

# IMPLEMENTING DYNAMIC SYSTEM MODELS IN THE ASSET SIMULATION FRAMEWORK

Stephen D. Derry\*, Jeffrey Maddalon\*

NASA Langley Research Center  
24 W. Taylor Road  
Hampton, VA 23681

## Abstract

The Aircraft System Simulation Environment and Toolkit (ASSET) project is being undertaken as a proof of concept, aimed at demonstrating a rapid model prototyping and modification capability by utilizing modern software technologies and design concepts. Initially, the project is focused on a self-contained batch simulation, but it is designed to ultimately run embedded within other software environments, including real-time. This report focuses on the design and implementation of the model software, which strictly follows a block-diagram design paradigm. The initial implementation, a six-degree-of-freedom simulation of an F-16 airplane, is presented, along with current status and near-term plans.

## Introduction

NASA Langley Research Center utilizes flight simulation, both batch and real-time (pilot in the loop), to support research in a number of disciplines, ranging from control system development and handling qualities evaluation to human factors. Because these simulations are used for research rather than operational purposes (such as training), they are subject to frequent change.

---

\* *Computer Engineer, Systems Development Branch, Airborne Systems Competency.*

Copyright © 2000 by the American Institute of Aeronautics and Astronautics, Inc. No copyright is asserted in the United States under Title 17, U.S. Code. The U.S. Government has a royalty-free license to exercise all rights under the copyright claimed herein for Governmental purposes. All other rights are reserved by the copyright owner.

Several of the research disciplines require frequent modifications to the simulation model itself, to either the aircraft plant, the associated control system, or both. To support a rapid design iteration rate, the researcher should be able to modify the model and test the simulation in a desktop workstation environment without the support of software specialists. This desktop batch simulation must be capable of producing trim solutions and time-history evolutions; additionally, linearized models, possibly of reduced order, must be extracted from the full non-linear model. Finally, the non-linear model must be portable to a real-time environment for piloted simulation and/or flight test, preferably without recoding.

ASSET will attempt to demonstrate rapid model software development and modification in a desktop workstation batch simulation environment that supports the three primary operations described above (trim, linearization, and time-history evolution).

## ASSET Software Overview

To facilitate development of simulation model software by non-software experts, ASSET is designed to be functionally de-coupled and easy to learn. In this context, functional de-coupling refers to the design goal of isolating the software context in which models operate. Model software can then be developed and/or modified without knowledge of the context in which the model is to be run (i.e. whether the model is a stand-alone system or part of a larger system), and with little knowledge of the software architecture of the other parts of the ASSET framework. Functional de-coupling should also enhance the re-usability of the software.

Object-oriented design techniques are employed throughout the ASSET project. To maintain simplicity of design and ease of learning, the class hierarchy (levels of inheritance) is intentionally kept shallow – typically two or three levels deep at the most.

ASSET uses the Java programming language [1]. Java was chosen primarily for its portability, support for the object-oriented design paradigm, and ease of learning, as well as the breadth of industry support and the availability of already-developed software (such as vector and matrix classes and graphical user interface foundational classes).

ASSET consists primarily of six major components: applications, models, a data store, data collectors, a user interface, and the framework of common software. A given simulation project will typically have a unique application and models, but use standard ASSET classes for the remaining components. This paper focuses on the ASSET concept of a model and its software realization. A companion paper [2] discusses the data store and the data collectors.

Initially, ASSET is targeted to run in a stand-alone batch environment, providing a desktop (non real-time) simulation environment with a graphical user interface (GUI). Long-term plans are to embed ASSET into other simulation software environments, including commercial products in a desktop batch environment and, ultimately, real-time environments for piloted simulation. Once ASSET is embedded into a simulation environment, then models developed for ASSET will run in that environment.

When embedding ASSET models into a non-ASSET environment, the embedded ASSET components will typically include the models, data store, data collectors, and portions of the framework, along with interfacing software specific to the environment in which the ASSET components are embedded.

The design of ASSET allows for it to be embedded into a real-time environment, although ASSET itself does not provide a real-time environment. To this end, ASSET does not create any objects in the run-time portion of the software, which eliminates the necessity of running garbage collection in real-time.

Garbage collection is a mechanism used by Java to reclaim unused memory to make it available for the creation of new objects. Garbage collection is undesirable for real-time operations because it is time consuming and its invocation generally occurs at unpredictable times.

One final aspect of ASSET is that it is designed not just for flight simulation. While many components of the framework are specific to flight simulation (equations of motion, etc.), ASSET can be used to support the modeling and simulation of any set of dynamic systems.

## **Software Implementation of Models**

### **Basic description and class design**

The Model super-class is the basic abstraction for all mathematical models in ASSET. A Model object represents a block in a block diagram. Models have inputs, outputs, and perhaps states. Note that a block diagram typically depicts the inputs and outputs for the individual blocks, but the states are not explicitly shown. Similarly, the input and output vectors for ASSET Models are visible to the block's "owner", but the states are generally not. The "owner" of a model object is usually the object which created it.

All classes which implement models are sub-classes (descendants) of the Model super-class, either directly or indirectly. The Model super-class manages certain bookkeeping functions, such as mode and time, whereas the sub-class implements the mathematical state and output equations that govern the behavior of the model.

All models have essentially the same software interface. Most of the methods defined for a model are implemented in the super-class; the remaining methods defined in the sub-class usually over-ride methods of the super-class. Rarely are additional, model-specific, methods needed. The uniformity of the software interface for all models simplifies the software design, both for the models themselves and the upper layers of software that use the models.

### **Signals and model interface vectors**

Model input and output vectors are implemented as Java arrays of double's, with one double for each input and output signal. Each of the signals has a

unique name, and both the names and values for the input and output signals are accessible to the model's owner. To improve run-time execution performance, signal name lookups are typically done only once, at construction time, and the corresponding indices are retained for use during run-time.

A model may also contain states, which are defined and maintained by the model and not generally visible outside. ASSET supports both continuous and discrete states, and a given model may contain either or both types. The state vectors are also implemented as arrays of `double`'s with unique names for each state. State and state derivative vectors are maintained for the continuous states, whereas state and next-state vectors are maintained for discrete states. Special methods in the Model super-class provide access to the state-related vectors, primarily to support linear model extraction; these methods are not intended for general use.

An additional interface, the parameter vector, is used by the application to pass signals to the model without intervention by the model's owner. These parameters can be used to adjust gains, etc. for control systems, or to trigger failures or other anomalous events.

Finally, a model may select certain of its quantities to be sampled for recording and analysis purposes; these may include internal signals that are not otherwise visible. The sampling and recording are accomplished via the data store and data collectors and are described in [2]. Other than defining which of its quantities are to be sampled, the model sub-classes do not participate in the sampling process.

### Units

All quantities in ASSET are internally maintained (i.e. in variables) in a set of coherent internal units. A set of units is coherent if it has exactly one unit for each type of quantity, and is mutually related by multiplication and division without conversion factors [3]. For example, the area unit would be the distance unit squared, and the unit of velocity would be the distance unit divided by the unit of time.

The modeler should make no assumptions about what actual units are used internally—as long as all quantities are maintained in internal units, the results will be con-

sistent. Whenever a numerical quantity is used, whether as an input, output, or a hard-coded constant, the corresponding unit conversion must be performed so that the quantity is stored internally with the correct coherent unit.

ASSET can be configured to use either of two internal coherent unit systems: the International System of Units (SI) [3] or a customary set of units based on the foot, slug, second, and degree Rankine. As long as all unit conversions are properly specified, the actual selection of the internal unit system is transparent and arbitrary. The SI system is generally used.

### Embedding models

Just as a block in a block diagram may itself be specified as a lower-level block diagram, an ASSET model may embed models within it. In this case, the embedded model is a separate object which is “part of” the containing model. The containing model uses the same interface (input and output vectors, etc.) and methods to communicate with the embedded model.

The software that implements the embedded model has no dependency at all on the containing model. This means that models may be designed with no knowledge of the context(s) in which they are to be used, whether they are to stand alone or to be part of a larger system. This aspect of ASSET's design facilitates testing as the model software is not changed between unit testing and incorporation into a larger system. It also facilitates re-use since the model software is not sensitive to its context.

A model's states are often contained in its embedded models. The ASSET framework will include basic models, such as integrators and transfer functions, which are typically employed to manage the states of a larger model. Due to the uniform model interface, it is transparent to a model's owner whether the model's states are contained directly in the model or within embedded models. To maintain this transparency, the special methods (required to support linearization) for accessing and manipulating a model's state vector are recursive: they operate on all states contained in the model itself and in all embedded models.

Each model is given a name by its owner; this name is unique among its siblings (i.e. among all models created by the same owner object). Each model also has a hierarchical name which is unique among all models defined in the system. The hierarchical name for a model is its given name appended to the hierarchical name of its parent (containing) model, if any. For example, the name of an aileron model contained within a standalone model of an open-loop F16 aircraft might be "F16OpenLoop.Aileron".

### Model execution cycle

A model execution cycle starts with the model's owner establishing the appropriate quantities in the model's input vector. The owner then invokes the `runModel()` method which is defined in the Model super-class. The `runModel()` method performs the following actions, if required, depending on the current mode (modes are described in a separate section below) and other model settings:

1. Advance time
2. Propagate states to the new time by invoking the sub-class method `propagate()`
3. Invoke the sub-class method `calculateModel()`
4. Invoke model sampling via the DataStore

After this completes, the model's current outputs are available in its output vector.

The `calculateModel()` method, which is implemented in the sub-class for each specific type of model, begins by initializing the model's states if required for the current model mode. The initial values for the states are determined either by forcing steady-state conditions based on the current inputs, or by utilizing special input signals which serve solely to provide state initialization data from outside the model (see Model residualization below). After the states are initialized (if necessary), the primary model calculations are performed: state derivatives for continuous states, next values for discrete states, and model outputs. These calculations should be the same, regardless of the model mode. This helps insure that the model behavior is consistent between modes by minimizing special "reset" code. If the model embeds other models, the `runModel()` method should be invoked on the embedded models.

The `propagate()` method, which is also implemented in the sub-class for each type of model, propagates the states to the next time step. Continuous states are integrated using the state derivative calculated for the previous step (and perhaps other data, depending on the specific integration method used), and the discrete states are advanced to their next values which were also calculated in the previous step. The `propagate()` method need only be defined if the model directly maintains its own states. Frequently, states are embedded in standard models such as filters and integrators, in which case an implementation for `propagate()` is not necessary.

The ASSET framework currently includes a small set of standard integrators and transfer functions which are implemented using single-pass algorithms suitable for use in a real-time environment. This set of integrators and transfer functions will be expanded as necessary, and it is anticipated that these standard models will be used to maintain most states.

### Time management

A variety of time and mode management services are provided by the Model super-class. The primary time management functions are the maintenance of the time and time step for each model. The time step for top-level models (those which are not embedded inside a containing model) is established by the model's owner. By default, embedded models inherit the time step of their containing model; however, the time step may be set to a multiple or a sub-multiple of the containing model's in order to sub-rate or super-rate the model. This capability allows different parts of the system to run at different rates, transparently to the model and application software.

Super-rating an embedded model means that several cycles of the embedded model are run for each cycle of the containing model. This is useful for modeling a subsystem which contains higher frequency dynamics than the full system without requiring the entire system to be modeled at the smaller time step. Sub-rating is the opposite: the embedded model is only evaluated once for several cycles of the containing model. Sub-rating is primarily provided to reduce execution time for subsystems whose dynamics are appreciably slower than

the system as a whole. To make sub-rating useful for a real-time environment, a set of sub-rated models may be phased. For example, a model of a twin-engine airplane may run its engine models on alternating time steps. Super-rating and sub-rating are handled entirely by the Model super-class.

### Modes

ASSET models can be in one of three modes: Initialize, Steady, and Dynamic. When a model is in Initialize mode, the states are initialized prior to the calculation of the state and output equations, and time does not advance. In Steady mode, the states are initialized similar to the Initialize mode, but time advances. In Dynamic mode, time and states are advanced and integrated prior to the calculations. Other than the state initializations for the Initialize and Steady modes, the mode should be transparent to the modeler and the model sub-class. In addition to simplifying the software, this helps insure consistent results.

The Initialize mode is used to establish and/or trim the initial conditions, while the Steady and Dynamic modes are used to produce a time-history evolution starting from the initial conditions. A model in the Dynamic mode will respond according to the dynamics described by its state and output equations, while a model in the Steady mode is forced to produce a steady-state response as described in the next section (model residualization).

To initialize the system, all models are placed in the Initialize mode. To run the system (over time), all models are placed in either the Dynamic or Steady mode. Normally, all models will be set to Dynamic mode; however, selected models may be placed in Steady mode to reduce the complexity of the dynamics being studied. For example, actuator or engine models could be placed in Steady mode so that their steady-state response is produced instantaneously while removing their transient response. This can be used to produce a reduced-order linear model of the aircraft.

While running (i.e. not initializing), the system as a whole may also be suspended, which means that the passage of time and propagation of states does not occur. While suspended, only the normal state and output

calculations, including (re)initialization of states if in Steady mode, are performed by the `runModel()` method. This feature is used to produce linear models of the system by allowing the application to perturb each state and input, in turn, and observe the system response in the output vector and the state derivatives. Special methods are provided by the Model super-class to access the state and state derivative vectors for this purpose. Note that although only the top-level model's inputs and outputs are visible, the state and derivative vectors include the states and derivatives of all embedded models.

Suspension and the special state vector access methods could also be used to support the use of an external integration algorithm, although this has not been fully designed yet. (Additional methods would be needed in order to manually adjust the time and time step.) Such external integration methods would be implemented separately from the model and would allow the user to provide alternate integration algorithms, including multi-pass algorithms such as Runge-Kutta, possibly with variable step size.

### Model residualization

In ASSET, certain models are considered to be residualizable, meaning that the states can be initialized such that the model is placed in a steady-state condition for the current inputs. Steady state condition is defined to occur when the residual error is zero, where the residual error is the root-sum-square of the continuous state derivatives and the discrete state differences (next state – current state).

This feature allows the user to optionally remove the dynamics of selected residualizable subsystem models, such as aircraft actuators, by placing the subsystem models in Steady mode while the rest of the system is in Dynamic mode.

One area of difficulty for determining a steady-state solution is that some models contain implicit loops in their state equations.<sup>1</sup> A standard approach is to iterate

---

<sup>1</sup> Implicit state equations occur when the state derivatives are computed as functions of several quantities, including the state derivatives or equivalent forms. This

the calculation of the states and derivatives until the residual error is driven suitably close to zero.

Classes which implement residualizable models identify themselves to the Model super-class as such. This enables the super-class to automatically calculate the residual error and iterate the invocation of `calculateModel()` if the model is in Initialize or Steady mode.

Note that models which are not residualizable require special inputs for initializing the states. Since these states require external participation in their initialization, iterative calls to `calculateModel()` are not performed on these models.

### Signal wiring

One of the more time-consuming parts of writing software for models in a block-diagram paradigm is the passing of signals from one model to another, i.e. wiring the connections. For example, ASSET's model for the aircraft rigid equations of motion has over two hundred input and output signals.

ASSET provides several framework classes to assist with this chore. One set of classes implement busses, which aggregate a number of signals. Busses provide a structure for managing the names for a group of signals, and for mapping the names so that the two sides of the "connection" can name the signals differently. Input, output, state, and parameter vectors can be defined in terms of busses as well as individual signals. Standard busses are defined for basic aggregate quantities, such as vectors, matrices, and quaternions, which are used frequently and are composed of multiple signals. Another class, the Signal List, automates the transfer of quantities between the interface vectors and primitive data types which are used in the model code.

If a model can be described completely in terms of a set of embedded models wired together, the CompositeModel super-class can be used to gain significant leverage in the coding of such model classes. CompositeModel, which is a sub-class of Model, manages the execution of such models. In this case, the model devel-

---

is common in aerodynamic models, where aerodynamic coefficients are functions of the angle-of-attack derivative and pitch-rate derivative, for example.

oper merely has to define a sub-class of CompositeModel with a constructor; no other methods need to be defined in the sub-class. The constructor must create the embedded models, establish their wiring and execution order, and describe the data which should be included in the sampling. The CompositeModel super-class provides the methods, such as `calculateModel()`, which are needed for run-time execution. Standard algebraic models, such as gains, summers, and limiters, have been added to the framework to support the use of composite models.

### External model interfaces

The block-diagram paradigm, with successive levels of decomposition of a model into embedded models which are "part of" the containing model, has its limitations and cannot be used as a basis for all inter-model communications. Namely, there is occasionally a need for models which are not part of the same system to communicate (i.e. exchange data) with each other. For example, consider the case of an aircraft and an atmosphere. Models can be built for each, but neither entity is "part of" the other; each can exist without the other, but the presence and characteristics of one has a great influence on the behavior of the other!

ASSET uses the Java interface mechanism<sup>2</sup> so that one model can invoke specialized methods of another model. This use of the Java interface adds flexibility to the object-oriented design of the modeling software. However, because it is a special-purpose mechanism, its use is kept to a minimum to avoid excessive complexity in the software design.

### **Early Implementation – F16 Simulation**

The first substantial system to be simulated in ASSET is a simplified F-16 model, described in [4]. This system provides an illustrative example of a hierarchy of embedded ASSET models with several external interfaces.

---

<sup>2</sup> Java interfaces basically provide declarations of methods, including parameters and return types, without defining the implementations of those methods. A Java class can be declared to implement an interface, which means that it must define an implementation for all of the methods declared in the interface.

An object diagram for this system is shown in Figure 1, and a class diagram is shown in Figure 2. Note that all of the model classes are either sub-classes of Model or CompositeModel. All of the states in this system are maintained in the KinematicBody and Integrator objects.

The six-degree-of-freedom rigid-body equations of motion are implemented by the AircraftRigidEOM class, which is a sub-class of CompositeModel. These equations are mathematically similar to those used in the current piloted flight simulation framework at Langley, LaSRS++. LaSRS++ is described in [5] and is written in C++ using object-oriented techniques.

The DynamicRigidBody model receives as inputs the external forces and moments acting on the aircraft as well as the aircraft mass and inertia, and computes the resulting inertial accelerations, both translational and rotational. These signals are then passed to a KinematicBody model, which maintains and propagates the aircraft's position and velocity states, both translational and rotational. All of the kinematic states are referenced to an earth-centered inertial Cartesian coordinate system. KinematicBody has built-in integrators for maintaining the rotational and translational velocities (second-order Adams Bashforth) and translational position (truncated Taylor series). It maintains the aircraft attitude as a quaternion, which is propagated by a discrete algorithm described in [6].

Both the DynamicRigidBody and KinematicBody classes interface with an external World model for calculating world-relative positions, velocities, and accelerations as well as the gravity vector. This allows a variety of world models to be used without modifying the basic dynamics classes. Currently, only a flat earth model is implemented, but an ellipsoidal earth model will be developed in the near future.

AirFlow is a CompositeModel which embeds an AirPath model and a GasDynamics model. The AirPath model interfaces with an external Wind model to compute the aircraft's air-relative velocity, given its current world-relative velocity. The GasDynamics model computes such quantities as Mach number, equivalent and calibrated airspeeds, dynamic and total pressures, and total temperature. It takes the true airspeed as an input,

and interfaces with an external Atmosphere model to obtain the ambient atmospheric characteristics (static temperature and pressure, density and density gradient, and speed of sound) at the current location of the aircraft. The AirPathAccel model is used to calculate the derivatives over time of the air-relative velocity, such as the derivatives of the angles of attack and sideslip. It is a separate model because the aircraft accelerations are not available when the AirFlow model is invoked.

The current atmosphere model is based on the 1976 U.S. Standard Atmosphere, with provisions for local temperature and pressure deviations. The only wind model implemented at this point is one which provides a constant wind. Additional wind models, along with stochastic turbulence models, are anticipated to be added in the future.

The simplified F-16 vehicle model is taken from [4], and consists of several ASSET models, all embedded within the F16OpenLoop class which is a CompositeModel.

The aerodynamic forces and moments are computed in the F16Aerodynamics class. The aerodynamics model is non-linear and includes eighteen one- and two-dimensional function table lookups with linear interpolation.

The engine dynamics and thrust are computed in the F16Engine class. The engine model includes three two-dimensional thrust tables, and the engine dynamic state is maintained by an ImplicitTrapezoidalIntegrator which is embedded in the F16Engine model.

The three aerodynamic control surfaces defined in the aerodynamics model are positioned by three instances of the ActuatorOrder1 model (namely the aileron, elevator, and rudder objects), each of which embeds an ImplicitTrapezoidalIntegrator.

Any combination of the engine and/or actuator models may be residualized in order to eliminate the effect of their dynamics and thereby reduce the complexity of the system as a whole. This feature may be used to reduce the model for linearization and/or time-history evolution.

Finally, the aircraft mass and inertia properties, as well as the location of the CG, are produced by the F16MassGeometry model (the location of the CG is set via a parameter to this model). The F16SumForces model combines the forces and moments from the engine and aerodynamics models and transforms the moment to the actual aircraft CG, and AircraftRigidEOM is invoked to compute the dynamics of the aircraft.

This implementation consists of a system of twenty-two ASSET models, including a hierarchy of nineteen models for the F-16 aircraft and three independent models for the environment.

### Current Status

The portions of the ASSET framework that are essential for simulating 6-DOF rigid aircraft in a simplified environment (flat earth with constant wind) have been developed, along with two simple aircraft models (the F-16 mentioned above and a linearized model of a North American Navion). The Navion model and the rigid aircraft six-degree-of-freedom equations of motion have successfully passed an initial validation by comparison against an established simulation described in [7].

To date, ASSET development activities have taken place on Windows NT 4.0 platforms using the Sun JDK 1.2.2 Java development environment, which includes a just-in-time compiler. Performance measurements of the F-16 model running on a 550 MHz. Intel Pentium III with 256 MB of RAM have resulted in the following times:

- 230 microseconds per step with sampling each step
- 195 microseconds per step without sampling

The entire F-16 model, including equations of motion, was exercised each step, and the sampling recorded 210 signals. These tests were performed in a batch environment with no user interface.

Near-term development plans include a graphical user interface, implementation of simple control systems to demonstrate closed-loop simulation, and incorporation of trim and linearization capabilities. (Linearization has already been demonstrated in ASSET on a simple linear actuator model.) Following this, one or more additional aircraft models which are of current research interest will be developed in ASSET in order to assess the soft-

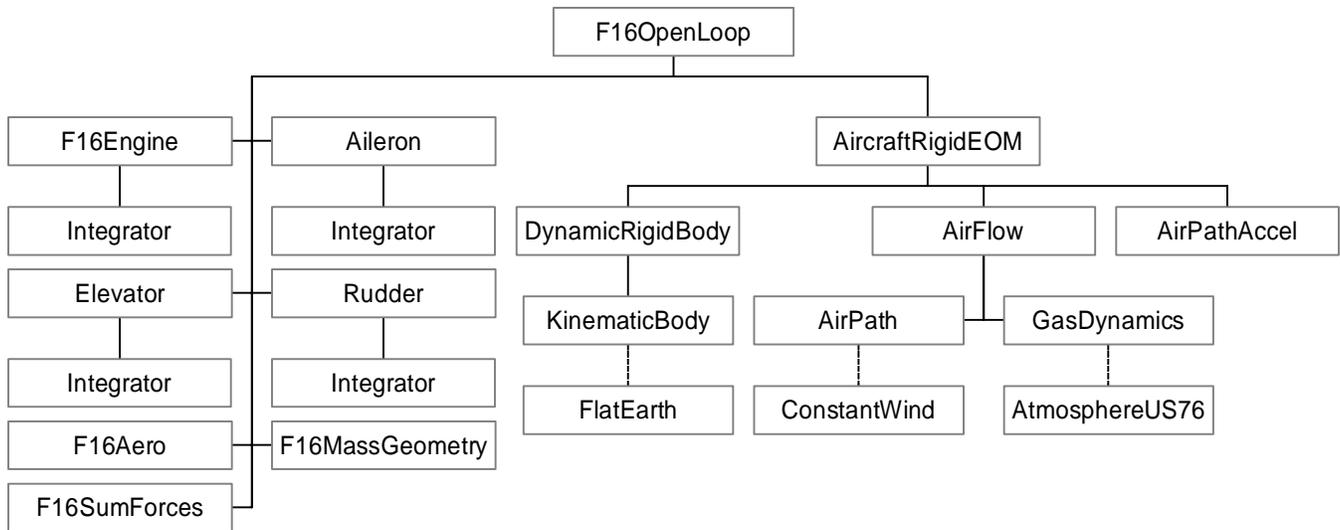
ware development effort required and to demonstrate ASSET's usefulness in supporting Langley's research mission.

### Conclusions

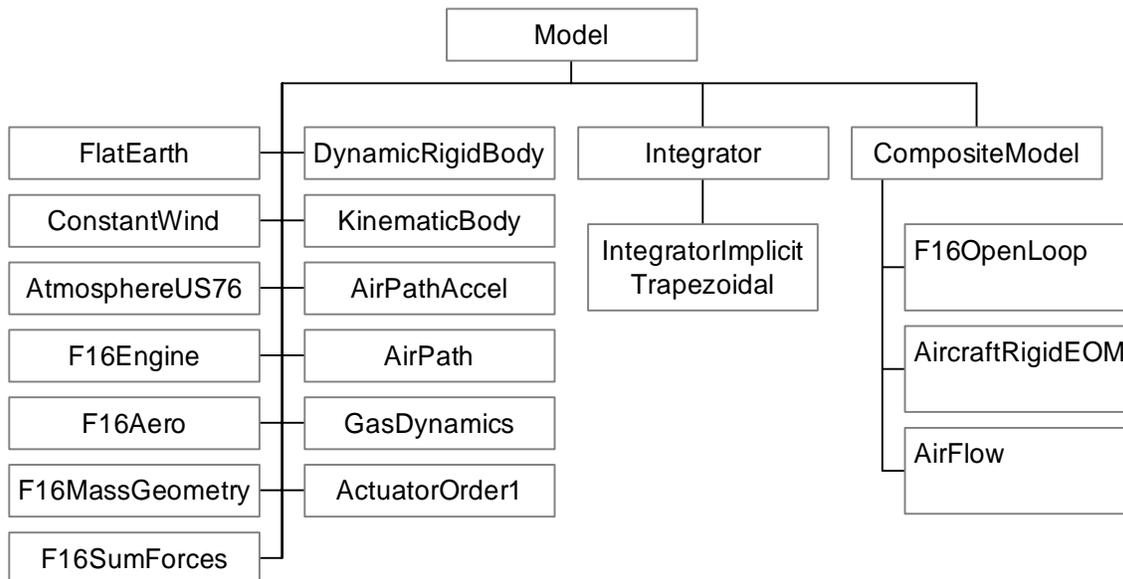
Progress with ASSET to date demonstrates that the block diagram paradigm is a useful design for simulation model software, and that Java is a viable platform for simulation software development. Follow-on efforts will seek to determine ASSET's effectiveness in reducing model software development time.

### References

- <sup>1</sup> Arnold, Ken; Gosling, James. *The Java Programming Language, Second Edition*. Addison Wesley Publishing Company, Reading, MA, 1998. ISBN 0-201-31006-6.
- <sup>2</sup> Maddalon, Jeffrey; Derry, Stephen. *Data Management in the ASSET Simulation Framework*. AIAA 2000-4500. Modeling and Simulation Technology Conference, Denver, CO, August 2000.
- <sup>3</sup> Taylor, Barry N. *The International System of Units (SI)*. NIST Special Publication 330, National Institute of Standards and Technology, August 1991.
- <sup>4</sup> Stevens, Brian L.; Lewis, Frank L. *Aircraft Control and Simulation*. John Wiley and Sons, Inc., New York, NY, 1992. ISBN 0-471-61397-5.
- <sup>5</sup> Leslie, R.; Geyer, D.; Cunningham, K.; Madden, M.; Kenney, P.; Glaab, P. *LaSRS++: An Object-Oriented Framework for Real-Time Simulation of Aircraft*. AIAA-98-4529, Modeling and Simulation Technology Conference, Boston, MA, August 1998.
- <sup>6</sup> Barker, Lawrence E. Jr.; Bowles, Roland L.; Williams, Louise H. *Development and Application of a Local Linearization Algorithm for the Integration of Quaternion Rate Equations in Real-Time Flight Simulation Problems*. NASA TN D-7347. December 1973.
- <sup>7</sup> Jackson, E. Bruce; *Manual for a Workstation-Based Generic Flight Simulation Program (LaRCsim) Version 1.4*. NASA TM-110164, May 1995.



**Figure 1. F-16 Model Object Diagram**



**Figure 2. Model Class Diagram**

