab and Simulink, but the user does not have the control over the synchronization at time-step boundaries that a 6DOF implementation provides. For run-time execution, a 6DOF implementation can be refactored to run in a distributed or multithreaded configuration or run in a federation with other system simulations (for example, high-fidelity radar or threat simulations); at this time, Matlab and Simulink do not have these capabilities. This control of the model equation propagation and information flow between models is particularly important for the nonlinear, highly coupled designs currently being developed. 6DOF performance predictions for these systems are a valuable tool for risk and cost reduction during missile system development and deployment.

## REFERENCES AND NOTES

- <sup>1</sup>DoD *Modeling and Simulation (M&S) Glossary*, DoD 5000.59-M, Defense Modeling and Simulation Office, Washington, DC (15 Jan 1998).
- <sup>2</sup>Tsiolkovsky, K. E., “The Exploration of Cosmic Space by Means of Reaction Device,” *Sci. Rev.* (5), in Russian (1903).
- <sup>3</sup>Etkin, B., *Dynamics of Atmospheric Flight*, Dover Publications, Mineola, NY (2005).
- <sup>4</sup>Korn, G. A., “Continuous-System Simulation and Analog Computers: From Op-Amp Design to Aerospace Applications,” *IEEE Control Syst. Mag.* 25(3), 44–51 (June 2005).
- <sup>5</sup>Hayes, B., “The Post-OOP Paradigm,” *Am. Sci.* 91(2), 106–110 (Mar–Apr 2003).
- <sup>6</sup>Subramaniam, G. V., and Byrne, E. J., “Deriving an Object Model from Legacy Fortran Code,” *Proc. 1996 Int. Conf. on Software Maintenance*, Monterey, CA, pp. 3–12 (4–8 Nov 1996).
- <sup>7</sup>Ross, J. M., and Zhang, H., “Structured Programmers Learning Object-Oriented Programming,” *SIGCHI Bull.* 29(4), 93–99 (Oct 1997).
- <sup>8</sup>Scope is an enclosing context where values and expressions are associated; for example, global scope implies the variable is visible and modifiable everywhere in the program. Local scope means that the variable is visible and modifiable only in the local (subroutine) context.
- <sup>9</sup>Private indicates that the data are hidden within the class and out of scope outside that class, and it is a keyword in C++ and Java.
- <sup>10</sup>Encapsulation refers to the principle of information hiding in which the interface of a class (its *public* operations) are separated from its implementation (its *private* data and *private* operations). This technique allows the code developer to change the implementation without violating the contract with the user stated by the public interface.
- <sup>11</sup>Inheritance is a generalization of a base class’ implementation. A derived class inherits data and behavior from its base class as it reuses and extends its operations. For instance, an Autopilot is—a ModelObject indicates that Autopilot should inherit from ModelObject. Inheritance therefore produces a hierarchy between classes of objects.
- <sup>12</sup>Polymorphism lets the user invoke the functions specified by the base class’ interface but achieve the derived class’ behavior. This binding of behavior can be done at run time and yields plug-and-play swapping of ModelObjects. Another form of polymorphism, parametric polymorphism, refers to the compile-time polymorphism achieved with C++ templates or Java Generics.
- <sup>13</sup>Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Upper Saddle River, NJ (1995).
- <sup>14</sup>The Common Object Request Broker Architecture, or CORBA, defined by the Object Management Group is a middleware standard for the Broker pattern. Implementations of CORBA enable simulations on different platforms written in different programming languages to execute together.
- <sup>15</sup>A thread is a sequence of computer instructions executed by a CPU core. A thread shares the address space with other threads in the same process. A single CPU core time-slices the execution of multiple threads, but multiple threads can run simultaneously on separate CPU cores.
- <sup>16</sup>IEEE Computer Society, *IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)—Framework and Rules*, IEEE Standard 1516-2000, doi: 10.1109/IEEESTD.2000.92296 (2000).
- <sup>17</sup>Lansberry, J. E., *Momentless Missile Dynamics Model with a Rotating Earth*, Technical Memorandum A1E(04)U-2-001, JHU/APL, Laurel, MD (9 Feb 2003).
- <sup>18</sup>Murtaugh, S. A., and Criel, H. E., “Fundamentals of Proportional Navigation,” *IEEE Spectrum*, 75–85 (December 1966).
- <sup>19</sup>Chiu, H. Y., *A1E OO Working Group Presentation Viewgraphs, 26 February 1997*, Technical Memorandum A1E(97)-2-040, JHU/APL, Laurel, MD (5 Mar 1997).
- <sup>20</sup>Palumbo, N. F., Reardon, B. E., and Blauwkamp, R. A., “Integrated Guidance and Control for Homing Missiles,” *Johns Hopkins APL Tech. Dig.* 25(2), 121–139 (2004).

# The Authors

**Patricia A. Hawley** is a member of APL's Senior Professional Staff in the Guidance, Navigation, and Control (GNC) Group of the Missile Systems Branch in the Air and Missile Defense Department. She holds a B.A. in astronomy and physics, an M.S.E.E. from Purdue University, and an M.S. in applied physics from The Johns Hopkins University Whiting School of Engineering. Ms. Hawley has pursued additional graduate courses in electrical engineering and mathematics at the University of New Hampshire. Her area of expertise is in 6-degree-of-freedom simulations of guidance, navigation, and control systems. She is a member of the Institute of Electrical and Electronics Engineers and the American Institute of Aeronautics and Astronautics. **Ross A. Blauwkamp** received a B.S.E. degree from Calvin College in 1991, and an M.S.E.



Patricia A. Hawley



Ross A. Blauwkamp

degree from the University of Illinois in 1996; both degrees were in electrical engineering. He continues to pursue a Ph.D. from the University of Illinois. Mr. Blauwkamp joined APL in May 2000 and currently is the supervisor of the Advanced Concepts and Simulation Techniques Section in the GNC Group. His interests include dynamic games, nonlinear control, and numerical methods for control. He is a member of the Institute of Electrical and Electronics Engineers and the American Institute of Aeronautics and Astronautics. For further information on the work reported here, contact Patricia Hawley. Her e-mail address is [patricia.hawley@jhuapl.edu](mailto:patricia.hawley@jhuapl.edu).