Check for updates

THE AMERICAN SOCIETY OF MECHANICAL ENGINEERS
Three Park Avenue, New York, N.Y. 10016-5990

99-GT-347

The Society shall not be responsible for statements or opinions advanced in papers or discussion at meetings of the Society or of its Divisions or Sections, or printed in its publications. Discussion is printed only if the paper is published in an ASME Journal. Authorization to photocopy for internal or personal use is granted to libraries and other users registered with the Copyright Clearance Center (CCC) provided $3/article is paid to CCC, 222 Rosewood Dr., Danvers, MA 01923. Requests for special permission or bulk reproduction should be addressed to the ASME Technical Publishing Department.

Copyright © 1999 by ASME All Rights Reserved Printed in U.S.A.

COMPUTATIONAL SIMULATION OF GAS TURBINES: PART II — EXTENSIBLE DOMAIN FRAMEWORK

John A. Reed and **Abdollah A. Afjeh**
Mechanical, Industrial & Manufacturing Engineering Dept.
The University of Toledo
Toledo, Ohio, 43606, USA
E-mail: jreed@eng.utoledo.edu, aafjeh@eng.utoledo.edu

Keywords: *object-oriented modeling, gas turbines, computational simulation*

ABSTRACT

This paper describes the design concepts and object-oriented architecture of *Onyx*, an extensible domain framework for computational simulation of gas turbine engines. Onyx provides a flexible environment for defining, modifying and simulating the component-based gas turbine models described in Part I of this paper. Using advanced object-oriented technologies such as design patterns and frameworks, Onyx enables users to customize and extend the framework to add new functionality or adapt simulation behavior as required. A customizable visual interface provides high-level symbolic control of propulsion system construction and execution. For computationally-intensive analysis, components may be distributed across heterogeneous computing architectures and operating systems. A distributed gas turbine engine model is developed and simulated to illustrate the use of the framework.

1 INTRODUCTION

The design of gas turbine engine simulation software has traditionally focused on developing single, custom-built systems. As a result, leveraging software design has generally been done in an ad hoc manner. It is widely recognized, however, that such an approach, while adequate for simple systems, is not cost effective for realizing large, complex systems. Systematic software reuse—the process of creating software systems from predefined software components—offers a methodology for moving from the practice of hand-crafted systems to a more formal approach based on engineering principles (Prieto-Díaz, 1990). Object-oriented technology (OOT) has been embraced by many as a sufficient means for software reuse, but OOT does not, by itself, promote large-scale system reuse.

Domain analysis and modeling is an emerging collection of software design methodologies which aim to identify the essential features, components, capabilities, interfaces, abstractions, etc., of a family of systems in a domain (Batory et al., 1995). Based mostly on object-oriented techniques, they enhance those techniques with respect to design for reusability. In recent years, two approaches to domain modeling have emerged: integrative modeling and generative modeling. Integrative modeling directly extends the object-oriented modeling methodologies and typically defines an integrated set of sub-models in the form of object composition/aggregation and generalization/specialization hierarchies. Generative models primarily deal with software architecture issues such as identifying the fundamental programming abstractions in the domain, creating libraries of interoperable software components, and defining knowledge-representation languages for combining and coupling software "building blocks." Two special cases of generative models—*design patterns* and *object-oriented frameworks*—have been shown to be beneficial to the development of reusable and flexible, domain-specific software systems.

Design patterns (Gamma et al, 1995) are a promising technique for achieving widespread reuse of software architectures. A design pattern is a recurring solution to problems that arise when building software in various domains. Patterns aid the development of reusable components and frameworks by expressing the structure and collaboration of participants in a software architecture at a level higher than (i) source code or (ii) object-oriented design models that focus on individual objects and classes (Schmidt, 1997). 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.

While patterns are useful in promoting reuse, they are not sufficient to create flexible software by themselves. Patterns are, in general, language-independent and do not directly produce reusable code (Schmidt, 1995). It is therefore important to apply patterns in a manner which capitalizes on the reuse of abstract design, architecture knowledge and code. Object-oriented frameworks do precisely this.

An object-oriented framework is a set of classes that embodies an abstract design for solutions to a family of related problems (Johnson and Foote, 1988). The set of classes define "semi-complete" applications that capture domain-specific object structures and functionality. Specific functionality in new applications is realized by inheriting from, or composing with, framework components.

Presented at the International Gas Turbine & Aeroengine Congress & Exhibition
Indianapolis, Indiana — June 7–June 10, 1999
This paper has been accepted for publication in the Transactions of the ASME
Discussion of it will be accepted at ASME Headquarters until September 30, 1999

Downloaded from http://asmedigitalcollection.asme.org/GT/proceedings-pdf/GT1999/78583/V001T01A010/4215578/v001t01a010-99-gt-347.pdf by guest on 02 October 2022

A major product of domain modeling is the identification of software components—self contained software elements which can be controlled dynamically and assembled to form applications (Englander, 1997). 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.

This paper describes the architecture of *Onyx*, an object-oriented domain framework for the modeling and simulation of gas turbine systems. Onyx leverages object-oriented design techniques—design patterns, frameworks, and software components—to provide a flexible and extensible environment which can be used to compose new engine component models, to inspect and edit existing models, and simulate and display execution results. These models, which are based on the Common Engineering Model described in Part I of this paper, provide plug-compatible software components which users can combine to form increasingly complex gas turbine models. The following section provides an overview of the structure of the Onyx framework and the Java™ programming platform (Arnold and Gosling, 1996) which is used by Onyx.The next three sections describe the major features of the framework, including the graphical user interface, distributed services, and execution control capabilities. The final section illustrates how Onyx is used to build and simulate gas turbine engine models. An example engine model is visually composed using the example engine model introduced in Part I, and a distributed simulation demonstrated.

2 ONYX

Onyx is arranged as a layered collection of four individual framework:

- **Engine Component Framework (ECF).** This framework contains collections of software components which represent physical components in the gas turbine domain (e.g., compressor, turbine, combustor, etc.). The components are based on the Common Engineering Model defined in Part I and are represented in the ECF by EngElement and Port objects, which are connected by Connector objects.
- **Visual Assembly Framework (VAF).** This framework "sits" on top of the Engine Component Framework and provides visual representations of the objects in the Engine Component Framework. The VAF forms a major part of the Onyx Graphical User Interface (GUI) and provides tools to visually assemble and manipulate simulation models in the ECF.
- **Connection Services Framework (CSF).** The CSF is an extensible mechanism which provides both local and distributed connection services for the ECF and VAF. The CSF is responsible for locating and loading both local and distributed components, and making them available for instantiation in the ECF and VAF.
- **Execution Control Framework (XCF).** This framework provides a set of tools with which to control Onyx simulations. The XCF defines a high-level abstraction mechanism for creating execution strategies for the simulation, as well as registration and selection of numerical solvers for solution of the system of equations.

This layering of frameworks enhances Onyx's flexibility as the individual frameworks are loosely coupled. This allows each framework to be used more-or-less independently. For example, the Engine Component Framework, Connection Services and Execution Control Frameworks can be used without the Visual Assembly Framework. From a design perspective, the reduction of coupling between frameworks is beneficial as it helps to minimize the effect that a design change to one framework has on another.

3 VISUAL ASSEMBLY FRAMEWORK

Engineers often use schematic drawings to represent gas turbine systems and subsystems. It is natural then to represent computational simulations of such entities using this visual metaphor. The Visual Assembly Framework (VAF) provides a high-level architecture for visually creating, assembling and manipulating engine schematics based in engine component models developed using EngElement classes.

The design requirements of the Visual Assembly Framework were: (i) visual elements are needed to represent the objects which form Onyx's engine component model; (ii) the concept of component composition developed in Part I must also be supported visually; and (iii), the framework must take care of managing basic graphical functions—window management, displaying objects, moving and dragging visual elements, tracking mouse movements, etc. This reduces the programming burden for developers using the framework.

In addition to these goals were some constraints. First, the framework should decouple the visual user interface (UI) objects from their counterparts in the engine component framework. Such an approach would allow a component's UI to be changed easily, possibly at run-time without affecting the structure of the engine component object. Second, the design should allow the developer to override the default visual representations as much as is practically possible.

The Java platform was selected for developing Onyx in part due to its integrated graphical support. Onyx uses the Swing component set—a subset of the Java Foundation Classes (JFC) (Weiner and Asbury, 1998)—to implement it's graphic interface. Because Swing components are lightweight (i.e, not based on the native windowing system), they are very flexible and have the ability to support multiple look-and-feel standards (e.g, Windows™, Motif™ and Macintosh™).

3.1 Overview Of The Visual Assembly Framework

A gas turbine simulation model is developed by building a schematic diagram of the gas turbine in a SchematicFrame window (see Fig. 1). Individual engine components, represented graphically as SchematicIcons, are selected from the ToolBox window and dragged into a SchematicFrame. The SchematicIcons are then interconnected to form a schematic representation.

A SchematicIcon is composed of a VEngElement and one or more VPorts. VEngElement is the visual analog of the EngElement class in the Engine Component Framework (ECF). By default, it is composed of an Icon, which presents an image of the engine component, and a Label, which displays the name of the EngElement object it represents. VPorts are the visual representation of the Port objects in the ECF. VPorts are the connection points between VEngElements, and are color-coded to represent the Port's type (e.g.,

Downloaded from http://asmedigitalcollection.asme.org/GT/proceedings-pdf/GT1999/78583/V001T01A010/4215578/v001t01a010-99-gt-347.pdf by guest on 02 October 2022

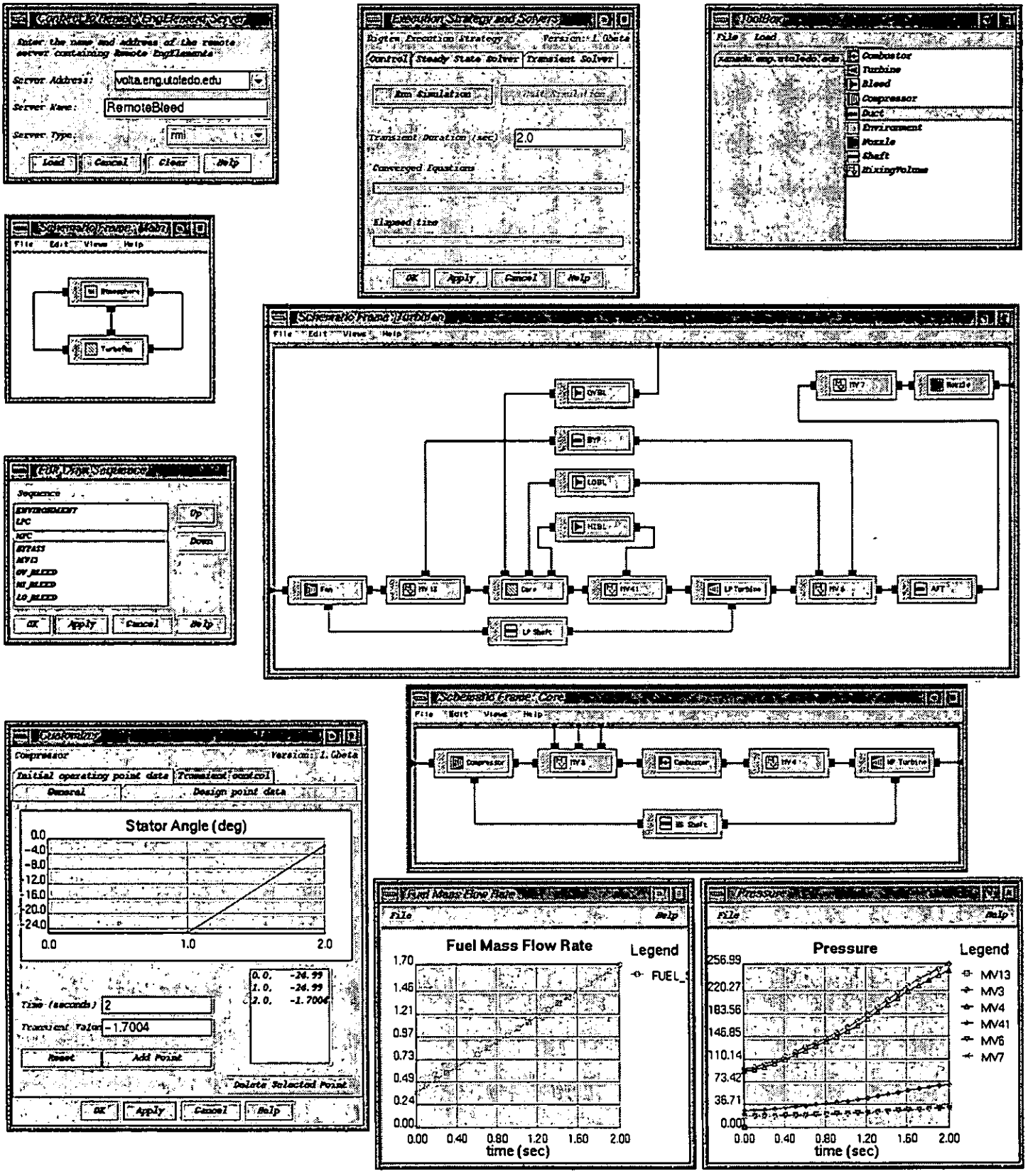

Figure 1: Overview of Onyx Visual Assembly Framework.

Downloaded from http://asmedigitalcollection.asme.org/GT/proceedings-pdf/GT1999/78583/V001T01A010/4215578/v001t01a010-99-gt-347.pdf by guest on 02 October 2022

fluid transfer, mechanical, etc.). VConnectors (i.e., *visual* Connectors) define a graphical connection between two VPorts, and represent the Connector objects in the ECF.

Hierarchical engine schematic structures can be created in the VAF using SchematicIcons and SchematicFrames. The approach is based in-part on the work of Curlett et al. (1995). A hierarchical model is graphically constructed by adding a Composite SchematicIcon to a model in an existing SchematicFrame. This creates a new SchematicFrame which represents a deeper (more refined) level in the structural hierarchy. At the same time, it creates a CompositeEngElement in the ECF; this is represented by the SchematicFrame. Additional SchematicIcons can then be dragged into this new SchematicFrame to form a sub-model schematic. In Fig. 1, the SchematicIcon labeled Core is a CompositeEngElement. Each of the EngElements in the Core composite are also represented by SchematicIcons. These are seen as a separate schematic in the SchematicFrame window titled Core.

SchematicIcons in the Core sub-model are connected to SchematicIcons in the parent model by *promoting* the sub-model SchematicIcon's VPort. Every SchematicFrame in a simulation (except the top-level SchematicFrame), has a *promotion border*, which appears as a solid border around the inner edge of the SchematicFrame. When a connection is made from a VPort into the promotion border, the VPort is connected to a BorderVPort in the promotion border, and a new VPort is added to the Composite SchematicIcon in the parent model. For example, in Fig. 1, the Compressor SchematicIcon's input VPort (on the left-hand side of the SchematicIcon) has been promoted. The VPort is depicted in the Turbofan SchematicFrame as a VPort on the left-hand side of the Composite SchematicIcon. This VPort may now be connected to any compatible VPort in the parent model schematic. In this case, it is connected to the MV13 SchematicIcon. Similarly, the HP Turbine SchematicIcon's exit VPort has been promoted and appears on the right-hand side of the Composite SchematicIcon. It is connected to the MV41 SchematicIcon in the parent model schematic.

The location of a promoted VPort on the Composite SchematicIcon corresponds to the relative location of the BorderVPort location in the promotion border. A BorderVPort can be dragged to any point within the promotion border, and the corresponding VPort is moved accordingly.

Each VEngElement, VPort and VConnector has a popup menu associated with it. The menu allows the user to access various functions such as moving, deleting, copying, etc. In the VEngElement, the popup menu has a special item for "customizing" the component. When selected, a Customizer object is displayed. Customizers are graphical interfaces which allow the user to change an EngElement's attributes (see lower-left corner of Fig. 1). Typically, these are used to supply user-defined data to the EngElement's Domain Model. The VAF also provides additional tools for plotting, editing files, and displaying Web browsers to search on-line documentation.

3.2 SchematicIcons

An objective of the Visual Assembly Framework (VAF) is to provide a SchematicIcon for each type of EngElement available in the Engine Component Framework (ECF). Each EngElement type may have different numbers and types of Ports, as well as different Domain Models. Consequently, the VAF must provide SchematicIcons with VPorts and Customizers that match the EngElement's type. One way to

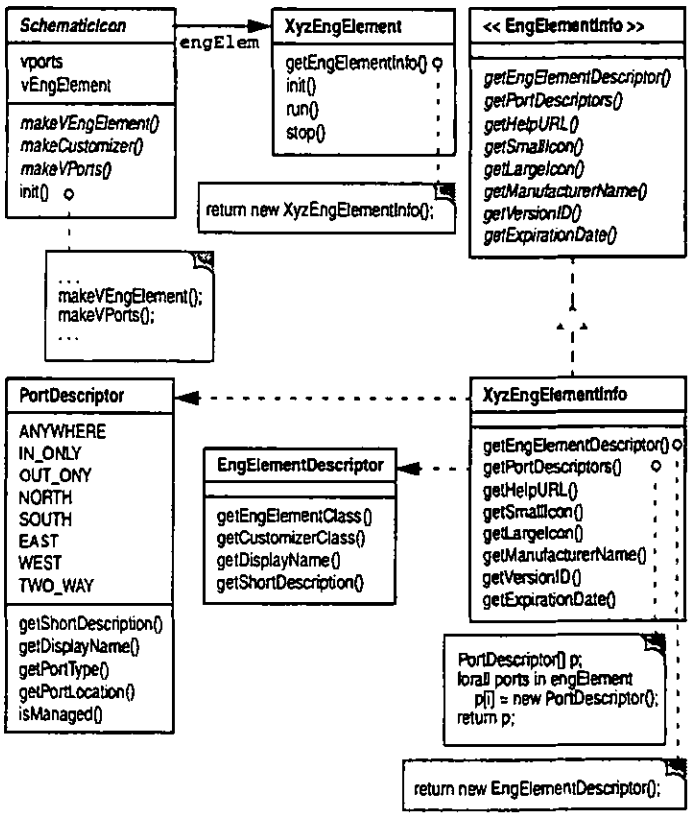

Figure 2: SchematicIcon "Info class" structure diagram.

account for this variability in the VAF design is to represent SchematicIcon as an abstract class. Subclasses can then define appropriate Icons, Customizers, and VPorts for a particular EngElement type. This approach, however, leads to a very broad and shallow inheritance tree, indicating poor use of inheritance.

A more flexible approach is to create SchematicIcons using object composition. The VAF uses a variation of the JavaBeans™ "BeanInfo" class approach (Englander, R., 1997) to compose SchematicIcons with the correct attributes (i.e, Icon, Customizer and VPorts). In this technique, each EngElement is provided with a separate auxiliary "Info class" which identifies and encapsulates its UI-specific attributes. The benefit of using an Info class is that it separates the UI part of the engine component representation from the underlying component model defined by the EngElement. This is extremely useful in reducing the complexity of the EngElement. It also provides flexibility, since the EngElement's UI may be redefined (at runtime, if desired) without affecting the EngElement itself.

The "Info class" mechanism centers around the use of the EngElementInfo interface (see Fig. 2). The interface defines a set of methods for obtaining UI information for an EngElement. For example, the `getSmallIcon` method returns a reference to the Icon used in the SchematicIcon. The interface also defines methods for obtaining information for use in objects other than the SchematicIcon. The `getHelpUrl` method is used by the Onyx help system to obtain the Uniform Resource Locator (URL) pointing to the location of the EngElement's help file.

Downloaded from http://asmedigitalcollection.asme.org/GT/proceedings-pdf/GT1999/78583/V001T01A010/4215578/v001t01a010-99-gt-347.pdf by guest on 02 October 2022

The getEngElementDescriptor and getPortDescriptors methods in the EngElementInfo interface are particularly important. The getEngElementDescriptor method returns an EngElementDescriptor object which encapsulates information about the EngElement. This information includes the EngElement's Class name, its display name, a short description of its function, and the Class name of the Customizer used to customize its Domain Model. The method getPortDescriptor returns information concerning the number, type, display name and location of the VPorts to be included in the SchematicIcon. This information reflects the number, type and names of the Port objects in the EngElement.

The EngElementInfo class is implemented by other classes. In the figure, the XyzEngElementInfo class implements the interface and defines concrete methods to return information about the XyzEngElementInfo class. XyzEngElement returns its EngElementInfo class when its getEngElementInfo method is invoked.

The abstract SchematicIcon class' init method defines a series of steps for creating a SchematicIcon to represent an EngElement object. It obtains information about the EngElement object's UI by getting the "Info class" objects, then uses that information to create UI-specific versions of the VEngElement and VPorts, which are then added to the SchematicIcon to define its appearance.

3.3 Customizer

A customizer is a UI element which allows the simulation user to edit an EngElement's attributes. These attributes include general data, such as the EngElement's name, Class name, and description, as well as the data contained in it's Domain Model. A natural consequence of this relationship is that there is a tight coupling between the two classes: the structure of the customizer is dictated by the data structure of the EngElement. This posed a challenge in developing a customizer class structure. As a graphical component, a customizer should be defined as part of the VAF. However, attempting to create a "universal" customizer for all types of EngElements would be very difficult. It is more practical to let the developer of the EngElement create the customizer, than it is to define its structure as part of the VAF. To successfully support this approach, the design must address two key issues: 1) customizers which are created by the developer must be *pluggable* into the VAF; that is, they must be easily integrated into the framework; and 2) developers should be able to create customizers as easily as possible.

In order to be pluggable into the VAF as a customizer, a class must implement the Customizer interface (see Fig. 3). This Java interface defines two methods. The first method, setTarget, is invoked after Customizer construction to identify the EngElement object that it is being customized. The commitChanges method is called by the VAF when the user accepts any changes made in the customizer's data fields. Besides implementing these methods, a customizer must be a subclass of java.awt.Component. This requirement is necessary so that it can be added to an instance of CustomizerFrame.

To maximize ease of use, the VAF allows developers to subclass the CustomizerPage class, compose it with desired UI objects, and add it to BasicTabbedCustomizer. CustomizerPage provides methods to handle common issues such as laying out components. Since BasicTabbedCustomizer adds instances of Customizer, it is also possible to add classes which inherit from java.awt.Component and implement Customizer. Figure 1 depicts an instance of BasicTabbedCustomizer, with instances of several CustomizerPage classes added. These

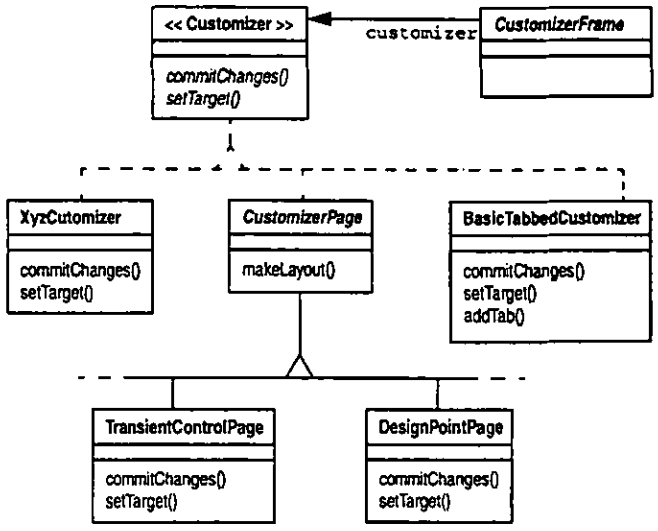

Figure 3: Customizer class structure diagram.

pages, titled *General*, *Design point data*, *Initial operating point data*, and *Transient control*, contain specific groupings of data fields and other UI widgets.

The Customizer class structure provides considerable flexibility. It allows the user to compose the UI or inherit functionality and structure when developing a customizer. By adhering to an interface, users can develop different customizers and plug them into the VAF as desired. Furthermore, a customizer is a subclasses of java.awt.Component, so users can use Java Integrated Development Environments (IDEs) to quickly construct customizers from AWT or Swing JavaBean GUI components.

3.4 Organizing Class Information

The design of the VAF emphasizes the distribution of behavior among different classes. An EngElement's behavior is delegated to its Domain Model; its UI is spread out among the EngElementInfo, PortDescriptor, EngElementDescriptor and Customizer classes. Onyx uses Java Archive (JAR) files to organize and maintain these collections of files. An EngElement class, its auxiliary classes and any data (such as the UI's Icon image), can be archived in a JAR file, and loaded into Onyx for use in both the VAF and ECF. The JAR file format is also useful for delivering EngElement classes across the Internet, as they can be compressed to reduce file size. Furthermore, JAR files can have a digital signature added to identify the author of the files contained in the JAR.

4 CONNECTION SERVICES FRAMEWORK

Higher-fidelity, multidisciplinary computational simulation of the gas turbine engine is computationally intensive. It is estimated that a single disciplinary analysis, such as an aerodynamic, unsteady, three-dimensional, viscous simulation of a complete engine, requires on the order of 10^{12} floating point operations per second, with full engine multidisciplinary analysis 2 to 3 orders of magnitude in excess of that estimate (Holst et al, 1992). While such resources are not currently available, advanced computing techniques are expected to make integrated interdisciplinary analysis practical in the near future. It is

Downloaded from http://asmedigitalcollection.asme.org/GT/proceedings-pdf/GT1999/78583/V001T01A010/4215578/v001t01a010-99-gt-347.pdf by guest on 02 October 2022

essential, therefore, that future gas turbine propulsion simulation systems be capable of accessing extensive computational resources.

Much of the recent research aimed at increasing computational performance has concentrated on parallel processing, wherein a large computational problem is decomposed into many smaller tasks. Until recently, these efforts have focused on the use of massively parallel computers, which combine thousands of processors and hundreds of gigabytes of memory in a single unit, to give enormous computational power. However, with the emergence of low-cost, high-performance microprocessor-based computers and local- and wide-area networks, there is growing movement away from highly centralized structures to decentralized distributed structures. *Distributed parallel computing*, which uses general-purpose workstations connected by a network as a large parallel computing resource, is a promising trend in parallel processing. With this technique, existing computing resources can be utilized effectively to achieve supercomputer-level power without the additional cost of a supercomputer.

The object-oriented paradigm is naturally suited towards distributed computing as object-oriented techniques encourage the decomposition of the problem space into pieces—objects—that collaborate to accomplish more complex tasks. Objects may be defined to perform a specific piece of the computation and placed on different computers; these objects then communicate with one another via messages to carry out the complete computation.

In contrast to *local* objects, which exist in a single address space, *distributed* objects exist in different address spaces. A consequence of this is that the pointers—entities used to identify the objects location in memory—in a local address space are not valid in another (remote) address space. To circumvent this problem, distributed object infrastructures—generally referred to as an *object request broker*, or *ORB*—provide an underlying mechanism to eliminate the boundary between local and remote computing. Using an ORB, an object reference is created locally and bound to a remote object. Methods may be invoked on the reference as if it were a local object. The ORB transparently intercepts these method invocations and transmits the method request and its arguments to the proxy object. The ORB marshals and unmarshals the method arguments and return values for inter-address space communication.

Distributed object infrastructures permit the developer of distributed applications to adopt a unified model of object interaction which supports location, platform, and programming language transparency. Such transparency is not without its costs, however, and over the years, the designers of some distributed object architectures have elected to forego some neutrality in exchange for perceived improvements in performance, applicability to specific tasks, and/or ease of use. As a result, there are number of distributed object infrastructures in use today. The most popular and widely used are: *RMI* (Remote Method Invocation), the Java platform's distributed object infrastructure (Wollrath et al., 1996); *Voyager*, a full-featured Java ORB architecture from ObjectSpace (Voyager, 1997); *CORBA* (Common Object Request Broker Architecture), a platform-neutral standard which provides flexible object communication and activation in distributed heterogeneous object-oriented computing environments (Vinoski, 1997); and *DCOM* (Distributed Component Object Model), Microsoft Corporation's Windows-centric distributed object protocol (Brown and Kindel, 1998).

4.1 Web-based Distribution

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" (Berners-Lee, 1996). These protocols have been extended and integrated with other new related technologies that provide for the delivery of content that is much more dynamic in nature. The most important of these related developments has been the introduction and widespread adoption of the Java programming language as a standard for Internet-based computation.

The integration of the Web and Java represents a technological advancement that enables a fundamentally new approach to modeling and simulation. Java's platform and language independence, combined with the Web's efficient and ubiquitous delivery mechanism, facilitate the widespread deployment of computational models. Standardized models, in the form of software components, can be placed on Web servers, downloaded across the Web to client machines, and loaded into simulation systems for use by engineers. Such an approach fosters model reuse, and collaboration in the design process. As a result, web technology has the potential to significantly alter the ways in which simulation models are developed (collaborative, by composition), documented (dynamically, using multimedia), analyzed (open, widespread investigation) and executed (using massive distribution) (Page et al, 1998).

4.2 Distributed and Web-based Connection Strategies in Onyx

Onyx's Connection Services Framework (CSF) provides developers with a simple and extensible mechanism for integrating both distributed object computing and web-based modeling strategies into a gas turbine engine simulation. The main objective in designing the CSF was to develop a uniform and consistent mechanism for instantiating and/or referencing EngElement. In order to meet this objective, the CSF must be capable of accommodating the various available distributed object infrastructures which might be utilized (e.g, CORBA, RMI, etc.), as well as incorporating local object instantiation and web-based object downloading. Furthermore, the CSF was designed so that future distribution strategies can be incorporated within the CSF. Figure 4 illustrates how these various strategies are implemented in Onyx. In (i), the *distributed object strategy*, EngElements are placed on remote servers and connected to Onyx by various distributed object infrastructures (e.g, RMI, CORBA, etc.). In (ii), the *web-based distribution strategy*, Java classes, in the form of byte-codes, are downloaded from a Web server to the local file system, loaded into a Java VM and instantiated for use in Onyx. And in (iii), the *local connection strategy*, EngElement class files are read from the local file system, loaded into a Java VM and instantiated for use in Onyx. This flexible approach offers several advantages:

- Ability to distribute a computationally intensive process across a number of processors;
- Ability to leverage legacy code limited to platforms offering specific programming and/or operating systems;
- Facilitation of collaborative exchange of EngElements in design process; and,

Downloaded from http://asmedigitalcollection.asme.org/GT/proceedings-pdf/GT1999/78583/V001T01A010/4215578/v001t01a010-99-gt-347.pdf by guest on 02 October 2022

- 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.).

Connections to distributed and local resources are made by the user from a graphical user interface component known as the ToolBox (see top right corner of Fig. 1). Selections made from ToolBox's Load menu bring up appropriate Dialogs to: load a local EngElement; load a Web-based EngElement; or connect to a distributed EngElement on a remote machine (top left corner of Fig. 1). Once connections are established, Onyx lists the EngElements available from the selected host. In the case of local and distributed servers, the name of the host appears on the right as tabbed panes, while the names of available EngElements appear in the list on the right. For web-based EngElements, the byte-codes are downloaded to the local file system, so the name of the EngElement appears in the list for the locally available EngElements. EngElements are instantiated only when selected by the user from the list for placement into a SchematicFrame. This technique, known as lazy initialization, is used to reduce Onyx's memory footprint.

Onyx currently supports local and web-based EngElement loading, and distributed objects using RMI and Voyager. Support for CORBA will be added in the near future.

5 EXECUTION CONTROL FRAMEWORK

Model execution is a key step in the process of performing a simulation. In most cases, it involves finding a solution to the mathematical equation(s) which define the model's behavior. The exact nature of the solution process is dependent on a number of issues, such as the complexity of the problem geometry, the nature of the defining equations, and the imposition of special boundary and initial conditions. In cases where the equations, geometry and boundary conditions are of general form, the solution can be obtained using standard numerical approaches. In complex cases, however, the solution algorithm is frequently tailored especially to handle the specific characteristics of the model.

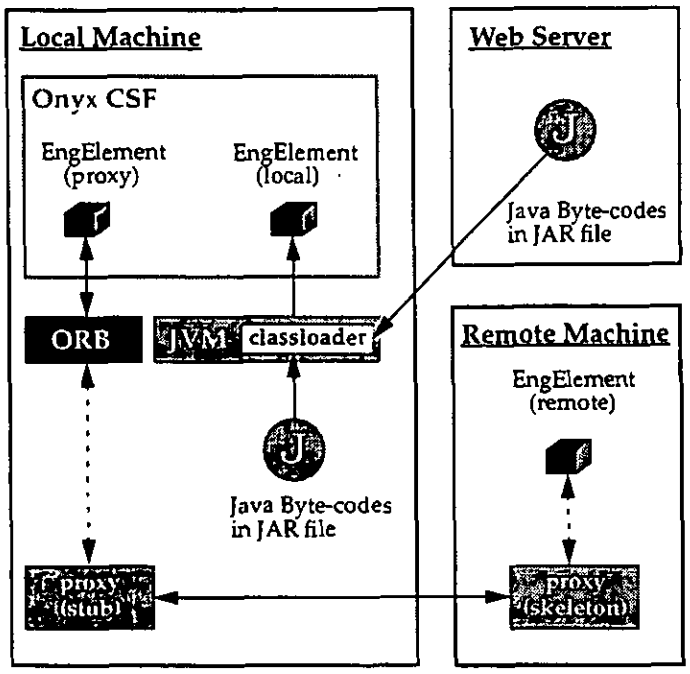

Figure 4: Overview of the services provided by CSF.

This relationship between solution algorithm and model is important in the design of Onyx. As described in Part I, Onyx's Common Engineering Model allows users to develop heterogeneous simulations based on collections of hierarchically connected models. These models can represent the system at different levels of abstraction, and consequently may have various levels of complexity. It is possible that each different model may require a specific solution algorithm, thus precluding the use of a single, generalized approach for executing the overall simulation. This situation forces us to adopt an approach which allows different solution algorithms to be integrated into the framework.

A solution algorithm primarily defines two entities: a *solver* and a *controlling algorithm*. The solver is the software equivalent of the numerical method which is used to obtain an exact or approximate solution to the equation or sets of equations. The controlling algorithm provides support for the solver, through actions such as setting boundary conditions, updating counters, streaming results to a file, etc. In most simulation software, there is no distinction made between the two; the solution algorithm *is* the solver.

In heterogeneous simulations, it is advantageous to provide separation between solver and controlling algorithm. This allows for the construction of a single controlling algorithm for the complete simulation. Such an approach can be essential to carry out complex simulations in which the objective functions of the individual solvers are competitive. It is also beneficial to define a separate controlling algorithm to act as a *coordinator* between two or more coupled models.

The XCF defines an ExecutionCoordinator class to encapsulate a simulation model's execution strategy (see Fig. 5). The analyze method outlines the controlling algorithm used to perform the simulation. Subclasses of ExecutionCoordinator are expected to redefine this method to tailor the algorithm to the specific characteristics of the model. Generally, this will involve creating Solver objects which are used to solve the model's equations. ExecutionCoordinator keeps a reference to a CompositeEngElement object, which represents the model being controlled by the ExecutionCoordinator. Thus, each assembly of EngElements represented by a CompositeEngElement can have its own ExecutionCoordinator and its own Solver. This allows nesting of Solvers and ExecutionCoordinators in a hierarchical model. A nested approach, such as this, is useful (and possibly) essential in multimodels. It allows models with different time scales to be handled by different solvers, which is important in dealing with mixed CFD and cycle models.

The XCF design defers the creation of solvers to the developer of the ExecutionCoordinator. However, to help developers identify EngElement models which declare equations to be used in a solver, the XCF defines the Solvable interface and a Solver class. Solvable is a Java interface which defines no method and no constants. It is a *marker* interface, which can be used to identify classes of objects (not only EngElements) which have equations to be solved. The design allows Onyx to identify objects which are to be handled by a solver, but lets developers decide how the data in those objects should be accessed. Class Solver is an abstract class which defines a single concrete

Downloaded from http://asmedigitalcollection.asme.org/GT/proceedings-pdf/GT1999/78583/V001T01A010/4215578/v001t01a010-99-gt-347.pdf by guest on 02 October 2022

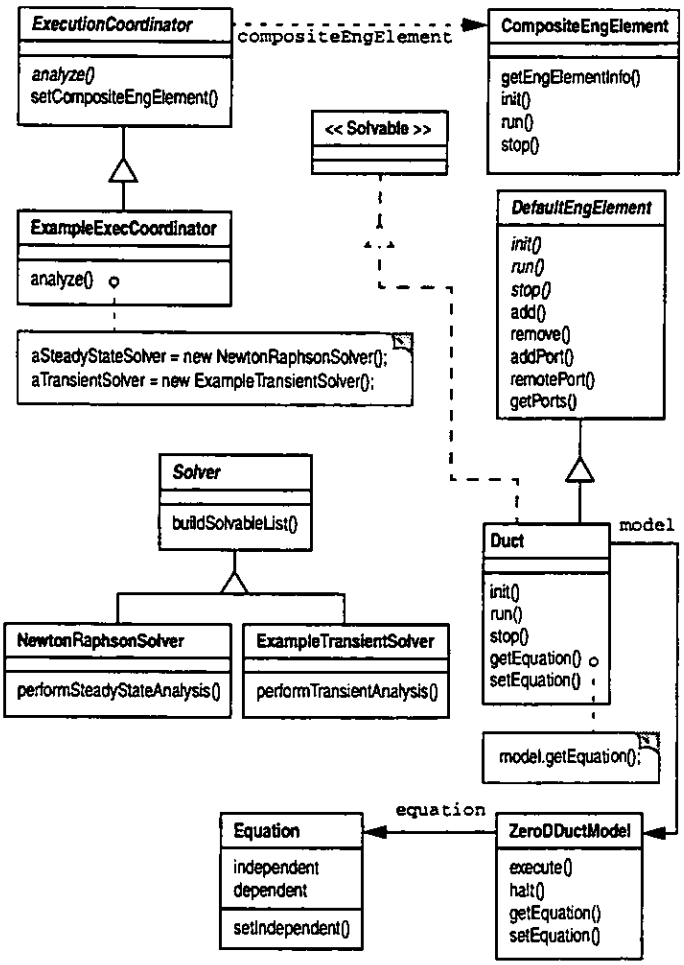

Figure 5: XCF class structure diagram.

method, `buildSolvableList`. This method returns a list of Solvable EngElements (i.e., instances of EngElement classes implementing the Solvable interface) for the CompositeEngElement it references (see Fig. 5).

6 EXAMPLE

In Part I, we developed a set of component-based models to simulate the E^3 gas turbine engine. We now utilize those component to graphically build and execute a distributed gas turbine simulation. The first step in the process is to add the "Info class" data—Customizer and EngElementInfo classes—to the component classes defined in Part I. We describe the process only for the Compressor component; the other components are similarly modified.

6.1 Compressor Customizer

A Customizer for the CompressorModel was created to allow users to interactively change the Domain Model data when the simulation is run using Onyx's Visual Assembly Framework. The CompressorCustomizer class extends BasicTabbedCustomizer (see Fig. 3):

```
public class CompressorCustomizer
    extends BasicTabbedCustomizer {
```

When the CompressorCustomizer is instantiated by the Visual Assembly Framework, its constructor calls the `buildTabbedPane` method to create four CustomizerPage objects:

```
public CompressorCustomizer()    {
    buildTabbedPane();
  }

  private void buildTabbedPane() {

    CustomizerPage page1 = new CompressorGeneralPage();
      addTab(page1, "General");
      CustomizerPage page2 = new CompressorDesignPtPage();
      addTab(page2, "Design point data");
      CustomizerPage page3 = new CompressorInitOpPtPage();
      addTab(page3, "Initial operating point data");
      CustomizerPage page4 = new CompressorTransientControlPage();
      addTab(page4, "Transient control");
  }
}
```

Four new CustomizerPage subclasses were defined: CompressorGeneralPage, CompressorDesignPtPage, CompressorInitOpPtPage, and CompressorTransientControlPage. CompressorGeneralPage has an input fields for setting the name of the Compressor object. CompressorDesignPtPage has input fields for defining design point values for mass flow rate, adiabatic efficiency, stator geometry angle and stator angle bias. The CompressorInitOpPtPage defines the initial operating point value of the stator angle. The CompressorTransientControlPage provides a TransientControlSchedule editor to define how the stator geometry angle varies during a transient. Using this CustomizerPage, the user creates *time-value* data points which are then added to the schedule. These values are displayed textually in a list on the page, and graphically by an XY plot of the value as a function of time. Linear interpolation is used to determine the stator angle value at times between the specified data points. The CompressorCustomizer is shown in the lower-left corner of Fig. 1.

6.2 CompressorInfo

As described in section 3.2, the Visual Assembly Framework composes SchematicIcons based on information defined by the EngElementInfo interface (see Fig. 2). The CompressorInfo class implements this interface:

```
public class CompressorInfo
    implements EngElementInfo {
```

The `getEngElementDescriptor` method returns an EngElementDescriptor object which encapsulates information about the Compressor. This information includes the Compressor's Class name, its display name, and the Class name of the CompressorCustomizer.

```
public EngElementDescriptor getEngElementDescriptor() {

    Class compClass = Class.forName(packageName+"Compressor");
    Class customizerClass = Class.forName(packageName
                                          +"CompressorCustomizer");
    EngElementDescriptor c = new EngElementDescriptor(compClass,
                                                      customizerClass);
```

Downloaded from http://asmedigitalcollection.asme.org/GT/proceedings-pdf/GT1999/78583/V001T01A010/4215578/v001t01a010-99-gt-347.pdf by guest on 02 October 2022

```
    c.setDisplayName("Compressor");
    return c;
  }
```

CompressorInfo's getPortDescriptor method returns information concerning the number, type, display name and location of the VPorts to be included in the SchematicIcon. As described in Part I, section 5.2, Compressor defines two OneDFluidPort objects and a single OneDMechPort object.

```
public PortDescriptor[] getPortDescriptors() {

    PortDescriptor fluidInPortDesc = new PortDescriptor(
        "Fluid In",                  // Display name
        "Fluid flow into Compressor",// short description
        PortDescriptor.IN_ONLY,      // I/O type
        PortDescriptor.WEST);        // Initial location

    PortDescriptor fluidOutPortDesc = new PortDescriptor(
        "Fluid Out", "Fluid flow out of Compressor",
        PortDescriptor.OUT_ONLY, PortDescriptor.EAST);

    PortDescriptor mechInPortDesc = new PortDescriptor(
        "Shaft In", "Shaft into Compressor",
        PortDescriptor.IN_ONLY, PortDescriptor.SOUTH);

    PortDescriptor[] ports = {fluidInPortDesc, fluidOutPortDesc,
                                                mechInPortDesc};
    return ports;
}
```

The getPortDescriptor method creates three PortDescriptor objects, each parameterized with appropriate Strings describing the, display name, short description, type, and location of the Ports. Each PortDescriptor object is added to the ports array and retumed. The PortDescriptors are extracted by Onyx to build VPorts. Note that the order in which the PortDescriptors are added to the ports array is *extremely* important; the order must correspond to the order in which the Port objects were added to the ports Vector in the Compressor class. This is a consequence of the loose coupling between the UI objects and the EngElement objects. The only coupling between the Compressor class and the UI objects is the getEngElementInfo method defined by the EngElement interface (see Part I, Fig. 3). In the Compressor class, this method appears as:

```
public EngElementInfo getEngElementInfo() {
  return (new CompressorInfo() );
}
```

The other methods defined by EngElementInfo interface are also implemented in the CompressorInfo class, but are not listed here for brevity.

6.3 The Compressor JAR File

The Compressor, CompressorModel, DataTable, OneDFluidPort, OneDMechPort, CompressorCustomizer, CompressorGeneralPage, CompressorDesignPtPage, CompressorInitOpPtPage, CompressorTransientControlPage, CompressorInfo and miscellaneous utility classes needed by Compressor (such as the TransientControllerSchedule class), were compiled using the Java compiler, *javac*, on a Silicon Graphics workstation (JDK 1.1.6). The resulting byte-code (.class) files were archived along with Icons representing the Compressor using the Java Archive (*jar*) tool following the description of section 3.4.

6.4 Setting Up The Example Simulation

Customizers and "Info classes" for each of the EngElement classes (i.e., Bleed, Combustor, Duct, etc.) described in Part I were developed in a similar manner as for the Compressor. Each EngElement, its Domain Model, Customizer, and miscellaneous classes were archived into JAR files as just described. With the exception of the Combustor JAR file, all of the JAR files were copied to the machine running Onyx, and put in Onyx's *jar* directory. Jar files located in this directory are automatically loaded into Onyx when it is started.

The Combustor JAR file was placed on the University of Toledo Mechanical, Industrial and Manufacturing Engineering Department's Web server. The Combustor JAR file was then downloaded into Onyx and instantiated during model construction to demonstrate the ability to load component models from the Web. To further demonstrate Onyx's distributed capabilities, a Bleed EngElement was distributed to a remote machine and connected to the local machine running Onyx using RMI. The EngElement class, called RemoteBleed was placed on the remote machine, a DEC Alpha 255/300, located on the University of Toledo's College of Engineering network, and compiled using DEC's Java compiler. A simple server was written to export the RemoteBleed object to the RMI Registry. A RmiDistMgr class was written to handle the connection and referencing services using the RMI protocol. Because RMI treats remote objects as types of java.rmi.Remote rather than java.lang.Object, a wrapper called RmiProxyEngElement was written to wrap the remote EngElement proxy on the local machine. This class was defined as a subclass of DefaultEngElement, and thus could be loaded into Onyx. The class maintained a reference to the remote object proxy, and delegated any requests to the proxy.

6.5 Constructing the Engine Model

Using the local, remote and Web-based EngElements, a model representing the overall turbofan engine structure was constructed in the Visual Assembly Framework (see Fig. 1). Three sub-models were defined: Core, Turbofan, and Main.

The Core sub-model, represents the high-pressure core sub-system, and includes the High-Pressure Compressor, Combustor, High-Pressure Turbine and the High-Speed Shaft. Fluid connections were defined by instances of BasicFluidVConnector and mechanical connections were defined by instances of BasicMechanicalVConnector. Connections to the Turbofan sub-model were defined by promoting the input VPort on the High-Pressure Compressor and the exit VPort on the High-Pressure Turbine. MixingVolumes were placed between non-Shaft objects to define boundary conditions and handle inter-component fluid dynamics.

The Turbofan sub-model represents the low-pressure sub-system, and includes the Fan, Core, Low-Pressure Turbine, BypassDuct, Nozzle, and Low-Speed Shaft. Again, MixingVolumes were placed between non-Shaft objects to define boundary conditions and handle inter-component fluid dynamics. Three Bleed components were used to provide cooling air from the High-Pressure Compressor to the High-Pressure Turbine, Low-Pressure Turbine, and customer (over-board). The Bleed flow is extracted from MixingVolume 3, located downstream of the High-Pressure Compressor. Connections between the Bleed components and the other components were defined by promoting the output Port of each Bleed. These promoted Ports appear on the boundary of the Core SchematicIcon in the Turbofan schematic.

Downloaded from http://asmedigitalcollection.asme.org/GT/proceedings-pdf/GT1999/78583/V001T01A010/4215578/v001t01a010-99-gt-347.pdf by guest on 02 October 2022

In the Main SchematicFrame, the complete engine model is represented by the Turbofan SchematicIcon which is connected at its inlet and exit to the Environment. This represents the placement of the engine in the environment, or in a test-cell.

6.6 Controlling the Simulation

An ExecutionCoordinator subclass, ExampleExecCoordinator (see Fig. 5), was created to define the controlling algorithm for executing the example simulation. For simplicity, only a single ExecutionCoordinator is used in the simulation, and it was (by default) associated with the top-level CompositeEngElement (which in this case is called Main). The coordinator defines steps for initializing the EngElements, performing a steady-state analysis at the engine's initial operating point, and a transient excursion from that point. The controlling algorithm is defined in the overridden analyze method:

```
void analyze(Object obj) {

  Solver aSteadyStateSolver = new NewtonRaphsonSolver();
  Solver aTransientSolver = new ExampleTransientSolver();

  compositeEngElement.init();

  aSteadyStateSolver.performSteadyStateAnalysis(
                                        compositeEngElement, obj);

  aTransientSolver.performTransientAnalysis(
                                        compositeEngElement, obj);
}
```

For the example, two subclasses of Solver were created (see Fig. 5). The steady-state analysis was managed by an instance of NewtonRaphsonSolver, which defines the method performSteadyStateAnalysis to converge the system to steady-state using a Newton-Raphson numerical method. The transient analysis was managed by an instance of ExampleTransientSolver, which defines the method performTransientAnalysis to run the transient.

The init message is sent to the top-level compositeEngElement, compositeEngElement, which in turn calls init on each of its children EngElements (see Part I, Fig. 3). The init method is defined differently by each EngElement in the example. In most cases, it is used to determine *correction coefficients*, which are used for numerical stability in components which read data from performance maps (Reed, 1993).

6.7 Determining Steady-State Engine Balance

Before proceeding to the transient analysis, the simulation first attempts to drive the engine to balanced (steady-state) conditions at the initial operating point. This ensures that the engine model is in a consistent, physically valid operating state before beginning the transient.

The NewtonRaphsonSolver class uses a numerical algorithm based on the Newton-Raphson method (Gerald and Wheatley, 1984), to drive the system to convergence. In order to apply this method, the NewtonRaphsonSolver class must identify the EngElements in the system which define differential equations, and obtain their derivative terms. EngElements defining differential equations, which are to be used in a Solver, are identified by implementing the Solvable interface (see Fig. 5). In the example engine system, the Duct, MixingVolume and Shaft classes implement the Solvable interface. Each class implementing the Solvable interface defines getEquation and setEquation methods which are used to get/set instances of the Equation class. The Equation class, as its name implies, represent differential or algebraic equations. Instances of Equation are used to encapsulate the independent-dependent relationships of the domain model's equations. The system of equations is obtained by using the abstract Solver class' buildSolvableList method, which returns a list of Equations in the complete engine model. For the example engine system, this returns the following list of EngElements: Duct BYPASS, MixingVolume MV13, MixingVolume MV3, MixingVolume MV4, Shaft HSS, MixingVolume MV41, Shaft LSS, Duct AFTERBURNER, MixingVolume MV6, and MixingVolume MV7.

The NewtonRaphsonSolver object uses the getDependent method on each Solvable EngElement in the solvable list, and applies the Newton-Raphson algorithm to the system of equations to compute new values of the independent terms. These are placed back into their respective equations using the setIndependent method. The effects of the new independent terms on the system are evaluated by "running" the engine. The solver sends the run message to compositeEngElement and it's runSequence method is called, which in turn calls run on each of its children EngElements (see Part I, Fig. 3).

The execution order of the children EngElements is important in the example system. Calculations of certain fluid properties, such as fuel-air ratio, and enthalpy are dependent on the fluid conditions in the "upstream" component. Thus, the order is defined by the user for the entire model using the graphical sequence editor (see Fig. 1). For the example, the order is: ENVIRONMENT-->LPC-->HPC-->BYPASS-->MV13-->OV_BLEED-->HI_BLEED-->LO_BLEED-->COMBUSTOR-->MV3-->HPT-->MV4-->HPT_ROTOR-->LPT-->MV41-->LPT_ROTOR-->AFTERBURNER-->MV6-->NOZZLE-->MV7

6.8 Transient Engine Analysis

In unsteady simulations, the independent variables, $\mathbf{x}(t)$, are functions of time, and may be implicitly integrated to predict their value at a future time based on known past values. This is typically done using a family of numerical methods known as Predictor-Corrector methods (Camahan et al., 1990). In these methods, an initial estimate of the independent variable at the end of a discrete time step, Δt, is estimated using a Predictor equation. The system is evaluated using the estimated independent to calculate the dependent derivative term. The derivative is then used in a Corrector equation that computes a new estimate of the independent at the end of the time step. The predictor estimate and the corrector estimate are compared. If the difference is less than some user-defined tolerance, then the equation is converged. If not, the Corrected is applied again using the most recent estimate of the independent.

The same approach as described for the steady-state case was used for the transient, except in this case, the time value was varied and the independent calculated at the new time value. An algorithm employing the Euler Implicit Predictor-Corrector method was used to estimate new values for each independent in the set of equations.

Downloaded from http://asmedigitalcollection.asme.org/GT/proceedings-pdf/GT1999/78583/V001T01A010/4215578/v001t01a010-99-gt-347.pdf by guest on 02 October 2022

6.9 ExecutionCoordinator Customizer

An ExecutionCoordinatorCustomizer was written to provide interactive control over the simulation, and to allow users to change the Solver input data. The approach is essential the same as was described for the Compressor Customizer class. Three subclasses of CustomizerPage were created: ExampleControlPage, NewtonRaphsonSolverPage, and ImplicitEulerSolverPage. The ExecutionCoordinatorCustomizer is seen at the top center of Fig. 1. The ExampleControlPage has an input field for users to set the transient duration, and buttons to start and halt the simulation. Two progress bars are provided to give the user visual feedback about the progression of both the steady-state balancing and the transient simulation. The NewtonRaphsonSolverPage and ImplicitEulerSolverPage pages provide input fields for setting specific control values for the operation of each solver.

6.10 Running the Simulation

Once the engine model was constructed, each EngElement's design and initial operating point data were fully defined, and the Solver parameters were set, the simulation was started by pressing the "Run Simulation" button in the ExecutionCoordinatorCustomizer's *Control* page. The engine was balanced successfully at the initial operating point in three iterations using the Newton-Raphson solver, and a two second transient was then run. During the transient, fuel flow rate in the Combustor was increased linearly from 0.37 to 1.7 lbm/s. (see bottom center of Fig. 1) During that time, the Fan and HP Compressor stator angles were adjusted according to the TransientControlSchedule defined in the CompressorCustomizer to improve flow into the blades. Component parameters for each component in the simulation model can be plotted using Onyx's plotting facilities. For example, Fig. 1 (bottom right) shows a plot of the stagnation pressure (psi) for each of the MixingVolumes in the model.

7 CONCLUSION

The Onyx domain framework described in this paper provides an ensemble of framework components which, together, form an integrated framework for propulsion system simulation. Onyx utilizes advanced object-oriented technologies such as object-oriented frameworks and design patterns to produce a reusable component-based architecture which can be extended and customized to meet future application requirements.

Onyx promotes the construction of aerospace propulsion systems, such as jet gas turbine engines, in the following ways. First, it provides a Common Engine Model which (i) encapsulates the hierarchical nature of the physical engine model, (ii) is capable of housing multidisciplinary and multifidelity analysis models, and (iii) enforces component interoperability through a consistent interface between components. Second, it enables the construction and of engine models and customization of the simulation at a high level of abstraction through the use of visual representation in the Visual Assembly Framework. Third, it provides a general, yet powerful, abstraction mechanism for distributing the computational requirements of a gas turbine simulation using services such as RMI, CORBA, Voyager or DCOM. And finally, it supports the integration of user-defined execution strategies based on the solvers being used, as well as the objective of the simulation.

Acknowledgments

The work described in this paper was made possible by funding from the NASA Lewis Research Center Computing and Interdisciplinary Systems Office, and the University of Toledo. We would like to thank Greg Follen at NASA Lewis for his continued support.

References

Arnold, K. and Gosling, J., 1996, "The Java Programming Language," Addison Wesley Publishing Company, Inc., Reading, MA.

Batory, D., McAllester, D., Coglianese, L. and Tracz, W., 1995,"Domain Modeling in Engineering of Computer-Based Systems," Proc. of the 1995 International Symposium and Workshop on Systems Engineering of Computer Based Systems, Tucson, Arizona.

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

Brown, N. and Kindel, C., 1998, "DCOM; "Distributed Component Object Model Protocol," Microsoft Corporation.

Carnahan, B., Luther, H. A.and Wilkes, J. O., 1990, "Applied Numerical Methods," R.E. Krieger Pub. Co., Malabar, Fla.

Curlett, B. P., Haas, A. R., and Naylor, B. A., 1995, "Adaptive Graphical User Interface Framework for Object-Oriented System Simulations," NASA TM-106890.

Englander, R., 1997, "Developing Java Beans," O'Reilly & Associates, Inc., Sebastopol, CA.

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.

Gerald, C. F. and Wheatley, P. O., 1984, "Applied Numerical Analysis," Addison-Wesley Publishing Company, Inc.

Holst, T. L., Salas, M. D., and Claus, R. W., 1992, "The NASA Computational Aerosciences Program—Toward Teraflops Computing," AIAA Paper No. 92-0558.

Johnson R. E. and Foote, B., 1988, "Designing Reusable Classes, The Journal Of Object- Oriented Programming," Vol. 1, No. 2, pp. 22-35.

Page, E. H., Buss, A., Fishwick, P. A., Healy, K. J., Nance, R. E., and Paul, R. J., 1998, "The Modeling Methodological Impacts of Web-Based Simulation," Proceedings of the 1998 SCS International Conference on Web-Based Modeling and Simulation, pp. 123-128.

Prieto-Díaz, R., 1990, "Domain Analysis: An Introduction," ACM SIGSoft Software Engineering Notes Vol. 15, 2, pp. 47-54.

Reed, J. A., 1993, "Development of an Interactive Graphical Aircraft Propulsion System Simulator," MS Thesis, The University of Toledo, Toledo, OH.

Schmidt, D. C., 1995, "Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software," Communications of the ACM, Vol. 38, 10, pp. 65-74.

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.

Vinoski, S, 1997, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," IEEE Communications, Vol. 14, No. 2.

Voyager, 1997, "Voyager: The Agent ORB for Java," http://www.objectspace.com/.

Weiner, S. R. and Asbury, S., 1998, "Programming with JFC," Wiley Computing Publishing, New York, NY

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

Downloaded from http://asmedigitalcollection.asme.org/GT/proceedings-pdf/GT1999/78583/V001T01A010/4215578/v001t01a010-99-gt-347.pdf by guest on 02 October 2022