Improving the Aircraft Design Process Using Web-based Modeling and Simulation

John A. Reed[†], Gregory J. Follen[‡], and Abdollah A. Afjeh[†]

[†]The University of Toledo

2801 West Bancroft Street

Toledo, Ohio 43606

[‡]NASA John H. Glenn Research Center

21000 Brookpark Road

Cleveland, Ohio 44135

Keywords: Web-based simulation, aircraft design, distributed simulation, Java™, object-oriented

Supported by the High Performance Computing and Communication Project (HPCCP) at the NASA Glenn Research Center.

Abstract

Designing and developing new aircraft systems is time-consuming and expensive. Computational simulation is a promising means for reducing design cycle times, but requires a flexible software environment capable of integrating advanced multidisciplinary and multifidelity analysis methods, dynamically managing data across heterogeneous computing platforms, and distributing computationally complex tasks. Web-based simulation, with its emphasis on collaborative composition of simulation models, distributed heterogeneous execution, and dynamic multimedia documentation, has the potential to meet these requirements. This paper outlines the current aircraft design process, highlighting its problems and complexities, and presents our vision of an aircraft design process using Web-based modeling and simulation.

1 Introduction

Intensive competition in the commercial aviation industry is placing increasing pressure on aircraft manufacturers to reduce the time, cost and risk of product development. To compete effectively in today's global marketplace, innovative approaches to reducing aircraft design-cycle times are needed. Computational simulation, such as computational fluid dynamics (CFD) and finite element analysis (FEA), has the potential to compress design-cycle times due to the flexibility it provides for rapid and relatively inexpensive evaluation of alternative designs and because it can be used to integrate multidisciplinary analysis earlier in the design process [17]. Unfortunately, bottlenecks caused by data handling, heterogeneous computing environments and geographically separated design teams, continue to restrict the use of these tools. In order to fully realize the potential of computational simulation, improved integration in the overall design process must be made. The opportunity now exists to take advantage of recent developments in information technology to streamline the design process so that information can flow seamlessly between applications, across heterogeneous operating systems, computing architectures programming languages, and data and process representations.

The World Wide Web has emerged as a powerful mechanism for distributing information on a very large scale. In its current form, it provides a simple and effective means for users to search, browse, and retrieve information, as well as to publish their own information. The Web continues to evolve from its limited role as a provider of static document-based information to that of a platform for supporting complex services. Much of this transformation is due to the introduction of object technologies, such as Java and CORBA (Common Object Request Broker Architecture) [36] within the Web. The integration of object technology represents a fundamental (some would say, revolutionary) advancement in web-technology. The web is no longer simply a document access system supported by the somewhat limited protocols. Rather, it is a distributed object system with which one can build general, multi-tiered enterprise intranet and internet applications.

The integration of the Web and object technology enables a fundamentally new approach to simulation: *Web-based simulation*. A Web populated with digital objects — models of physical counterparts — will lead to model development by composition using collaborative Web-based environments [9]. Model execution will occur across networks using Web-based technologies (e.g., Java) and distributed simulation techniques (e.g., CORBA). Finally, simulation execution, models, and other related data will be documented using forms of hypermedia (hypertext, video, virtual models, etc.).

Web-based simulation has the potential to provide the necessary tools to improve the aircraft design process through integration and support for collaborative modeling and distributed model execution. In the remainder of this paper, we examine how this might be achieved. In Section 2, we provide a brief overview of the aircraft design process, drawing attention to the complexities of the process and its inherent problems. Section 3 provides a review of the area of Web-based simulation, and singles out several principles of Web-based simulation that we believe are important in the aircraft design process. In Section 4, we present an example scenario illustrating how Web-based modeling and simulation might be used in that process, and discuss aircraft model development and distribution using the Onyx simulation framework. Onyx's object-oriented component model, visual environment for model assembly, and support for both Web-based and distributed object execution are explained in context of the integration of a jet engine within the aircraft. Lastly, in Section 5, the relationships to the Web-based simulation principles outlined in Section 3 are identified and discussed, as are general implications of Web-based simulation on the design process.

2 The Aircraft Design Process

The aircraft design process can be divided into three phases: *conceptual design*, *preliminary design*, and *detailed design*. The conceptual design phase identifies the various conditions of the mission, and synthesizes a set of initial aircraft configurations capable of performing the mission. For commercial aircraft, the mission is defined by airline company demands, which typically

include payload requirements, city-to-city distance along a proposed service route, traffic volume and frequency, and airport compatibility. If the conceptual design effort confirms the feasibility of the proposed mission, management may decide to proceed with one or more preliminary designs. In the preliminary design phase, more detail is added to the aircraft design definition. Here the aerodynamic shape, structural skeleton and propulsion system design are refined sufficiently so that detailed performance estimates can be made and guaranteed to potential customers. In the final design phase, the airframe structure and associated sub-systems, such as control systems, landing gear, electrical and hydraulic systems, and cabin layout, are defined in complete detail [17].

The design of an aircraft is an inherently complex process. Traditional preliminary design procedure decomposes the aircraft into isolated components (airframe, propulsion system, control system, etc.) and focuses attention on the individual disciplines (fluid dynamic, heat transfer, acoustics, etc.) which affect their performance. The normal approach is to perform disciplinary analysis in a sequential manner where one discipline uses the results of the preceding analysis (see Fig. 1). In the development of commercial aircraft, aerodynamic analysis of the airframe is the first step in the preliminary design process. Using the initial Computer-aided Design (CAD) geometry definitions resulting from the conceptual design studies, aerodynamic predictions of wing and fuselage lift and drag are computed. Key points in the flight envelope, including take-off and normal cruise, are evaluated to form a map of aerodynamic performance. Next, performance estimates of the aircraft's propulsion system are made, including thrust and fuel consumption rate. The structural analysis uses estimates of aerodynamic loads to determine the airframe's structural skeleton, which provides an estimate of the structure weight.

Complicating the design process is the fact that each of the disciplines interacts to various degrees with the other disciplines in the minor analysis loop. For example, the thrust requirements of the propulsion system will be dependent on the aerodynamic drag estimates for take-off, climb and cruise. The values of aerodynamic lift and yaw moments affect the sizing of the horizontal and vertical tail, which in turn influence the design of the control system. For an efficient design

process, fully-updated data from one discipline must be made accessible to the other disciplines without loss of information. Failure to identify interactions between disciplines early in the minor design cycle can result in serious problems for highly integrated aircraft designs. If the coupling is not identified until the system has been built and tested experimentally, then the system must undergo another major cycle iteration, further increasing the time and expense of product development.

There are many factors that can make the design process less efficient. These include:

(1) *Lack of interoperability.* Numerous software packages — CAD, solid modeling, FEA, CFD, visualization, and optimization — are employed to synthesize and evaluate designs. These tools are often use different, possibly proprietary, data formats. As a result, they generally do not interoperate, and require manual manipulation when passing data between applications. Although in some cases, custom translation tools are available to "massage" the data into the appropriate format, users still spend considerable time and effort tracking data and results as well as preparing, submitting and running the computer applications [28].

(2) *Heterogeneous computing environments.* The aircraft design computing environment is extremely heterogeneous, with platforms ranging from personal computers, to Unix workstations, to supercomputers. To use the various software required in the design process, users are forced to become familiar with different computer architectures, operating systems and programming languages.

(3) *Geographically separated design groups.* Multidisciplinary design and analysis is frequently carried out by geographically dispersed engineering groups. In special cases, entire subsystems may be designed and developed by third-party contractors or companies. The propulsion sub-system, for example, is designed and built separately by the propulsion company, and delivered to the aircraft company for installation in the aircraft. In any case, geographic separation places pressure on the designers to maintain a high level of interaction during the design process so that loss of data is minimized.

Improving the design process, therefore, requires the development of an integrated software environment which provides interoperability standards so that information can flow seamlessly across heterogeneous machines, computing platforms, programming languages, and data and process representations. We believe that web-based simulation tools can provide such an environment.

3 Principles of Web-based Simulation

Since its inception in 1990, the World Wide Web (WWW or Web) has quickly emerged as a powerful tool for connecting people and information on a global scale. Built on broadly accepted protocols, the WWW removes incompatibilities between computer systems, resulting in an "explosion of accessibility" [2, 30]. Within the simulation community this proliferation has led to the establishment of a new area of research — *Web-based simulation* — involving the exploration of the connections between the WWW and the field of simulation. Although the majority of work in web-based simulation to date has centered on re-implementation of existing distributed and standalone simulation software using Web-related technologies, there is growing acknowledgement that web-based simulation has the potential to fundamentally alter the practice of simulation [11].

In one of the first papers to explore the topic of web-based simulation, Fishwick [8] identifies many potential effects of web-based simulation, with attention given to three key simulation areas: (1) education and training, (2) publications, and (3) simulation programs. He concludes that there is great uncertainty in the area of Web-based simulation, but advises simulation researchers and practitioners to move forward to incorporate Web-based technologies. Building on Fishwick's observations, Page and Opper [25] present six principles of web-based simulation which capture the vision of future simulation practice: (1) digital object proliferation, (2) software standards proliferation, (3) model construction by composition, (4) increased use of "trial and error" approaches, (5) proliferation of simulation use by non-experts, and (6) multi-tier architectures and multi-language systems.

In the remainder of this section, we briefly review several of these principles. In the following sections, we will examine in more detail how each apply to both the development of a simulation environment, and to the improvement of the aircraft design process.

3.1 Digital Objects.

In the mid 1960's a pioneering simulation language called Simula-67 [3] was developed to more faithfully model objects in the physical world. Simula-67 introduced many of the core design concepts (e.g., classes and objects) which form the foundation for the object-oriented programming paradigm. Since that time, object-oriented technologies, such as object-oriented programming (OOP), design (OOD) and analysis (OOA), have had a major impact on the field of simulation. Today, the majority of simulation languages, as well as many of the most successful general purpose-languages, are object-oriented.

The importance of objects in simulation applications naturally leads us to consider their use as part of the WWW infrastructure. The WWW, however, is currently based on documents, rather than objects. In the future, though, it is envisioned that the Web will be populated by digital objects, with documents being just one type of object. The objects, representing models and data for use in simulation environments, will be made available for use through publication on the WWW [9].

Indications of a transition to an object-based WWW are currently evident in the successful application of *mobile code* and *distributed object* technologies. Mobile code — programs which can be transmitted across a network and executed on the client's computer — make it possible to deliver digital objects, in either executable or serialized form across the WWW. Several programming languages which can produce mobile code have been developed [4, 32, 33, 34]; the most well known and widely supported is Java [1]. Compiled Java code, known as byte-code, can be downloaded across the Web to the client where it is executed by a Java Virtual Machine. The Java run-time system, incorporated within the Java Virtual Machine, provides an extensive class library that can be accessed by the compiled code.

Component Object Model (COM) [29], and High Level Architecture (HLA) [21]. Alternatively, a component architecture may be defined by the particular simulation application in which the objects are to operate. This is often the case in domain-specific simulation environments, where the component architecture must be crafted to meet specific requirements of the domain. The Onyx simulation environment, described in the following section, is such an example; it defines a component architecture which is oriented towards physical modeling of aerospace systems.

3.4 Heterogeneous Modeling and Simulation

The digital objects of our Web-based simulation future will populate a Web that is highly heterogeneous. Digital objects will certainly be developed using different programming languages and programming styles (e.g., object-oriented, procedural, functional, etc.). The digital objects will themselves be highly variable. Some will be based on mobile code which can move across the Web (e.g., agents), while others will form object busses which provide services from specific locations on the Web. Applications will become more complicated as a result, with complex multi-tier architectures becoming the standard. In order to operate effectively in such an environment, Web-based simulation will need extensive enabling technologies such as search engines to locate appropriate digital objects and models, translators to convert models and data to appropriate formats, and expert systems to guide non-experts in the use of Web-based simulation models.

4 An Example Scenario

In this section, we present a scenario illustrating how Web-based modeling and simulation can be used in the aircraft design process. Our goal is to discuss both the technical issues related to the design, development and publication of digital objects, as well as organizational issues concerning the roles engineers and programmers play in the Web-based design process. Although the discussion is oriented towards the aircraft design process, we believe that it is applicable to engineering processes used in many fields.

4.1 Onyx

The modeling and simulation environment for our research is the *Onyx* simulation system [26, 27]. The major features of Onyx include the following.

- A set of object classes and interfaces for representing the physical attributes and topology of the aircraft system is included. These classes comprise an object-oriented *component architecture* capable of housing the analytical and geometric views of the various aircraft components employed in the design process. The architecture facilitates and ensures object interoperability among separately developed software components.
- A *visual assembly interface* is included for graphical creation and manipulation of aircraft system models. It enables users to establish model design, control model execution and visualize simulation output.
- A dynamically-defined, run-time *simulation executive* is included to control complex, multi-level simulations.
- A *persistence engine* capable of transparently accessing geometry and data stored in either relational or object database management systems is included.
- A *connection service* provides access to federated model and data repositories using standard internet protocols. Various connection strategies to access Web- and server-based distributed objects are included.

Our goal in creating Onyx is to develop a simulation-based design system that promotes collaboration among aerospace designers and facilitates sharing of models, data and code. Special emphasis is placed on developing a distributed system which fosters reuse and extension in both the models and the simulation environment. To achieve these goals, we have made extensive use of object-oriented technologies such as *object-oriented frameworks*, *software components*, and *design patterns*.

An object-oriented framework is a set of classes that embodies an abstract design for solutions to a family of related problems [19]. Onyx is designed as a layered collection of frameworks, with individual frameworks for the visual assembly interface, persistence engine, connection services,

simulation executive and component architecture. The set of classes in each framework define a "semi-complete" structure that captures the general functionality of the application or domain. Specific functionality is added to Onyx by inheriting from, or composing with, framework components. In the example in the next section, we will illustrate this by deriving new classes to represent the components in an aircraft engine, then assembling instances of those classes to form a complete engine model.

A key characteristic of Onyx, and object-oriented frameworks in general, is its inverted control structure. In traditional software development, the application developer writes the main body of the application which defines a series of calls to various libraries of subroutines. These libraries provide reusable code, while the main body is customized by the application developer. In framework design, the control structure is defined by the framework, with predefined calls going to methods that the application developer writes. In this approach, the design or structure of the application — which is domain-specific — is reused, and the specific functionality of the application is provided by the developer. Using this approach, Onyx reduces the burden for aircraft engineers and modelers, allowing similar aircraft component models to be developed faster and more efficiently. The concept of reuse is best illustrated for models that are assembled from a library of components (i.e., composition), and for models that are made in several versions with minor differences (i.e., inheritance).

A major product of object-oriented design is the identification of software components — self contained software elements which can be controlled dynamically and assembled to form applications. The central step in identifying them is recognizing recurring fundamental abstractions in the domain. By identifying these abstractions and standardizing their interfaces, these components become interchangeable. Such components are said to be "plug-compatible" as they permit components to be "plugged" into frameworks without redesign. Onyx's software components use a variant of the JavaBeans [7] component architecture to define standard interfaces and abstractions. These components represent the "plug-compatible, digital objects" with which the Web-based models of the aircraft and its subsystems are developed.

Throughout the Onyx environment, design patterns — recurring solutions to problems that arise when building software in various domains [13] — are used to achieve reuse. Patterns aid the development of reusable software components and frameworks by expressing the structure and collaboration of participants in a software architecture at a level higher than source code or object-oriented design models that focus on individual objects and classes [31]. Patterns also are particularly useful for documenting software architectures and design abstractions. They provide a common and concise vocabulary which is useful in conveying the purpose of a given software design.

The Onyx simulation environment is designed to be both multi-tiered and platform independent so as to provide the greatest flexibility when modeling complex aircraft systems. Java was chosen as the implementation language as it offers extensive class libraries, a distributed object model (i.e., Java RMI), and byte-code interpreters on a wide range of computer architectures, among other benefits. As a result, the Onyx system is extremely portable and accessible. The visual assembly interface (described below), for example, can be run in the context of a Web browser, which are widely available, while computationally intensive components run on dedicated, distributed servers.

Java is also the preferred language for programming Onyx software components, as models written in Java are easily downloaded across a network and dynamically loaded into the Onyx environment. In cases where it is desirable or necessary to use a programming language other than Java, software components may be accessed from Onyx using CORBA. CORBA's ability to deal with the heterogeneous nature inherent in distributed computing environments makes it particularly suitable for leveraging legacy applications not written in Java. This is especially useful for simulation of aerospace systems in which the majority of existing analysis programs have been written in procedural languages, such as FORTRAN and C. The use of CORBA adds flexibility to the Onyx system allowing it to "wrap" these existing programs, rather than having to replace or abandon them.

4.2 Engine-Aircraft Integration Scenario

This scenario illustrates our vision of how Web-based modeling and simulation may be used in the process of development and integration of an aircraft subsystem within the complete aircraft. As stated earlier, the aircraft design process generally follows a hierarchical decomposition of the aircraft system (see Fig. 2a) into major airframe components, e.g., Fuselage, Rudder, Wing and Propulsion System (i.e., Engines). Individual engineering groups are responsible for establishing the conceptual and preliminary designs for each respective component. These teams work together, exchanging information as necessary, to develop the individual component designs, and as the process progresses, to integrate them into a final design.

We have selected for our example the integration of the propulsion subsystem into the aircraft because it represents one of the more complex aspects of aircraft design. Propulsion system performance, size and weight are important factors in the overall aircraft design. Engine size and thrust, for example, influence the number and placement of engines, which in turn affects aircraft safety, performance, drag, control and maintainability. Furthermore, because the engine is designed and developed by an external manufacturer — i.e., an engine company — this example illustrates the challenges faced by designers separated both geographically and organizationally. We intend to show how Web-based modeling and simulation can address these and other issues.

4.2.1 Model Authoring. As in the aircraft company, engineering design groups in the engine manufacturer are generally organized according to a physical decomposition of the engine, with individual teams responsible for developing the major engine components: Fan, Compressor, Combustor, Turbine, Mixer, etc. (see Fig. 2b). In each team, a *model author*, having expertise in the given design area, establishes a conceptual model of the component. During early phases of design, model resolution is kept relatively coarse to speed simulations and enable more complete exploration of the design space. Such a model typically consists of a set of algebraic and/or linearized ordinary differential equations which describe the component's gross behavior. At this stage in the design knowledge of component characteristics is incomplete, so empirical data gathered from rig-testing of previously developed components are scaled to approximate the

current model. These data, commonly referred to as "performance maps," attempt to capture component characteristics within their operating range, and serve to provide closure to the equations.

4.2.2 Component Authoring. Once a conceptual model is validated, a *component author*, working closely with the model author, maps the model to the computational domain, creating a software component which encapsulates the model abstraction. As pointed out in section 3, the mapping is largely dependent on the choice of component architecture being used. The Onyx component architecture used here is based upon a *control volume* abstraction. The use of control volumes is standard engineering practice, wherein the physical system is divided into discrete regions of space — control volumes — which are then analyzed by applying conservation laws (e.g., mass, momentum, energy) to yield a set of mathematical equations describing physical behavior (see Fig. 3). A component architecture predicated on this approach provides a convenient and familiar mapping mechanism for modeling physical systems, and ensures that a simulation component resembles the conceptual model developed by the model author. A brief overview of the Onyx component architecture is presented below; a complete description can be found in ref. [26].

4.2.3 Overview of Onyx Component Architecture. There are four basic entities in the Onyx architecture: *Element*, *Port*, *Connector* and *DomainModel* (see Fig. 4). The Java interface **Element** represents a control volume, and defines the key behavior for all engineering component classes incorporated into Onyx. It declares the core methods needed to initialize, run and stop model execution, as well as methods for managing attached **Port** objects. Classes implementing this interface generally represent physical components, such as a compressor, turbine blade, or bearing, to name a few (see Fig. 3b). However, they may also represent purely mathematical abstractions such as a cell in a finite-volume mesh used in a CFD analysis. This flexibility permits the component architecture to model a variety of physical systems.

Consider, for example, a component author in the Compressor design team wanting to develop a representative Compressor digital object for use in simulations during preliminary design. The

author begins by defining a concrete implementation of the Element interface, such as SimpleCompressor (see Fig. 4). Here the author extends the abstract class DefaultElement, which captures common implementation aspects of the Element interface, as well as maintaining a list of Port objects associated with its subclasses. Alternatively, the author could implement the interface directly, explicitly defining each interface method. This feature is used through the architecture to provide flexibility: the component author may select to utilize the default functionality of the common abstract class, or inherit from another class hierarchy and implement the interface directly.

An Element may have zero or more Port objects associated with it. The interface Port represent a surface on a control volume (i.e., Element) through which some entity (e.g., mass or energy) or information passes. Ports are generally classified by the entity being transported across the control surface. For example, the SimpleCompressor has two FluidPort objects — representing the fluid boundaries at the Compressor entrance and exit — and a StructuralPort object, representing the control surface on the Compressor through which mechanical energy is passed (i.e., from a driving shaft). The Port interface defines two methods to set and retrieve the data defined by the Port. These data may be stored in any type of Java Object, such as Hashtable or Vector. The common abstract class, DefaultPort, defines default functionality for these methods, and maintains a reference to the Connector object currently connected to the Port.

The common boundary between consecutive control volumes is represented by a Connector object. The interface Connector permits two Element objects to communicate by passing information between connected Port objects (see Fig. 3c). It is also responsible for data transformation and mapping in situations where the data being passed from Ports of different type. The need for such data transformation can range from simple situations, such as conversion of data units, to very complex ones involving a mismatch in model fidelity (e.g., connecting a 2-D fluid model to a 3-D fluid model) or disciplinary coupling (e.g, mapping structural analysis results from a finite-element mesh to a finite-volume mesh used for aerodynamic analysis).

For all but the simplest cases, the algorithms needed to perform the data transformation/ mapping will tend to be very complex. To improve reusability, **Connector** delegates transformation/mapping responsibilities to a separate **Transform** object (see Fig. 3c) which encapsulates the necessary intelligence to expand/contract data and map data across disciplines. The **Transform** interface (see Fig 4) defines a general method, transform, which is implemented by subclasses to carry out a particular transformation algorithm.

A similar situation is found with the mathematical model used to define component behavior. As described above, the mathematical models used to describe Compressor (or any other component) behavior during preliminary design are relatively simple and may be solved analytically or using basic numerical methods. However, models used in latter phases of design can be quite complicated. In these cases, approximate solutions are obtained by discretization of the equations on a geometrical mesh and applying highly specialized numerical solvers. The presence of these complex mathematical models and the numerical tools needed to solve them suggest that it is desirable to encapsulate these features and remove them from the **Element** structure. This enhances the modularity of **Element**, allowing new **Element** classes to be added without regard to the mathematical model used, and conversely to add new models without affecting the **Element** class. To achieve this, Onyx utilizes the Strategy design pattern [13] to encapsulate the mathematical model in a separate type of object called **DomainModel** (see Fig. 4). The benefit of this pattern is that families of similar algorithms become interchangeable, allowing the algorithm — in this case the **DomainModel** — to vary independently from the **Elements** that use it. This admits the possibility of run-time selection of an appropriate **DomainModel** for a given **Element**; however, this is currently not used in Onyx. Furthermore, encapsulating the **DomainModel** in a separate object also encourages the "wrapping" of pre-existing, external software packages. For example, the Fan **DomainModel** in Fig. 3d might "wrap" a three-dimensional (3-D) Navier-Stokes or Euler flow solver to provide steady-state aerodynamic analysis of fluid flow within the Fan. This approach allows proven functionality of

existing software analysis packages to be easily integrated within an Element. Some of the advantages of this concept is illustrated later in this section.

The **DomainModel** interface is designed to be very general, due to the complicated nature of the various models which might be encapsulated in an **Element**. The intent is not to restrict the use of any algorithm or the "wrapping" of external software packages by overly defining the **DomainModel** interface. Consequently, the interface defines only two methods, execute and halt, which are used to start and stop the execution of the **DomainModel** code. Additional methods are obviously needed to access and make the data internal to the **DomainModel** available to the **Element**, but because these are specific to the particular **DomainModel** structure, they are not included in the interface. For our example, the component author has defined a **SimpleCompressorModel** class (see Fig. 4) to encapsulate the set of ordinary differential equations and performance maps needed to model compressor behavior.

After the Compressor class definitions (i.e., **SimpleCompressor**, **FluidPort**, **StructuralPort** and **SimpleCompressorDomainModel**) are established, the component author compiles, verifies and tests their operation. When complete, the class' byte-code files and any auxiliary data (e.g., performance maps) are combined to form a single Compressor software component in the form of a Java Archive (JAR) file. The JAR file format is useful for encapsulating components as they can be compressed to reduce file size, digitally signed for added security, and easily transferred across the Web.

4.2.4 Publishing the Component. The Compressor software component is "published" by deploying it on a Web server where it can be accessed by others in the engine company. We envision that each engine component design team will maintain its own Web server, hosting the software components it has developed (see Figure 5). However, it may be easier and more efficient to maintain all components on a single company-wide Web server. In either case, publishing the software component is the responsibility of the *component deployer*, who has expertise in system and Web server administration. This expertise is necessary, since, in addition

to simply placing components on a Web server, the component deployer is responsible for addressing server configuration issues of component identification and security.

4.2.5 Accessing Components. One of the problems facing a user of a Web-based simulation system is locating appropriate software components, objects or data, for use in a simulation. A text-based search engine, similar to those used on the Web today, is one possible method to find objects and components [9]. However, these tools suffer from the fact that they are oriented towards HTML documents, rather than objects. A more object-oriented approach is to use *naming* and *directory* services to catalog available simulation objects and components. Using a naming service, the component deployer associates names with objects, providing the means to look up an object given its name. CORBA and RMI are examples of distributed object systems that employ naming services. Directory services extend naming services by adding attributes, making it possible to search for objects given their attributes. These attributes may be used by the component deployer to describe and hierarchically organize each component. For example, the attributes may be specified which describe the component class name, model fidelity and discipline, model author, or version number, as well as the manufacturer's name and component group, to name a few. Queries can be made to the directory service to find and return references to objects matching one or more attributes. Lightweight Directory Access Protocol (LDAP) [38] and NetWare Directory Service (NDS) [23] are examples of directory services which are used today.

Another important responsibility of the component deployer is establishing and maintaining security policies controlling access to published software components. These components represent significant investments in both time and money for the manufacturer. To protect their intellectual property against theft through reverse engineering, it is important to ensure that relevant data and software components can only be accessed by authorized users. Protection is accomplished through the use of *authentication* and *authorization* mechanisms. Authentication refers to the presence of an authentication protocol (e.g., password, Kerberos ticket [24],or public key certificate (X.509 [16], PGP [39], etc.) that identifies the requesting party (the principle), while authorization grants access only if the principles identity (credentials) is included in a

specific list (the access control list), or if the principle can assume a specific role (role-based authorization). Both authentication and authorization mechanisms are typically included as part of the naming and directory services, or as part of the Web server services. Using these mechanisms, the component deployer can control who gains access to the server, and what data can be read.

Communication channels between a client and the Web server are also a source of security concern. If the communication channel is a dedicated network connection (i.e., intranet or extranet), security problems are minimized due to physical isolation. If, however, the communication channel is the Web, physical isolation is impossible, and encryption mechanisms, such as Secure Socket Layers (SSL) [15], must be used.

4.2.6 Building the Engine. Once the engine component design teams publish their preliminary component objects, a *system integrator*, having expertise in system-level engine design, combines individual component objects to create a first-order engine model. The system-level engine model is developed using Onyx's visual assembly interface. Icons, representing individual engine components (i.e., Elements), are selected from a *component browser*, dragged into a workspace window, and interconnected to form a schematic diagram (see Fig. 6).

The component browser, as its name implies, is a tool for browsing the objects and data stored in a naming or directory service (see bottom-right corner of Fig. 6). Onyx currently supports access to common naming and directory services, such as NDS, LDAP, CORBA Naming Service (COS Naming), and RMI Registry, through the Java Naming and Directory Interface (JNDI) [18]. JNDI is an API that provides an abstraction that represents elements common to the most widely available naming and directory services. JNDI also allows different services to be linked to together to form a single logical namespace called a *federated* naming service. Using the component browser, Onyx users are ale to navigate across multiple naming and directory services to locate simulation data, objects and components.

For security purposes, the component browser requires users to authenticate themselves before they can retrieve any information from a naming or directory service. Once authentication has been successfully completed, the user can browse or search (using attribute keywords) the entire

namespace (subject to any authorization restrictions). Authentication and authorization capabilities are provided through JNDI and the Java Authentication and Authorization Service (JAAS) [22] framework. These tools allow the component browser to remain independent from the underlying security services, which is an important concern when working in a heterogeneous computing environment such as the Web.

Dragging an icon from the component browser to the workspace window causes the selected software component to be downloaded from the server to the client machine. Components comprised entirely of Java classes, such as the Compressor described above, are downloaded from a Web server to the local file system where the byte-codes are extracted from the JAR file, loaded into the Java Virtual Machine and instantiated for use in Onyx. Components developed in other programming languages are not downloaded, but remain on the server. Instead, a proxy object (stub), representing the component, is downloaded and used to connect to the remote component using a distributed object service, such as RMI, Voyager [37], CORBA, or DCOM. The need to use remote components in the aircraft design process is discussed at the end of this section.

Onyx supports the creation of hierarchical component models, and an icon can represent both a single component or an assembly of components. A component with subcomponents is called a *composite* or *structured* component. Components that are not structured are called *primitive* components, since they are typically defined in terms of primitives such as variables and equations. Composite components are represented by the **CompositeElement** class, which is part of the **Element** hierarchy (see Fig. 4). The class structure, based on the Composite design pattern [13], effectively captures the part-whole hierarchical structure of the component models, and allows the uniform treatment of both individual objects and compositions of objects. Such treatment is essential for providing the object interoperability needed to perform Web-based model construction by composition.

Figure 6 shows a composite model representing an aircraft turbofan engine. The icon labeled Core is a composite of components which are displayed in the lower schematic. Each icon has one or more small boxes on its perimeter to represent its **Ports**. Connecting lines are drawn between

the ports on different icons by dragging the mouse. A Connector object having the correct Transform object needed to connect the two ports is created automatically by Onyx. Each icon has a popup menu which can be used "customize" the attributes of its Element, Port and DomainModel objects. When selected, a graphical Customizer object is displayed (see upper-right corner of Fig. 6), which can be used to view or edit the selected objects attributes. The visual assembly interface also provides tools for plotting (see the lower-left corner of Fig. 6), editing files, and browsing on-line documentation. More information on the design and implementation of the visual assembly interface can be found in ref. [26].

4.2.7 Engine-Aircraft Model Integration. The system integrator, working with the model and component authors, performs a series of simulations to evaluate and improve the performance of the first-order engine model. Component conceptual models are refined and new software components developed, deployed and integrated, until all preliminary engine design requirements are satisfied. The engine model is then "passed" to engineers in the aircraft design group for use in their design process. This is accomplished by publishing the engine model as a CompositeElement object in the same process as described above, except that the engine component is deployed on a Web server accessible from networked locations outside the engine company (i.e., extranet). In the aircraft company, airframe designers use the preliminary engine component (now a sub-component in the airframe system model) to design the control system, size the airframe and design the planform (see Fig 5). An *aircraft* system integrator takes the engine component and, using the Onyx visual assembly interface, assembles an airframe model using components (e.g., rudder, fuselage, and wing) developed by aircraft design groups (see Fig. 6) in a process similar to the one described for the Compressor component. This model can then be used to simulate gross aircraft performance.

4.2.8 Hierarchical Models. While the preliminary engine component is being used by the aircraft design teams, the engine component teams continue to refine their designs. The refinement requires sophisticated models which give a detailed description of the underlying physical processes within the component. For instance, although the air flow through the

Compressor might be adequately modeled as a quasi-one-dimensional, inviscid fluid in early phases of design, the actual fluid flow is unsteady, three-dimensional (3-D) and characterized by turbulence, boundary-layers and shocks. Similarly, at an early stage of design the Compressor blades can be modeled as rigid, but for more detailed investigations it may be necessary to account for blade deformation due to material elasticity and thermal loading. Thus, simulating the behavior of complex components requires the development of a hierarchy of models, or *multimodel*, which represent the component at differing levels of abstraction [10]. These models may include: lumped-parameter models, such as the one used to model the Compressor component in preliminary design, or distributed parameter models such as fluid dynamics (CFD) or structural mechanics (FEA). Each model is implemented using a numerical method best suited to the application; e.g., an ordinary differential equation solver (ODE) for state-space models, finite-element solvers for structural mechanics or finite-volume solver for fluid dynamics. The specific numerical method implementation is encapsulated within the model. Figure 2c shows a multimodel representing the Compressor blade and flowfield at differing levels of fidelity. At the lowest level of fidelity, both the blade and flowfield are modeled using simple differential equations and empirical data. At higher fidelities, both are modeled using sophisticated numerical methods such as finite element analysis or computational fluid dynamics.

4.2.9 High-fidelity Distributed Components. The use of multimodels in Web-based modeling and simulation is important because it allows designers to selectively refine the fidelity of their model given the constraints (i.e., level of detail needed, the objective, the available knowledge, given resources, etc.) of the simulation. However, digital objects containing higher-fidelity models cannot be deployed in the same manner as the simple models described previously. High-fidelity CFD and FEA software packages are (generally) not written in Java, and thus cannot be run in the clients Java virtual machine. Even if this were possible, the packages are computationally intensive, making them unsuitable for execution on the client computer. Therefore, high-fidelity models are deployed as remote objects using distributed object services such as CORBA. This approach offers several advantages:

(1) Ability to distribute a computationally intensive process across a number of processors

(2) Ability to leverage legacy code limited to platforms offering specific programming and/or operating systems by "wrapping" it in a remote object

(3) Specialization of computer execution environment (i.e., placement of codes on appropriate computing platforms: such as visualization codes on high-end graphic workstations; computationally intensive codes on supercomputers, etc.).

As with the preliminary component models, the high-fidelity component models can be integrated into a system-level engine model by the engine system integrator, and used to simulate engine operation. An engine simulation using a model composed of high-fidelity components would provide detailed knowledge of the interaction effects between its components. Although these interactions can be critical to engine performance, they are not currently quantifiable by engine designers and therefore are unknown until after expensive hardware testing [5, 14]. Evaluation of these effects will allow engine engineers to make better design decisions earlier in the design process, before the principle design features have been frozen. Each high-fidelity component would perform its computations using a wrapped analysis package located on one or more remote computers. For example, in Fig. 5, the Fan component is run on a supercomputer, while a parallel software package is used to simulate Compressor operation using a cluster of computers.

The high-fidelity engine model is also a valuable resource to aircraft designers, and once the model is published, can be used in the aircraft model. The engine model allows aircraft designers to investigate the flowfield around aircraft nacelle (the cowling structure around the engine) and fuselage. Detailed descriptions of flow features at the engine exit (e.g., shocks and expansion waves), could allow aircraft designers to better predict the drag caused by the jet exhaust flowing along the aircraft surface. Engine designers would also benefit from a high-fidelity, integrated engine-aircraft simulation. For example, an integrated simulation could allow engine designers to study distortions in the airflow entering the engine when the aircraft is at a high angle of attack. Evaluation of this operating condition is important because distortions can cause the compressor to stall and the engine to lose thrust. A detailed engine-aircraft integration study would provide

valuable information which engine and aircraft engineers could use to better and more quickly design the aircraft.

5 Concluding Remarks

The design of complex systems involves the work of many specialists in various disciplines, each dependent on the work of other groups. When a single designer or core team is able to control the entire design process, difficulties in communication and organization are minimized. However, as design problems become more complex, the number and size of disciplinary groups increases, and it becomes more difficult for a central group to manage the process. As the design process becomes more decentralized, communications requirements become more severe. These difficulties are particularly evident in the design of aircraft, a process that involves complex analyses, many disciplines, and a large design space [20]. The lack of enabling software supporting disciplinary analysis by geographically dispersed engineering groups further aggravates these problems.

In this paper we have argued that Web-based simulation has the potential to improve the aircraft design process, allowing companies to become more competitive through condensed cycle times and better products. This improvement is due, in part, to the ability of the Web to support collaborative modeling and distributed model execution in a heterogeneous computing environment. A central focus of this strategy is the move towards a Web based on digital objects which can be published and reused to form new models.

Using a component architecture such as the one defined in the Onyx environment, digital objects can be developed which represent the hierarchical topology of physical systems, making them ideal as models of aircraft systems. Furthermore, these objects can encapsulate multimodels, including geometry models, multidisciplinary models and models having multiple levels of fidelity. Such models are ideal for concurrent design environments, since all of the modeling information is available in one place. The component architecture class structure provides the

capability to wrap existing software packages. This is extremely important in providing collaborative and integrative environment for the aircraft design process.

A World-Wide Web populated with digital objects provides the foundation for modeling by composition. Onyx's component architecture defines the standard interfaces needed to dynamically compose new objects and the visual assembly interface makes composition simple and easy. This promotes model reuse, as well as reducing new model development time.

The Onyx environment supports the distribution of simulation models across the Web. Both Web-based model distribution (in the case of Java-based models) and distributed services approaches (e.g., CORBA, COM) are provided. Each of these increase Onyx's usability, as models can be placed virtually anywhere. The CORBA bindings make it possible to integrate non-Java language distributed objects and legacy code. Also, since Onyx is written entirely in Java, it is portable without modifications to any computing platform which supports the Java Virtual Machine. Heterogeneous computing support makes the Onyx Web-based simulation system extremely viable for use in the heterogeneous computing environments typical of aircraft companies. Most importantly, it allows access to existing legacy codes and access to codes which must operate on specific architectures or operating systems.

References

[1] Arnold, K. and Gosling, J., 1996, *The Java Programming Language*, Addison Wesley Publishing Company, Inc., Reading, MA.

[2] Berners-Lee, T., 1996, "WWW: Past, Present, and Future," *Computer*, **29**(10) p. 69.

[3] Birtwistle, G., Dahl, O., Myhrhaug, B. and Nygaard, K., 1973, *Simula begin*, Petrocelli Charter, New York.

[4] Cardelli, L., 1994, "Obliq: A Language with Distributed Scope," Research Report 122, Digital Equipment Corporation Systems Research Center, Palo Alto, CA. On-line document. Available at http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-122. html.

[5] Claus, R. W., Evans, A. L., Lytle, J. K., and Nichols, L. D., 1991, "Numerical Propulsion

System Simulation," *Computing Systems in Engineering*, Vol. 2, pp. 357-364

[6] Eddon, G. and Eddon, H., 1998, *Inside Distributed COM*, Microsoft Press, Redmond, Washington.

[7] Englander, R., 1997, *Developing Java Beans*, O'Reilly & Associates, Inc., Sebastopol, CA.

[8] Fishwick, P.A., 1996, "Web-Based Simulation: Some Personal Observations," *Proceedings of the 1996 Winter Simulation Conference*, J.M. Charnes, D.J. Morrice, D.T. Brunner and J.J. Swaim (eds.), pp. 772-779, Coronado, CA.

[9] Fishwick, P.A., 1998, "Issues with Web-Publishable Digital Objects," *Proceedings of SPIE: Enabling Technologies for Simulation Science II*, pp. 136-142, Orlando, FL, April 14-16.

[10] Fishwick P. A. and Zeigler, B. P., 1992, "A Multimodel Methodology for Qualitative Model Engineering," *ACM Transactions on Modeling and Computer Simulation*, Vol. 12, pp. 52-81.

[11] Fishwick, P.A., Hill, D.R.C. and Smith, R., Eds., 1998, *Proceedings of the 1998 International Conference on Web-Based Modeling and Simulation*, SCS Simulation Series **30**(1).

[12] Freeman, E., Hupfer, S., and Arnold, K., 1999, *JavaSpaces™ Principles, Patterns, and Practice*, Addison-Wesley.

[13] Gamma, E., Helm, R, Johnson, R., and Vlissides, J., 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Publishing Company, Inc., Reading, MA.

[14] Hall, E.J., Delaney, R.A., Lynn, S.R. and Veres, J.P., 1998, "Energy Efficient Engine Low Pressure Subsystem Aerodynamic Analysis," AIAA Paper No. 98-3119.

[15] Hickman, K.E.B., 1995, *The SSL Protocol*. Available at http://home.netscape.com/eng/security/SSL_2.html.

[16] Housley, R., Ford, W., Polk, T., and Solo, D., 1999, "Internet X.509 Public Key Infrastructure Certificate and CRL Profile. Request for Comments 2459," Internet Engineering Task Force. Available at http://www.imc.org/rfc2459 .

[17] Jameson, A., 1997, "Re-Engineering the Design Process through Computation," AIAA Paper No. 97-0641.

[18] Java Naming and Directory Interface. Available at http://java.sun.com/products/jndi/index.html.

[19] Johnson R. E. and Foote, B., 1988, "Designing Reusable Classes, *The Journal Of Object-*

Oriented Programming," **1**(2), pp. 22-35.

[20] Kroo, I., Altus, S., Braun, R., Gage, P., and Sobieski, I., 1994, "Multidisciplinary Optimization Methods for Aircraft Preliminary Design," AIAA Paper No. 94-4325.

[21] Kuhl, F., Weatherly, R. and Dahmann, J., 1999, *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*, Prentice Hall.

[22] Lai, C., Gong, L., Koved, L., Nadalin, A. and Schemers, R. 1999, "User Authentication And Authorization In The Java™ Platform," To appear in *Proceedings of the 15th Annual Computer Security Applications Conference*, Phoenix, AZ.

[23] Lindberg, K.J.P., 1998, *Novell's Netware 5 Administrator's Handbook*, IDG Books Worldwide.

[24] Neuman, B.C. and Ts'o, T., 1994, "Kerberos: An Authentication Service for Computer Networks," *IEEE Communications*, **32**(9), pp.33-38.

[25] Page, E.H. and Opper, J.M., 1999, "Investigating the Application of Web-Based Simulation Principles within the Architecture for a Next-Generation Computer Generated Forces Model," *Future Generation Computer Systems*, to appear.

[26] Reed, J.A., 1998, "Onyx: An Object-Oriented Framework for Computational Simulation of Gas Turbine Systems," Ph.D. dissertation, The University of Toledo, Toledo, Ohio.

[27] Reed, J.A., and Afjeh, A.A., 1998, "An Object-Oriented Framework for Distributed Computational Simulation of Aerospace Propulsion Systems," *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, Santa Fe, New Mexico.

[28] Ridlon, S. A., 1996, "A Software Framework for Enabling Multidisciplinary Analysis and Optimization," AIAA Paper No. 96-4133.

[29] Rogerson, D., 1996, *Inside COM*, Microsoft Press, Redmond, Washington.

[30] Schatz, B.R., and Hardin, J.B., 1994, "NCSA Mosaic and the World Wide Web: Global Hypermedia Protocols for the Internet," *Science*, 265, p. 895.

[31] Schmidt, D. C., 1997, "Applying Design Patterns and Frameworks to Develop Object-Oriented Communications Software," *Handbook of Programming Languages*, Volume I, P. Salus, ed., MacMillian Computer Publishing.

[32] Smith, R.B., and Ungar, D., 1995, "Programming as an Experience: The Inspiration for Self," *Proceedings of ECOOP'95*.

[33] Watters, A., van Rossum, G., and Ahlstrom, J., 1996, *Internet Programming with Python*, MIS Press/Henry Holt Publishers.

[34] Wirth, N. and Gutknecht, J., 1989, "The Oberon System," *Software: Updated Practice and Experience*, **19**(9), p. 857.

[35] Wollrath, A., Riggs, R. and Waldo, J., 1996, "A Distributed Object Model for the Java™ System," *Proceedings of the Second USENIX Conference on Object-Oriented Technology and Systems (COOTS)*, pp. 219-231.

[36] Vinoski, S, 1997, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communications*, **35**(2), pp. 46–55.

[37] Voyager, 1997, "Voyager: The Agent ORB for Java" Online document. Available at http://www.objectspace.com/.

[38] Yeong, W., Howes, T., and S. Kille, "Lightweight Directory Access Protocol", Request For Comments 1777," Internet Engineering Task Force. Available at http://www.ietf.org/rfc/rfc1777.txt.

[39] Zimmerman, P. 1994, *PGP User's Guide*, MIT Press, Cambridge, 1994.

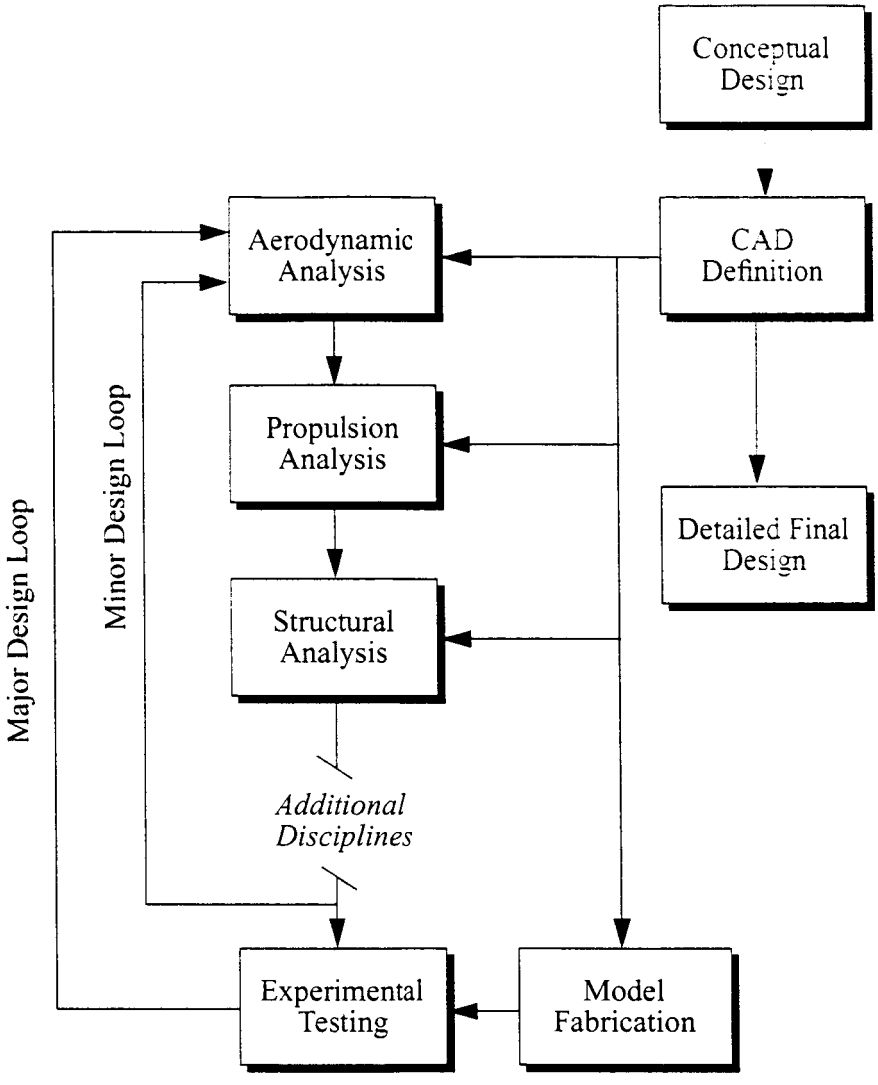

Figure 1: The Aircraft Design Process. The process involves conceptual, preliminary and detailed final design phases. The preliminary design phase includes both major and minor design loops. In the minor design loop, separate disciplinary analysis such as aerodynamic, propulsion, and structural analysis are carried out. Additional disciplinary analysis, such as controls, loading, stability, acoustics, etc. have been omitted for clarity. Once a design is converged upon in the minor loop, it is experimentally tested in the major design loop. After convergence of the major design loop, the detailed final design phase is executed.

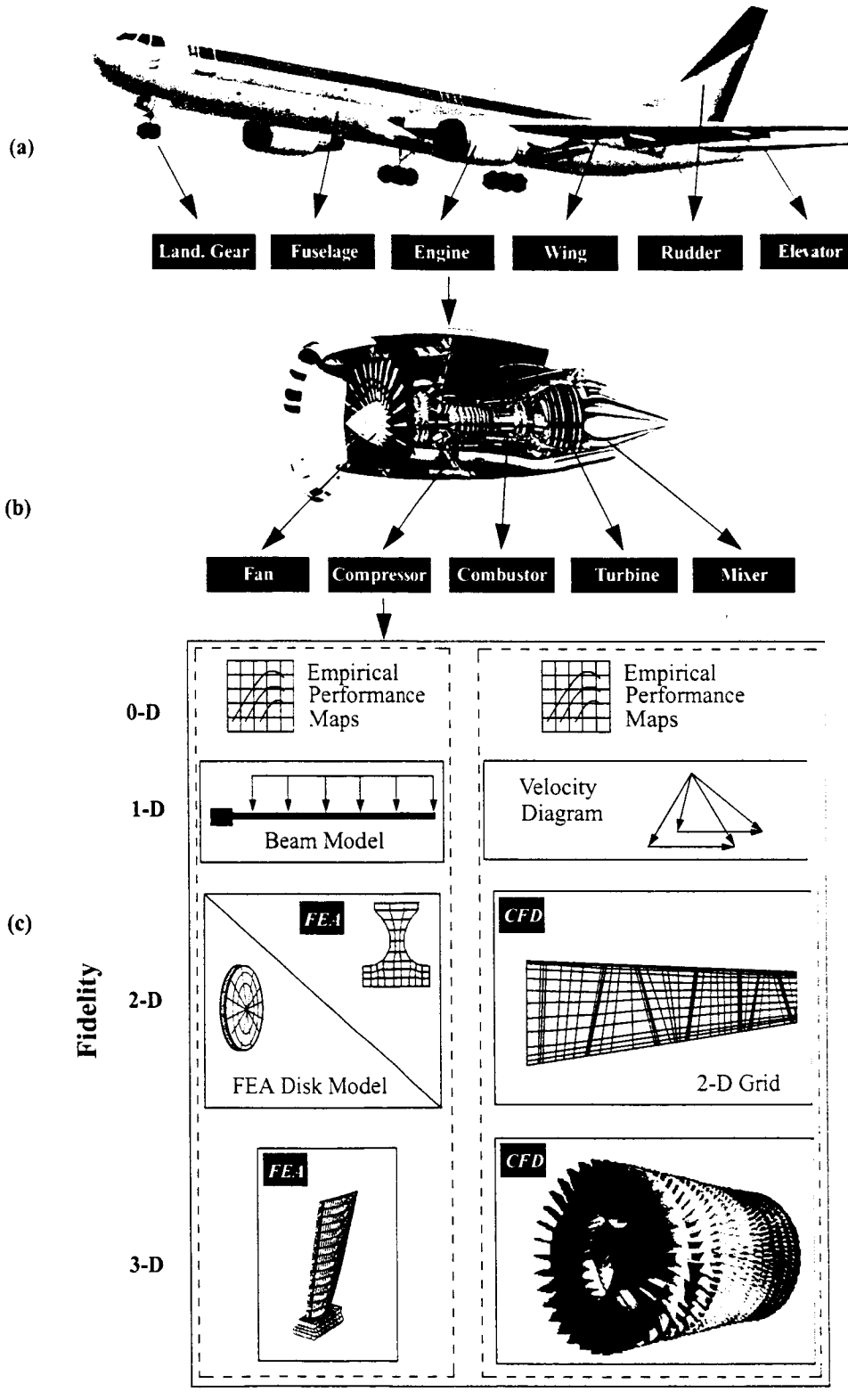

Figure 2: (a) Decomposition of aircraft into high-level components; (b) decomposition of engine component; and (c) collection of models (multimodels) at differing levels of fidelity and discipline for Compressor component.

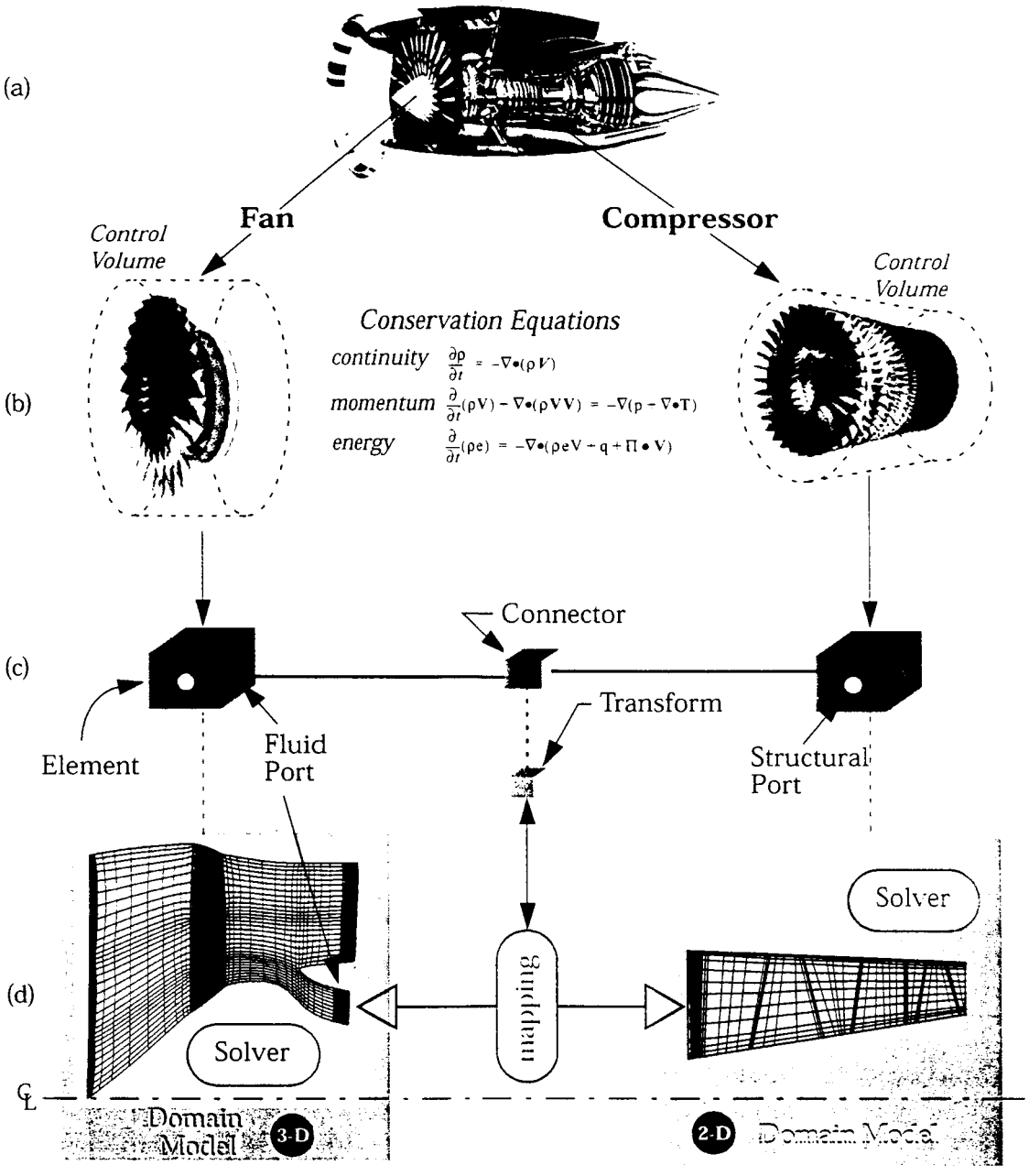

Figure 3: Mapping of engine physical domain to computational framework. (a) Engine is decomposed into separate components, such as the Fan and Compressor. Component control volumes are defined (b), with behavior defined by conservation laws. Components are represented in Onyx as **Elements** (c), whose **Ports** are connected by **Connectors**. Component behavior is defined by a **DomainModel** (d) which may apply numerical discretization methods to solve the conservation equations. Data exchange at control volume boundaries is passed via **Ports** and **Connectors**, with multifidelity and interdisciplinary mapping handled by **Transform** objects.

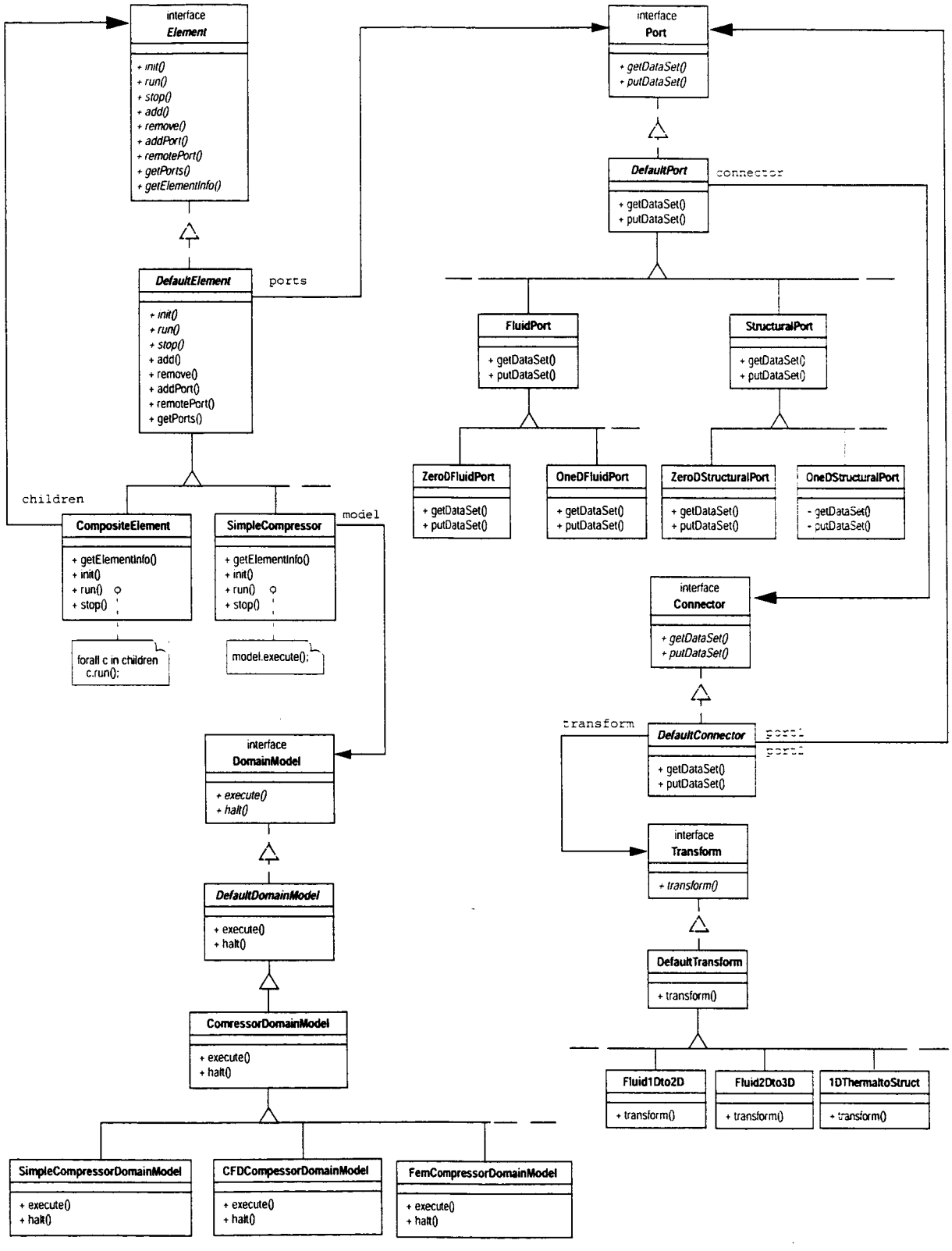

Figure 4: A portion of the Onyx component architecture class structure.

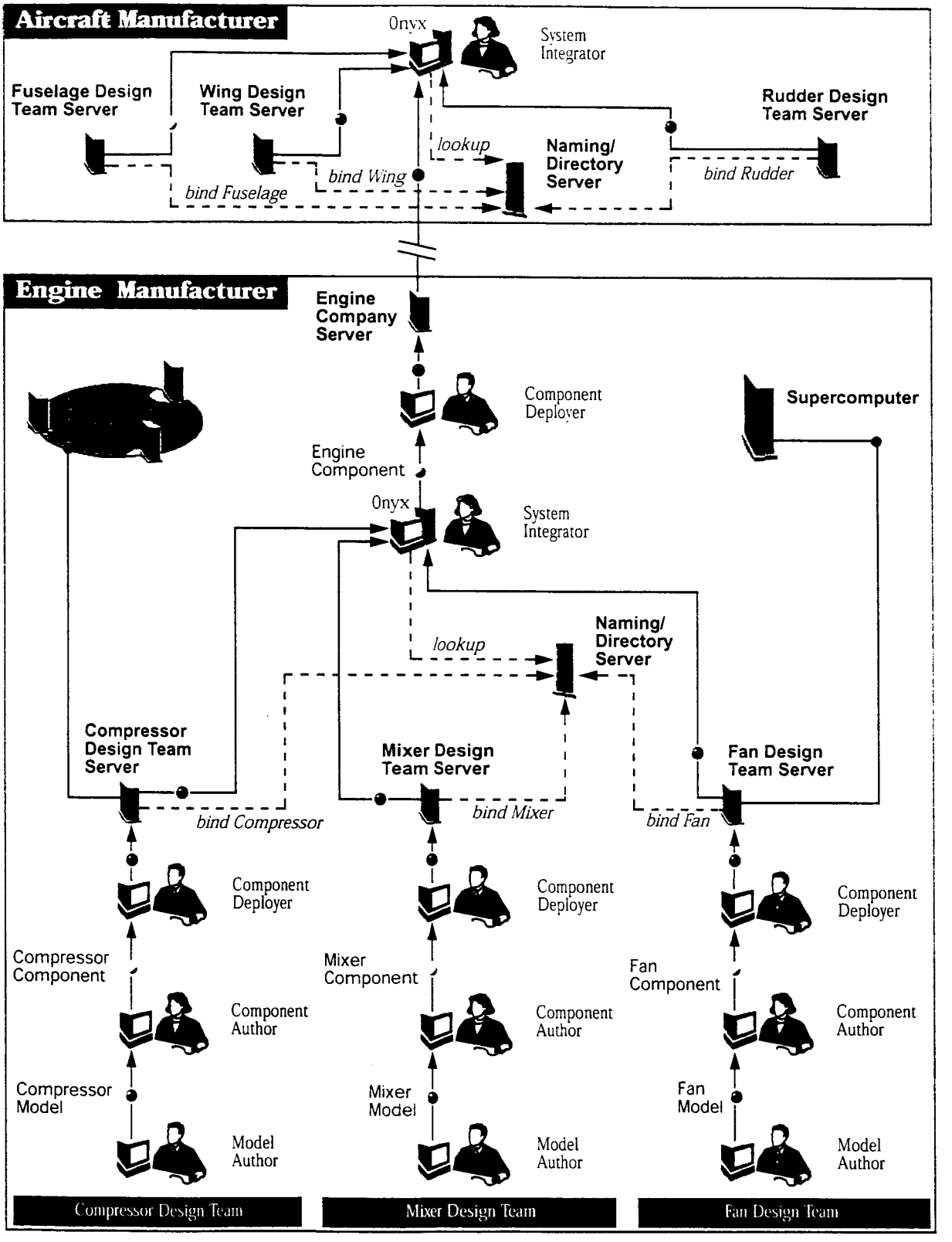

Figure 5: Exchange of digital objects in a Web-based simulation environment.

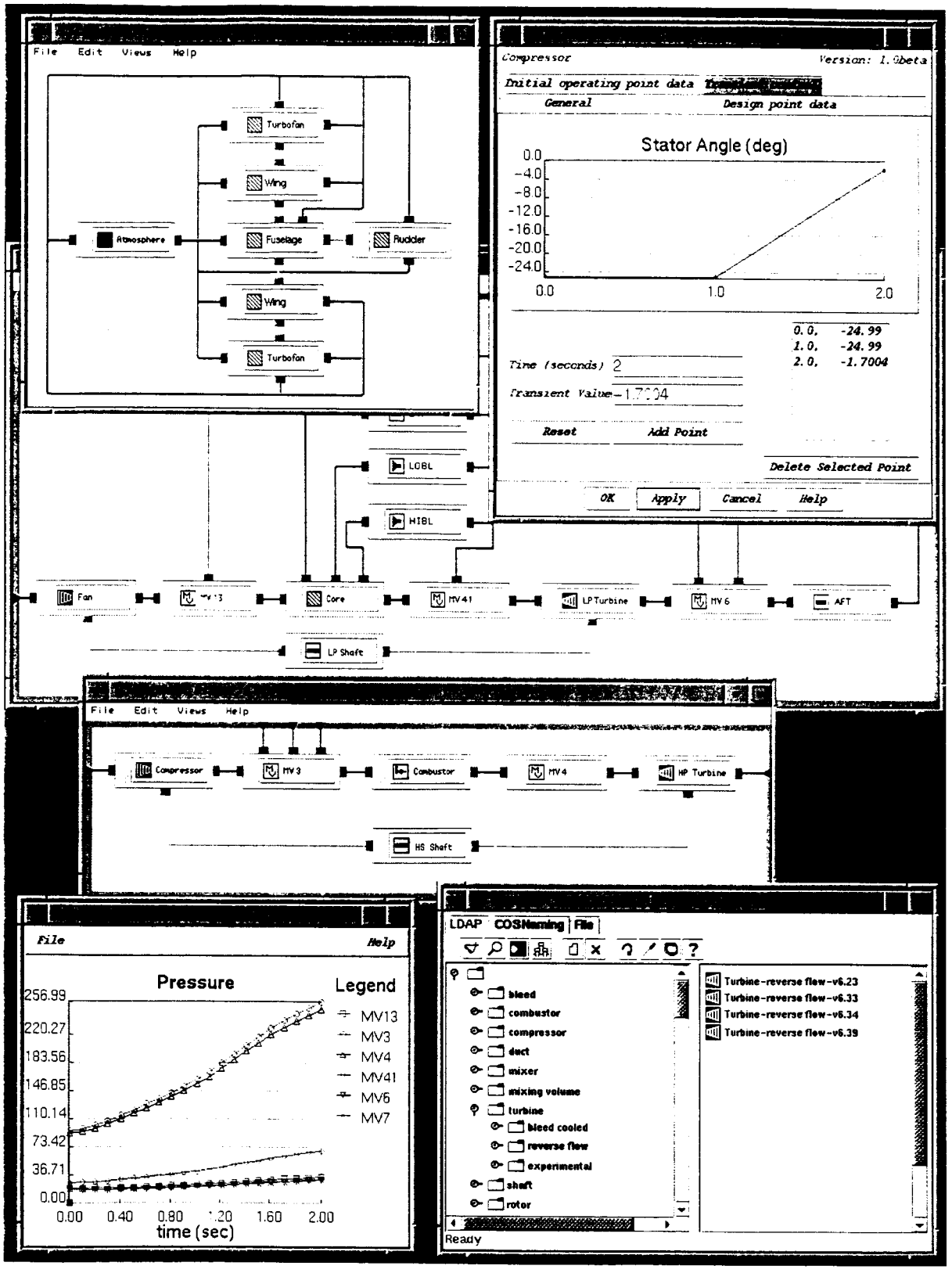

Figure 6: Overview of Onyx Visual Assembly Interface.