IN-07
53547
P. 34

NASA Technical Memorandum 106970

Object-Oriented Approach for Gas Turbine Engine Simulation

Brian P. Curlett and James L. Felder
Lewis Research Center
Cleveland, Ohio

July 1995

National Aeronautics and
Space Administration

Object-Oriented Approach for Gas Turbine Engine Simulation

Brian P. Curlett and James L. Felder
National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio

Summary

An object-oriented gas turbine engine simulation program was developed. This program is a prototype for a more complete, commercial grade engine performance program now being proposed as part of the Numerical Propulsion System Simulator (NPSS). This report discusses architectural issues of this complex software system and the lessons learned from developing the prototype code. The prototype code is a fully functional, general purpose engine simulation program, however, only the component models necessary to model a transient compressor test rig have been written. The production system will be capable of steady state and transient modeling of almost any turbine engine configuration.

Chief among the architectural considerations for this code was the framework in which the various software modules will interact. These modules include the equation solver, simulation code, data model, event handler, and user interface. Also documented in this report is the component based design of the simulation module and the inter-component communication paradigm. Object class hierarchies for some of the code modules are given.

1.0 Introduction

The aerothermodynamic simulation of the engine plays a pivotal role in the entire life cycle of turbine engines. It is the first tool engineers use to evaluate new engine concepts. It is very heavily used during development, testing and certification. As a generator of customer decks, it calculates the performance levels guaranteed to the customer on which a company stakes its reputation and capital. Finally, it is used to provide diagnostics for engines in service.

The engine simulation program calculates not only engine performance, it also is the nexus for most multidisiplinary studies. It provides: 1) key dimensions for flowpath design, 2) temperatures, pressures and rotational speeds needed for structural analysis, 3) thrust levels and fuel consumptions for mission analysis, and 4) transient information needed to build and test control systems. In fact, the aerothermodynamic simulation provides critical information for every type of study from cost, to environmental impact, to reliability.

The engine simulation computer code is so important to every aspect of the aircraft engine design, development, and maintenance, it is imperative that the US aircraft engine manufactures have the best simulation tool possible to help maintain their leadership in the world market place. To this end, NASA Lewis has started an effort to develop a new engine simulation program, called

STEPP. This new software will support the aerothermodynamic system simulation process for the full life cycle of an engine. It will be designed to do system simulation from conceptual design through development and testing as well as support engines in the field. To provide this capability, STEPP will be designed to perform steady state and transient performance prediction, test data reduction, cycle optimization, customer deck generation, detailed parasitic flow analysis, and many other tasks. New standards for interfacing with other disciplines will also be established. This code will have the capability to "zoom" to models of greater fidelity and it will "couple" more directly to modeling tools for other disciplines. Furthermore, this new program will take advantage of modern computational techniques and programming languages. It will have an easy to use graphical user interface and distributed processing capabilities for fast turnaround times.

A software project of this size requires a well thought out plan, lots of input from end users, and proven methodologies to succeed. Much like what is done in the development of a new aircraft or engine, we determined that a functioning prototype was needed to insure a full understanding of the problem and to reduce the technology risks in the development of STEPP. This report covers the prototyping efforts that helped uncover many of the potential pitfalls of this project and the solutions applied to overcome these problems. This prototyping effort was extremely valuable in terms of gathering requirements, testing ideas, and scoping out the size of the final software development task.

The prototype system consists of four main modules and some utility classes. The main modules are the simulation (or analysis), the data model, the user interface, and the solver. The latter three are documented in references 1, 2, and 3, respectively. This report covers the simulation module and the framework it uses in cooperation with the other modules. A synopsis of the data model, the solver module, and some of the engineering related utility classes is included in this report.

The simulation code completed thus far is functional and general purpose. A transient compressor test rig model exercises most of the codes capabilities and is adequate for testing the system architecture. For these reasons only the code for the components necessary to model the compressor rig were written. The code architecture is complete enough to model other engine configurations if other component models are added. The components written include: inlet, duct, compressor, nozzle, shaft, load, and motor.

This report first gives background information on some related engine performance modeling activities. Next the system requirements and design for the prototype simulation code are presented. Then the framework, data model, and class structure are discussed. A method for inter-component communications is then presented along with the rational for this selection. Next the report elaborates on the components and other engineering modules. Finally, the compile/link events during the model building process are discussed. We assume the reader has a basic understanding of turbine engine simulation (ref. 4), object-oriented programming terminology and the C++ programming language (ref. 5).

2.0 Background

This research was conducted as part of a larger effort that involves the complete multidisiplinary analysis of airbreathing propulsion systems. This larger effort is call the Numerical Propulsion System Simulator (NPSS) (ref. 6). The NPSS project has been involved with overall engine cycle performance modeling from its inception. This section reports some of the previous efforts to do cycle simulation as part of the NPSS project. This information is important in understanding our decision to develop an all new, object-orient engine simulation program.

There have been both in-house and contracted study efforts for engine cycle simulation as part of NPSS. Most of the early focus has centered around the transient simulation code DIGTEM and using it to demonstrate different analytical frameworks for zooming. Zooming is the process of going from a low fidelity model (one-dimensional analysis) to a high fidelity model (two or three-dimensional analysis) and back again transparently. Zooming is considered one of the key new technologies that is part of NPSS. An example of zooming is an engine cycle simulation program invoking a streamline code for calculating compressor performance instead of using a table lookup. Current engine cycle programs with FORTRAN common block architectures would make this type of code addition difficult or impossible. Object-oriented architectures offer the greatest chance to successfully implement zooming.

Under contract no. NAS3-25951, General Electric Aircraft Engines developed an object-oriented framework in Lisp with component models based on the DIGTEM code (ref. 7). Although this effort was not focused on developing production software, we did learn much about how to (and how not to) develop object-oriented system simulations. One major conclusion from this work is that it is often better to write new code from scratch than to try to re-engineer an old FORTRAN code, like DIGTEM, and force it into an object-oriented framework. This is particularly true when the FORTRAN subroutines make extensive use of global common blocks. If the FORTRAN code had been written more modular (without common blocks) it could have been more easily called from an object-oriented wrapper, however, this is not the case with most code currently in use.

Another object-oriented implementation of DIGTEM was done by John Reed, et. al. in the AVS framework (ref. 8). This code was delivered to industry as NPSS Mod0. The AVS framework was useful as an experimental tool for distributed object-orient programs, however, industries feedback was that this is not a suitable environment for a production system. The two limiting factors were the high licence fees for the commercial software and the inability to modify the AVS framework.

The NASA Engine Performance Program (NEPP, a.k.a. NNEP89) was evaluated as the NPSS one-dimensional steady state code, referred to as a level 1 code in NPSS nomenclature (ref. 9). Although NEPP does a good job of the aerothermodynamic simulation of an engine cycle, it lacks some of the major capabilities that industry needs in a full featured simulation system. The shortcomings are; (1) no transient modeling, (2) no data reduction capability, (3) no multipoint design capability (not to be confused with multimode capability in NEPP), (4) an inflexible structure that limits its expandability, and (5) an unsuitable framework to support zooming.

It would be possible to modify the program structure of NEPP such that it could evolve into a more full featured system. However, it has been our experience that this would be a more difficult task, and would produce less desirable results than to write a new program from scratch. There are several reasons for this. One is that a large percentage of the NEPP program deals with moving data into and out of common blocks. Changing this common block structure would result in an almost complete reworking of the code. Secondly, a clean start allows us to implement the program in an object-oriented language. Lastly, doing a complete rewrite allows us to apply more rigorous software quality control procedures, hence reducing future maintenance efforts.

The same concerns about the NEPP code apply to other engine performance programs in the public domain. Thus there is no adequate engine performance prediction program that is suitable to the needs of NASA and industry in the public domain. Higher level modeling needs, modern computer technology, and new uses for system simulations all lead to the decision to develop a new code.

3.0 Requirements

This section describes some of the high level requirements imposed of the prototyping effort. These requirements come from the project office, our industry partners, and/or are self-imposed by the development team based on our own experiences from developing similar simulation programs. These requirements will most likely carry forward to the production effort but are still subject to approval by our industry partners.

The basic requirement of this code was to perform a one-dimensional steady state and transient performance prediction of a gas turbine engine. The focus of this effort was to prototype a possible object-oriented framework and not produce an end product, the component and solver algorithms were taken from available sources and not re-engineered (ref. 10, 11).

The most important design criterion for the simulation framework is that the class structure must have physical meaning. That is, physical components of an engine must be modeled as computational components (henceforth just components) in the simulation. This lead to the decision to adhere to a control volume approach for component models. The component will have control surfaces through which all mass and energy flux must pass. All the significant processes present in the physical component must be performed inside the computational component.

A second requirement is that there must be no restrictions on the types or numbers of components in the simulation. At least not in the highest levels of the class structure. The structure needs to be extendable for new component types and other disciplines without requiring modification to the basic architecture. The effect of this criterion leads to the selection of C++. The object-oriented nature of this language allowed us to construct an open framework from which new component types and variations can be derived. The dynamic memory capability allows any number of components and any given type to appear in the simulation.

Third, this software must facilitate zooming. That is, it must be able to dynamically swap out simple one-dimensional, empirical computational models for higher ordered (two or three-dimensional) computational models. This adds the requirement to support distributed processing because it cannot be assumed that the higher ordered component models will run on the same computer platform as the rest of the system. Separate hardware can be required for computation efficiency of the high ordered models.

Fourth, the code must be able to run in both graphical and batch modes. For batch mode operation an easy to edit file format was required for program inputs. This requirement a human readable storage format eliminated the consideration of the use of persistent object storage.

Fifth, we were enjoined to not base the code on any commercial software. As a result of this we developed some general utility type routines, such as list management. The time spent developing these low level classes detracted from that available to creating higher level engineering classes unique to this problem domain. We did make use of some third party software for distributed processing and for the graphical user interface. In these cases we required the design to minimize the impact of future changes to other software packages.

Sixth, this code must be structured so as to fully support system analysis needs for the entire life cycle of the engine. This includes conceptual design, preliminary design, development and testing, certification, derivative or growth analysis, and field support. Full life cycle support also mandated that this code be able to perform transient analysis.

4.0 Design

The system design meets the requirements outlined in section 3.0 as well as all the detailed requirements intrinsic to any engine simulation program. The design is composed of several layers. At the top is the system framework. The framework specifies how the major modules of the program (simulation code, user interface, solver, and data model) work together. The system framework is the topic of section 5.0. An effort has been made to make each of these major modules of the framework as independent as possible. Independent, although somewhat parallel, class structures were designed for each of these major modules.

The design of the simulation module is based on component models and their interconnections. Our design for the simulation module has three basic concepts; components, ports, and connectors. The idea is that any system can be composed of interconnected components. Each component includes a set of ports to which other components can be connected. The ports are of standard types and contain standard information of interest to other components; for example, a port may be an object representing the flux of gas out of a turbine. There are no limits on the number or types of ports a component can have.

An assembly of components is made by using connectors to "wire" together ports of different components. The connector carries data directly between the ports on the connected components. This greatly reduces the global name space. It also eliminates routines otherwise required to manage global data. These connectors are not hard coded into the components. Rather the connections between components are specified in the engine model and are constructed from an input file or at run time with the graphical user interface.

Having assemblies and subassemblies of components is vary important to our design. Making an assembly a type of component allows the engine model to be organized into nested subassemblies and subsystems. The depth of the hierarchy is arbitrary. A subassembly is treated as any other component with input and output ports. Assemblies are explained more in section 7.0.

This design of components, ports, and connectors is totally free of global data and is infinitely extensible (within the available hardware resources). This design separates the data communication and engineering models as much as possible. Inter-component communication is discussed more in section 8.0.

5.0 System Framework

The system framework is based on what we call the Model-Data-View architecture. That is, the Model, or simulation, portion of the code is separated from the View, or graphical, portion of the code by a data model. In this way the data used by both the user interface and the simulation portions of the code only needs to be described once. The user interface is adaptive. If the simulation adds data to or modifies a datum in the data model the user interface will automatically adapt to these changes and update the display. Figure 1 is a high level view of this architecture. Notice that the simulation and user interface have no direct communication. The flow of information is through the data model and the event handler. The data model, user interface, solver, and event handler are documented in references 1, 2, and 3, respectively. The event handler portion of this architecture is documentation in a internal memo [*Visual Environment for Distributed Program Chaining*, Curlett, B.P.]. This report focuses on the simulation module and how it works with these other modules.

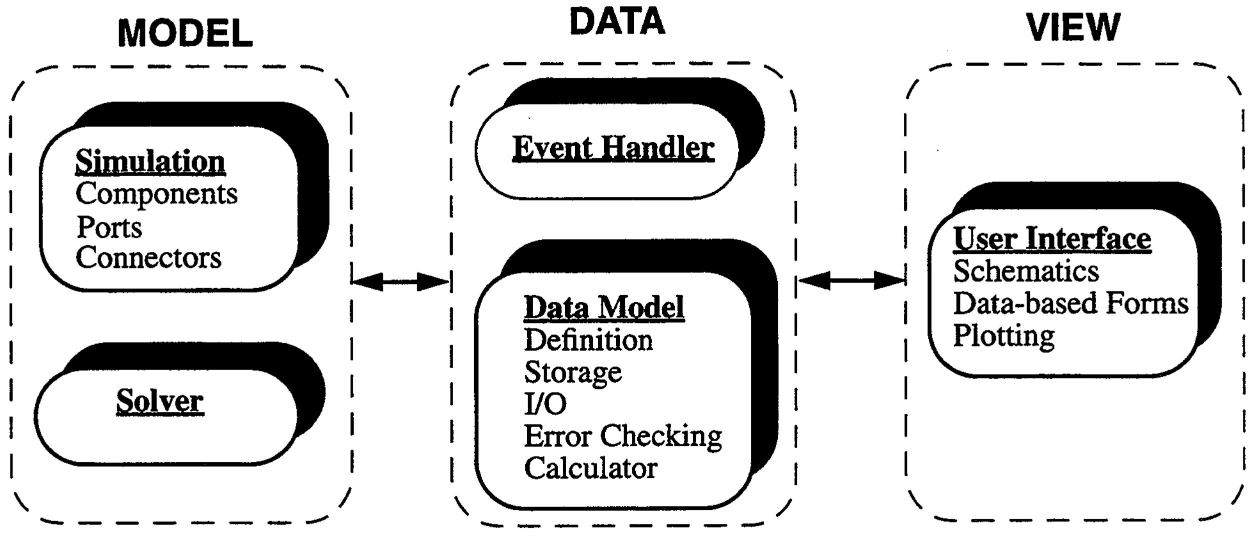

FIGURE 1. System Framework: The Model-Data-View Architecture

When a component, port, or connector is created the framework first constructs a simulation object and, if applicable, a GUI object. The simulation object does the "engineering" work and is responsible for loading its data model. The same data model is shared by the simulation and GUI objects. Almost all communication between the simulation and the GUI is handled through the data model and event handler. In this way the simulation code can be developed completely independent from the user interface and with little regard for the interface itself. This is a most desirable feature for the engineers who are typically assigned the task of writing simulation models without the aid of a professional programmer. The engineer must understand the use of the data model but is not required to actually perform any graphics programming in this framework.

The solver is also programmed to communicate with component models through the data model. The data model allows the solver to have safe, easy access to all system parameters by name.

The simulation code provides at least three methods of integration with the user interface.

1. It can be loosely integrated; that is, the user interface can create a file from the data model which can be used to invoke a simulation code. This is the simplest solution and it provides the most separation of the engineering and GUI development task.
2. It can be tightly integrated; that is, the simulation code is compiled along with the GUI and shares the same data model. This solution provides the user with the quickest interaction between the GUI and simulation. This allows the user to perform rapid "what if" scenarios.
3. It can be remotely connected; that is, the simulation code communicates with the GUI through a remote connection. This gives the user several advantages; 1) multiple simulations can be run in parallel from a single user interface, 2) simulations can be executed on compute servers while the GUI runs as a local process, and 3) other codes can be integrated with the cycle simulation, i.e. program chaining and program zooming can be preformed using this method.

Support for all three of these methods is built into the simulation, user interface and data model modules. The best method for integrating the user interface with the simulation code will depend on how the simulation is built (see sec. 12.0). The prototype simulation code is loosely integrated with the prototype graphical user interface. The architectures of both of these pieces permit tighter integration. The production versions of these codes will be more tightly integrated using methods 2 and/or 3 while still providing for loose integration.

6.0 Data Model

The internal data model is responsible for the description of the data, organization of the data, and all data input/output (I/O). It is also the mechanism used to communicate information among the major modules in the framework. This section explains some of the basic concepts of the data model which is necessary to the understanding of remained of this report.

This simulation framework uses a simple data model based on *variable tables*. Tables like these are sometimes called "data dictionaries". Variable tables are hierarchal in nature; similar to directories in a DOS or UNIX file system. We use one table, possibly with subtables, for each component in the simulation. Each variable in a table represents an attribute in the simulation. In addition to the value of the attribute, the variable class provides storage for other relevant information, such as: a description of the attribute, a default value, and limits.

We found these variable tables to be a good mechanism for: defining and cataloging our data, user friendly i/o, standardizing access functions, and insuring integrity of data that would otherwise have to be made global. We also added methods to the variable classes for interprocessor data transfer and an interpretive language for programming relationships between numerical variables.

There are six basic variable classes that are used for six different data types, these include: string, file, numerical, boolean, map, and enumeration data types. There are also a couple of special types. The *VarLink* class is a template class to link variable information to data stored elsewhere. The *VarTable* class holds other variables and tables and provides search routines for locating variables. All variable classes are derived from the *Var* class which stores information common to all types, such as name and label string.

In addition to cataloging data and performing data I/O, the variable tables provide an excellent mechanism for moving data. Ports (boundary objects on components) can use the variable data types listed above to build a VarTable of all their important attributes. Data can then be copied from one components port to another components ports simply by calling the VarTable copy function. This is how the send() method of the connector class is implemented. Connectors are explained more in section 8.0. Furthermore, the variable classes have built-in methods for reading and writing themselves to and from Parallel Virtual Machine (PVM) message buffers. This permits an easy extension to the connector class to send data to remote processes.

If component and port attributes could not be placed in a VarTable, or similar structure, each type of component and port would have to have its own methods for I/O, copying, remote distribution, and user interface display. Furthermore, data access methods would have to be added for the system solver to get at component and port data and calls to these methods would have to be hard coded because attributes could not be located by name at run time. At the surface the data model used here may look like a breakdown of the object-oriented paradigm. However, if you view the variable classes as a means of making "smart attributes", and the VarTable class as a means of providing standard interfaces for organizing and locating these attributes, you will see how this model is an enhancement to the object-oriented message passing paradigm.

7.0 Simulation Class Structure

This section covers the high level class structure for the simulation code. This class structure embeds some of our fundamental design decisions. However, this is not necessarily the final class structure of the production system. This is the structure of the prototype to be tested and evaluated. Other variations will be explored. Figure 2 shows the class structure of the engine simulation

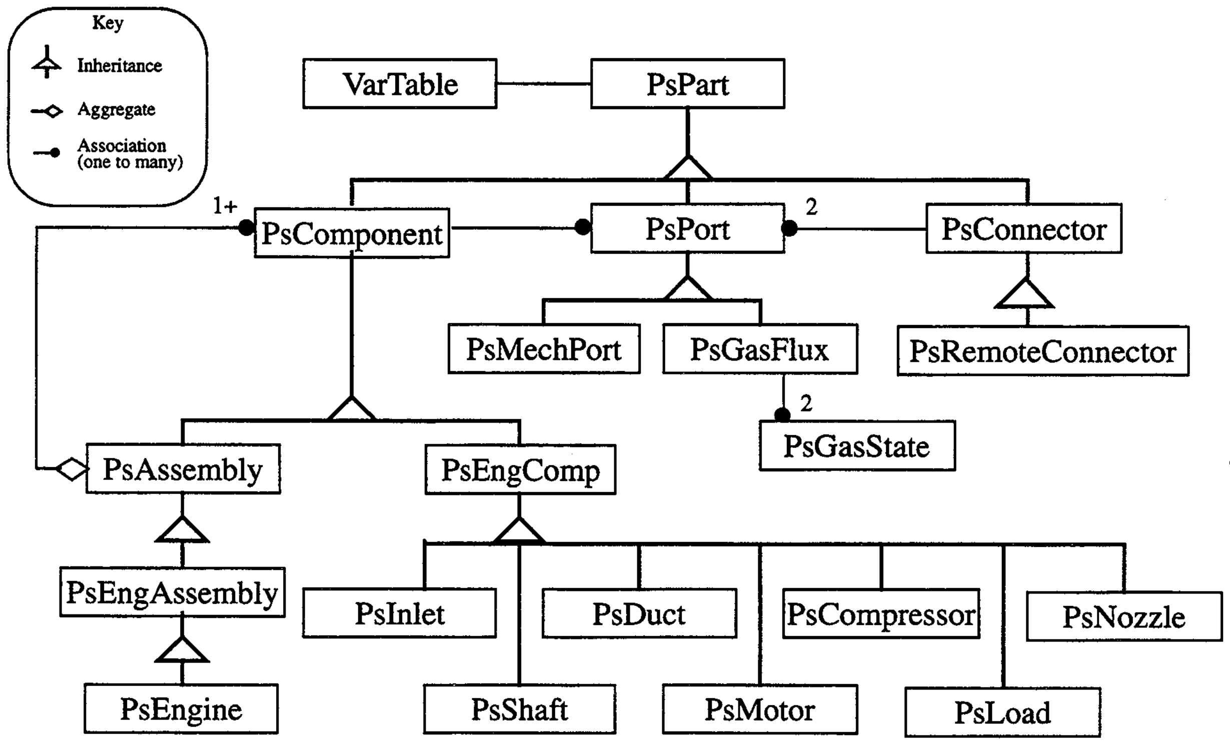

FIGURE 2. High Level Class Structure with Associations and Aggregations

module. In our design inheritance plays a more important role in achieving the correct polymorphic behavior rather than code reuse. The use of utility classes and functions for achieving code reuse is emphasized in our design (see section 11.0).

There are three major parts to building a simulation, these are: components, ports and, connectors. All three of these are derived from the PsPart class. The PsPart class does not have physical meaning but is necessary to hold common information and methods that all parts need, such as a relationship with a corresponding VarTable.

The PsComponent class usually represents a physical piece of hardware. However, no assumption of what the component does is made by this class. For example, a component could be a compressor, burner, or turbine in a jet engine simulation, or a boiler, generator, or condenser in a power plant simulation. This permits the simulation framework to be reused for other types of simulations. The PsComponent class handles things that are common to all components from any type of simulation. For example, all components must have ports to handle information flow into or out of the component. The management of these ports is handled by the PsComponent class.

The PsPort class is a base level class that provides for information flow into and out of components. Each component will have one or more ports; there is no upper limit to the number of ports on a component. Components can be thought of as control volumes and ports as the control surfaces on that volume. An example of a port is the PsGasFlux class which represents the flux of gas across a control surface. The PsGasFlux class is derived directly from the PsPort class and is discussed in more detail in section 11.2.

The PsConnector class permits two components to communicate with one another by passing information from one component's port to another component's port. The connector checks to make sure the ports are of compatible types. If the ports are incompatible an error would be reported and a component that translates from one port type to the other must be inserted by the user. The PsRemoteConnector is functionally the same as the PsConnector but it allows data to be transmitted between ports on different processors using the PVM message passing protocol.

Derived from the PsComponent class, the PsEngComp (Engine Component) class holds attributes and methods that are associated with gas turbine engine components. This intermediate class is used mostly as a way to distinguish between physical components and logical components. This can be done using the `isA()` function, which tests class type. A number of specific component classes were written. The seven class derived from the PsEngComp are the seven engine component models developed for testing the system architecture. Having working code applied to a real problem greatly helped in validating this architecture.

The PsAssembly class is a type of component that holds other components. No assumptions are made on the types of components contained in the assembly. Because a PsAssembly is a type of PsComponent, an assembly may be part of another assembly. This permits subassemblies to be made to any level.

The PsEngAssembly class is derived from the PsAssembly class. The PsEngAssembly holds attributes and methods associated with an assembly of engine components. This class also contains an association with engine cycle solver, methods for calculating the computational path through a series of engine component, and a method for executing the components in this predetermined order. If a PsEngAssembly is used to make a subassembly then this subassembly would have its own solver. This may be useful if an engine's high spool had a much different response time than its low spool. The time constants for the high spool subassembly could be different then those of the rest of the system. If a subassembly is made using the PsAssembly class then it would use the same solver as the parent PsEngAssembly. Thus the PsAssembly is only used for encapsulating some of the models complexities, it otherwise does not effect the solution of the problem.

The PsEngine class is an assembly of engine components plus the methods for calculating the overall engine performance. These attributes include: gross thrust, net thrust, SFC, etc. Multiple PsEngine objects may coexist. This would permit multimode engine modelling as well as being a useful feature if this simulation was used as a subprogram for an aircraft synthesis code modelling a multi-engine aircraft.

Several other class structures were considered. One idea was to derive PsAssembly from PsEngComp, therefore, an assembly of engine components could be an engine component. The problem with this is we could not have an assembly of non-engine components. Another idea was to inherit PsEngAssembly from both PsAssembly and PsEngComp. This would be ideal but we would not be able to cast down from PsComponent because it would have to be made a virtual base class.

8.0 Inter-component Communications

One of the major design decisions for the simulation architecture of STEPP was to determine what constitutes a component and how components are connected. A component is defined as a code module (or object) that takes inputs and produces outputs and has little other interaction with the system. A component is an object that usually represents a physical piece of hardware such as a compressor, burner, or nozzle. A component has or is associated with boundaries.

All the communications concepts evaluated share the basic idea that components transmit information at their physical and/or logical boundaries by providing a port to which a port of another component may be connected. The connection process could be the sharing of the same port (boundary data) or the transmission of data between two different logical ports that represent the same physical boundary. As a matter of record the different connection concepts evaluated are presented in Appendix A with pros and cons. The two port/connector model was chosen for the initial version of STEPP because the new architecture must demonstrate zooming and distributed processing and this concept lent well to these goals.

The two port/connector model assumes that there are two copies of the "station" data, one in each component object. Each component is responsible for creating and destroying its own copy of the station data. The data is moved (copied) from one component to the next by a PsConnector object. In this model the connector class has no built-in logic other than that to transfer data. The connector does not calculate any flow error or preform any other type of data transformation.

Figure 3 shows a two port/connector model. The large boxes in the figure represent components, the smaller boxes contained in these are ports, and the arrows are connectors. Three types of ports are shown in the figure, they are: flow-in (fi), flow-out (fo), and mechanical (m).

The connector class (PsConnector) works in a push or pull mode. That is, the compressor component C1 can tell the connector to send its flow-out (fo) data to the burner component B1, or B1 can

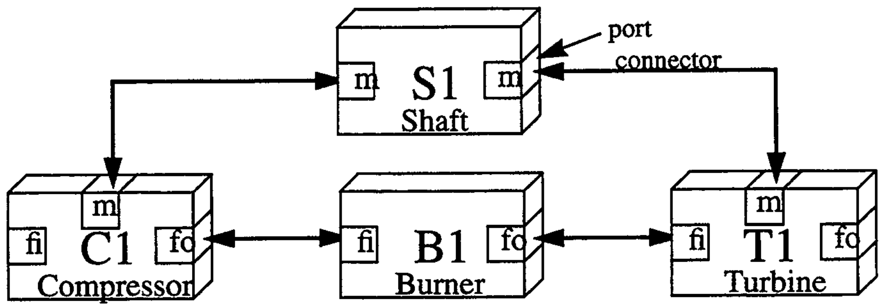

FIGURE 3. Sample Two Port/Connector Schematic

request the connector to send the upstream data when it needs it. Data can flow in the other direction as well. Our cycle program is implemented mostly as a pull system.

Because each component has its own copy of the boundary data, the two port model can be easily extended for distributed processing. The PsConnector is simply replaced by a PsRemoteConnector. The system behavior is unchanged. The problem with using a single copy of the port's data between two components is that there is never a call to the connector to transmit data and hence the extension of this method for distributed processing would require considerable changes to the component's algorithms.

The connector moves data from one port to the other simply by invoking the copy method on the port's VarTable. This copies all the variables in one port's table to the other's. Note that both ports need not contain exactly the same data, because the VarTable copy command only copies data between variables of the same name. This is why the connector class does not enforce a strict port type match between ports in the connection; instead it validates the connection based on information supplied in the port's VarTable.

Although the normal send and receive operations on the connector copy all similar data, it is sometimes desirable to send or receive just a single value. Recall port data is stored in or linked to a VarTable class. Because any variable in the VarTable may be accessed by name, the send and receive methods on the connector where modified to take this name as an argument and pass just this one parameter. Without this feature it would not be possible to implement ports which require simultaneous, bidirectional transmission of data because of data overwrites.

One advantage of this system is that components are more self contained units and the connection of components is simply a system level problem that has nothing to do with the engineering problem. Because the engineering and system problems are decoupled the two port/connector model will result in the most extendable architecture.

Some disadvantages of this architecture have been identified. It requires twice the storage for the port data as the single port model, and it will slow execution speed over the single station model. However, it was felt on today's hardware these concerns are minimal and out weighted by the benefits this model provides.

9.0 Component Models

We can now discuss what is inside of a component model. The answer is basically what ever you want. This framework assumes very little about the internal structure of components. A component may be as simple as multiplying a number on an input port by two and writing it to an output port; or it may be as complex as a 3D Navier-Stokes code with ports containing large grids of data.

We mandated that components derived from PsEngComp (Engine Component) class must follow a control volume approach to encapsulating the physical process being model. However, our class structure does not enforce this as strictly as some of the other proposed class structures. Our rationale for this design is similar to the rational for the design of the C and C++ programming languages; which is to provide flexibility and trust the programmer to use it wisely.

We developed components to model the aerothermodynamics of gas turbine engine components. They are specifically designed to function in the framework and to take advantage of the engineering utilities. Only enough types of components to model a compressor test rig were developed for the prototype code. These include an inlet, compressor, duct, nozzle, shaft, load, and motor. The engineering specification for the compressor component is included in appendix B; the corresponding source code is included in appendix C as an example.

All of our components are derived from the base class PsEngComp, which is derived from PsComponent, which in turn is derived from PsPart. The PsPart class will construct a VarTable with the name given to the component. The constructor of the component must load its data definition file and make VarLinks for any internal data it wishes to share with the rest of the system. The data definition file contains information to be loaded into the data model, such as: labels, default values, and limits. The constructor will usually lookup the addresses of any variables created when the data definition file was loaded. The component's destructor must delete any ports created by the constructor. Any connectors to these ports will automatically be deleted. The base class PsPart will clean up the variable tables.

Other than the constructor and destructor every component must have a *run()* method. This method is called by the system solver to execute the component. The run method should initialize any data that could not be initialized during the construction process. It will then request the incoming ports to update themselves, perform any calculations the component must do, and write the results to the outgoing ports. Data in the variable table not implemented as VarLinks, must be updated before exiting the *run()* method.

Components with complex calculations in the *run()* method are broken down into smaller methods. This is done to aid understanding of the algorithm and to make creating a new class by deriving off an existing class easier. This is especially true if the difference between the different classes is entirely contained within one of these smaller methods. For example, suppose a new compressor component was needed with a different surge margin calculation. The class PsNewCompressor could be derived from the PsCompressor class (see Appendix C). Only the *SurgeMargin()* method would need to be re-implemented in PsNewCompressor; everything else would be inherited from PsCompressor.

10.0 Solver

The system solver is a cooperative set of several classes. The class structure is shown in Figure 4. These consist of an equation solver class (PsSolver) that uses a Newton-Raphson matrix method to solve a set of n linear partial differential equations with n unknowns. The coefficients of the equations are gathered into a square matrix referred to as the Jacobian matrix. In this solver, the coefficients are approximated by measuring the change in residual error terms to a small change in a given independent parameter. The methods to generate a jacobian matrix and those required to use that matrix to drive a system to solution are quite independent. Therefore, all functions related to generating and using the Jacobian matrix are encapsulated in a separate class (PsJacobian). The PsSolver class contains only those methods required to iterate a system to solution using a known Jacobian matrix. If the PsSolver determines that a new Jacobian is required, it then simply "asks" the Jacobian matrix to regenerate itself. When the generate method on the PsJacobian object returns, a new conditioned and inverted Jacobian matrix is available to start predicting changes to the independents required to solve the system.

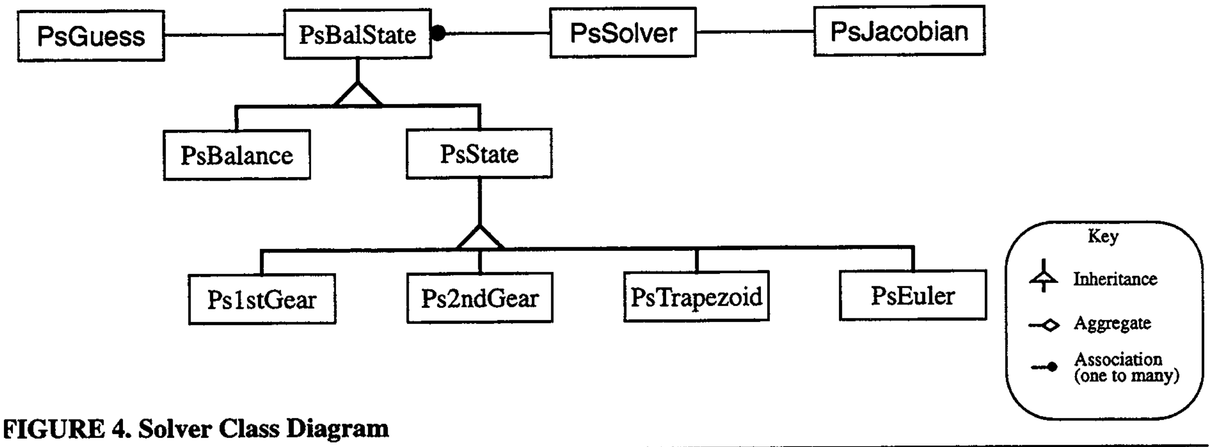

FIGURE 4. Solver Class Diagram

The PsSolver and PsJacobian classes only deal with error terms and independent parameters. As such they make up a very general equation solver. How the error terms are calculated and how new values of the independent parameters are set back into the simulation are handled by a set of classes that we will collectively refer to as equation classes. The base class for all the equation classes is PsBalState. Virtual functions defined in this class set the public interface of these objects. How these functions are implemented in the derived classes defines what is unique about each equation class. Each equation object instantiated from these classes handles one independent and calculates one error term. A system with n independent and n dependent terms will require n equation objects. A list of pointers to these objects is maintained in the PsSolver classes.

Separating how equations are solved from how the terms of the equations are calculated allows the equation solver to solve a set of equations without regard to how the error term is calculated. It could be 1) a corrector equation calculating the change in a state value over a time step, or 2) between two dependent parameters to be algebraically driven to be equal independent of time, or 3) a mix of the two. The solver algorithm and class structure is discussed in detail in reference 3.

11.0 Engineering Utilities

Engineering utilities are classes and functions that simplify the construction of higher level models. Examples of utilities are: equation solvers, table interpolation/extrapolation class, and gas properties classes. Three noteworthy utilities are presented in this section, the gas state class, the gas flux class, and the table interpolation classes.

11.1 Gas State Class

Many computer programs for aerothermodynamic simulation contain numerous lines of codes for tracking the state of a gas at different points in space and time. State information consists of chemical composition, phase, temperature, pressure, enthalpy, ratio of specific heats, gas constant, etc. The programmer often struggles with organizing this information, determining what is known and when to call the appropriate calculation routine to determine unknowns. In our code an object of type PsGasState is used to store the state information at a given location in space and time. This object tracks what information is known about the state and has methods for calculating the unknowns.

The PsGasState class is simple to use. The programmer enters the knowns about a state. When values are later needed, the programmer just calls the appropriate state method. If the value was a given it is simply returned, else the value is calculated from the knowns. An error message will result if the problem is underspecifed. This type of data encapsulation reduces errors, simplifies the programming task, and decreases maintenance efforts.

11.2 Gas Flux Class

The program will need to know more about the fluid then just its state. It will also need to know the flow rate, velocity, mach number, and stagnation conditions of the gas at each point in time and space. The PsGasFlux works much like the PsGasState in calculating these additional properties. The simplest PsGasFlux will contain two PsGasStates; one for static conditions and one for stagnation (or total) conditions. The PsGasFlux class tracks knowns much like the PsGasState class and has methods for calculating unknowns. Flow rate and area can be used to calculate Mach number, or flow rate and Mach number can be used to calculate area. The programmer can specify what parameters are to remain fixed and what parameter change as the flow rate varies. For example, at the design point the Mach number is usually given and the area calculated. The area can then be frozen for the off-design calculation and the Mach number is calculated.

Most cycle simulation programs use lumped parameters at each flow station in the engine. However, planned additions to the gas flux class will allow multiple gas states to be combined to make up a single gas flux. This will allow streamline and even more complex codes to be used in place of simple empirical component modules. Lumped parameters can still be obtained from the gas flux class by "asking" it to integrate itself. This is a great example of how object-oriented architectures can be very accommodating when designed correctly.

11.3 Table Interpolation Classes

A set of classes was developed that makes handling of component maps and other tabular data much easier. These classes decouple the data, file parser, interpolation routines, and scaling relations for flexibility and efficiency while maintaining a single, simple, consistent interface.

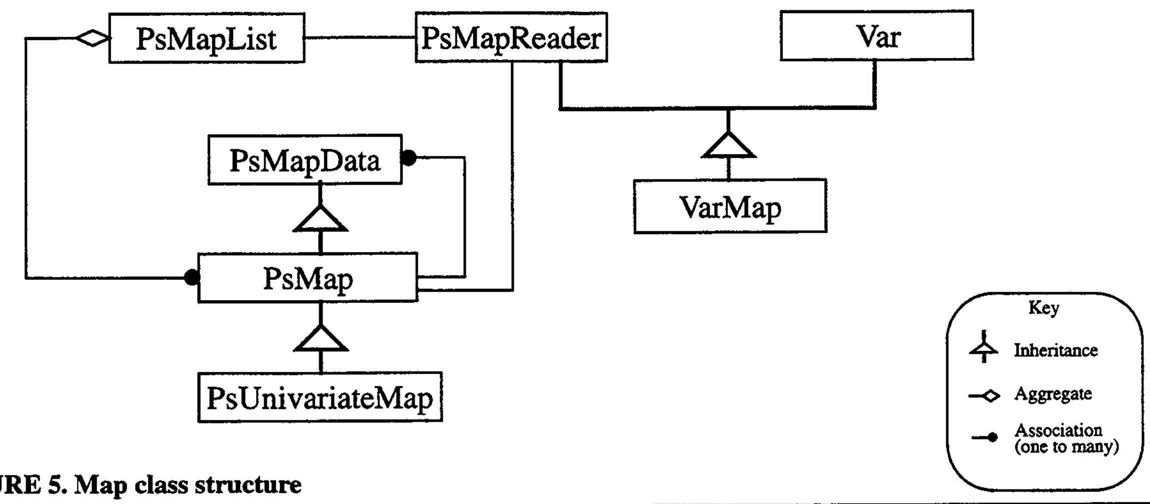

FIGURE 5. Map class structure

These classes make no assumption about the number of independent parameters in the table or about the storage format of the data. A parser for NEPP input file format is provided and is used by default. In addition, the code is capable of using user supplied interpolation routines. Extrapolation can be turned on or off.

A PsMap object is created for each table (or map). As the maps are read in from disk each PsMap that is created is stored in the PsMapList. The user does not work directly with the PsMap class. Instead the PsMapReader class provides the interface between the PsMap object and the user. This allows the same map to be used by more then one component at a time. The PsMapReader class stores an association with the PsMap and the map scaling factors for a specific component.

The VarMap class is derived from the Var class and the PsMapReader class allowing maps to be handled as any other data type in the data model as discussed in section 6.0. This simplifies the map declaration process. The file name in which the map is stored and the map identifier can be entered in the data model input file with the rest of the components data by using the VarMap class.

12.0 Compile/Link Events

Currently, there are two predominate ways in which cycle programs are produced. The first way is to build a single executable computer code that has the flexibility of modeling all engine configurations. The second method is to build an executable computer code out of standard parts and custom code for each engine configuration. There are advantages and disadvantages to both of these methods and this point can be debated at length. Table 1 summarizes some of the arguments made for each approach. The choice of method considerably influences the architecture of the program and, therefore, the project plan and development time. Hence, this will be one of the first issues addressed for the production version of STEPP.

One general executable	Configuration specific executable
more dynamic (configuration can be changed on-the-fly from user interface)	more flexible
better suited for conceptual/preliminary design (configuration changing)	better suited for final design (configuration not changing)
easier to maintain	user can add their own code
easier to setup	slightly better performance
simplifies configuration management	closer to customer deck representation
optimizer (AI) can modify configuration	smaller executables
better suited for a GUI	

Table 1: Pros and cons of build processes

Figure 6 shows the proposed source code compile/link event procedure. The boxes in the middle represent executable codes. On the left are the input sources to those executable. On the right are the models that function with those executables. The top box in the diagram is the "standard" NASA release of the program. It is recognized that each company will add site specific source code to the standard version and develop a site specific executable. It is the goal of this development effort that this executable, with its capability of running interpretive code, will be general enough to handle most all the modeling needs at the companies. However, we recognize at times there will be a need to build fast production engine specific models. As shown in the diagram, there will be a mechanism to allow this. Finally, the source, model data, and the customer deck API can be compiled into a customer deck for distribution. It is our intent to automate, as much as possible, the process of moving from the generalized NASA model to the customer model.

13.0 Concluding Remarks

The development of an industry standard simulation program of this size and complexity is a daunting task - many such projects fail. Although far from the final solution to this problem, we feel that this prototyping provided much useful information. It surfaced many issues that may have been glossed over in the requirements gathering and design phases of the commercial software system. Among these are the high level class structure, the communications paradigm, and the equation solver design. It also provided a training ground for object-oriented design and C++ programming. Hence, we are now much better prepared to effectively address the task of developing a commercial grade program.

Much was learned from this prototyping effort. A system framework was designed, tested and proved to be very effective. Although somewhat complex in implementation, the framework's design is straightforward and very extensible. Class structures for all the major modules of this program have also been designed, tested, and evaluated. The design of the solver, gas properties, and map classes presented in this report provide an excellent starting point for the final models.

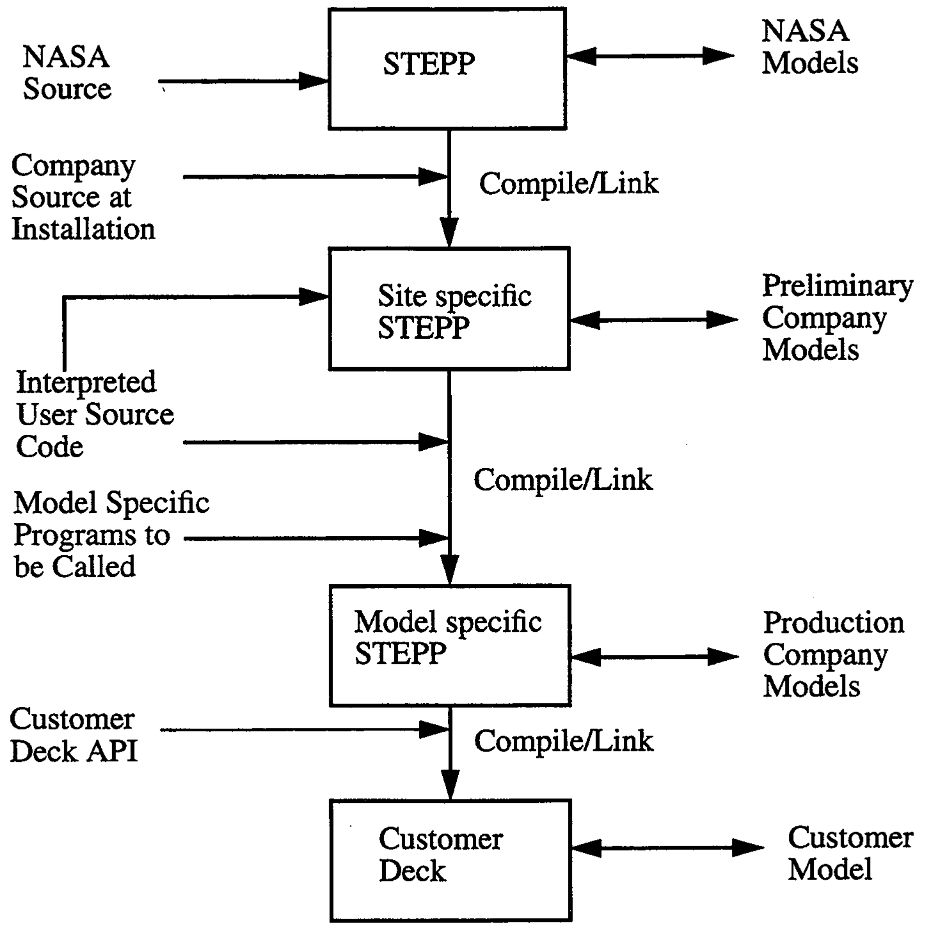

FIGURE 6. Source Code Compile/Link Event Summary

Some work will continue on this prototype. It provides a quick and dirty method for testing ideas and validating concepts. However, it is now the time to start a more formal approach to engineering this software. NASA Lewis in cooperation with the US gas turbine engine manufactures and the Department of Defense have now initiated an effort to develop an all new, commercial grade, industry standard engine performance program.

Acknowledgment

We would like to thank Jack Gould, Bret Naylor and Ken Moore for their contributions to this effort.

References

1. Curlett, B.P.; and Gould, J.: *Flexible Method for Inter-object Communication in C++*. NASA TM-106649, 1994.
2. Curlett, B.P.; Haas, A.R.; and Naylor, B.A.: *Adaptive Graphical User Interface Framework for Object-Oriented System Simulations.* NASA TM-106790, 1995.

3. Felder, J.L.; and Moore, K.: *Solver for the STEPP Simulation Program.* NASA CR to be published.

4. Oates, G.C.: *The Aerothermodynamics of Aircraft Gas Turbine Engines.* AFAPL-TR-78-52, 1978.

5. Stroustrup, B.: The C++ Programming Language. Second Edition, Addison-Wesley, Reading, Massachusetts, 1991.

6. Claus, R.W.; Evans, A.L.; and Follen, G.J.: *Multidisiplinary Propulsion Simulation Using NPSS*, AAIA 92-4709, 1992.

7. Holt, G.; and Phillips, R.: *Object-Oriented Programming in NPSS - Phase III System Documentation,* NASA Contract No. NAS3-25951, 1992.

8. Reed, J.: *Development of an Interactive Graphical Aircraft Propulsion System Simulator*, MS Thesis, University of Toledo, 1993.

9. Plencner, R.; and Snyder, C.: *The Navy/NASA Engine Program (NNEP89) - A User's Manual.* NASA TM-105186, 1991.

10.Klann, J.L.; and Snyder, C.A.: *The NEPP Programmers Manual.* NASA TM-106575, 1994.

11.Pratt & Whitney Government Engines Business: *System Design Specification for the ROCETS System - Final Report*, NASA CR-184099, 1990.

Appendix A: Data Communications Models

This appendix explains some of the alternative approaches to data communication between components.

A.1 Single Station Model

The single station model is the simplest architecture and the one used in most existing engine simulation programs. The concept is to create components and their boundary data separately. Examples of boundary data are PsGasFlux or PsMechPort classes. Boundaries here are called stations and are each uniquely named. Each component is constructed then the stations are constructed and finally the components and stations are told how they are connected. Each component stores a pointer to its stations.

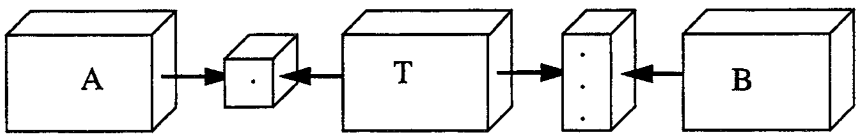

FIGURE 7. Single Station Mode

Figure 7 shows a single station model. Component A shares a single data set with component T. Component T is a data transfer function for going from 1D to 2D data, and is modelled as any other component. Component B shares the same 2D data station with component T.

It is important to note that the stations in this model contains only one set of parameters. For example, the PsGasFlux class would hold flow rate, temperature, and pressure. This class contains no storage for flow error terms. If component T expects a different weight flow rate then provided from component A, than component T would have to store a flow error term within itself for the solver to use. The reason for this decision is it is not possible to determine in general what error terms need to be saved. For example, the transient PsDuct class has a pressure discontinuity to be solved for. Anyhow, we prefer that the system not make assumptions about the error terms to store.

A remotely connected object would require one station on each computer and a PsRemoteConnector object to keep the two copies in sync. The problem with this is that there is no corresponding connector concept for local connections and hence is an awkward addition to the system.

The advantage of a single station system is that it is simple and applicable to most (not all) problems. It uses the least memory and would result in the fastest execution speed because of the shared memory.

In this approach the system (or PsEngine component) determines the type of station to be constructed not the component. This is advantageous in a simple system where the engine would know if it needed to construct a simple PsGasFlux class or a PsGasFlux with chemical equilibrium. However, the PsEngine class cannot know how to construct the station when more complex data are involved. Only the component can know how to construct these objects. Therefore, it is felt that a major limitation of this approach is that it is not as pure object-oriented as the two port/connector model, in which the component owns, creates and destroys its boundaries.

A.2 Double Data/Single Station

The double data/single station model is a combination of the double port and the single station model. This model has one named station between each component but that station holds two sets of data. One for the upstream component and one for the downstream component.

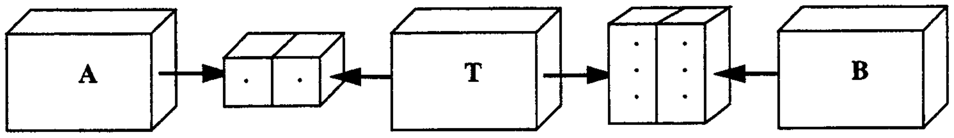

FIGURE 8. Double Data/Single Station

The advantage of this system is that error terms can be calculated in the station. For distributed processing the station can have one set of data on one machine and another set of data on another machine. The station acts as a remote connector between the two data sets.

The station in this model is not "smart", i.e. the station can not preform any type of data transformation between the two sets of data. For example, a PsGasFluxSwirl cannot be converted into a PsGasFlux. A translation component would have to be entered between the two gas flux classes in this case.

The problem with this architecture is that it is not as pure object-oriented as the two port/connector architecture because the components have no ownership over their boundaries.

A.3 Smart Connector with Double Data

This variation has a smart connector between two components. The connector is referred to as "smart" because it can connect two different types of data if it has a suitable translation function to go from one type to the other. This only differs from the single station architecture in that the transfer function from one data type to another is part of the connector class and not a separate component. Although, this can seem like a small difference it does significantly change much of the system.

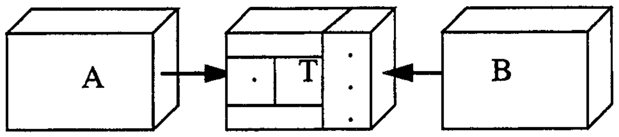

FIGURE 9. Smart Connector with Double Data

This reduces to the single station architecture if no translation function is needed. This can be a source of confusion when developing the system. The single station model does not require any type of execution call to be made to the station. However, in this model the system must know when to and when not to call the translation function in the station.

The advantage of this method is that it appears to simplify the system from the users point of view. The disadvantage of this method is that it puts engineering in the connector. The connector is no longer simply a data transfer mechanism.

A.4 Two Port with Smart Connector

In the two port/smart connector architecture the components hold their own boundary data as in the two port/connector architecture. The difference is that if a translation function from one port type to another had to occur the connector would create an instance of that translation function within itself. The connector and translation function holds no data, only pointers.

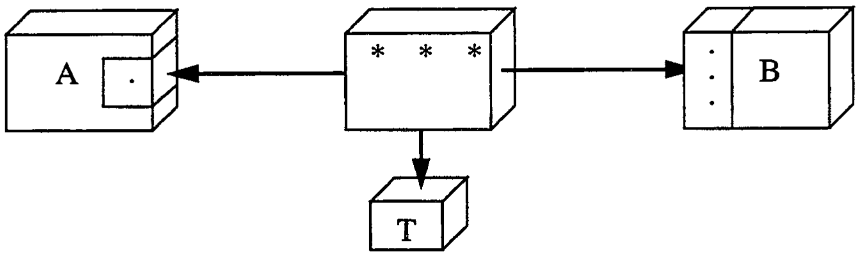

FIGURE 10. Double Port with Smart Connector

This "smart" connector would handle distributed processing itself. In this case a local copy of the data on each side of the transfer function would exist on both machines and a remote connector, as part of this connector, would make sure both copies of each data set are in sync.

The advantage of this system is the components once again hold their own boundary data. Again the concern with this architecture is putting "smarts" in the connector can complicate the system. One object is trying to do to many jobs.

A.5 Conclusion about Connectors

The team has decided the two port/connector architecture offers the best hope for an expandable system. Although this architecture can appear overly complex for a simple engine cycle model it does decouple the data handling and component models which in theory should result in a more extensible system.

Appendix B: Sample Component Engineering Specification

This appendix contains the engineering specification of the compressor component. This is provided as an example along with the corresponding source code in appendix C.

The PsCompressor class computes the thermodynamic performance of compressors. The compression process is divided into entry conditions, surge margin, work and bleed flow calculations.

B.1 Assumptions

- Gas flows are adiabatic.
- As is customary in aircraft applications, no working gas corrections are made when scaling compressor performance maps. Thus it is assumed that variations in γ from temperature change have a negligible effect on compressor performance.
- There is only one compressor bleed. Bleed work is given by the user specifying the fraction of total compressor power saved because of the interstage bleed.

B.2 Symbols

h	enthalpy
J	778.16ft-lbf/BTU
N	rotational speed
P	horse power
r	pressure ratio
R	ray line on compressor map
S	surge margin
W	mass flow rate
η	efficiency
θ	corrected temperature (T_t/T_{std})

Subscripts

corr	Corrected
1	incoming flow condition

2	exit flow condition
B	bleed flow condition
i	ideal
S	condition at surge

B.3 Method of Calculation

B.3.1 Entry Conditions

The rotational speed (N) of the compressor is determined from the connected shaft. The corrected speed is then determined by

$$N_{corr} = \frac{N}{\sqrt{\theta}}$$

At design, the corrected mass flow rate is determined from the incoming flow stream. Design values for corrected speed, pressure ratio, compressor efficiency, R and stator angle are input. These are used to set scale factors in the corresponding map objects.

At off-design the given R, corrected speed and stator angle are used to determine the compressor flow rate, pressure ratio, and efficiency

$$W_{corr} = fn\,(R, N_{corr}, \alpha)$$

$$r = fn\,(R, N_{corr}, \alpha)$$

$$\eta = fn\,(R, N_{corr}, \alpha)$$

Note that the corrected weight flow rate from the map will not match the entering corrected flow rate from the upstream component at off-design conditions. A system balance will have to be added to balance this two flow rate. For example, vary the R value of the compressor until the flow rate out of the upstream component equals the updated flow rate into the compressor.

B.3.2 Surge Margin

The surge margin is calculated from

$$S = 100\left(\frac{(W_{corr})_1 / (W_{corr})_S}{r/r_S} - 1\right)$$

The subscript S means the value at surge. This occurs when R=1.0, that is

$$r_S = fn(1.0, N_{corr}, \alpha)$$

$$(W_{corr})_S = fn(1.0, N_{corr}, \alpha)$$

B.3.3 Conservation of Mass

The compressor class allows for only one bleed flow. The amount of bleed flow is input as a ratio to total incoming flow. The exit mass flow rate is then the incoming flow rate minus the bleed flow rate.

$$W_2 = W_1 - W_B$$

B.3.4 Compressor Work

The exit pressure of the compressor is found by

$$P_{t2} = rP_{t1}$$

The exit temperature of the compressor is found by multiplying the entrance relative pressure by the pressure ratio to obtain the exit relative pressure. The exit temperature is then determined from the exit relative pressure using the thermo function in the PsGasState class.

With the exit temperature and pressure known, the ideal exit enthalpy is determined by the gas state class. The ideal work is then the difference between the exit and incoming enthalpies

$$\Delta h_i = h_{2,i} - h_1$$

The required work or enthalpy rise is then evaluated from

$$\Delta h = \Delta h_i / \eta$$

The actual exit enthalpy is

$$h_2 = h_1 + \Delta h$$

B.3.5 Bleed Flow

The fraction of work (F_B) on the bleed flow not required do to interstage bleed is input. This value is 0.0 if the bleed is at the compressor exit, and 1.0 if the bleed is at the compressor entrance. An iteration is necessary to solve for the bleed outlet total pressure which matches the value F_B. The convergence condition is

$$\left| P_{tB} / P'_{tB} - 1 \right| \leq 10^{-6}$$

When this condition is satisfied, the assigned value of F_B is equaled to four decimal places. Convergence is rapid and does not require an iteration counter. The initial guess of the bleed pressure ratio is the compressor pressure ratio itself

$$P_{tB} = rP_{t1}$$

The enthalpy of the bleed flow is then

$$h_{B,i} = \Delta h_i (1 - F_B) + h_1$$

This is used to find the relative pressure at the bleed flow exit. From this a new pressure ratio is determined and hence a new bleed exit pressure. If the convergence criterion is not satisfied, the new value of bleed total pressure replaces the former value and the iteration continues.

B.3.6 Power

The horse power required to pump bleed stream is

$$P_B = -JW_B \Delta h_B / 550$$

where J is the conversion from BTUs to ft-lbf and 550 is the conversion from ft-lbf/s to horse power.

The total horse power required for the compressor is then

$$P = -JW_3 \Delta h / 550 + P_B$$

Appendix C: Sample Component Model Code

This appendix contains an annotated piece of code for a simple compressor component as it would be written to work in the simulation framework discussed in this report. Appendix B contains the corresponding engineering specification for this code.

```
//--------------------------------------------------------------------------
//  PROJECT:NPSS, Object-Oriented Engine Simulation Framework
//
//  FILE:PsCompressor.H
//
//  CONTENTS:
//    PsCompressor is intended to demonstrate the construction of a specific
//    component using the basic building blocks of the simulation framework.
//
//  This class will be given four explicit boundaries:
//    in    - fluid boundary with upstream component
//    out   - fluid boundary with downstream component
//    bleed - fluid boundary with bleed system
//    mport - mechanical boundary with shaft
//
//  AUTHOR:Brian Curlett
//          NASA Lewis Research Center
//          Cleveland, Ohio
```

```
//
//  DATE:6/94
//
//--------------------------------------------------------------------------
#ifndef _PSCOMPRESSOR_H
#define _PSCOMPRESSOR_H

#include "PsEngComp.H"        // include base class header
#include "PsGasFlux.H"        // include port classes
#include "PsMechPort.H"

//------------------------------------------------------------------------
class PsCompressor:  public PsEngComp
{

  // ATTRIBUTES -----------------------------------------------------------
 private:
 protected:

  // mode switches
  VarBoolean *design;   // parent design flag TRUE if running a design case
  VarBoolean *redesign;// TRUE if doing a component redesign

  // maps used by compressor
  Boolean maps_init;
  VarMap *map_Wcorr;
  VarMap *map_PR;
  VarMap *map_Eff;

  // dependent parameters from maps
  double v_Wcorr;
  double v_PR;
  double v_Eff;

  // independent parameters on maps
  double v_mapR;
  double v_Ncorr;
  double v_alpha;

  // design point input parameters for scaling
  double v_PR_design;
  double v_Eff_design;
  double v_Ncorr_design;

  // bleed conditions
  double v_bleed_ratio;
  double v_bleed_work_charged;

  // other inputs
  double v_surge_margin;

  // these temporary variables are use in more then one function
  // but not input or output to/from this class
  double PR_mapdp;            // pressor ratio at map design point
  double delta_h;             // enthapy rise across the compressor
  double delta_h_ideal;       // ideal enthapy rise across the comp.
  double Win;                 // incoming flow rate before map lookup
  // output parameters
  double v_PW;                 // req. power for compressor
  double v_PWb;                // req. power to pump bleed flow
  double v_Werror;             // flow rate error


  // connections
  PsGasFlux *in;               // upstream flow port
  PsGasFlux *out;              // downstream flow port
  PsGasFlux *bleed;            // bleed flow port
  PsMechPort *mport;           // shaft port

  // METHODS -------------------------------------------------------------
 private:
  void makeLinks();

 protected:
  // Calculation Methods -------------------------------------------------
  void InitPorts();
  void InitMaps();
  void EntryConditions();
  void SurgeMargin();
  void Work();
  void BleedFlow();

 public:
  // Constructors and Destructor -----------------------------------------
  PsCompressor (const char *name, PsEngAssembly *parent,
                const char *file=NULL);
  virtual ~PsCompressor();

  // Access Methods ------------------------------------------------------
  virtual void run();

  // Class Identification Methods ----------------------------------------
  virtual PsClassType classType( ) const { return PsCompressorClass; }
  virtual int isA (PsClassType type) const {
    return (type == PsCompressorClass || PsEngComp::isA(type) );
  }
};

#endif // _PSCOMPRESSOR_H


//----------------------------------------------------------------------------
//
// PURPOSE: calculates the thermodynamic performance of compressors.
//
// Note that this file has verbose comments because it servers as an
// example for other component classes.
//
// Programmed By:
//      Brian Curlett
//      NASA Lewis Research Center
//      6/94
//----------------------------------------------------------------------------
```

```
#include <math.h>              // include the standard C math functions
#include <PsMapReader.H>       // include the map handling classes

#include "PsEngine.H"          // include the engine assembly classes
#include "PsCompressor.H"      // include the header for this class


//---------------------------------------------------------------------------
// Constructor
// This function creates a PsCompressor object given the name of the object,
// a parent engine assembly, and the name of the definition file. The base
// class for this component will construct a variable table named `tab'.
//---------------------------------------------------------------------------
PsCompressor::PsCompressor (const char *name, PsEngAssembly *parent,
                      const char *file)
  : PsEngComp(name, parent)
{
  // initialize some values
  maps_init = FALSE;
  v_Wcorr = 0.0;
  v_Werror = 0.0;
  v_PR = 0.0;
  v_Eff = 0.0;
  v_mapR = 1.4;
  v_Ncorr = 0.0;
  v_alpha = 0.0;
  v_PR_design = 0.0;
  v_Eff_design = 0.0;
  v_Ncorr_design = 0.0;
  v_bleed_ratio = 0.0;
  v_bleed_work_charged = 0.0;
  v_surge_margin = 0.0;
  delta_h = 0.0;
  delta_h_ideal = 0.0;
  v_PW = 0.0;
  v_PWb = 0.0;

  // if a definition input file name is given it is loaded
  if (file) {
    tab->load(file,FALSE,TRUE);
    // set type information in VarTable for use in checking
    // valid connection types, etc.
    tab->setTemplateName("PsCompressor");
    // this call will link in definition information from the data model
    makeLinks();
  }

  // construct the ports for this compressor
  in    = new PsGasFlux ("FI", this, "PsGasFlux.def");
  out   = new PsGasFlux ("FO", this, "PsGasFlux.def");
  bleed = new PsGasFlux ("FB", this, "PsGasFlux.def");
  mport = new PsMechPort("MP", this, "PsMechPort.def");
}

//---------------------------------------------------------------------------
// Destructor
// This method is called to do cleanup work when this compressor is
// no longer needed in this simulation.
//----------------------------------------------------------------------------
PsCompressor::~PsCompressor ()
{
  // delete all ports
  delete in;
  delete out;
  delete bleed;
  delete mport;
}

//----------------------------------------------------------------------------
// Lookup variable addresses
// Some data used by this object is located directly in the variable
// table. These data are stored in Var objects that were created when
// the definition file was read from the constructor. This method locates
// that data by name and stores a pointer to that data in this object. This
// will save us the time and trouble of having to locate this data each time
// it is used.
//-------------------------------------------------------------------------
void PsCompressor::makeLinks()
{
  // This type of access will "link" a local double precision number to
  // a variable in the variable table thus permitting other class such as the
  // solver class to access this variable by name.

  // link inputs
  in_tab->lookupDouble("alpha")->link(&v_alpha);
  in_tab->lookupDouble("PR_design")->link(&v_PR_design);
  in_tab->lookupDouble("Eff_design")->link(&v_Eff_design);
  in_tab->lookupDouble("Ncorr_design")->link(&v_Ncorr_design);
  in_tab->lookupDouble("bleed_ratio")->link(&v_bleed_ratio);
  in_tab->lookupDouble("work_charged")->link(&v_bleed_work_charged);
  in_tab->lookupDouble("mapR")->link(&v_mapR);


  // link outputs
  out_tab->lookupDouble("PW")->link(&v_PW);
  out_tab->lookupDouble("PWb")->link(&v_PWb);
  out_tab->lookupDouble("Wcorr")->link(&v_Wcorr);
  out_tab->lookupDouble("PR")->link(&v_PR);
  out_tab->lookupDouble("Eff")->link(&v_Eff);
  out_tab->lookupDouble("Ncorr")->link(&v_Ncorr);
  out_tab->lookupDouble("surge_margin")->link(&v_surge_margin);
  out_tab->lookupDouble("Werror")->link(&v_Werror, VarReadOnlyAccess);

  // This type of variable table link stores a pointer locally to data
  // data stored in the variable table.
  map_Wcorr = in_tab->lookupMap ("map_Wcorr");
  map_PR    = in_tab->lookupMap ("map_PR");
  map_Eff   = in_tab->lookupMap ("map_Eff");
  redesign = in_tab->lookupBoolean("Redesign");
  design = tab->lookupBoolean("..\\In\\DesignPoint");
}
```

```
//---------------------------------------------------------------------------
// Executes the module
// This method is called from the parent PsEngAssembly object.
//---------------------------------------------------------------------------
void PsCompressor::run()
{
  // on the first call to this method initialize the map data
  if (!maps_init)
    InitMaps();

  // ask connectors to update the incoming flow stream ports.
  InitPorts();

  // save incoming mass flow to calculate mass flow error in EntryConditions
  Win = in->W();

  // calculate the entrance flow conditions
  EntryConditions();

  // calculate the surge margin of this compressor
  SurgeMargin();

  // set exit flow rates - conservation of mass
  out->copyTotal(*in);
  bleed->copyTotal(*in);

  bleed->W( in->W() * v_bleed_ratio );
  out->W( in->W() - bleed->W() );

  // calculate the work done by the compressor
  Work();

  // adjust work requirements do to bleed
  BleedFlow();

  // calculate required horse power
  v_PW = -1.415 * out->W() * delta_h + v_PWb;

  // set power in the port connected to the shaft
  // so that the shaft may calculate delta N
  mport->PW(v_PW);
}

//-------------------------------------------------------------------------
// Setup maps - called only once to initialize maps readers and
// set the map design parameters.
//-------------------------------------------------------------------------
void PsCompressor::InitMaps()
{
  map_Wcorr->Init();
  map_Eff->Init();
  map_PR->Init();

  map_Wcorr->setMapDesVals (v_mapR, v_Ncorr_design, v_alpha);
  map_PR->setMapDesVals    (v_mapR, v_Ncorr_design, v_alpha);
  map_Eff->setMapDesVals   (v_mapR, v_Ncorr_design, v_alpha);
  PR_mapdp = map_PR->lookup(v_mapR, v_Ncorr_design, v_alpha);
  maps_init = TRUE;
}

//----------------------------------------------------------------------
// This function request connector to send data to input ports
//----------------------------------------------------------------------
void PsCompressor::InitPorts()
{
  // get input gas flux from upstream component
  in->Recv();

  // get mechanical data from shaft
  mport->Recv();
}

//----------------------------------------------------------------------
// Entry condition are calculated by the incoming gas flux object
//----------------------------------------------------------------------
void PsCompressor::EntryConditions()
{
  // get rotational speed from shaft and correct for temperature
  double N = mport->N();
  double theta = in->Tt()/Tstd;
  v_Ncorr = N/sqrt(theta);

  if ( *design==TRUE || *redesign==TRUE) {
    // Design Point Case
    // get flow rate from upstream component
    v_Wcorr = in->Wcorr();

    // set pressure ratio and efficiency to input design values
    v_PR = v_PR_design;
    v_Eff = v_Eff_design;

    // the map design values are already set in the map reader object
    // these calls set the components design values in the map reader
    // which are used to calculate scaling values
    map_Wcorr->setCompDesVals (v_mapR, v_Ncorr, v_alpha, v_Wcorr);
    map_PR->setCompDesVals    (v_mapR, v_Ncorr, v_alpha, v_PR);
    map_Eff->setCompDesVals   (v_mapR, v_Ncorr, v_alpha, v_Eff);


    // PR scales as (PR-1)/(PRmap-1), not PR/PRmap, so need to
    // override default scaling relationship
    double PRscaler = (v_PR - 1.0) / (PR_mapdp - 1.0);
    map_PR->setScaler(3,PRscaler);

    // tell fluxes to calculate area from mach number and flow rate
    // during the design phase
    in->FloatArea();
    out->FloatArea();
    bleed->FloatArea();

    v_Werror = 0.0;
  } else {
    // Off-design Point Case

    // flow rate, efficiency and pressure ratio are calculated from
    // maps given the mapR, rotational speed and stator angle.
    // note that mapR is varied by the solver to balance incoming flow
    // rate
    v_Wcorr = map_Wcorr->lookup (v_mapR, v_Ncorr, v_alpha);
    v_Eff   = map_Eff->lookup   (v_mapR, v_Ncorr, v_alpha);
    v_PR    = map_PR->lookup    (v_mapR, v_Ncorr, v_alpha);

    // tell fluxes to calculate new mach number due to change in
    // flow rate during off-design execution
    in->LockArea();
    out->LockArea();
    bleed->LockArea();

    // set new flow rate in incoming gas stream. the solver will need
    // to balance this with the flow rate on the connected port
    in->Wcorr(v_Wcorr);
    v_Werror = 2.0 * (Win - in->W()) / (Win + in->W());
  }
}


//--------------------------------------------------------------------------
// Calculate Surge Margin
// Surge line is define where mapR = 1.0
// This is not the only possible way to define surge margin
//--------------------------------------------------------------------------
void PsCompressor::SurgeMargin()
{
  double Wratio  = v_Wcorr / map_Wcorr->lookup(1.0, v_Ncorr, v_alpha);
  double PRratio = v_PR / map_PR->lookup   (1.0, v_Ncorr, v_alpha);
  v_surge_margin = 100.0 * (Wratio/PRratio-1.0);
}


//--------------------------------------------------------------------------
// Calculates the work required by this compressor
//--------------------------------------------------------------------------
void PsCompressor::Work()
{
  // set exit total pressure
  out->Pt( in->Pt() * v_PR );

  // set ideal exit temp. from relative pressure and pressure ratio
  out->Prel_t(in->Prel_t() * v_PR);

  // find ideal work
  double h2 = in->ht();
  double h3_ideal = out->ht();
  delta_h_ideal = h3_ideal - h2;

  // find work given efficiency
  delta_h = delta_h_ideal / v_Eff;
  out->ht(h2+delta_h);
}
```

```
//------------------------------------------------------------------------
// Bleed flow calculation
//------------------------------------------------------------------------
void PsCompressor::BleedFlow()
{
  if (bleed->W() <= 0.0) {
    // no bleed flow
    bleed->Tt(0.0);
    bleed->Pt(0.0);
    return;
  }

  // fraction of power saved do to interstage bleed
  double Fb = 1.0 - v_bleed_work_charged;

  if (Fb >= 1.0) {
    // bleed is at exit
    bleed->Tt( out->Tt() );
    bleed->Pt( out->Pt() );
    return;
  }

  // set enthapy rise
  double hb_ideal = delta_h_ideal * Fb + in->ht();
  bleed->ht( hb_ideal );

  double Prel2  = in->Prel_t();
  double Pt2 = in->Pt();

  // guess pressure
  double Ptb = in->Pt() * v_PR;
  bleed->Pt(Ptb);

  // find pressure of bleed stream - loop on total pressure
  double Prelb, PRb, Ptb_last, hb;
  do {
    Prelb = bleed->Prel_t();
    PRb = Prelb/Prel2;

    Ptb_last = Ptb;
    Ptb = Pt2*PRb;
    bleed->Pt(Ptb);
  } while (ABS(Ptb_last-Ptb) > Ptb*0.00001);

  // set actual enthapy of bleed stream
  double delta_hb = delta_h * Fb;
  hb = delta_hb + in->ht();
  bleed->ht( hb );

  // calculate horse power req. to pump bleed stream
  v_PWb = -1.415 * bleed->W() * delta_hb;
}
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE July 1995	3. REPORT TYPE AND DATES COVERED Technical Memorandum

4. TITLE AND SUBTITLE Object-Oriented Approach for Gas Turbine Engine Simulation	5. FUNDING NUMBERS WU–505–69–50
6. AUTHOR(S) Brian P. Curlett and James L. Felder	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Lewis Research Center Cleveland, Ohio 44135–3191	8. PERFORMING ORGANIZATION REPORT NUMBER E–9731
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, D.C. 20546–0001	10. SPONSORING/MONITORING AGENCY REPORT NUMBER NASA TM–106970

11. SUPPLEMENTARY NOTES

Responsible person, Brian P. Curlett, organization code 2420, (216) 977–7041.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified - Unlimited Subject Category 07 This publication is available from the NASA Center for Aerospace Information, (301) 621–0390.	12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

An object-oriented gas turbine engine simulation program was developed. This program is a prototype for a more complete, commercial grade engine performance program now being proposed as part of the Numerical Propulsion System Simulator (NPSS). This report discusses architectural issues of this complex software system and the lessons learned from developing the prototype code. The prototype code is a fully functional, general purpose engine simulation program, however, only the component models necessary to model a transient compressor test rig have been written. The production system will be capable of steady state and transient modeling of almost any turbine engine configuration. Chief among the architectural considerations for this code was the framework in which the various software modules will interact. These modules include the equation solver, simulation code, data model, event handler, and user interface. Also documented in this report is the component based design of the simulation module and the inter-component communication paradigm. Object class hierarchies for some of the code modules are given.

14. SUBJECT TERMS Computer code; Engine performance; Cycle analysis; Turbine engine	15. NUMBER OF PAGES 34
	16. PRICE CODE A03

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT