An Unmanned Aerial Vehicle Simulation Framework for Training Simulation and Simulation-based Design

Dr. Michael Selig
University of Illinois Urbana-Champaign
m-selig@uiuc.edu

Walter Malley
Naval Air Systems Command

Daniel Lavery
National Technology Associates, Inc.

Frank McCoy
FPM Enterprises, Inc.

Jeffrey Wallace
Space and Naval Warfare Command Systems Center San Diego
jwallace@spawar.navy.mil

ABSTRACT

We present a completely general object-oriented flight vehicle simulation which is written in C++ and used to train unmanned aerial vehicle (UAV) pilots. To illustrate the utility of the design, an IMPORT CASE tool simulation-based design implementation of a more detailed UAV scenario is shown.

This re-uses the high fidelity vehicle dynamics from the C++ framework in the IMPORT simulation. Then, using the flexibility of IMPORT, we demonstrate how to quickly prototype a detailed, automatic control system and electromechanical device simulation framework (ACS/EMD) to implement the UAV avionics system, mechanical actuators, and sensors.

INTRODUCTION

The Department of Defense (DOD) is beginning to rely on simulation to do training, design new systems, and develop courses of action for the myriad of decisions facing DOD personnel, both military and civilian. To simplify the task of managing modeling and simulation, we advocate a technology approach that maximizes simulation software reuse. That is, if we have a real-time pilot training simulation for UAV operators, we want the design of the permit simulation-based design on the same UAV system.

To accomplish this goal, we present the six-degree-of-freedom equation of motion formulation implemented in the flight vehicle modeling (FVM) framework. The conceptual design for the FVM framework is presented in Object Modeling Technique (OMT) [Rumbaugh 91]. We then demonstrate how to employ the C++ flight vehicle modeling framework in an IMPORT simulation-based design problem requiring very high level of detail modeling. To implement these requirements, as illustrated in Figure 1, we describe a framework for modeling automatic control systems and electromechanical devices to model UAV avionics, actuators, and sensors.

In the process of presenting the ACS/EMD framework, the parallel semantics and synchronization constructs of the process based, object-oriented simulation CASE tool IMPORT [Whitehurst 95] will be explained.

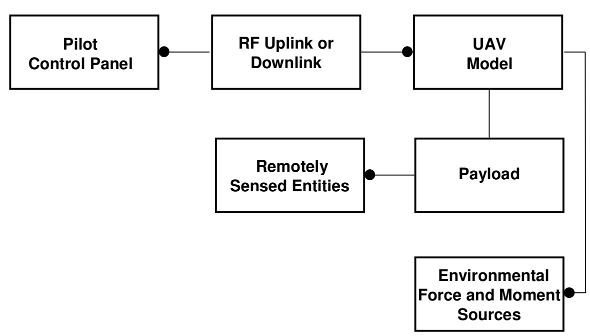

Figure 1. OMT Conceptual Model of UAV Simulation.

MVC-BASED DESIGN

The general purpose FVM approach features the Model-View-Controller (MVC) framework [Leonard 97]. We explain how the MVC paradigm is ideal for model reuse in our example. We enumerate several advantages of the MVC philosophy:

The MVC paradigm permits a wide variety of information and event probes, or views to be attached to the same model. The model is unaware of the presence of zero or many views. This is at the heart of model reuse.

Views are employed to encapsulate information outflows from models. In this case, X-window display gauges, real-time 3D rendering on Silicon Graphics workstations, and data logging for analysis purposes are examples.

Controllers may be interchanged, allowing different user interaction devices. Here we could have many different input devices and different user input modes, such as input file for non-interactive, analysis modes, instead of human-in-the-loop inputs real-time modes.

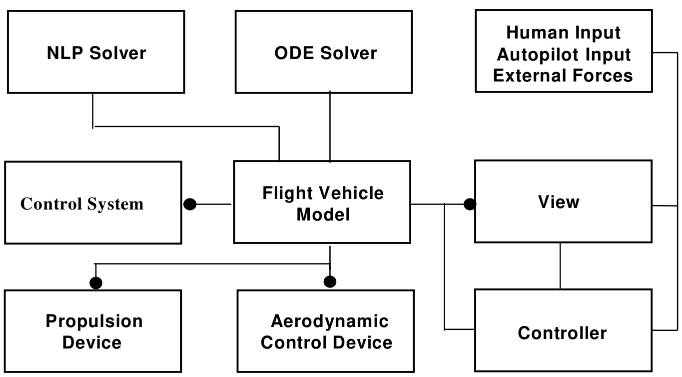

Figure 2. OMT Conceptual Model of FVM Framework..

This approach provides a propagation of change (dependent updates), or cause-and-effect object-oriented design pattern. This cause-and-effect analysis and design approach is inherent in how we conceptually represent interactions in the world, and can now be implemented easily and efficiently.

FLIGHT DYNAMICS MODELING

A general flight vehicle simulation framework must also allow for:

1. wide ranges of speed, flight vehicle configuration, and maneuvers
2. a wide range of environmental conditions.

A widely accepted coordinate system for vehicles flying through an air mass uses the following notation, shown in Figure 3. The origin, **O**, is defined as the center of gravity, with the triple (X,Y,Z) = $\mathbf{p_B}$ denoting a local orthogonal coordinate system, an example of a `body-fixed' coordinate system. In the following discussion, vectors will be lower case bold letters with identifying subscripts to illustrate meaning.

For example, the symbol $\mathbf{p_B}$ means position in the body-centered coordinate system. Bold capital letters indicate matrices, such as **B**, which will be defined as the direction cosine matrix below. The remaining triples express the following quantities in terms of this coordinate system [Roskam 1979]:

(U,V,W) linear velocity along body axes, $\mathbf{v_B}$;
(P,Q,R) angular velocity along body axes, $\mathbf{a_B}$;
(F_x,F_y,F_z) $\mathbf{f_B}$, the forces acting on flight vehicle resolved along the body axes;
(L,M,N) **m**, the moments acting along the body axes.

In our formulation of the six-degree-of-freedom equations, however, we will use the position measured from an inertial reference frame, e.g. one measured from a point outside the flight vehicle, $\mathbf{p_I}$. Other quantities of interest are mass, the angle of attack, angle of side-slip, and some common control inputs:

m mass
α angle of attack[$\tan^{-1}(W/U)$]
β angle of side-slip[$\tan^{-1}(V/(U^2+W^2)^{1/2})$]
δe elevator deflection, positive down
δa aileron deflection, positive left
δr rudder deflection, positive nose left

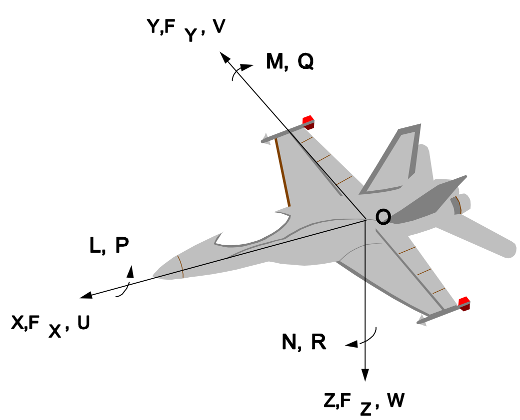

Figure 3. Axes and notation.

The mass characteristics are defined by the inertial tensor **J**. This may vary with time, in particular when the flight vehicle configuration changes during the simulated flight. When referred to a body frame of reference oriented from the center of mass (the body axis denoted by a subscript B), and an inertial axis, denoted by a subscript I [Rolfe 1991], the general equations of motion of a rigid flight vehicle flying in air are:

$\mathbf{q} = (q_0, q_1, q_2, q_3)$ quaternions, which measure orientation;
$\mathbf{x} = (\mathbf{p_I}, \mathbf{v_B}, \mathbf{a_B}, \mathbf{q})$ the 6 DOF state variables;
$\mathbf{u} = (\delta e, \delta a, \delta r, t, \mathbf{e_{SNE}})$ the control and synthetic natural environment inputs;

and the general form of the 6 DOF equations-of-motion are:

$$d\mathbf{x}/dt = \mathbf{A(x)\ x + b(x,u)}$$

$$\mathbf{B} = [b_{ij}] \quad \mathbf{C} = [c_{ij}] \quad \mathbf{D} = [d_{ij}] \quad \mathbf{J} = [j_{ij}]$$

$$b_{11} = q_0^2 + q_1^2 - q_2^2 - q_3^2$$
$$b_{12} = 2*(q_1*q_2 - q_0*q_3)$$
$$b_{13} = 2*(q_1*q_3 - q_0*q_2)$$
$$b_{21} = 2*(q_1*q_2 - q_0*q_3)$$
$$b_{22} = q_0^2 - q_1^2 + q_2^2 - q_3^2$$
$$b_{23} = 2*(q_2*q_3 - q_0*q_1)$$
$$b_{31} = 2*(q_1*q_3 - q_0*q_2)$$
$$b_{32} = 2*(q_2*q_3 - q_0*q_1)$$
$$b_{33} = q_0^2 - q_1^2 - q_2^2 + q_3^2$$

$$\begin{array}{lll} c_{11} = 0 & c_{12} = -R & c_{13} = Q \\ c_{21} = R & c_{22} = 0 & c_{23} = -P \\ c_{31} = -Q & c_{32} = P & c_{33} = 0 \end{array}$$

$$\begin{array}{llll} d_{11} = 0 & d_{12} = P & d_{13} = Q & d_{14} = R \\ d_{21} = -P & d_{22} = 0 & d_{23} = -R & d_{24} = Q \\ d_{31} = -Q & d_{32} = R & d_{33} = 0 & d_{34} = -P \\ d_{41} = -R & d_{42} = -Q & d_{43} = P & d_{44} = 0 \end{array}$$

$$\begin{array}{lll} j_{11} = I_{xx} & j_{12} = 0 & j_{13} = -I_{xz} \\ j_{21} = 0 & j_{22} = I_{yy} & j_{23} = 0 \\ j_{31} = -I_{xz} & j_{32} = 0 & j_{33} = I_{zz} \end{array}$$

$$d\mathbf{p_I},/dt = \mathbf{B}^T \mathbf{p_I}$$
$$d\mathbf{v_B}/dt = -\mathbf{C}\ \mathbf{v_B} + \mathbf{f_B}/m + \mathbf{B}\ \mathbf{g} \qquad (\mathbf{g} = \text{gravity vector})$$
$$d\mathbf{a_B}/dt = -\tfrac{1}{2}\mathbf{J}^{-1}\mathbf{CJ}\ \mathbf{a_B} + \mathbf{J}^{-1}\ \mathbf{t_B}$$
$$d\mathbf{q}/dt = -\tfrac{1}{2}\mathbf{D}^T\ \mathbf{q}.$$

Thus, integrating the four systems of differential equations above for $\mathbf{x} = (\mathbf{p_I}, \mathbf{v_B}, \mathbf{a_B}, \mathbf{q})$ given a set of initial conditions solves the 6 DOF flight vehicle simulation problem.

The FVM framework is focused around accumulating the force term, $\mathbf{f_B}$, and $\mathbf{t_B}$, the moment term, as a function of the control inputs and the interaction with the synthetic natural environment, $\mathbf{u} = (\delta e, \delta a, \delta r, t, \mathbf{e_{SNE}})$. The diagram in Figure 2 represents the relationships present among the base classes. Aerodynamic control sources include the base aerodynamic representation of the body, elevators, ailerons, rudders, landing gears, canards, trim tabs, various farings, rotors, and armaments. Propellers, rotors, and jet engines are examples of propulsion sources.

In the case of the U.S. Navy's Pioneer UAV, there are three aerodynamic control sources(ADCs), a rudder, elevator, and aileron, one propulsion source, a Sachs SF2 350 engine model with propeller effects. The local pattern UAV pilots have two joysticks to provide δe, δe, δe, and t. Forces from the SNE, such as wind, air pressure and density are also moderated through view and controller classes and are used in the equations of motion.

The joysticks and throttle are encapsulated by **view** and **controller** classes, and provide angle inputs to the ADCs. Along with the angles of attack and side-slip, the ADCs calculate coefficients which are provided to the FVM class for use in solving the equations of motion.

We now demonstrate how the twenty five core C++ classes in the FVM framework, originally designed for, and fielded as, a high fidelity physics-based virtual reality UAV pilot training simulator were reused to create a simulation-based design framework for UAVs.

SIMULATION-BASED DESIGN EXAMPLE

In this section we describe the development of a simulation-based design framework for UAVs, that is to be able to very accurately simulate every aspect of an operational UAV. To accomplish this, we reuse our C++ FVM, which implements flight vehicle dynamics accurately, in a simulation framework implemented with the IMPORT simulation CASE tool.

The FVM framework models motion due to airframe, powerplant, and mass characteristics of flight vehicles very well. It does not model the physics of the avionics systems and details such as the effects of control surface actuators in great detail, since this is not required for operator training. A complete UAV framework for simulation-based design, then, must model all system components shown in Figure 1, the subsystems at the highest levels of detail and fidelity, e.g. avionics and mechanical actuators.

Thus, we have a model of a ground control station, the *GCS* class, an uplink *Radio* class to send input commands to the UAV, a downlink *Radio* class to send UAV telemetry and payload data back to the *GCS*. The source code below executes a run of the SBD version of the Pioneer UAV simulation.

```
EXPORT Main = OBJECT

EXPORT CONSTRUCTOR METHOD ()
VAR
   infile    : InFile;
   input,
   output,
   fname     : STRING;
   gcs       : GCS;
   pioneer   : PioneerUAV;
   gcsRadio,
   uavRadio  : Radio;
   finaltime : REAL;
   clockRate : INTEGER;
BEGIN

 fname := ARGS();
 input := CONCAT(fname,".in");
 output := CONCAT(fname,".out");

 gcsRadio := NEW Radio(56,6);
 uavRadio := NEW Radio(56,6);
 gcs      := NEW
GCS(gcsRadio,26,4,input);
 pioneer  := NEW
PioneerUAV(input,output,uavRadio);

 ASK uavRadio TO ConnectTo(pioneer);
 ASK uavRadio TO TransmitTo(gcsRadio);
 ASK gcsRadio TO ConnectTo(gcs);
 ASK gcsRadio TO TransmitTo(uavRadio);

 TELL gcs TO ProcessInput();
 TELL pioneer TO Fly();

END METHOD;
END OBJECT;
```

A framework, called the Automatic Control System and Electromechanical Device (ACS/EMD) simulation framework was developed to represent the avionics systems and electromechanical devices required. The ACS/EMD framework can be used to model arbitrary feedback control system, automatic control systems(e.g. the avionics system), and electromechanical devices, such as mechanical actuators.

We first illustrate the mechanism by which we link the FVM framework into IMPORT simulations, which demonstrates the general capability in IMPORT to directly link in native C++ libraries. We then describe the ACS/EMD framework, and finally illustrate how the elements are combined.

Interfacing C++ to IMPORT

Linking in a previously written library for use in IMPORT is straightforward, as illustrated below. The `header:` variable in the `INTERFACE` statement points to the location of the header file which defines the library interface, and `library:` points to the executable library.

The class name must be the same as the class name in the C++ header, and the method names must be the same as well. The variable types of the arguments must match as well.

```
MODULE PIONEER
INTERFACE(header:"inc/Pioneer.h",library:
"src/pioneer.a")

EXPORT Pioneer = OBJECT

   EXPORT CONSTRUCTOR METHOD (IN infile :
   STRING; IN outfile : STRING);
   EXPORT ASK METHOD advance(IN numsteps
   : INTEGER; IN RTime, RTimeStep :
   REAL):REAL;
   EXPORT ASK METHOD setInputCmds(IN
   elev, ailer, ruddr, thrtl : REAL);
   EXPORT ASK METHOD getAlpha():REAL;
   EXPORT ASK METHOD getBeta():REAL;
   EXPORT ASK METHOD
   getDynamicPressure():REAL;
   EXPORT ASK METHOD
   getPitchFeedback():REAL;
   EXPORT ASK METHOD
   getRollFeedback():REAL;
   EXPORT ASK METHOD
   getYawRateFeedback():REAL;

END OBJECT;
END MODULE;
```

This will be used only for the solution to the 6 DOF aircraft model, not the avionics, etc. For that reason, we use the *Pioneer* class in the *PioneerUAV* class below.

```
PioneerUAV = OBJECT

    AMSubsystem : Pioneer;
    Autopilot: UAVControlSystem;
    EndTime,deltaT : INTEGER;

    EXPORT CONSTRUCTOR METHOD (IN
instr,outstr : STRING;IN radio : Radio)
    VAR
       infile : InFile;
       endtime : REAL;
       timestep : INTEGER;
    BEGIN
       infile := NEW InFile(instr);
       infile >> endtime;
       EndTime := TRUNC(1000.0*endtime);
       infile >> timestep;
       deltaT := TRUNC(1000.0/timestep);
       DISPOSE infile;
       AMSubsystem := NEW
Pioneer(instr,outstr);
```

```
      Autopilot := NEW
UAVControlSystem(AMSubsystem,instr);
      ASK Autopilot TO
AttachAMSubsystem(AMSubsystem);
      ASK radio TO ConnectTo(Autopilot);
      ASK Autopilot TO ConnectTo(radio);
      TELL Autopilot TO
UpdateControlLoop();
   END METHOD;

   EXPORT TELL METHOD Fly()
   VAR
      time : REAL;
   BEGIN
      WHILE ( SIMTIME() <= EndTime ) DO
         time := ASK AMSubsystem TO
advance(1,SIMTIME()/1000.0,deltaT/1000.0)
;
         WAIT(deltaT);
      END WHILE;
   END METHOD;
END OBJECT; <-- PioneerUAV
```

The ACS/EMD class is used to implement the *UAVControlSystem* class used in the *PioneerUAV* class. The statement: `TELL Autopilot TO UpdateControlLoop();` activates a process which samples the output of a network of over seventy components and subsequent simulation processes from the ACS/EMD class. An implementation shortcut was taken, and the components to represent onboard sensors and the actuators for aerodynamic control surfaces were instantiated in the *UAVControlSystem* class, since these are always present in almost all UAVs.

ACS/EMD Framework

The intent is to allow designer the ability to implement control system block diagrams easily. To do so, a number of classes will be presented to represent system inputs, allow system output sampling, gains, Laplace transform transfer functions, summing junctions, bandpass filters, and a structure from which to implement any function one may wish to implement in a block diagram structure. For example, to implement the mechanical actuators for the Pioneer UAV, a proprietary real valued function block was developed.

The BDElement class below implements a data structure which has as state variables: a real value, an array of BDElements which implements an arbitrary graph of BDElements with a maximum fan-in of MAXIN and a coefficient associated with each input. This will allow implicit chaining of a gain in series to an arbitrary block. The main abstractions are to allow blocks to connect, to get the input from connected blocks, and allow output sampling at any time.

The **addInput** method implements the connection protocol, and **calculateOutput** gets input from the connected blocks, and provides the output at any time. The method getValue implements the output sampling protocol. This method chains backward through the control system diagram to the input data blocks. This forces functional evaluation at the current simulation time of the block being sampled. This ensures causality correctness in case the code was executed in a parallel simulation environment.

```
BDElement = OBJECT

   Value : REAL;
   Inputs : ARRAY[MAXIN] OF BDElement;
   Coeffs : ARRAY[MAXIN] OF REAL;
   NumInputs : INTEGER;
   ID : INTEGER;

   EXPORT ASK METHOD
calculateOutput():REAL
   VAR
      I : INTEGER;
      value : REAL;
   BEGIN
      IF ( NumInputs <= 1 ) THEN
      RETURN Value;
      ELSE
      Value := 0.0;
      FOR I := 0 TO NumInputs-1 DO
         Value := Value + Coeffs[I]*(ASK
Inputs[I] TO calculateOutput());
      END FOR;
      RETURN Value;
      END IF;
   END METHOD;

   EXPORT ASK METHOD addInput(IN bde :
BDElement;
                              IN coeff :
REAL)
   BEGIN
      Inputs[NumInputs] := bde;
      Coeffs[NumInputs] := coeff;
      NumInputs := NumInputs + 1;
   END METHOD;

   ASK METHOD getValue(IN value :
REAL):REAL
   BEGIN
      IF NumInputs = 0 THEN
         RETURN Value;
      ELSE
         RETURN ASK Inputs[0] TO
calculateOutput();
      END IF;
   END METHOD;
END OBJECT; <-- BDElement
```

The *Junction* class implements summing junctions. *Junction* is also used as a base class for the *BlockElt* class, which specializes the *Junction* to serve as the base class for all block diagram components.

```
Junction = OBJECT (BDElement)

   JunctionID : ARRAY[MAXIN] OF INTEGER;
   JunctionCoeff : ARRAY[MAXIN] OF REAL;
   NumJunc     : INTEGER;
   BlockID     : ARRAY[MAXIN] OF INTEGER;
   BlockCoeff     : ARRAY[MAXIN] OF REAL;
   NumBlocks  : INTEGER;

   EXPORT ASK METHOD addJunction(IN id :
INTEGER;IN coeff : REAL)
   BEGIN
      JunctionID[NumJunc] := id;
      JunctionCoeff[NumJunc] := coeff;
      NumJunc:= NumJunc + 1;
   END METHOD;

   EXPORT ASK METHOD addBlock(IN id :
INTEGER;IN coeff : REAL)
   BEGIN
      BlockID[NumBlocks] := id;
      BlockCoeff[NumBlocks] := coeff;
      NumBlocks := NumBlocks + 1;
   END METHOD;

END OBJECT;
END MODULE; <-- End Junction
```

BlockElt defines the protocol which makes the diagram simulate, the ability to **TurnOn**.

```
BlockElt = OBJECT ( Junction )

   EndTime, deltaT : INTEGER;

   TELL METHOD TurnOn()
   BEGIN
   END METHOD; <-- tell method
END OBJECT; <-- BlockElt
```

The *DataBlock* class is used to encapsulate a system input, or do a zero order hold for output.

```
DataBlock = OBJECT ( BlockElt )

   EXPORT CONSTRUCTOR METHOD (IN et,dt :
INTEGER;IN id : INTEGER)
   INIT BlockElt(et,dt,id);
   BEGIN
      {Value := 0.0;}
   END METHOD;
   EXPORT ASK METHOD getValue():REAL
   BEGIN
      IF NumInputs = 0 THEN
         RETURN Value;
      ELSE
         RETURN ASK Inputs[0] TO
calculateOutput();
      END IF;
   END METHOD;
END OBJECT; <-- BlockElt
```

The *NoiseBlock* provides Gaussian noise injection into the system.

```
NoiseBlock = OBJECT (BlockElt)

   Mu,Sigma : REAL;
   normal : Random;

   EXPORT TELL METHOD TurnOn()
   BEGIN
      WHILE SIMTIME() <= EndTime DO
      WAIT(deltaT);
      Value := ASK normal TO
Normal(Mu,Sigma);
      END WHILE; <-- SIMTIME() <= EndTime
   END METHOD; <-- tell method
END OBJECT; <-- NoiseBlock
```

The *MultiplierBlock* implements a system gain.

```
MultiplierBlock = OBJECT (BlockElt)

   Multiplier : REAL;

   EXPORT TELL METHOD TurnOn()
   VAR
      input : REAL;
   BEGIN
      WHILE SIMTIME() <= EndTime DO
      WAIT(deltaT);
      input := ASK Inputs[0] TO
calculateOutput();
      Value := Multiplier*input;
      END WHILE; <-- SIMTIME() <= EndTime

   END METHOD; <-- tell method
END OBJECT; <-- MultiplierBlock
```

The *LimitBlock* implements a filter which limits output between a range of values.

```
LimitBlock = OBJECT (BlockElt)

   High, Low : REAL;

   EXPORT TELL METHOD TurnOn()
   VAR
      input : REAL;
   BEGIN
      WHILE SIMTIME() <= EndTime DO
      WAIT(deltaT);
      input := ASK Inputs[0] TO
calculateOutput();
      IF input <= Low THEN
```

```
         Value := Low;
      ELSEIF input >= High THEN
         Value := High;
      ELSE
         Value := input;
      END IF;
      END WHILE; <-- SIMTIME() <= EndTime

   END METHOD; <-- tell method
END OBJECT; <-- LimitBlock
```

The *TransformBlock* implements the transfer funtion

$T(s) = A(a)/B(s);$

$A(s) = a_0 + a_1 s + \ldots + a_n s^n;$
$B(s) = b_0 + b_1 s + \ldots + b_m s^m;$

where n <= m.

The transfer function is represented internally as an m-th order system of linear ordinary differential equations (ODE) using the **b** coefficients above in the right hand side calculation. The **X** variables below are the internal ODE state variables. The **TurnOn** process below collects the inputs, and evaluates the transfer function using that input. The **evaluateTransferFunction** method below calls **integrate**, which uses the method **computeRHS** in a fourth order Runga-Kutta numerical integration scheme. The output is then calculated using the **a** coefficients to calculate the output. If n = m, the **b** coeffiencts are used to calculate a complex residue.

```
TransformBlock = OBJECT (BlockElt)

   B     : ARRAY[9] OF REAL;
   A     : ARRAY[9] OF REAL;
   N,M   : INTEGER;
   Input : REAL;
   X     : ARRAY[9] OF REAL

EXPORT TELL METHOD TurnOn()
   BEGIN
      WHILE SIMTIME() <= EndTime DO
         WAIT(deltaT);
          Input := ASK Inputs[0] TO
          calculateOutput();
         ASK SELF TO
evaluateTransferFunction();
      END WHILE; <-- SIMTIME() <= EndTime

   END METHOD; <-- tell method

ASK METHOD computeRHS(IN  x : ARRAY[9] OF
REAL;IN rhs : ARRAY[9] OF REAL)
   VAR
      I : INTEGER;
      value : REAL;
   BEGIN
      FOR I := 1 TO M-1 DO
         rhs[I] := x[I+1];
      END FOR;
      value := 0.0;
      FOR I := 1 TO M DO
         value := value + B[I-1]*x[I];
      END FOR;
      rhs[M] := (Input - value)/B[M];
   END METHOD;

   ASK METHOD evaluateTransferFunction()
   VAR
      value : REAL;
      I : INTEGER;
   BEGIN

      ASK SELF TO
integrate(deltaT/1000.0);
      IF N = 0 THEN
         Value := A[0]*X[1];
      ELSE
         Value := 0.0;
         FOR I := 0 TO N-1 DO
            Value := Value + A[I]*X[I+1];
         END FOR;
         IF N < M THEN
            Value := Value + A[N]*X[N+1];
         ELSE
            value := 0.0;
            FOR I := 1 TO M DO
               value := value + B[I-
1]*X[I];
            END FOR;
            Value := Value +
                  A[N]*(Input -
value)/B[M];
         END IF;
      END IF;
   END METHOD;
END OBJECT;
```

The following code fragment illustrates the implementation of an elevator actuator. The objects are instantiated and then the computation graph is established using the **addInput** method each of the classes inherit. The objects are then activated by spawning the simulation threads using the TELL invocation.

```
EAj1 := NEW Junction(51);
EAj2 := NEW Junction(52);
EAtsf1 := NEW

  TransformBlock(a,b,0,1,EndTime,deltaT/
2,57);
```

```
EAtsf2 := NEW
   TransformBlock(a,b,0,1,EndTime,deltaT/
2,58);
EAm := NEW
     MultiplierBlock(30.8,EndTime,deltaT/
     2,63);
EAcut:= NEW
        LimitBlock(-
23.0,23.0,EndTime,deltaT/2,66);
EArl := NEW
   RateLimitBlock(uav,EOut,250.0,1,c,2,En
dTime,
   deltaT/2,69);

      ASK EAj1 TO addInput(EActIn,1.0);
      ASK EAj1 TO addInput(EAcut,-1.0);
      ASK EAtsf1 TO addInput(EAj1,1.0);
      ASK EAj2 TO addInput(EAtsf1,1.0);
      ASK EAj2 TO addInput(EAcut,-1.0);
      ASK EAm TO addInput(EAj2,1.0);
      ASK EArl TO addInput(EAm,1.0);
      ASK EAtsf2 TO addInput(EArl,1.0);
      ASK EAcut TO addInput(EAtsf2,1.0);
      ASK EOut TO addInput(EAcut,1.0);
TELL EAtsf1 TO TurnOn();
      TELL EAm TO TurnOn();
      TELL EArl TO TurnOn();
      TELL EAtsf2 TO TurnOn();
      TELL EAcut TO TurnOn();
```

PERFORMANCE RESULTS

We presented a flight vehicle framework which, in accordance with the DOD initiative to employ virtual environments, allows the design, testing of flight vehicles in a virtual environment.

This simulation was executed using the IMPORT object-oriented simulation CASE tool on a 166 MHz Pentium computer. The 14th order nonlinear ODE which computes the 6 DOF equation-of-motion using fourth order Runga-Kutta and was executed at 100 Hz. Each of the more than seventy active block diagram elements was executed at 200 Hz. Twelve of these were Laplace transform transfer functions, and so were embedded ODEs. This executed six times faster than real time on the Pentium. We consider this to be quite acceptable, considering the ease with which this simulation was created.

ACKNOWLEDGEMENTS

We would like to thank the Naval Air Systems Command PMA 263 for their support. We would like to thank Jeffrey Wallace, SPAWARSYS-SD, Code D44202 for his OOA, OOD, and OOP assistance. We would also like to thank Dr. Albert Herman, Dr. Eric Hirschorn, and Donald Mowbray for their programming support.

REFERENCES

Leonard G., L. Peterson, and J. Caldwell. 1997. "Using the Model-View-Controller Framework as a Simulation Development Methodology," In Object-Oriented Simulation 1997, Wallace J. *et al* Eds., Simulation Councils, Inc., January 1997.

Rolfe J. M. and J. K. Staples. Flight Simulation, Cambridge University Press, Cambridge, UK, 1991.

Roskam J. Airplane Flight Dynamics and Automatic Flight Controls, Roskam Aviation and Engineering Corporation, 1979.

Rumbaugh J., et al. Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, NJ, 1991.

Whitehurst R. A. and V. P. Morrison. 1995. "IMPORT: A Language for Simulation of Complex Systems," In Object-Oriented Simulation 1995, Roberts C. *et al* Eds., Simulation Councils, Inc., January 1995.