AN OBJECT-ORIENTED FRAMEWORK FOR BLADE ELEMENT ROTOR MODELLING AND SCALABLE FLIGHT MECHANICS SIMULATION

Mike Theophanides* and Daniel Spira, CAE Inc., Montreal, Canada

Abstract

A novel object-oriented rotor and aerodynamic modelling framework was recently developed by CAE and deployed on a Handling Qualities Simulator. This framework fulfils two goals: it provides an environment for the development of flight mechanics simulations that are easily reconfigurable for rapid model prototyping; and it provides a unified structure for flight mechanics models of scalable fidelity from engineering to training simulation, ranging from faster-than-real-time, interactive generic desktop simulation to pilot-in-the-loop simulators for handling qualities studies, Level D and mission rehearsal applications. This paper describes the new framework, example applications and its implementation on the Handling Qualities Simulator.

NOTATION

ACL	Aerodynamic Component Library
AFCS	Advanced Flight Control System
API	Application Programming Interface
DLL	Dynamic-Link Library
DOF	Degree of Freedom
HCL	Helicopter Component Library
HQS	Handling Qualities Simulator
IO	Input - Output
OO-BERM	Object-Oriented Blade Element Rotor Model
XML	Extensible Markup Language
g	Gravitational acceleration
L_p	Roll damping derivative
$L_{\delta_{lat}}$	Roll control derivative
p	Roll rate
v	Lateral velocity
δ_{lat}	Lateral cyclic control position
ϕ	Roll attitude

1. INTRODUCTION

Object-oriented design facilitates the re-use of common software elements and the extension of existing elements for specialized applications. Simulation frameworks such as NASA's LasRs[1] and CAE's STRIVE[2] were among industry's first to apply object-oriented principles to large scale vehicle simulation. These frameworks provide base classes and services in support of modelling, simulation and interfacing in multi-vehicle simulations. The flight mechanics group at CAE developed an object oriented framework in C++, known as the Object-Oriented Blade Element Rotor Model (OO-BERM), for aerodynamic modelling and blade element simulation.

The object-oriented framework differs from procedural and object-based simulation tools by the use of abstract interfaces (APIs) to encapsulate individual library components, rather than enforcing data exchange through hard-coded static structures. This approach improves interoperability with other simulation frameworks and eliminates intrusion from other systems by ensuring that each object's internal data and behaviour are decoupled from other parts of the system. Legacy procedural software can be encapsulated behind object-oriented interfaces to leverage existing mature libraries. Polymorphic methods are implemented in base classes to provide a foundation of behaviour that derived classes inherit or override to provide specialized behaviour. When a specialized class inherits from one or more abstract interfaces the interface delineates its responsibilities within the larger integrated simulation.

The object-oriented paradigm lends itself naturally to physics-based modelling methods[3,4] through the use of component objects representing individual physical models. The component-based modelling approach adds clarity to the program structure by having software objects simulate the behaviour of real-world components or physical phenomena. The composition of these objects constitutes the complete simulation model. Components are extensible without copying or modifying base source code. This enables new features to be added by extending base classes to address new operating

* Presented at the 35th European Rotorcraft Forum, September 22-25, 2009 in Hamburg, Germany.
Corresponding author: Mike Theophanides, CAE, Dept. M63B - Military Aircraft Software, 8585 Cote-de-Liesse, Saint-Laurent, QC, Canada, H4T 1G6. email: mike.theophanides@cae.com

environments and requirements while maintaining backward compatibility. The result is simplified software management and deployment.

The abstraction of low-level programming through the use of pre-defined library components facilitates the rapid-prototyping of aircraft models of scalable fidelity. The user can create simulation models of varying complexity by swapping components in and out using configuration files without the need to write software to integrate them. In contrast to closed comprehensive codes requiring extensive option selection, deselection and configuration, component-based modelling through load-time composition results in more efficient design since only instantiated components need to be configured.

In rotary-wing flight mechanics, the burden of implementing equations of motion for articulated bodies with multiple linkages is removed from the user by providing pre-defined objects encapsulating equations for the solution of each part's motion.

By assembling and deploying aircraft models using pre-compiled components, the user is distanced from the problems associated with automatic code-generation such as the debugging of human un-readable machine-generated code. Software configuration management is simplified since the source code evolution is visible and traceable. Long-term software maintenance is promoted since library models can be extended or specialized without copying, regenerating or recompiling the framework or user-extended classes.

2. SIMULATION FRAMEWORK

CAE's OO-BERM framework is based on Foundation Layers[2] providing base classes, services and pre-defined re-usable objects as shown in Figure 1. The OO-BERM consists of two main packages: structural component modelling to resolve the dynamics of multibody systems without the need to program equations of motion; and aircraft system modelling permitting the user to develop aerodynamic models or reuse pre-developed components supplied as base classes in libraries.

The helicopter dynamics are computed by assembling components available in the Helicopter Component Library (HCL), which is based on the dynamics solution of articulated tree structures. The helicopter rotor model is configured at simulation load time by instantiating the aircraft dynamic objects, such as the helicopter body, rotor hinges and blade structural segments from the library. Aerodynamic models such as rotor inflow, interference models, blade and aircraft body air loads, ground reactions and other external forces and moments, as well as atomic components that evaluate single coefficients or simple functions, are developed with a user-extensible Aerodynamic Component Library (ACL).

The second layer consists of pre-developed, re-usable models, components, adapters, data types and IO services that are packaged in libraries. Models and components have configurable attributes that can be modified for a specific application. Adapters provide services to access CAE's shared memory structures and perform data type validation. The adapters are extensible to interact with other data structures or protocols. Mathematical data types such as Matrix and Vector have been developed to easily perform mathematical operations and have overloaded operators to simplify the IO these types from XML configuration files. The third

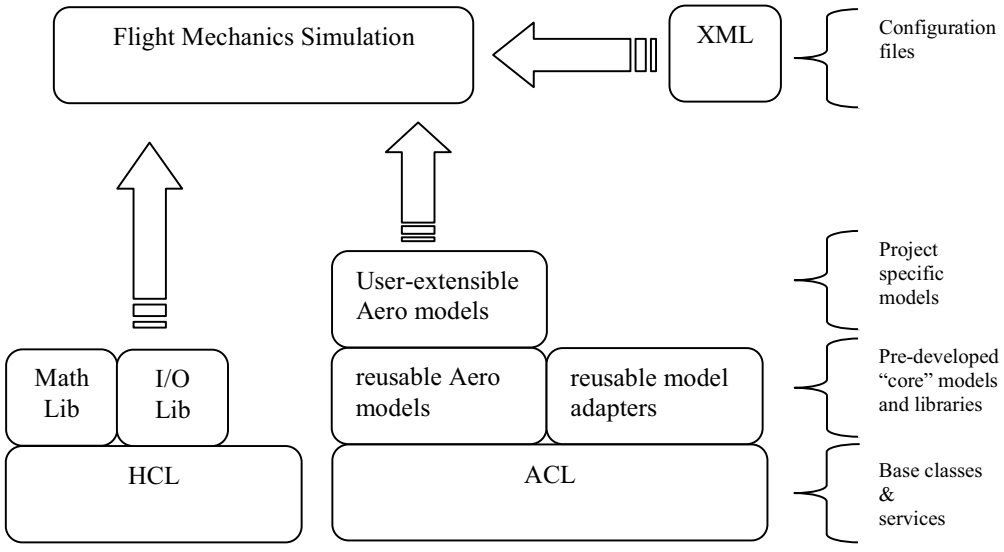

Figure 1 - Framework Foundation Layers

layer represents specialized models to address specific requirements of a given project that may not necessarily be re-usable on other projects. Configuration files in XML format are used to define which parts of the libraries constitute the complete flight mechanics simulation. All layers are packaged as dynamic-link libraries (DLL), which eliminates the need for source-code deployment and compilation on simulator host computers.

2.1. Helicopter Component Library (HCL)

The generic equations of motion solver for n-DOF dynamics of an articulated tree automatically computes the solution based on the tree structure of the connected parts. The dynamic solution of flexible, articulated tree is a global recursive code based on Katz et al[5]. Figure 2 demonstrates the modular approach to modelling with the analogy of a helicopter modelled as an inverted articulated tree.

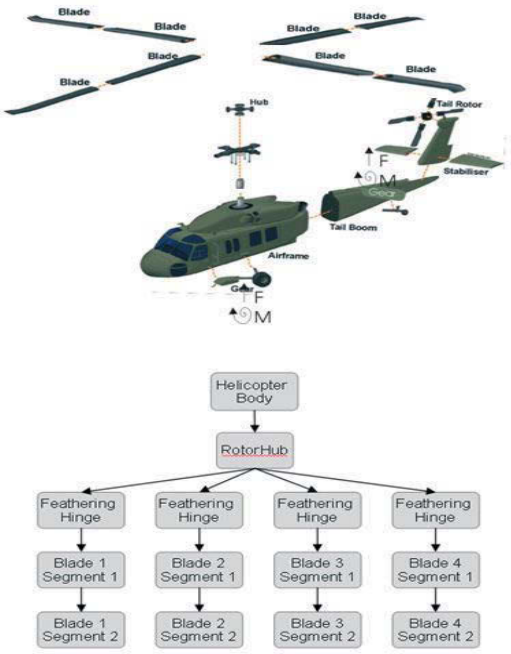

Figure 2 - Modular Dynamic structure of a Rotorcraft

The structural parts are pre-defined dynamic elements available in the HCL that can be automatically assembled into a complete aircraft dynamics model. The solver recursively iterates through the composition of the rotor tree and solves the dynamics of each part. It accumulates the forces and moments of all the branch elements and applies it to the entire branch of the tree, treating it as a rigid body with the instantaneous configuration of the branch. It then recursively applies the correction due to motion within the branch. This recursive solver integrates for the velocity and position of each part below the root. The 6-DOF root integration is the final step of the solution. The advantage of the global recursive solution is that small-mass and zero-mass parts such as pure hinges do not introduce singularities as the model becomes more detailed. Aero-elastic blade bending can be simulated by dividing each rotor blade into a series of rigid segments inter-connected by spring-damper hinges representing blade elastic properties. Therefore, through simple composition of the parts and the hinge orientation of each part, it is just as easy for the user to simulate the dynamics of teetering rotors, fully articulated rotors, rigid rotors, including or excluding aero-elastic blade bending.

All parts inherit from a parent called *Part*, as shown in Figure 3, which provides the basic services common to all parts of the rotor tree dynamics. The HCL also provides four specialized classes that derive from *Part*: *HeloBody*, *RotorHub*, *FeatheringHinge* and *BladeSegment*.

The base class *Part* contains the routines for computing the global moment of a branch that is used to compute the angular acceleration. Any *Part* or child of *Part* is permitted to rotate about an arbitrary axis with respect to its parent in the tree. For example, if one wanted to model a “Teetering Hinge”, one way to do it would be to introduce a new class representing a massless hinge deriving from *Part* . The *HeloBody* class derives from base Part and is usually the root part of the Tree. A helicopter body part includes external forces acting on the fuselage (e.g. undercarriage, tail rotor, slung loads, weapons). The *FeatheringHinge* part is capable of handling additional rotations due to feathering after the basic part rotation, and automatically feathers all structural segments outboard along its branch. This automatic feature can be de-activated resulting in a simple hinge rotation only. The *BladeSegment* class derives from base *Part* and manages the dynamics of a blade segment hinge that is also capable of carrying external forces. Its key role is to

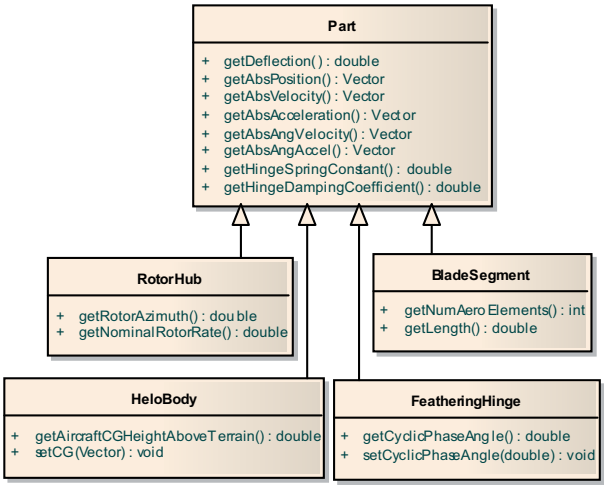

Figure 3 - Helicopter Component Library classes

accumulate the forces and moments for each aerodynamic blade element on the blade segment. The *RotorHub* part represents the hub of the rotor and has additional interfaces to allow the application of engine torque, and also provides specialized rotor hub-related services such as azimuth and revolution counter.

Each derived part can be hinged to its parent or rigidly attached. Each hinge can accept nonlinear spring and damping models. Each part is represented as a dynamic object containing local and global quantities. The local quantities are associated with the part by itself as a free body, such as the mass, moment of inertia, angular momentum, or the external force applied directly to the part. The global quantities are the corresponding values for the entire branch including that part and all its descendants.

While the HCL supports coupled rotor-body models, an aircraft model is not required to contain a rotor. The user is free to define an aircraft model consisting of only an isolated helicopter body as the root part. The first example in Section 3.1 illustrates the value of this feature for scalable fidelity.

2.2. Aerodynamic Component Library (ACL)

The Aerodynamic Component Library (ACL) is composed of reconfigurable and extensible classes that are used to create forces and moments and their constituent models, called Components. The ACL base classes are shown in Figure 4.

The aerodynamic model build-up is done in a similar modular approach as the structural composition described above. A *PartForce* is used to create any physical model whose output results in a Force and Moment that will contribute as a forcing function of the HCL part's dynamics. Classes such as Blade Aerodynamic Element, Tail Rotor or Undercarriage derive from *PartForce* to define the external forces and moments that can be applied to each dynamic Part. A *PartForce* is created in a user's library extension by writing a C++ class that derives from *PartForce*, as in the C++ code:

```
class ABC : public PartForce
```

A *PartForce* is attached to or "owned by" an HCL part when it is declared in the XML configuration files in the hierarchy under the HCL node (i.e. between <part> and </part>) as illustrated in the following example:

```
<part name  = "helo_body"
      type  = "HELO_BODY"
      parent= "NONE">
  <class>
  ….  Attributes section …
  </class>

  <partforce name="ABC">
    <class method = "entryPointABC"
           dll    = "MyUserLib.dll">
    … Data …
    </class>
  </partforce>
  <partforce name="XYZ">
    <class method = "entryPointXYZ"
           dll    = "MyUserLib.dll">
       … Data …
    </class>
  </partforce>
</part>
```

The number of *PartForce* objects that can be attached to a *Part* is unlimited. The accumulation of all the forces is done automatically by the Force

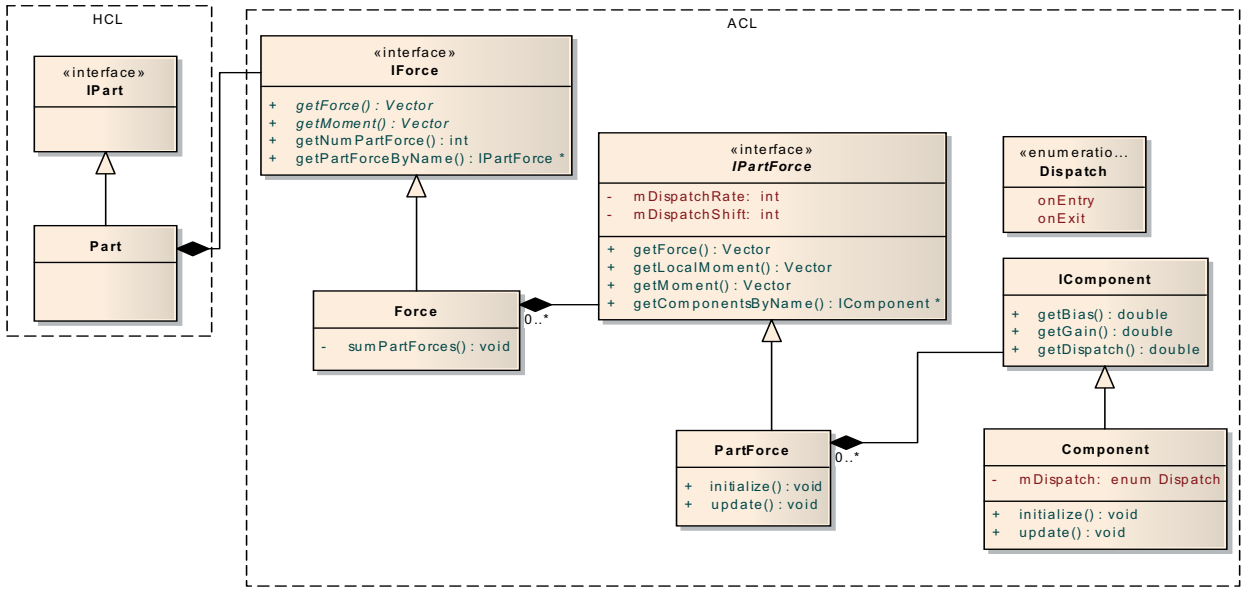

Figure 4 – Aerodynamic Component Library classes

class. This object maintains a list (collection) of all the *PartForce* instances and iterates through the collection at each time step to get the individual forces. These forces are accumulated and applied to the (HCL) part's dynamics automatically, without the need to recode force summation equations or reserve spare "hooks" or entry points as procedural programming requires. Each HCL part contains a pointer to *IForce* and is used to get the sum of all external forces and moments acting on that *Part* at their respective local points of application and at the Part's centre of gravity.

Classes deriving from *Component*, such as CL, CD, CM, Rotor Downwash, Blade Stall, define models that compute equations and/or use data lookup tables. A *Component* class is used to implement any model or computation that does not return a force. In abstract terms, a *Component* represents the smallest possible re-usable, re-configurable or extensible physical model or computation. The user can then choose which components to include or exclude from the simulation depending on which features are required.

For example, referring to Figure 5, a basic or generic blade aerodynamic model would only include the first component *BasicBladeAerodynamics* in the XML input files, while a more sophisticated model could include all three. Typically, a force generator will contain references to many components in order to create models with a useful level of complexity for flight mechanics simulation while retaining a desirable level of abstraction, modularity and extensibility. The above example illustrates in particular how implementing individual rotor blade aerodynamic coefficients in separate components decouples the blade model from each coefficient's implementation details. This also permits load-time swapping of equation-based components with ones obtaining outputs from data lookup tables. In CAE's deployment of the OO-BERM framework on a Handling Qualities Simulator (HQS), *Components* fulfilled a variety of responsibilities, including aerodynamic coefficients, finite state machines (e.g. vortex ring state simulation), malfunction processors

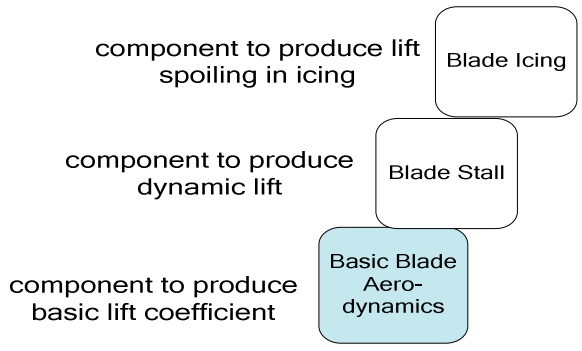

Figure 5 - Component abstraction

(e.g. tail rotor failure), inflow model constituents and interactional aerodynamic models.

The helicopter articulated tree topology, all aerodynamic components, as well as objects fulfilling abstract interfaces for the flight control system, engine/power train and atmospheric models are specified in XML configuration files and resolved dynamically at simulation load time. Furthermore, when deriving a *Part*, *PartForce* or *Component* in the user's library extension, the user is free to define its attributes settable at load time and those attributes' corresponding XML name tags. Thus, without the need for code generation or recompilation, the OO-BERM can simulate arbitrary rotor and hinge arrangements, rigid or elastic bending blades and interchangeable aerodynamic models, as well as integrate with simulator of varying sophistication.

3. EXAMPLE APPLICATIONS

3.1. Reduced Order Aircraft Model for Handling Qualities Study

This example demonstrates how simple reduced-order vehicle models, valuable for handling qualities studies, can be created using the OO-BERM. The simulator motion study of Schroeder and Chung[7] investigated the effects of roll and lateral motion cues on pilot workload during a 2-DOF sidestep task. In this study the vehicle equations of motion were represented by the following state-space model:

$$\begin{aligned} \dot{p} &= L_p\, p + L_{\delta_{lat}}\, \delta_{lat} \\ \dot{\phi} &= p \\ \dot{v} &= g \sin\phi \end{aligned} \qquad (1)$$

No other degrees of freedom were modelled. Implementing this vehicle model in the HCL is simple since it does not require the user to instantiate a rotor. The design and implementation of this example are illustrated in Figure 6. The specialized *PartForce*, class *TwoDofRollLat*, implements a simple first-order transfer function for the lumped aircraft roll rate response. It uses the lateral cyclic input and body roll rate to compute the rigid-body rolling moment. This is equivalent to the roll acceleration since the inertia tensor is set to identity in the XML configuration file. The HCL and ACL provide interfaces for real-time data exchange that are required by a client, including aircraft states and control angles (roll rate and lateral cyclic, respectively in this case). The stability and control derivatives' values are set at load-time from the XML configuration file.

An alternative design illustrated in Figure 7 permits run-time modification of the derivatives' values through the specialized class *SettableComponent*.

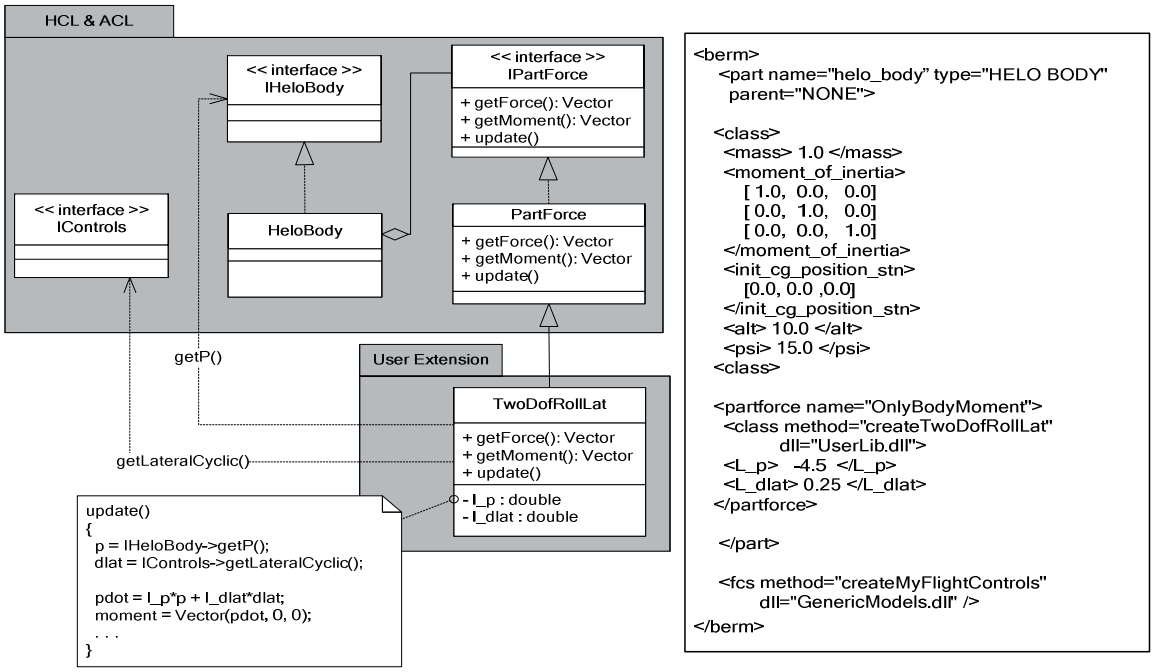

Figure 6. One possible implementation of 2-DOF aircraft model to reproduce a roll-lateral handling qualities study [7]. Fixed values of L_p and L_{dlat} are specified as XML configuration data.

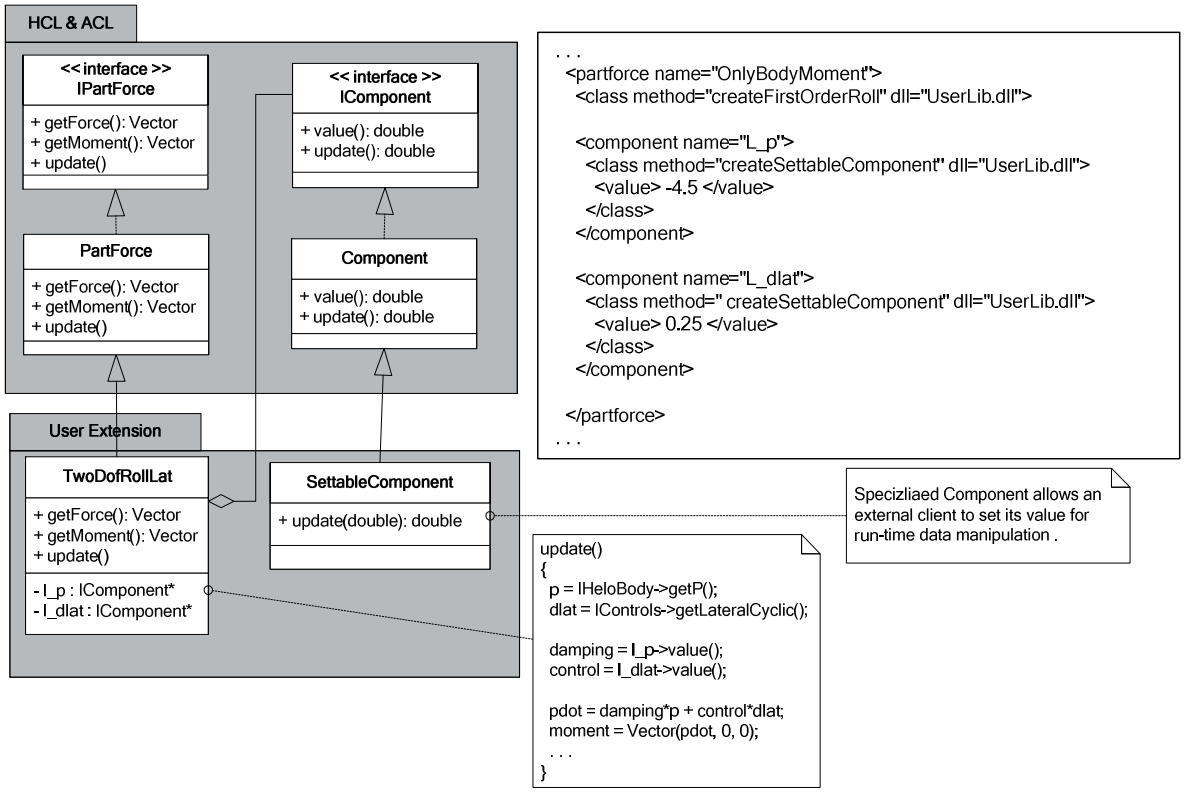

Figure 7 - Second possible implementation of 2-DOF aircraft model to reproduce a roll-lateral handling qualities study[7]. Lp and Ldlat are implemented as components to allow run-time modification.

Instead of owning static values of L_p and $L_{\delta_{lat}}$, TwoDofRollLat refers to the abstract interface *IComponent* to obtain those values during run time from any class fulfilling that interface. Here, this is fulfilled by the specialized class *SettableComponent*, which extends the base Component to allow an external client, such as a simulator operator's console, to override its value while the simulation is running. This capability is invaluable for handling qualities parametric studies such as the one performed by Mitchell et al[8]. It should be noted that *SettableComponent* is completely independent of the mechanism or console that would need to assign it a new value. It is reusable in the compiled library across a range of applications in any simulation environment.

The two designs presented for this application demonstrate scalable fidelity. A traditional blade-element code could either not support the simple 2-DOF simulation or would require significant effort to null the rotor dynamics and aerodynamics as required for this case. Similarly, object-based systems reliant on code generation would need to generate two separate executables for the blade-element and 2-DOF applications. In contrast, the OO-BERM permits the same compiled library to be used for a complete nonlinear blade element simulation as for a rudimentary 2-DOF rigid-body state-space model. Only load-time configuration specified in XML files differentiates the two applications.

3.2. Modeling Abnormal Conditions for Level-D Simulation

The HCL treats each main rotor blade as a separate body with independent properties. As structural elements, each blade segment must fulfil the same abstract interface as all other structural parts. One service in this interface (`IPart->setMass()`), allows a client to modify the part's mass during run time. This service allows an atmospheric simulation module to accrete and shed of ice mass on the fuselage, rotor hub and rotor blades using the same interface. This also permits the implementation of blade damage effects by extending base classes with only a few lines of code.

In the simplest case, blade damage may be simulated as the loss of one blade's tip cap, causing in a reduction in mass and inboard CG shift on the affected blade and resulting 1/rev vibration. This physical model would have been implemented in traditional procedural code with a conditional statement built into the core blade dynamics routine, such as in the following pseudo-code:

```
c = blade_cg_data
L = blade_length_data
M = blade_mass_data
Mtip = blade_tip_mass_data

for (i_blade = 1 to NumBlades)
  if (tip_loss_active AND i_blade=1)
    local_mass = M - Mtip
    local_cg = (M*c - Mtip*L)/local_mass
  else
    local_mass = M
    local_cg = c
  endif
  . . .
  {compute blade 'i' dynamics using
   local_mass and local_blade_cg}
end
```

The disadvantage of this implementation is that it binds the core dynamics module to a simulator feature whose requirements change between projects. This results in non-portable software. An equally unpalatable alternative would have one blade use a specific dynamics routine dependent on the malfunction flag while the others use a pure dynamics routine, which would complicate configuration management significantly.

In contrast, in an object-oriented implementation illustrated in Figure 8, the core blade dynamics are not polluted with a project-specific malfunction implementation. Instead, the fundamental malfunction feature is implemented by the *TipLossEffect* class as a User Extension, but remains independent of specific clients; this makes the class reusable in any simulation environment as well as being extensible (such as to allow variable malfunction severity). Only the user-enhanced adapter layer is associated with a specific instructor interface, thereby shielding the malfunction policy and blade dynamics from any specific simulator architecture.

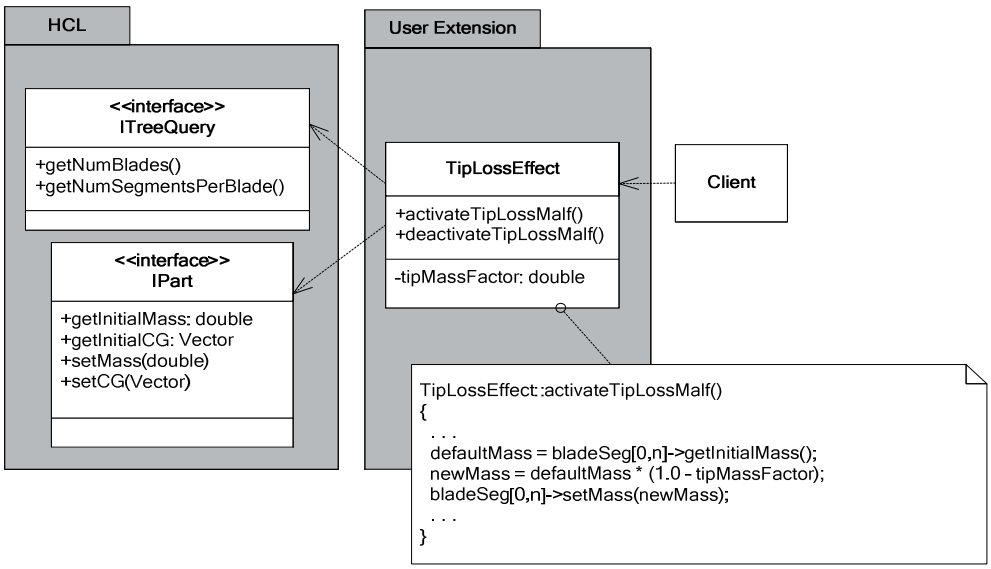

Figure 8. Object-oriented implementation of main rotor blade tip loss malfunction using HCL framework.

A typical full-flight simulator implementation is illustrated in Figure 9. The class *SharedMemConnector* provides a method for connecting to a specific address in shared memory by global variable name, for reading and writing to that address. *SharedMemConnector* could also enforce data type verification and provide other low-level services. *MalfSharedMemAdpater* is a specialized class that uses *SharedMemConnector* to read the global variable MALF_TIPLOSS, which could be set from a simulator instructor console. The class *MalfSharedMemAdpater* is solely responsible for activating and deactivating the malfunction at the correct instant, while the responsibility for actually executing the malfunction's effect remains with *TipLossEffect*.

An alternate implementation for a desktop simulation environment is illustrated in Figure 10. In this case, the simulator has a user-customizable console driven by keyboard input, which could be useful for software qualification. A specialized *ConsoleReader* interprets keyboard events as requests to activate simulator functions, such as toggling the blade damage malfunction state by pressing the "M" key.

In these examples, the *TipLossEffect* class remains independent of the mechanism used to activate the rotor blade malfunction, and the core dynamics classes are not bound to the malfunction. This demonstrates how the framework supports scalable fidelity and arbitrary simulator architectures. Once the user's extension DLLs contain the classes required to support both methods, they can be reused without recompilation on either simulation platform; only the XML configuration files would differ in order to instantiate either adapter dynamically at load time.

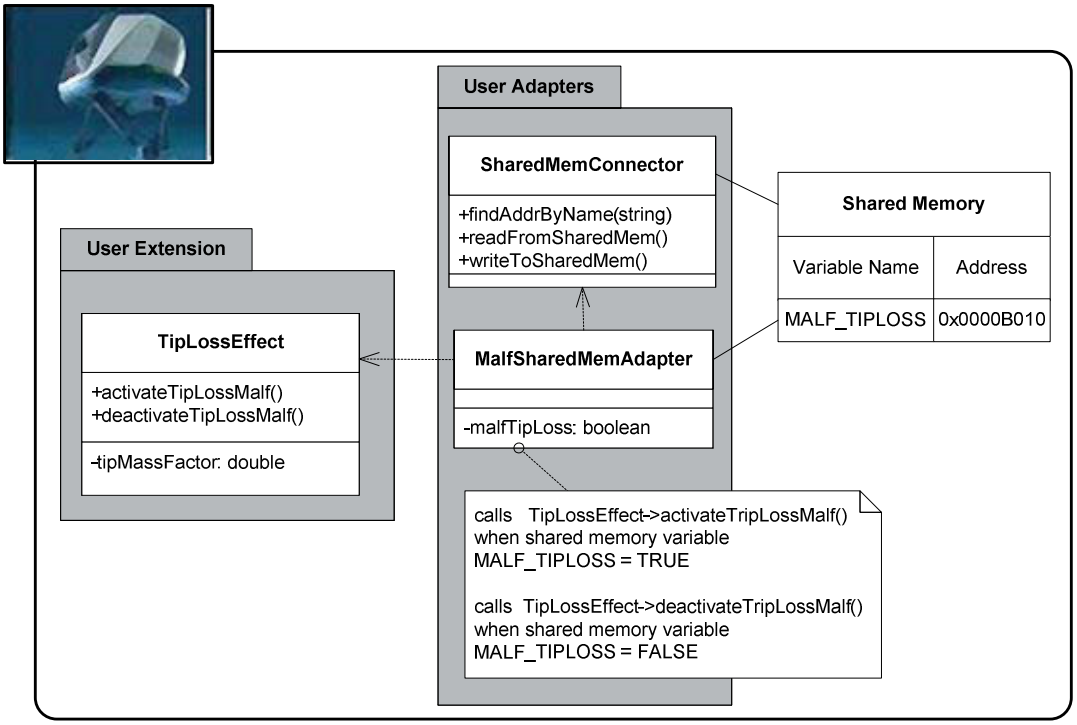

Figure 9. Illustration of malfunction adapter for shared memory data exchange, such as on a full motion simulator.

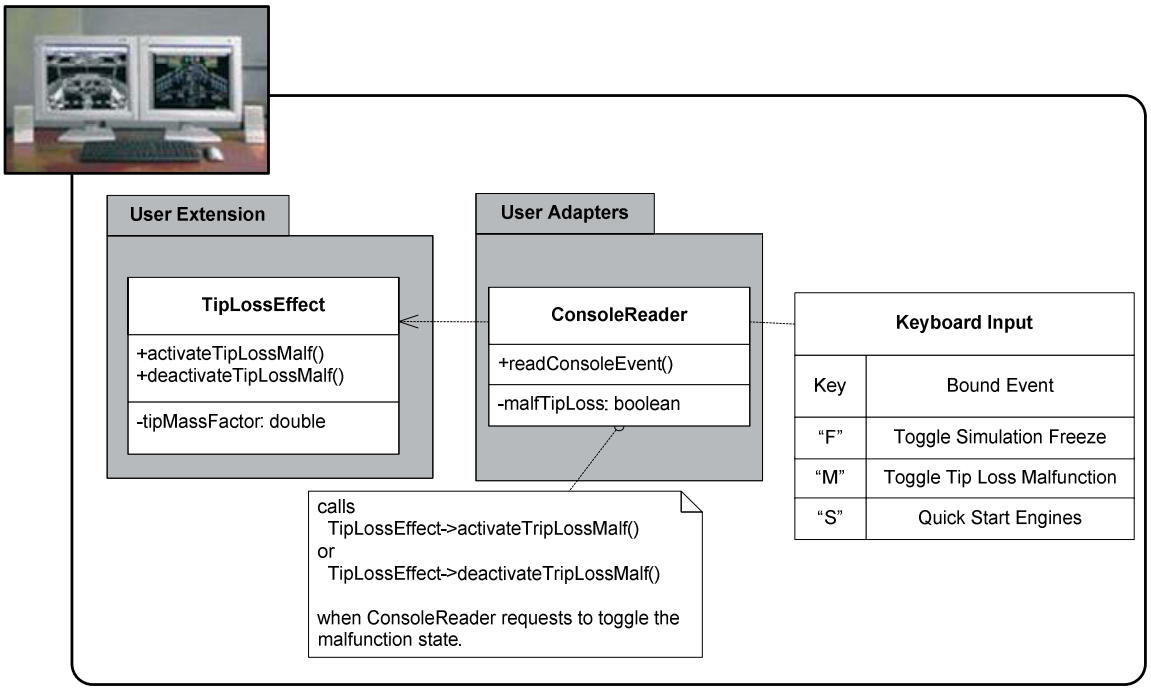

Figure 10. Illustration of malfunction adapter for simple desktop console application.

3.3. Handling Qualities Simulator (HQS)

The OO-BERM was integrated on a reconfigurable rotary wing Handling Qualities Simulator (HQS) as a design and development tool. The open framework and component approach to model building maximizes extensibility and re-usability across different platforms and aircraft types. The reconfigurable models provide the functionality required to attain Level D fidelity as aircraft data becomes available. The HQS is also equipped with a suite of CAE analysis tools for automated test driving, data analysis & visualization, online re-configurability and debugging. Figure 11 is a photograph of the HQS in a generic configuration.

3.3.1. Integration With Mature Legacy Software Libraries

The Handling Qualities Simulator employed a complex integration of dynamic models and simulator functions, as illustrated in Figure 12. Compiled user-extended libraries using the ACL and HCL included clean airframe and empennage aerodynamic models, interactional aerodynamic models, main rotor blade element, and tail rotor aerodynamics, and main rotor inflow including vortex ring state models. These were integrated with mature procedural software libraries for tightly coupled components, including CAE proprietary reconfigurable engine and fuel controller models, AFCS and flight controls models, the CAE Advanced Ground Handling Model[9], under slung / hoist load dynamics, mass properties and aerodynamic models for external stores and role equipment. The interfaces provided by the ACL and HCL and use of specialized adapters resulted in straightforward implementation of the HQS architecture with little or no modification to legacy libraries. The HQS was configured with generic data representative of a notional 8-tonne helicopter. The object-oriented

Figure 11 – Reconfigurable Rotary Wing Handling Qualities Simulator.

modules and ground reactions updated at 420 Hz. The remaining modules were updated at the simulator base execution rate of 60 Hz or slower. Synchronization of data exchange via the shared memory adapter layer was performed by a flight loop coordinator.

Figure 13 demonstrates the synchronization of the object-oriented model and the legacy engine, fuel controller and transmission models during a dual-engine start and rotor run-up to operating conditions on ground. Torque is provided from the transmission to the rotor through the engines' interface *IEngines->getTorque()*. The RotorHub requires this interface to obtain the propulsion system torque output. An adapter fulfils the *IEngines* interface by obtaining the propulsion system's torque output from CAE shared memory. After the dynamics updates all forces and moments, computes rotor acceleration and integrates for angular rate and position, another adapter uses the *IPart* provided interface to publish the resulting rotor speed to shared memory for use by the propulsion system and other procedural libraries. Figure 13 shows how the engines and rotor run up during the start sequence and the fuel controller is able to govern rotor speed at operating conditions with the expected stability.

Figure 14 shows how a shared memory adapter was used to provide external force and moments from an under slung load model running in a legacy software library. As described in Section 2.1, the *HeloBody* class, which represents the helicopter body in the dynamics tree, automatically accumulates external forces and moments from any class fulfilling the *IPartForce* interface that was declared under the *HeloBody* XML hierarchy. On the HQS, this was fulfilled by an adapter reading those forces and moments from shared memory. As with the engine simulation, the underslung load model requires helicopter body states such as acceleration and velocity in order to compute the load dynamics. These states are published from the OO-BERM through a shared memory adapter, which was excluded from the figure for brevity. It should be noted that these adapters are bound to names of variables in shared memory, not addresses. This soft binding is performed at load time. Therefore, the adapter does not to be recompiled if the sequence of shared memory variable declarations changes. This results in a portable compiled adapter library.

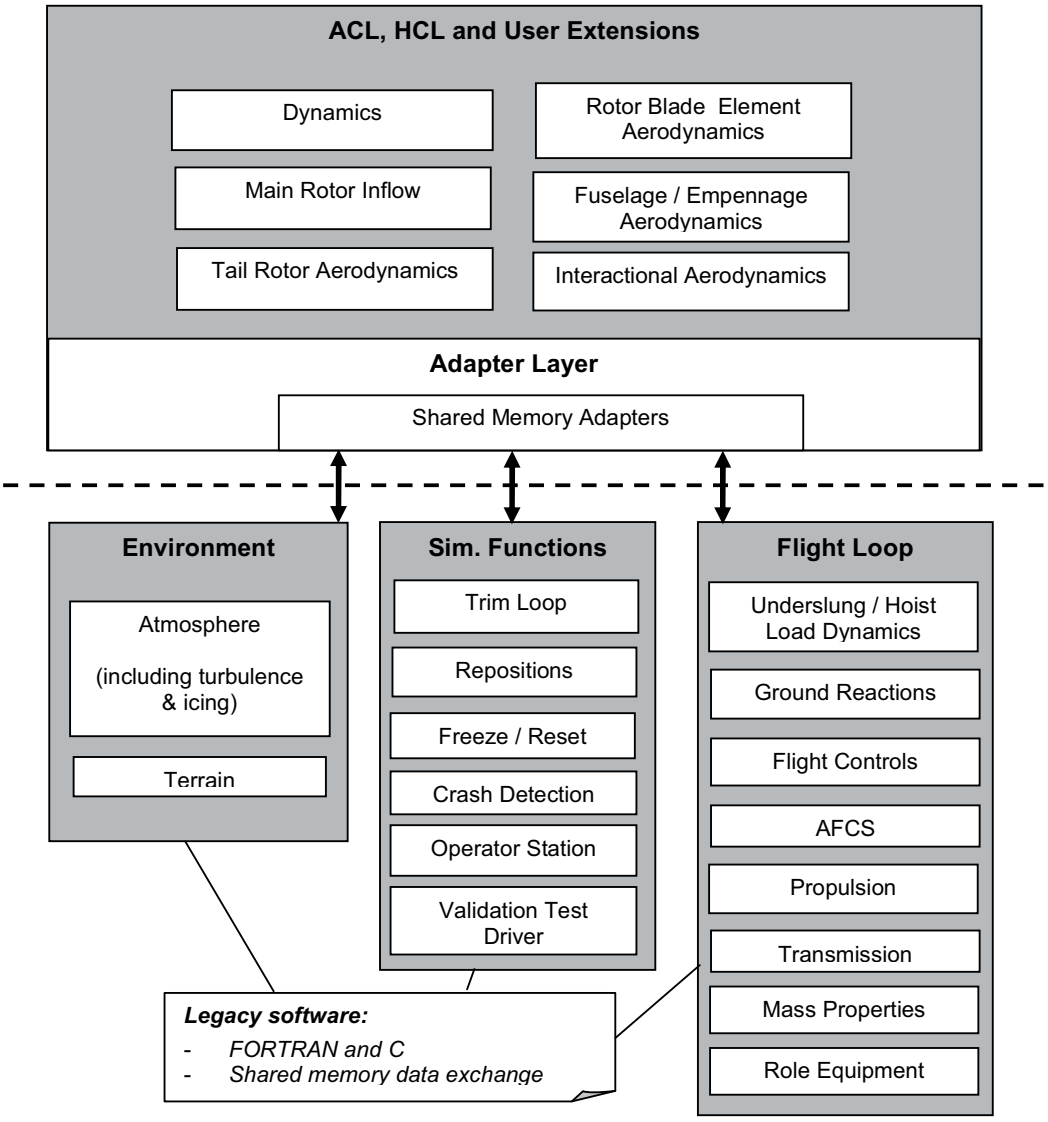

Figure 12. Handling Qualities Simulator software domains.

Figure 15 shows the result of a computer-generated longitudinal cyclic frequency sweep run on the fully integrated HQS. Like the preceding examples, this frequency sweep illustrates the synchronization of tightly interdependent flight loop components across the object-oriented and procedural domains. The validation test driver includes a math pilot that performs closed-loop flight path regulation. One criterion for control frequency sweeps is that the excitation produces responses within a linear region about the trim condition[6]. On HQS, the FORTRAN math pilot modified the raw commanded longitudinal cyclic on-axis inputs to maintain pitch attitude response within 5 degrees of trim, as can be seen by the increased input amplitude near the short period mode around t = 50s. The original commanded lateral cyclic input was zero change from trim. The notional helicopter model exhibited a high degree of pitch to roll inter-axis coupling, which the math pilot correctly suppressed by introducing lateral cyclic inputs to maintain the roll attitude response within 10 degrees of trim at the lowest frequency range before backing out of the loop correctly as the roll response attenuates at higher frequencies. This feedback loop closure between math pilot control inputs generated by a FORTAN module and aircraft response computed by models in the object-oriented framework demonstrates good synchronization between the two domains.

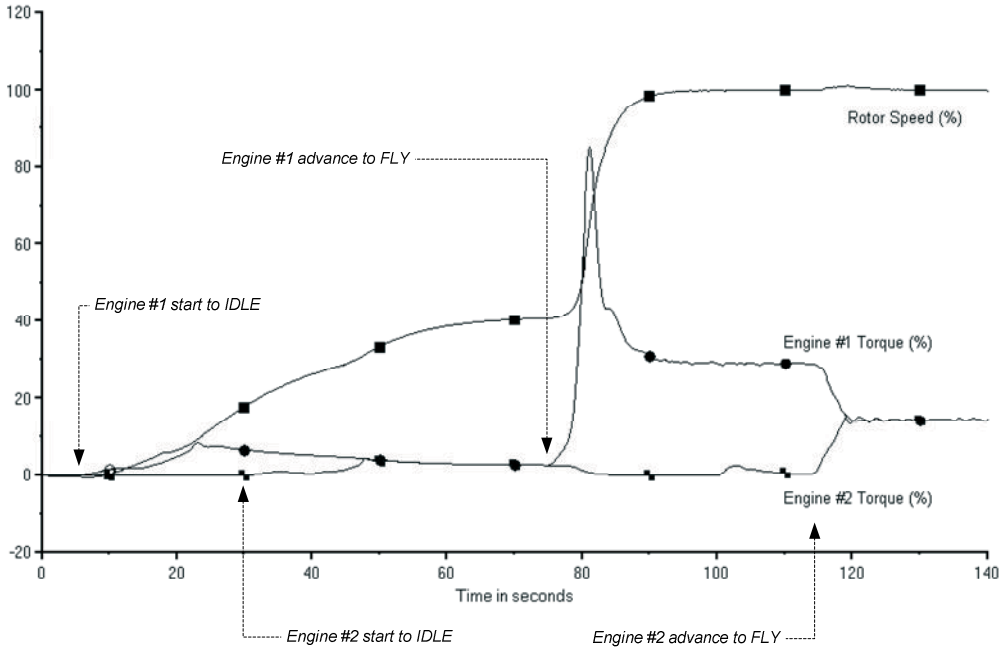

Figure 13. Result of dual-engine start run end-to-end on Handling Qualities Simulator.

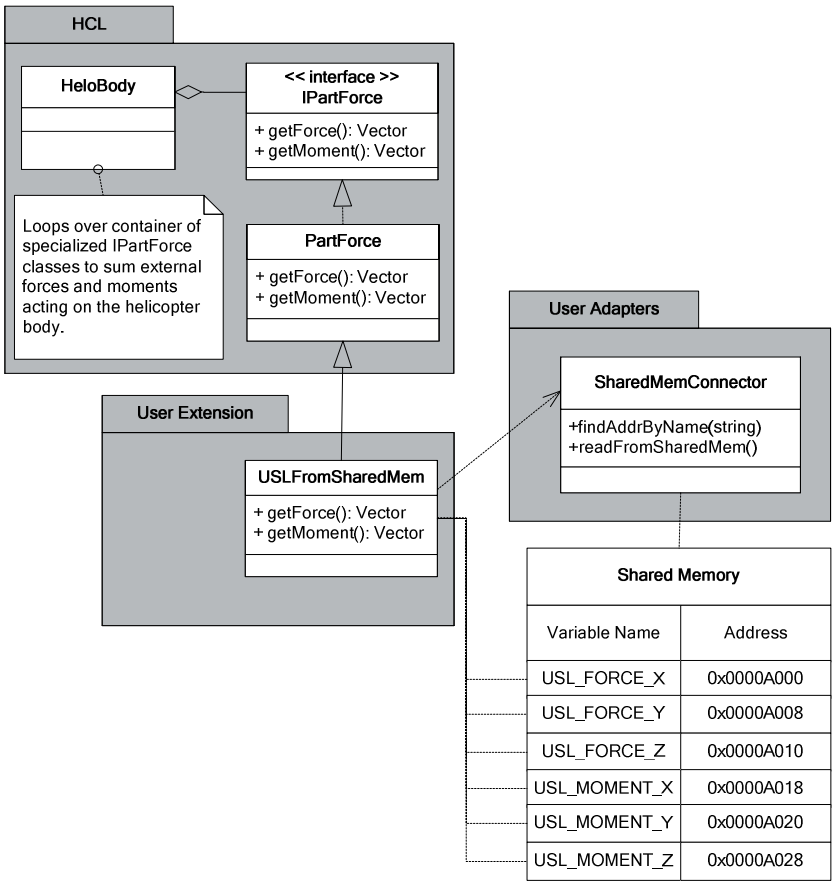

Figure 14. Underslung Loads shared memory adapter provides forces and moments computed by an external underslung / hoist loads model.

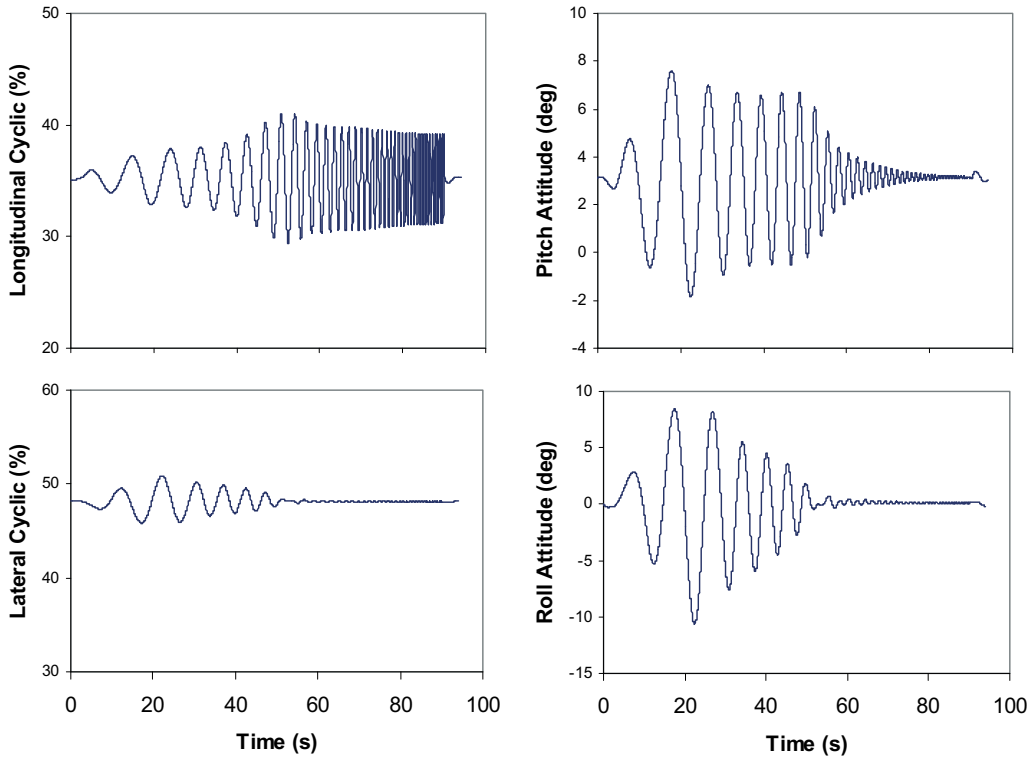

Figure 15. Result of longitudinal cyclic frequency sweep run end-to-end on Handling Qualities Simulator including on- and off-axis closed-loop stabilization.

4. CONCLUSIONS

An object-oriented blade-element and aerodynamic modelling framework, OO-BERM, was developed and deployed on a Handling Qualities Simulator. This framework provides an environment for the development of configurable flight mechanics simulations for research, development and aircrew training. OO-BERM provides the capability to develop and deploy vehicle simulation models of scalable fidelity on a range of simulator platforms without reliance on recompilation.

The user assembles components to create simulation models of varying complexity by swapping components in and out using configuration files at simulation load-time. Component-based modelling adds clarity and results in more efficient design than traditional comprehensive and object-based codes, which require intermediate code generation and/or extensive option selection and configuration.

OO-BERM models are extensible such that new features can be added by extending base classes to address new operating environments or additional requirements while maintaining backwards compatibility. This is accomplished without copying, regenerating or recompiling source code, resulting in simplified software management, deployment and maintenance.

Abstract interfaces decouple core models from specific clients and simulator environments. This facilitates interoperability with third-party frameworks and leveraging of mature procedural software libraries.

Examples were provided illustrating these features for handling qualities studies, Level-D simulation and mission rehearsal. Future work includes aeroelastic model research and handling qualities studies, and further expansion of core libraries.

ACKNOWLEDGEMENTS

The authors would like to recognize the OO-BERM and HQS development and integration team: MingXin Xie, Ara Agnerian, Pascal Martelli-Garon, Warren Dunn and Dave Demers.

REFERENCES

[1] Leslie R.A., Geyer D.W., Cunningham K. et. al. “LaSRS++ An Object-Oriented Framework for Real-Time Simulation of Aircraft”, AIAA-98-4529, 1998.

[2] Gagnon, F., Howells, P. and Simon, E., “On the Use of Frameworks for Real-Time Simulation Applications”, Standards Interoperability Workshop, Orlando, Florida, 1999.

[3] Spira D. and Martelli-Garon P., “Physics-Based Modelling of Asymmetric Control Response for Level-D Helicopter Simulation”, Proceedings of the American Helicopter Society 64th Annual Forum, Montreal, Canada, April 29-May 2, 2008.

[4] Heffner, K., "Reducing Aerodynamic Modelling Flight Test Data Requirements Using a Component Based Modeling Approach”, Royal Aeronautical Society Conference on Data for Simulation, London UK, November 2002.

[5] Katz, A. and Waits, T. “Global Recursive Dynamics of Articulated Tree Structures,” *Computational Mechanics*, Vol. 23, No. 4, pp. 288-298, May 1999.

[6] Tischler, M. B. and Remple, R. K., *Aircraft and Rotorcraft System Identification: Engineering Methods with Flight Test Examples*, American Institute of Aeronautics and Astronautics, Inc., Reston, VA, 2006.

[7] Schroeder, J. A. and Chung, W. W., “Simulation Motion Requirements for Coordinated Maneuvers,” Journal of the American Helicopter Society, Vol. 46, No. 3, pp. 175-181, July 2001.

[8] Mitchell, D.G., Hoh, R. H., He, C. and Strope, K., “Development of an Aeronautical Design Standard for Validation of Military Helicopter Simulators,” Proceedings of the American Helicopter Society 64th Annual Forum, Phoenix, AZ, May 9-11, 2006.

[9] Giannias, N., “An Advanced Ground Handling Model for Training Simulators”, Royal Aeronautical Society Conference – “Can Flight Simulation Do everything?” London, UK, May 19-20, 1999.