/N-07
22320
23 P

NASA Technical Memorandum 106723
AIAA–94–3095

Object-Oriented Technology for Compressor Simulation

C.K. Drummond and G.J. Follen
Lewis Research Center
Cleveland, Ohio

M.R. Cannon
Ohio Aerospace Institute
Brook Park, Ohio

Prepared for the
30th Joint Propulsion Conference
cosponsored by AIAA, ASME, SAE, and ASEE
Indianapolis, Indiana, June 27–29, 1994

National Aeronautics and
Space Administration

Object-Oriented Technology for Compressor Simulation

Colin K. Drummond[1] and Gregory J. Follen[2]
NASA Lewis Research Center
Cleveland, Ohio, 44135

M.R.Cannon[3]
Ohio Aerospace Institute
Brookpark, Ohio, 44142

Abstract

An object-oriented basis for interdisciplinary compressor simulation can, in principle, overcome several barriers associated the traditional structured (procedural) development approach. This paper presents the results of a research effort with the objective to explore the repercussions on design, analysis, and implemention of a compressor model in an object-oriented (OO) language, and to examine the ability of the OO system design to accommodate computational fluid dynamics (CFD) code for compressor performance prediction. Three fundamental results are that:

1. The selection of the object-oriented *language* is not the central issue; enhanced (interdisciplinary) analysis capability derives from a broader focus on object-oriented *technology*.

2. Object-oriented designs will produce more effective and reusable computer programs when the technolgy is applied to issues involving complex system inter-relationships (more so than when addressing the complex physics of an isolated discipline).

3. The concept of disposable prototypes is effective for exploratory research programs, but this requires organizations to have a commensurate long-term perspective.

This work also suggests that interdisciplinary simulation can be effectively accomplished (over several levels of fidelity) with a mixed-language treatment (i.e., FORTRAN-C++), reinforcing the notion that OO technology implementation into simulations is a "journey" in which the syntax can - by design - continuously evolve.

[1]Aerospace Engineer, Senior Member AIAA
[2]Software Manager, Member AIAA
[3]Currently at Reese Air Force Base

Introduction

Gas turbine engine simulations of elementary form began to appear about four decades ago, coinciding with the time two-spool engine technology was introduced. At that time, and with fuel-flow control problems as a backdrop, the increased engine technological complexity demanded a more complete understanding of dynamic system behavior, and there was a need for analysis methods (mathematical models and their computer implementation) leading to improved system control and performance (Fawke and Saravanamuttoo, 1971). Simulation contributions to the understanding of dynamic systems are as important today as they were in the 1950's. For example, the recent work of Tryfonidis et.al.(1994) is an excellent example of the use of simulation in the interpretation of stall data, the development of signal processing techniques, and stall model development (the original goal of quantifying stall precursor data shed considerable light on the path-dependent behavior of the transition to surge).

It is important to note that, historically, advances in simulation technology very closely follow refinements in dynamic system modeling, the evolution of computer languages, and improvements in computer hardware and operations. In the last two decades, major simulation software development paradigm shifts have been associated with the migration from analog to hybrid computers in the 1970's, a shift to digital computing platforms in the 1980's and subsequent advances in object-based compiler technology in the 1990's.

To embark on the development of object-oriented simulation is not a casual exercise since the resources required to move in that direction can be significant. The attraction, however, is the promise of *software reuseability* that, if realized, can produce roughly a 20:1 return on the investment (see; for example, Nordwall(1992)); the impetus for injecting object-oriented technology into simulation is a business decision, not a technical one.

Interdisciplinary Influence

Most physical processes involve some coupling between scientific disciplines, and a key issue in simulation development strategy is the extent of the coupling, and to what degree the coupling influences or dominates dynamic system behavior.

Simulations are derived largely along discipline lines in which model development perspective often requires simplifying assumptions about discipline decoupling. Of concern is that, in the case of problems that are genuinely interdisciplinary, to what extent accuracy is jeopardized during the process of system decomposition, element analysis, then system reconstitution (Denning, 1990). In fact, decomposition strategy is even an issue on a single discipline level, particularly with highly coupled inlet-compressor problems (see the CFD discussion on page 4).

More traditionally occuring in general simulation practice is an (interdisciplinary) blend of control system technology and aerothermodynamic cycle representations, typically directed at, for instance, the development of integrated flight/propulsion control (IFPC) systems (Shaw, 1988; Akhter, 1989). IFPC applications benefit from the ability to match time scales between the disciplines very closely. Advanced compressor performance requirements have brought to the forefront of interest active control problems once relegated to academic thought, but the challenge in matching the flow physics and control system time scales is aggravated. When advanced control concepts are combined with stability assessment techniques *and* CFD for multistage compressor instability control, what was once a simple initial-value problem is also now a boundary-value problem; the length and time scale for the component-level model is at least an order of magnitude lower than the CFD calculation requirement (for stability).

Another aspect of interdisciplinary coupling is aeroelasticity - aerodynamics coupled with structural dynamics. Consider an engine simulation intended to incorporate compressor blade flutter. Figure 1, taken from Carta(1989) presents the interdisciplinary nature of the problem very clearly. As in the case for CFD, Aeroelasticity transforms what was once an initial-value problem into what is now a combined initial-boundary-value problem. Thus, in the interdisciplinary area, some fundamental implementation issues are:

1. The "raw" number crunching capability required to solve unsteady aerodynamics and structures problems simultaneously,
2. Mismatched fidelity of the simulation modules
3. Developing effective (or standard) means to introduce geometric data into the simulation, and
4. Software manageability.

Scope of Work

Object-oriented (OO) technology has the appealing potential to bridge advanced computer technology and high-fidelity mathematical problem formulations to benefit interdisciplinary simulation and analysis of gas turbine compressors. *Although the OO approach sounds good in principle, what can we expect in practice?* An implementation exercise is an effective way to answer this question. In the present work, exercises involving the development of a prototype OO simulation (framework) are discussed; this project was motivated by the conspicuous absence of direct experience in assessing the benefits of OO environment for gas turbine applications. Activities to date include exercises in OO code development, external (someone other than the code author) review of subsequent software modules, post-development code modification experience, and the exercise of developing a simple interface between a map-based component level model (CLM) and a computational fluid dynamics (CFD) code.

Experiences with the prototype codes were intended to focus primarily on the compressor portion of the turbine engine, though at the outset the complete engine had to be considered in the analysis and design process. NASA Lewis Research on gas turbine OO programming has been underway for approximately four years, from which two prototype programs have been developed:

1. A prototype LISP OO code, developed for the complete engine system, and,
2. A C++ code, developed to explore compression system simulation issues.

Specific concepts of interest in code development include: creation of a traditional map-based compressor model, development of a computational fluid dynamics interface model, and use of the OO inheritance concept to create perturbations of these models. Exercises undertaken represent very specific attempts to quantify the benefit of the object-oriented programming approach (though largely from an aerothermodynamic standpoint).

Of particular interest is the LISP exercise, in which the prototype was instrumental in setting the stage for subsequent object-based research activities; the disposible prototype mentality liberated individuals to pick and choose modules from the LISP carcass.

In the discussion to follow, we first highlight some of the recurring simulation issues that were the catalyst for a look at the OO approach, and then consider issues associated with typical compressor representations (again, from a simulation point of view). After a brief introduction to key object-oriented concepts, the analysis and design of the prototype codes are presented and discussed.

Simulation Issues

Traditional Simulation Process

Gas turbine engine simulations are normally intended to mimic dynamic or steady-state engine behavior through a computer solution to a mathematical representation of the engine cycle. Figure 2, based on the work of Szuch et. al.(1982), illustrates seven key steps in the simulation development process. First, the formulation of the mathematical model involves the appropriate application and tailoring of conservation laws (discipline specific) to the perceived system attributes (physics); the complete mathematical method development includes equation solver strategies. After mathematical methods are established, it is then necessary to prepare data reflecting specific engine design detail and operating environment characteristics. The next step, implementation, links the methods and data through the creation of a computer code based on a computer language (syntax). Traditionally, *implementation expectations are highly coupled to formulation strategy.*

The fourth step in the process, simulation evaluation and validation, is where computed results are compared with design/performance data. Inevitably, a discrepency between calculated response and data occurs; this is usually attributable to either an error in the mathematical method or in the formula translation. Again, with Figure 2 in mind, a modification is usually necessary (to resolve the error) and the assessment of what is required may take the analyst back to either the formulation or implementation stage. Once the simulation results have been validated, documentation of the simulation design (and related methods) then (hopefully) takes place. Finally, if, for instance, the effect of design changes are of interest, the simulation development process just completed is repeated; it is often the intention for the simulation framework to be robust and applicable to a new systems with minimal effort.

Gas turbine simulation design usually evolves from a convenient and natural decomposition of the engine according to component functions (component level model), e.g., an engine is represented as the assembly of a compressor, burner, turbine, and interconnecting ducts. As an example, the turbofan engine schematic shown in Figure 3 has the component level model (CLM) representation shown in Figure 4.

Mathematically, the objective of modeling is to reduce the system to a set of ordinary or partial differential equations that represent the dominant physics of the system. For a typical real-time simulation, a vector of state variables (spool speed, pressure, temperature) is identified where generally the number of parameters in the state vector is a function of the engine fidelity (complexity) desired.

Four Key Issues

Numerous obstacles and limitations to existing simulation are described in the work of Drummond et. al.(1992), but only the following summary is essential to repeat here:

1. *Procedural code structures predominate* existing (large-scale) simulation codes and the ensuing approaches are constrained to be either general, simple, or accurate, but a simulation that is any combination of these is not currently available (in the public domain). Work-arounds to simulation constraints usually involve some compromise whereby, for instance, diminished model fidelity is exchanged for increased simulation execution speed, or where simulation generality (flexibility) results in a lack of program simplicity.

2. *Discipline isolation* is relatively predominant for the'usual' simulation. Interactions between, for instance, aerodynamics, structures and controls is actually fairly limited during mathematical modeling of component characteristics. A requirement also exists for the simulation to deal with a continuum of time and length scales, and for component geometric characteristics to be introduced in a manageable fashion.

3. *Simulation languages and architectures tend to assume a single-processor hardware environment* which impedes software portability to modern parallel and distributed computing environments; this leads to a "strategic fit" philosophy in which simulation methods are not designed to exceed perceived implementation limitations.

4. *Simulation initialization and balancing is non-trivial* for arbitrary engine configurations, especially for highly non-linear systems where a high fidelity simulation is required. Tools for initialization and balancing are never noticed (by the user) when the simulation runs without error. The "inexact science" needed to diagnose problems and take corrective action is frequently an understated aspect of system simulation.

Object-oriented technology can help alleviate the first three of these simulation issues; initialization and balancing problems don't go away - they just have a new look.

In the next section several issues specific to compresor simulation are discussed.

Compressor Simulation Issues

Compressor simulation is always accompanied by empiricizm (for unresolved scales) due to the diverse range of length and time scales associated with the governing physics. At one extreme, the simulation engineer may be interested in post-stall system behavior, for which the fidelity offered by a compressor 'map' representation will suffice. On the other hand, compressor performance predictions reflecting the influence of detailed flow structure require 3-D CFD analyses and sophisticated flow modelling. Traditional treatment has dealt with these extreme situations separately, requiring different hardware platforms and solution strategies for the effective use of computational resources. It has been suggested that advanced simulations should and can be structured to accommodate both in a seamless fashion.

The essence of the present work is to explore methods which integrate what heretofore have been isolated problem analysis techniques. More specifically, the concept of zooming is proposed as a tool for spanning a larger range of length and time scales in a problem than previously achievable; zooming is a temporary excursion to accomodate higer fidelity methods. The door to managing the implementation of the zooming concept opened when object-connector technology crossed paths with distributed processing.

To accomplish the goal of prediction or advanced simulation of compressor operating characteristics requires existing simulations entertain demanding new levels of fidelity and interdisciplinary coupling.

Non-Dimensional Maps

A major task in turbofan engine modeling is predicting the aerothermal performance of the major components of the engine. An acceptable compromise to the description of turbofan component performance is a representation based on non-dimensional analysis. This approach yields multivariate component "maps" which detail base component performance over a wide range of operating conditions.

To get an idea of component map implementation logic, a sample fan logic diagram from an F100 simulation is presented in Figure 5. Note the procedural nature of information flow; the complete simulation logic diagram represents a well-orchestrated (by the programmer, not the compiler) integration of controls and aerothermodynamic representations ... representations not easily modified!

The non-dimensional performance map is a relatively straight forward and an intuitively pleasing approach to compressor performance modeling. However, as a practical modeling technique, the approach has some significant drawbacks. Traditional performance maps are not easily scaled; therefore the maps are limited in their use for modification of component performance to account for new data and/or component sizing studies. Furthermore, modeling of component off-design performance can require additional maps consuming large amounts of computer memory. A key feature of hybrid simulation was the storage of component maps on the digital computer and the communication with and solution of the differential equations on the hybrid computer (in addition, of course, to the real-time capabilities of the hybrid computer).

The drawbacks in the traditional turbofan component performance maps led to the development of alternative methods of modeling component performance. Converse and Griffin (1984) developed a "backbone" performance fitting technique based on the physics of the component rather than curvefits of nondimensional parameters. The beauty of the approach is compromised by its complexity (of course, this diminishes considerably through concept familiarization). A very basic block diagram to describe the backbone approach, Figure 6, belies the significant additional procedural logic required for the backbone implementation.

Performance maps and backbone representations are appropriate techniques for the scope of many current and future simulation efforts. However, they represent static descriptions of component performance and create difficulties in developing, for instance, a high-fidelity dynamic system simulation study of rotating stall. Because this specific example is an active area of research, work-arounds (in the form of modeling) are beginning to emerge. Nevertheless, the traditional focus on aerothermodynamic (or any isolated discipline) performance precludes interdiscipinary system simulation. The role of component representations of a more fundamental nature (based on theories of fluid mechanics or elasticity) is becoming apparent.

CFD Predictions

An approach to component representation based on first principles is desireable, but as a practical matter is difficult to accomplish on a typical (single processor) simulation computing environment. Another issue is that several fundamentally different CFD approaches can be entertained, so there is extensive OO analysis and design work required before a transparent CLM-CFD handoff process can be accomplished. Central to this is the definition of compressor geometry, and the implementation of standards for blading. Modiano et. al.(1994) have recently explored object-oriented grid blocking techniques.

An innovative technique for effective CFD performance predictions is given by recent research by Tan(1994), in which methods are presented for assessment of compressor performance degradation due to inlet distortion. In that effort, an Euler solver is coupled with a model for body force terms; this provides a fairly computationally effective scheme that is currently being examined for possible porting to an object-based construct.

In exchange for the simplicity of the scheme, there arises the need to match (numerically) the upstream influence the compressor has on the 3-D inflow when the compresor is assumed axisymmetric. Further, when matching the distributed field solution with the lumped-parameter CLM model, the exact field integration technique is not always evident (see, Wyss et. al. 1993).

The Object-Oriented Perspective

A fundamental question to ask is: *What needs to be changed in current simulation practice?*

From a programming perspective the view of Schoeffler(1992) provides a good response:

1. Structure, structure, and more structure.
2. Modules that can be used like building blocks.
3. Modules which are application oriented.
4. Modules which can be reused.
5. Modules whose source code need not be changed to reuse them.

Reusability is a central issue (Meyer, 1987). One can begin to see that many of the conveniences associated with the 'usability' of FORTRAN hinder, in the context above, its 'reusability'. Furthermore, it starts to become clear that a computer language that has the attributes stated above would be the desired language for simulation – object-oriented languages appear to fit the requirements; specific features supporting this follow later.

In conjunction with the above, traditional (digital) simulation developers find it impossible to resist customizing code and tailoring solution techniques to the specific problem and performance window at hand (the "need for speed"); this has far-reaching manageability effects that include documentation, reliability, maintainability, and re-useability of the subsequent code. Application-driven efforts require flexibility in the multi-fidelity simulation.

In this section, we try to answer three frequently asked questions:

What is object-oriented programming?

How is the object-based approach different from a subroutine-based FORTRAN program

Why do we need another programming language?

It is proposed that an object-oriented (OO) design approach to gas-turbine simulation has the potential to overcome many of the simulation obstacles outlined above. In the OO approach, emphasis on modeling *objects* (instead of proceses) and the *relationships between objects*, holds the key to developing and managing simulations for complex gas-turbine systems. Robust strategies for implementing an OO perspective have been made possible by the recent accessability of object-based languages[2] and widespread availability of powerful computing platforms[3]: the enabling technology is ready and waiting.

The Terminology

New terminology is usually introduced with any new technology. Below are given our interpretation of the key terms used in the analysis and design process.

Classes

When looking at the actual code for an object-oriented program, nowhere will one find mention of objects. Objects are merely instances of the more general concept of a class. A class is usually described as a group of like objects (classes may also be thought of as templates for objects). A distinct object is created when a specific set of "values" is assigned to the attributes, that is, when an instance of an object is created (instantiated).

Objects

What is an object? Objects are defined in terms of classes. A class is a group of objects that have the same attributes (length, area, inlet pressure) and methods (equations for state variable time derivatives). The individual values of the attributes of a particular class are set by creating an instance of the class. For example, Figure 4 illustrates that there are several intercomponent ducts in the engine. The unique values of the duct's attributes are what distinguishes one duct from another. They are all members of the class of ducts and thus share the same analytic model.

Objects are the building blocks of an object oriented program. An object is a package containing a set of variables, called attributes, and a set of functions, called methods, that operate on the attributes. In an object-oriented program, the code is organized around the data. That is, data and functions to manipulate the data form a complete unit to model real world objects. In contrast, *procedural* codes are organized around the equations or functions, with the data being passed to the function when needed, or automatically via common blocks.

Attributes and Methods

Attributes and methods give the object its appearance and behavior. An attribute is basically a variable. It may be more appropriate to think of an attribute as a piece of data that helps define the state of an object. Methods are simply functions. However, they are very tightly coupled to the attributes of the object in which

[2]Which has been directly influenced by advances in compiler technology

[3]Fueled by the proliferation of inexpensive '486 computers and the availability of inexpensive C++ software.

they reside. That is, methods exist to perform operations on the the attributes.

Messages and Associations

Objects communicate by sending messages through connections called associations. A message is simply a call to one of an object's methods. Associations are the interactions between objects. The association defines such things as how messages are passed between objects, what knowledge an object has of other objects, and what methods are accessible to other objects. Messaging is a way of enforcing clarity in the realtionships between objects - "ask, don't touch" (Winblad et.al., 1990).

With the recent work of Schoeffler(1994) in mind, the philosophy of messaging is an enabling mechanism for distributed processing and the concept of "zooming." Zooming is a process in which adjacent components can communicate even when component fidelity levels are different (pressure and temperature grid resolution differences). Central to effective messaging in the present work are the use of connectors for scale integration (discussed in more detail later).

Inheritance

Inheritance is the sharing of methods and attributes by similar classes. When a class is drived from another, it inherits the essence of the base (parent) class.

An emphasis in our previous discussion has been to pose simulation obstacles in the context of a traditional simulation development approach. Here, we provide our perception of what constitutes an object-oriented (OO) approach, and remark on some differences between OO and traditional procedural code features. Then, some fundamental object representations are presented. Finally, the sections which follow provide the results of experiences with the LISP and C++ project explorations.

A Remark on the References

Literature on object-oriented analysis, design, and languages is abundant[4] and continues to grow. The work of, for example[5], Shaw (1992), Smith (1991), Steele (1984), Booch (1986), and Flaming (1991), provide a window to the world of OO technology. Of the numerous references consulted during the course of this research project, especially noteworthy introductory discussions are given in the work of Ince (1988) as a perceptive commentary on the 'software baroque', and by the work of Taylor(1991) on the motivation and rationale behind the object-oriented philosophy. Germaine to the present focus, however, are two reports. The first is the report of Holt and Phillips (1991), which deals explicitly with a Lisp object-based implementation of the DIGTEM(Daniele et. al., 1983) gas turbine engine code. The second is the work of Cannon(1993) dealing explicitly with the construction of C++ compressor objects.

[4]A trip to any major University bookstore is a stunning example of just the *textbooks* that are available.

[5]To be sure, there are numerous other papers and books.

Designs Oriented to System Objects

The system and compressor codes each began with an analysis of the system to be modelled and a design of the model to represent the system.

Designs based on object-oriented principles differ from traditional designs in the way the (perceived) system is decomposed. A traditional approach identifies the major steps in the overall *process* - a so-called functional decomposition - while OO approaches involve a system decomposition into physical entities, or *objects*[6]. Such a view of the system has lead to the notion of OO design and analysis as a way of 'modeling reality'; a more precise definition is evasive because the OO approach can be characterized as a *collection of concepts* which *together* describe a paradigm shift in software development (Taylor, 1991; Winblad et. al., 1990).

Certainly, the object-oriented paradigm invloves concepts which are not difficult to understand in themselves, but they collectively do imply (require) a new way of looking at and analyzing complex systems. Booch(1986) provides a useful outlook:

> "Read the specification of the software you want to build. Underline the verbs if you are after procedural code, the nouns if you aim for an object-oriented program."

Again, the intent in departing from traditional methods goes beyond the task of producing a working simulation for a specific component - it is moreso an ongoing effort to provide an infrastructure which inherently has the capability to rapidly produce a wide range of (new and existing) system configurations involving multiple disciplines and varying degrees of fidelity. The "correct" design will not change the outcome, just the journey.

The Initial Prototype

The first attempt at an object-based view of a gas turbine system was forged through a joint effort between GE (Lynn, MA) and the NASA Lewis Research Center. This effort was started in 1989 and contined for approximately 3 years. Although an operational code was developed, the more lasting impact of the work was the following:

1. The proof-of-concept for an object-oriented programming approach in gas turbine simulation.

2. A demonstration of the effectiveness of "shrink-wrapping" FORTRAN code with an object-oriented language.

[6]Generally speaking, object definitions are not restricited exclusively to physical entities; objects can also be, for example abstractions of *events*.

3. The identification and initial development of connector-object technology.

Prior to this exercise, there were no known object-based simulations (in the public domain) and the notion that an object-based simulation methodology would be successful was based simply on conjecture. To keep our results in the public domain as much as possible, a non-proprietary component level model (CLM) code was selected as the baseline simulation. It is fairly straightforward in the CLM structure (recall Figure 2) that each component of the system (i.e. fan, compressor, combustor, etc), as well as the generic mixing volumes, can be represented as an object. Each object has characteristics (like state variables) and functions (equations for state variable time derivatives) which are combined together to create a complete definition of the object. In general, when surveying a gas turbine, anything that is worth talking about is probably an object.

Analysis and Design Effort

The result of connecting the components and mixing volumes together with connector groups is a system. However, it is necessary to bring these objects together under an umbrella system object. The system defined by connecting components and mixing volumes together is strictly a static description of the system. It is necessary to define the system object (which contains the components, mixing volumes, and connector-groups) in order to provide the actual simulation methods (steady-state or transient).

A non-proprietary engine model, DIGTEM (Daniele et. al., 1983), was selected for decomposition and implementation in the Common Lisp Object System.[7] A graphical user interface was developed to simplify the creation and execution of the system.

A class hierarchy, shown in Figure 7, was created to provide a general framework for simulation model development. In principle, the framework allows simulation models across varying levels of fidelity and disciplines, and a structures code was identified to be married with the aerothermo cycle code (structural object definition was not completed).

Code Structure

The process of transforming the original DIGTEM code was a gradual effort. This suggests (correctly) one approach to object-oriented programming is that in which the code is not written *entirely* in, for instance, Lisp or C++. As mentioned earlier, the idea is not to assume that an object-oriented *language* is synonymous with the idea of object-oriented *technology*; although potentially clumsy, many different languages can be used to implement an object-based design.

[7]This novel use of Lisp resulted in this research sometimes being referred to as the "Lisp Project"

To reduce development cost of the prototype development, it was appropriate to reuse existing FORTRAN code wherever possible.[8] Such an approach does require some rehabilitation of the FORTRAN subroutines. For instance, all COMMON, DATA, and EQUIVALENCE statements must be removed if, within a given subroutine, the data could otherwise be passed through the subroutine argument list. In the DIGTEM case the "general cleanup" of the FORTRAN required:

1. A cataloging of all variables in the code to identify unnecessary variables, dummy variables repeatedly reused for different purposes (at different points in the procedural calculation sequence),
2. The eradication of all COMMON blocks, and migration of the effected variables to the subroutine calling argument list, and
3. Identification of modules fundamentally redundant.

The effect of this action was threefold:

1. There was a large increase in the argument lists for eash subroutine
2. The DIGTEM subroutines are now highly cohesive and loosely coupled to one another.
3. The communication between the subroutines is minimized, only the appropriate arguments are passed.

Also, subroutine names were changed to represent the actual components of the engine; this reflects a migration of the code to an object-based system. It is revealing to provide samples of actual code to reinforce these points. Sample class definitions are given in Figures 8; the instantiation and coupling of objects is shown in Figure 9. The FORTRAN-Lisp mixed language approach to the prototype is manifest in "foreign" object calls, as shown in Figure 10. Rehabilitation of FORTRAN compressor argument list - to eliminate COMMON and EQUIVALENCE statements - produces a generic compressor code, as illustrated in Figure 11.

An interesting feature of the prototype development is that the conversion was evolutionary. The basic LISP shell was operational, but a great deal more could be done to adhere to the religion of object-oriented design. Although inheritance and reuseability features were portrayed in the prototype to a limited extent, nonetheless, reusability was, in fact, demonstrated. An important message is that during the implementation of object-oriented technology for simulation, a paradigm *shift* does not mandate a software programming *revolution* (as long as the appropriate heirarchy and object-definition roadmap are in place). A derivative of the original graphic user interface, Figure 12, was, at the time, an impressive

[8]Again, the idea was to shrink-wrap FORTRAN routines with LISP

interface for the code operation, and the draging of icons across the screen was an enjoyable alternative to the traditional procedure involving a manual input deck build. This automation of relationship definition did, however, require a significant effort in the prototype effort. For complex system simulations, GUI development is absolutely essential, not just a "fun and interesting" thing to do.[9]

Connector Groups

Although the components and mixing volumes are the fundamental building blocks of the LISP simulation system, connector groups are the means by which components and mixing volumes communicate with one another. Connectors are represented as objects in the system; in the prototype simulation, key connector groups defined are parameter groups, zoom processors, and feedback connectors. Parameter connectors are a means to communicate individual parameters of a particular discipline between components and mixing volumes. A zoom processor connects component models of differing fidelity. Feedback connector groups permit the creation of closed-loop systems.

Parameter connectors allow for convenient interdisciplinary system definitions. Zoom processors assist in creating system simulations accommodating a variety of length and time scales (recall the mismatched fidelity issue discussed earlier). Schoeffler(1994) has extensively explored the connector concept, and his research describes the value of connectors as agents of message passing and scale integration – these concepts are relevant to the development of zooming and distributed processing capabilities.

Discussion

Different classes of objects in the LISP code share common methods and attributes through a mechanism called inheritance. For example, a variable compressor inherits most of its definition from the class of compressors. In turn, compressors are a kind of rotating part and thus inherit behavior from the class of rotating components. The goal of this approach is to eliminate redundant code development and maximize the generality of the model. With this general approach in mind, it is necessary to look for a generic starting point to define the system. In the prototype engine cycle the most general object is a fixed control volume. From this, components and mixing volumes are established. Components are associated with real physical entities (compressor, turbine) which transform energy from one form to another. Unlike mixing volumes, no energy can accumulate within the control volume. A more detailed description of the generic mixing volume concept is given in Holt and Phillips (1991).

[9] Again, see Windblad(1990) for further discussion

An extensive validation effort went into the original DIGTEM model, so the relative success of the object-based implemetation is manifest in the lack of difference between the original (FORTRAN) and Lisp state-variable profile predictions. The icon-based graphic user-interface simplifies system model development; it is not required of the user to modify any source code to create simulations of new configurations (really). An interesting byproduct of this effort was the definition of a very robust set of subroutines for the description of the physics (any reminants of the "solver" were removed from the subroutines). Subsequently, a DIGTEM-ADPAC zooming exercise (Reed and Afjeh, 1993) was made possible by the LISP effort for subroutine redefinition. An issue in that particular zooming exercise was the manner in which the CFD result is most appropriately integrated to match the CLM lumped-parameter specification (again, the subtle integration dilemmas presented by Wyss(1993) take on a new importance!).

Figure 13 is a comparison of the baseline FORTRAN and LISP code results for thrust, HP spool speed, and LP spool speed variations, driven by scheduled ramp changes in fuel flow and nozzle area. There is an offset in the curves associated with the definition of the compressor map data for the original FORTAN code that did not have an exact translation in the object-based environment. The map splits represented a mathematical convenience for which the FORTRAN code procedures were designed to accommodate. Object-oriented technology was pushing the fan to be treated – appropriately so – as a single component.

The Compressor Object Demonstration Exercise (CODE)

Two basic observations were the motivation for the C++ project. First, the zooming concept could not be tested within the framework of the LISP project, so an alternate simulation path needed to be in place. Second, in comparison to the widespread growth of C++ an object-oriented language, LISP programming was viewed more of as an AI language which exhibited a number of object-based characteristics.

Time did not permit a complete exercise in which compressor performance predictions went from Map objects to CFD, then back to the Map object, but a number of essential object-based features were established. Recall that the zooming concept involved the transparent replacement of the map-based compressor representation with a higher fidelity computational fluid dynamics (CFD) numerical model.

Analysis

Analysis of the problem first determined the expected function of the program: CODE would serve as a prototype compressor simulation that would demonstrate the ability to be easily modified, and therefore be positioned to address new simulation issues. For simplicity, the code would be based on the data and maps of the validated DIGTEM steady-state compressor model.

Many techniques exist for system analysis; one alluded to earlier was to write down a statement of the problem and picking out the nouns to represent objects and the verbs to represent associations. Objects and associations for the compressor are shown in Figure 14; again, these were based on the DIGTEM component level model philosophy.

Two basic kinds of associations were used in CODE. The 1:1 and 0:M symbols refer to the minimum and maximum number of objects that can exist at the other end of the association. The 1:1 association means that the compressor *must* be associated with one and only one upstream object (i.e. the fan). The 0:M association means that the compressor can have many bleeds or not at all.

Design

Whereas the analysis phase of the project determined what was to be simulated, the design phase determined how was to be done. During the design phase, the fundamental representations and appearance of the component objects and the structure of the associations were determined. In essence, a "blueprint" for the system was created.

The fundamental representation of a component object refers to the way in which the mathematics are descritized (organized) to represent the "real world". Extensive thought and discussion went into deciding on this representation. The pitfall in adhering to the CLM format is, for instance, in the representation of the control volumes. In the Lisp project, control volumes were required inbetween all (other) objects. Recalling that fundamental tenant of the object-oriented design philosophy is to model the real world, it became clear this situation was not satisfactory. The trap was a formulation based on the procedural mindset. Thus, the object-based control volume representation shown in Figure 15 illustrates the decision to eliminate a separate volume (in comparison with Figure 12, for instance).

It is worth remembering that, traditionally ,volume dynamics were moreoften included in simulations in order to assist in the *numerics* of the problem, not for resolution of the *gasdynamics*. This facet of simulation went unchecked in the initial Lisp prototype development, and thus volume elements had a prominent role in the system description. Frequently, the volumes used in the simulation had very little relation to the physical size of the component, and numerical values for volumes were often adjusted to ease numerical balancing or allow an increase in characteristic time.

Treatment of the performance maps also brought into question the basic architecture of the Lisp experiment. At first it seemed natural to make the maps as objects since there were both data and methods to operate on that data. In fact, an early version of the program employed map objects, but since the calculated performance of the compressor (a map-based one) depends on the map data, it appeared equally plausible that the data should be a part of the compressor object, and the methods to manipulate the data should exist alongside those for calculating such things as pressure. The latter representation was ultimately used. Another approach would be to make the maps objects, but to encapsulate them within the compressor object - that is, instantiate them within the compressor data. The essence of the system design is contained in the Instance Diagram shown in Figure 16. For brevity the attributes and methods are not shown, though the associations that were used are noted. A simple pointer was used to construct a 1:1 association. An unordered set object containing pointers to component objects was selected for the 0:M associations.

Simulation Framework

A simulation framework – a hierarchy of classes as shown in Figure 17 – was used to provide the necessary flexibility and reusability to allow the design to deal with advanced simulation issues.

The framework was structured such that class generality was commensurate with heirarchical position. For instance, the EngineElement class was the most general and contained all the characterisitcs common to all elements of the engine. One level down, the engine elements were divided into more specific classes of objects: those that rotate, those that resemble a control volume, those that have variable geometry, and those that have a performance map. Adding specificity to the lower classes in the heirarchy amounted to adding new methods and attributes to the basic features inherited from the based class (parent class) to distinguish them from other classes. For example, the MappedElement class took the fundamental functionality defined in the EngineElement class and added to it the ability to load, print, and interpolate DIGTEM's performance maps. Note that some methods were superseded (or overridden) by methods in the derived class.

The basic principle behind the Simulation Framework – a set of classes which define the appearance of objects but not their behavior – was derived from the Object Windows Library marketed by Borland International. The Object Windows application framework was designed to save developers of Microsoft Windows applications from having to define new methods to handle basic Windows tasks (like drawing windows on the screen) each time a new application is developed. Classes in the framework have the basic functionality needed to build objects that will run under windows, but they do not define specific behavior. The developer can derive a new class from the framework and inherit the functionality, then add methods to define the application's behavior.

Similarly, the compressor Simulation Framework was created to avoid having to define new attributes and methods to establish the basic functionality of new component classes when they are created. A new compressor class, which inherits the attributes and methods that relate it to a more general group of compressors, is derived and given new methods to define its behavior, such as the ability to calculate a new pressure.

Note the concept of multiple inheritance, deriving from more than one class base, allows a new class to take on an appearance resembling two other classes, such as combining the ControlVolume and RotatingElement classes to form a Compressor class. The use of multiple inheritance does tend to increase the complexity of the heirarchy, however, inheritance is central to avoiding duplicating code. Consider the diagram in Figure 18. Presumably, the methods necessary for handling map data would be the same for the two map-based objects. In panel A, the map routine must be duplicated in each of the mapped component classes. In panel B, they are defined once in MapElement and then inherited.

The Simulation Framework improved the reuseability of the program by decoupling the distinct behavior of the classes of objects from their basic functionality. Further, by separating this functionality into classes, the appearance of new classes can in effect be assembled as desired.

Sample Implementation

Figure 19 shows how the MappedCompressor class was derived from the Simulation Framework. Multiple inheritance was used to combine features nedded to define the basic functionality of the class, such as how to deal with performance maps. To give the new class its useful behavior, methods were added to calculate the various quantities associated with the compressor, such as pressure, temperature, and mass flow. Each of these methods was designed to calculate only one quantity. A complete lisiting of the class' methods is given in detail in the work of Cannon(1993). Some of the methods of the base class, such as its constructor, had to be explicitly called from the derived class to ensure proper operation. Further, due to multiple inheritance, it was necessary to declare from which base class a method was derived – in effect, telling the computer what path to take when searching for the method. This strategy is not unique to the compressor and, for instance, can be used for the construction of turbines.

A glimpse of the concept of encapsulation – hiding attributes and methods from other objects – can be seen in the fact that most of the calculation methods were declared "protected." The effect is that only the object itself can call those methods. The publicly declared RunSteady method takes care of calling the calculation methods in the proper order. This technique controls the access to the object and its data by explicity declaring the interface through which other objects may interact with the MappedCompressor object. Figure 20 is an example of C++ code illustrating the implementation of public and private data.

To create an operation program, the necessary objects – those defined back in the design phase – were instantiated from the appropriate classes. Instantiation of an object involves creation of a new object, and its preparation for operation within the program by assigning pointers and set objects to indicate the associations and calling the initialization methods. Once instantiated, the objects can be manipulated by the main function to form a functional program; in the present exercise, the main function consists of a very simple menu-type user interface.

Although the code is relatively simple in its operation, it provides a concrete demonstration of how object-based technology can be used to rapidly construct and integrate new classes of objects into a working simulation.

Discussion

Outside of the obvious technical advance in simulation practice the OO codes represent, three important lessons reinforced in the development process are (a) the value of the business decision to think in terms of disposable prototypes (versus evolutionary prototypes) during software development projects, (b) syntax complexity and relationship generalizations demand significant attention to code documentation, and (c) the time scale for introducing the object-oriented perspective into an organization is much larger than the time scale for the learning curve of the programming language itself. The failure to advance an awareness of the nature of the prototyping desired and the technology matriculation don't sound like compressor technology issues, but if ignored, these issues can be as difficult to overcome as the modeling of the physics.

Some surprising aspects of the program concerned what in hindsight might not seem exceptionally hard to have predicted. First, the development of an graphic user interface is an essential feature of the software system, not just "a fun and interesting thing" to do. A reasonable interface is transparent to the user – ***an interface is often only noticed when it does not work well.*** Second, the concept of "zooming" between differing levels of simulation fidelity (needed if the code is to truly be a bridge for diverse time and length scales) brings into light the benefit of standards (geometric and interface). Again, these issues do not have an evident "fluiddynamic" look and feel to them, but represent the technology discipline overlap one must deal with in attempting to "do things differently."

Our exploration into this new method of simulating compression system behavior is not marked by a significant difference in predicted component performance (we hoped it would be the same as before), but moreso by a new path to get there. In principle, this investment (time and money) is warranted only if the work to reproduce existing "dusty card deck" capabilities results in methods that truly have the potential to bridge disciplines and integrate codes in a seamless fashion across modern computing platforms.

Concluding Remarks

Object-oriented technology has been employed in this research project to identify an appropriate simulation framework for gas turbine components. Two programs were developed and were validated against DIGTEM. A mixed language implementation of the object-oriented design was sucessful. The concept of zooming was explored for the C++ compressor, but issues associated flowfield integration must be addressed before the zooming concept can be implemented within the necessary connector objects.

The object-oriented approach to the dynamic engine represents a major paradigm shift for system and component simulation; specific benefits are:

1. Object-oriented code modularity is amenable to distributed or parallel processing hardware platforms,
2. Methods and data are more closely related, and a rational hierarchy exists for the gas turbine system,
3. Strict (and enforceable) code syntax improves code maintainability and reusability.

It is proposed that this approach to code development, when executed on parallel/distributed processing environments (with the appropriate operating systems), now provides a realistic basis on which to explore simulations with sub-system component modules of differing fidelity (i.e., different length and time scales). This process of 'zooming' (entertaining various levels of fidelity with a given calculation sequence) holds great promise for those dynamic simulations where, for instance, performance at 'out-of-range' design conditions are unknown, or where a new compressor model behavior is of interest 'in-situ'.

The initial implementation of intercomponent mixing volumes for the LISP code did not strictly adhere to the OO philosophy, but the significant lesson learned was the need for the correct level of component abstraction during the analysis phase. Mixing volumes for traditional simulation derive from the convenience of procedural mathematics, and were not selected as the result of a rigorous OO analysis and design effort. In contrast, a more rigorous OO approach *was* taken in the CODE exercise, for which the fundamental premise for volume definition emerged in the appropriate light. Furthermore, the disposable viewpoint on the Lisp prototype allowed formulation errors to be viewed as lessons to be carried forward; as such, subsequent simulation development was not haunted by the need for software patches and work-arounds.

Although the prototypes mentioned in this work are, in fact, prototypes, it nonetheless has been instrumental in successfully demonstrating the salient features of an object-oriented perspective. Object-oriented design approaches for gas turbine system simulation *do* work ... they are working *now!*

References

[1] AGARD, 1994, "Turbomachinery Design Using CFD," AGARD Lecture Series 195.

[2] Akhter,M.M, J.H.Vincent, D.F.Berg, and D.S.Bodden, 1989, "Simulation Development for US/Canada ASTOVL Controls Technology Program," 20th Modeling and Simulation Conference, May 4-5, Pittsburgh, Pennsylvania.

[3] Booch, G., 1986, "Object-Oriented Development," IEEE Transactions on Software Engineering, Vol.SE-12, No.2, pp.211-220.

[4] Cannon, M., 1993, "Compressor Object Demonstration Exercise," OAI Internship Final Report.

[5] Carta, F., 1989, "Aeroelasticity and Unsteady Aerodynamics," Aircraft Propulsion Systems Technology and Design (G.C.Oates, editor), AIAA Educational Series. p.390-391, 394.

[6] Converse and Griffin, 1984, "Extended Parametric Representation of Compressor Fans and Turbines," NASA CR-174645.

[7] Daniele,C.J., Krosel,S.M. and Szuch,J.R., 1983, "Digital computer program for generating dynamic tubofan engine models (DIGTEM)," NASA TM-83446. (F100).

[8] Denning,P.J., 1990, "Modeling reality," American Scientist, V.78, pp.495-498.(reference in text is to p.497)

[9] Drummond,C.K., Follen, G.J, and Putt,C.W., 1992, "Gas Turbine System: An Object-Oriented Approach," NASA TM-106044.

[10] Fawke, A.J. and Saravanamuttoo,H.I.H., 1971, "Digital computer methods for prediction of gas turbine dynamic response," SAE Technical Paper 710550.

[11] Flaming, B., 1991, Turbo C++, New York: Wiley and Sons.

[12] Holt,G. and Phillips, R.E., 1991, "Object-oriented programming in NPSS," Phase II Report for NASA Contract NAS3-25951.

[13] Ince, D.,1988, *Software Development: Fashioning the Baroque*, Oxford University Press.

[14] Meyer, B., 1987, "Reusability: The Case For Object-Oriented Programming," IEEE Software, March, pp.50-64.

[15] Modiano, D., Steinthorsson, E., and Colella, P., 1994, "Object-Oriented Development in Computational Fluid Mechanics," AIAA Third Northern Ohio Technical Symposium, May 16, Cleveland, Ohio.

[16] Nordwall, B.D., 1992, "Defense Department Expects New Strategy for Improving Software to Save Billions," *Aviation Week and Space Technology*, June 22, pp.59-60.

[17] Reed, J.A., and Afjeh, A., 1993 "Development of an Interactive Graphical Aircraft Propulsion System Simulator," AIAA 94-3216.

[18] Schoeffler, R., 1992, "Concepts and Applications of Object-Oriented Programming," NASA Lewis Technical Seminar, March 18.

[19] Schoeffler, R., 1994, "An Object-Oriented Approach to Distributed Simulation," AIAA Third Northern Ohio Technical Symposium, May 16, Cleveland, Ohio.

[20] Shaw,P.D., 1988, "Design Methods for Integrated Control Systems," AFWAL TR-88-2601.

[21] Shaw,R.H., 1992, "Anatomy of a utility: Writing applications with C++," PC Magazine, February 25, pp.361-370.

[22] Smith,J.T., 1991, C++ For Scientists and Engineers, New York: McGraw-Hill.

[23] Steele, G., 1984, Common LISP, Digital Press, Bedford, Massachusetts.

[24] Szuch,J., Krosel,S.M. and Bruton,W.M., 1982, "Automated Procedure for Developing Hybrid Computer Simulation of Turbofan Engine," NASA TP-1851.

[25] Tan, C.S., 1994, *Inlet distortions on VSTOL Aircraft*, Final Report for NASA Grant NAG3-1567.

[26] Taylor, D.A., 1991, *Object-Oriented Technology: A Managers Guide*, Addison Wesley.

[27] Tryfonidis,M., Etchevers,O., Paduano,J.D., Epstein,A.H, and Hendricks,G. (1994), ASME Gas Turbine Conference, June.

[28] Winblad, A.L., Edwards, S.D., and King, D.R. (1990) *Object-Oriented Software*, Addison Wesley.

[29] Wyss,M.L., Chima,R.V., and Tweedt,D.L., 1993, "Averaging Techniques for Steady and Unsteady Calculations of a Transonic Fan Stage," NASA TM-106231.

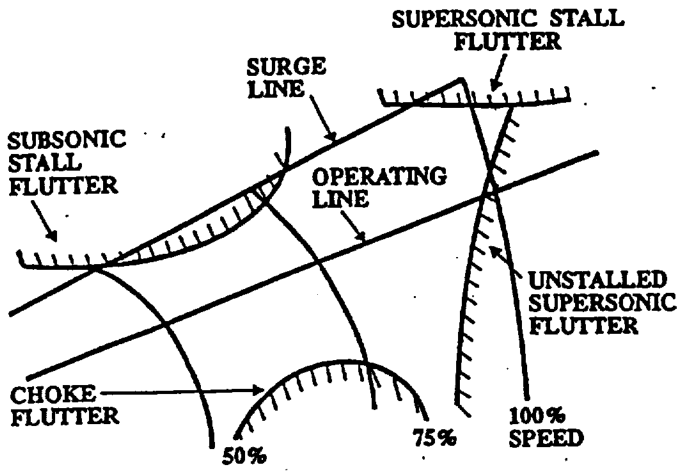


Figure 1. Compressor map showing flutter boundaries (from Carta, 1989).

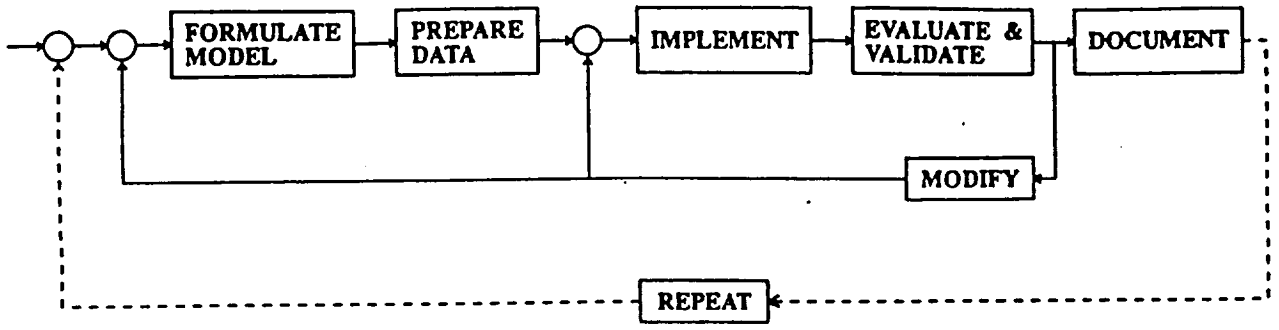


Figure 2. Simulation development process.

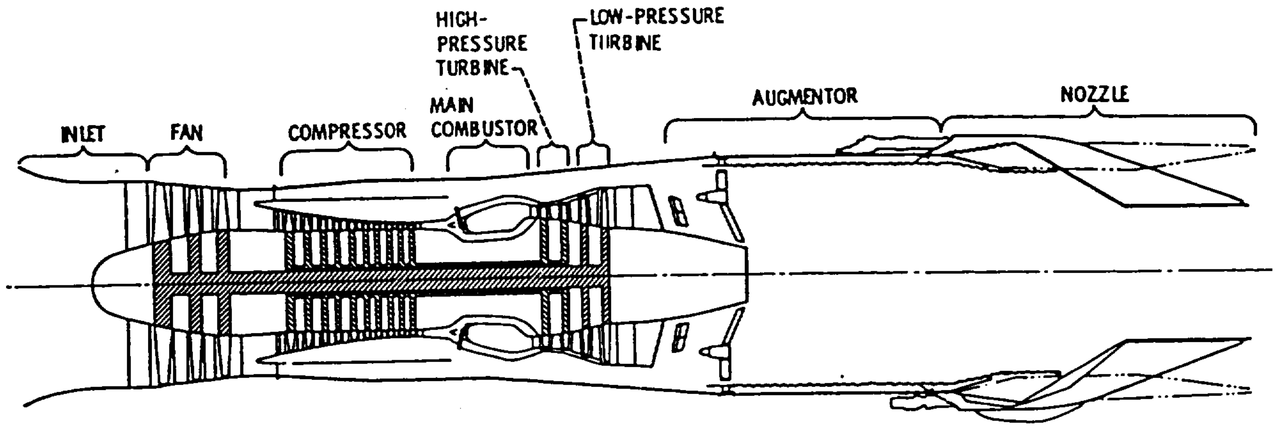
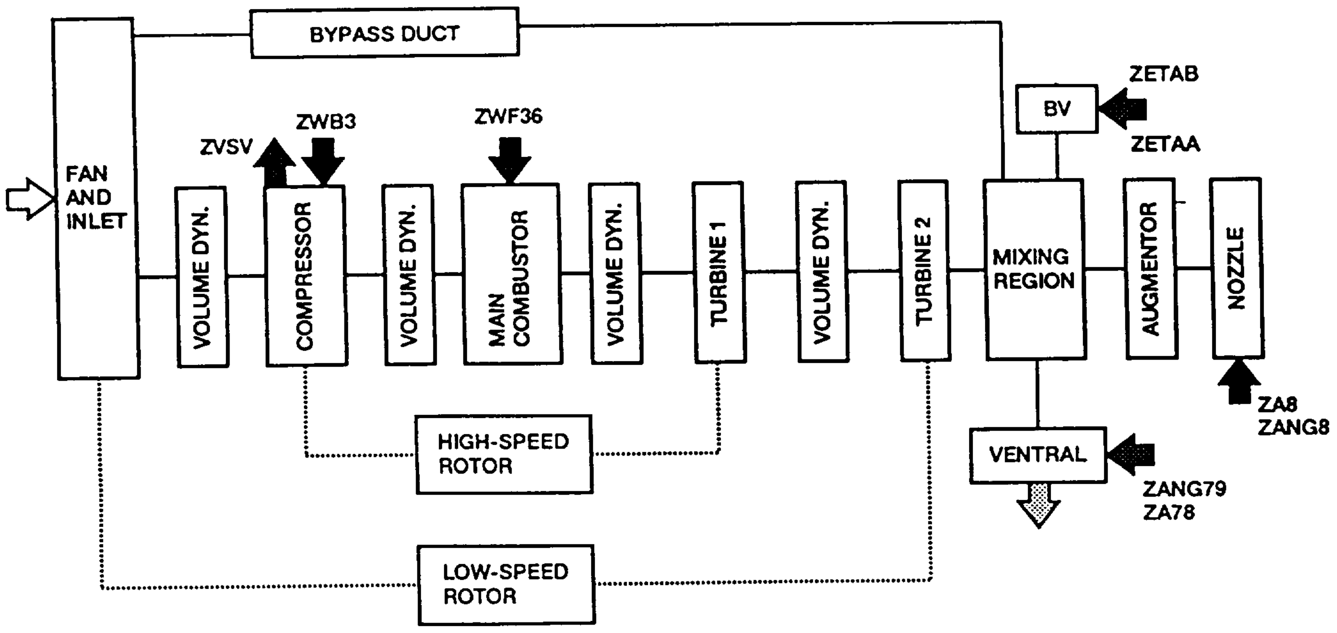


Figure 3. Turbofan flowpath.



Firgure 4. Propulsion system component level model (CLM).

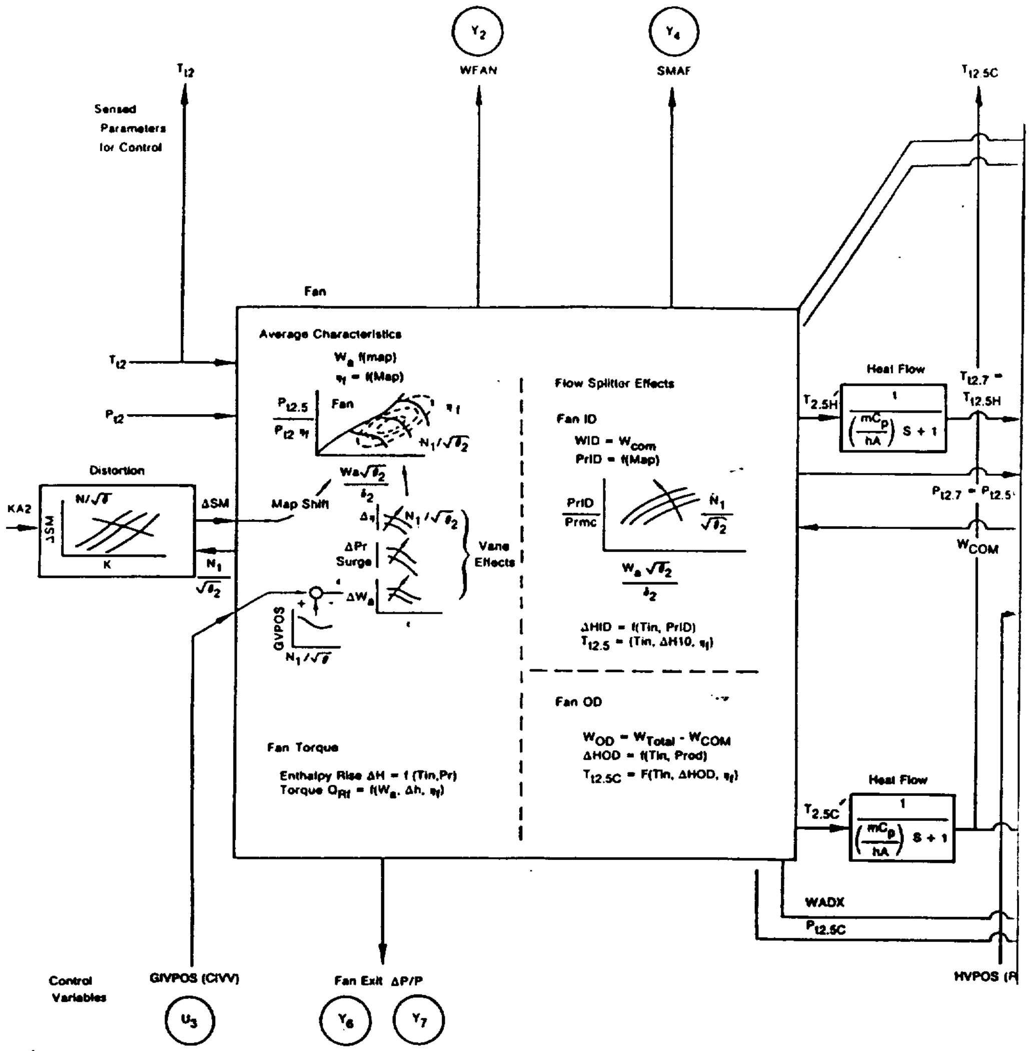


Figure 5. Typical map-based compressor module

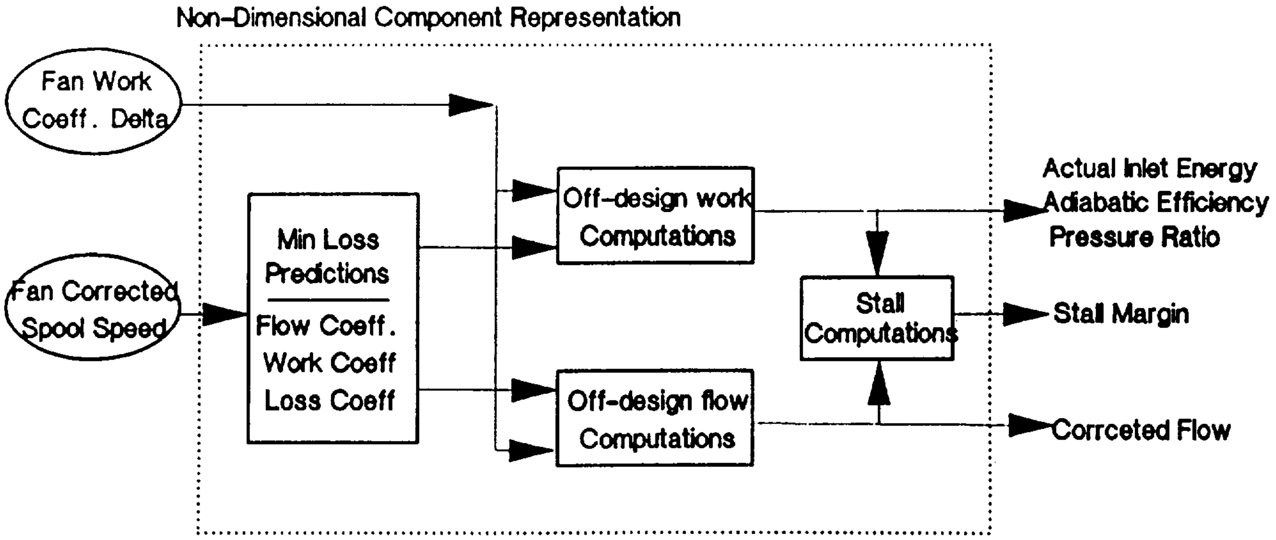


Figure 6. Backbone concept.

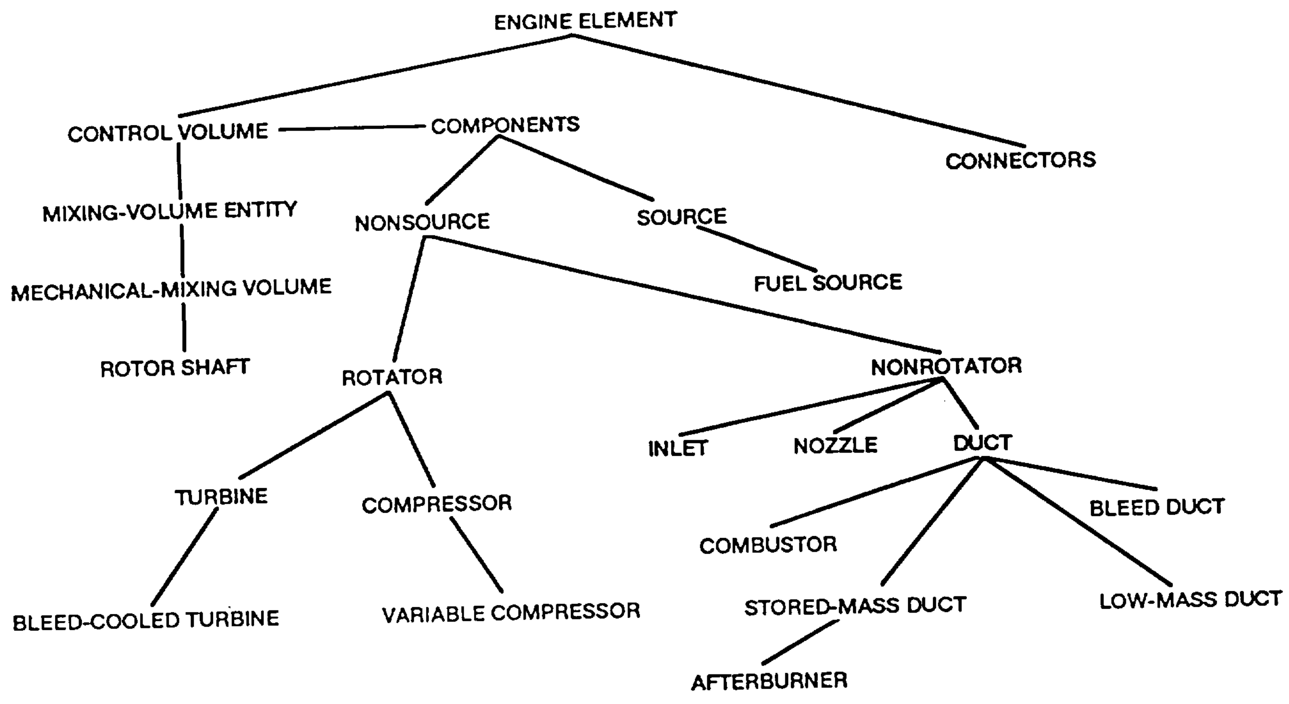


Figure 7. Baseline simulation heirarchy.

```
;;;--------------------
;;;   COMPRESSOR class

(defclass compressor (enthalpy-adding rotator)
  ((points-efficiency :initarg :points-efficiency
                      :accessor points-efficiency
                      :type list)
   (design-efficiency :initform (make-it-array 0.0)
                      :initarg :design-efficiency
                      :accessor design-efficiency
                      :type array)
   (efficiency :initform (make-it-array 0.0)
               :initarg :efficiency
               :accessor efficiency
               :type array)
   (temperature-correction-coef :initform (make-it-array 1.0)
                                :accessor temperature-correction-coef
                                :type array)
   (temp-interpolation-const :initform (make-it-array 0.0)
                             :initarg :temp-interpolation-const
                             :accessor temp-interpolation-const
                             :type array)
 (input-rotational-connector :initform nil
                             :accessor input-rotational-connector)
  )
 (:documentation "The compressor class"))

;;;--------------------
;;;   VARIABLE-COMPRESSOR class

(defclass variable-compressor (variable compressor)
  ())

;;;--------------------
;;;   BLEED-COOLED-COMPRESSOR Class

(defclass bleed-cooled-compressor (bleed-cooled compressor)
  ())
```

Figure 8. Example code for Compressor Lisp Classes.

```
(add-component 'variable-compressor
               :component-name "LPC"
               :points-mass-flow '(193.5 140.0 100.0 193.5 193.5)
               :points-efficiency '(0.8270 0.8097 0.7098 0.7803 0.7807)
               :points-variable-stator-angle '(-1.7004 -24.990 -24.990 -2.5004 -2.5)
               :bias-variable-stator-angle 0.0
               :base-perf-map "/usr/npss/map-dbase/digtem-lpc.map"
               :variable-perf-map "/usr/npss/map-dbase/digtem-vlpc.map")

(add-component 'variable-compressor
               :component-name "HPC"
               :points-mass-flow '(107.0 72.5 50.0 107.0 107.0)
               :points-efficiency '(0.8298 0.8249 0.7582 0.8189 0.8189)
               :points-variable-stator-angle '(4.0 -4.8 -20.0 4.0 4.0)
               :bias-variable-stator-angle 4.0
               :base-perf-map "/usr/npss/map-dbase/digtem-hpc.map"
               :variable-perf-map "/usr/npss/map-dbase/digtem-vhpc.map")



(connect-engine-elements "LPC-ROTOR" "LPC"
(connect-engine-elements "HPT" "HPT-ROTOR"
(connect-engine-elements "HPT-ROTOR" "HP-SHAFT"
(connect-engine-elements "HP-SHAFT" "HPC-ROTOR"
(connect-engine-elements "HPC-ROTOR" "HPC"
```

Figure 9. Example of instantiation and coupling

```
;;;--------------------
;;;   UPDATE-COMPRESSOR-LEVEL2-DIGTEM
;;;   The foreign function call to DIGTEM subroutine COMPRESSOR (eng1/eng2)

(defforeign 'update-compressor-level2-digtem
  :entry-point (convert-to-lang "compressor"
                                :language :fortran)

  :arguments '((simple-array single-float (1))    ;; pressure-in
               (simple-array single-float (1))    ;; design pressure-in
               (simple-array single-float (1))    ;; pressure-out
               (simple-array single-float (1))    ;; design pressure-out
               (simple-array single-float (1))    ;; temperature-in
               (simple-array single-float (1))    ;; design temp-in
               (simple-array single-float (1))    ;; temperature out
               (simple-array single-float (1))    ;; rpm spool
               (simple-array single-float (1))    ;; design rpm spool
               (simple-array single-float (1))    ;; design mass flow in
               (simple-array single-float (1))    ;; enthalpy in
               (simple-array single-float)        ;; base map 1
               (simple-array single-float)        ;; base-map 2
               (simple-array single-float)        ;; base map 3 -in/output
               (simple-array fixnum (5))          ;; base map 4 -in/output
               (simple-array single-float)        ;; var map 1
               (simple-array single-float)        ;; var map 2
               (simple-array single-float)        ;; var map 3 -in/output
               (simple-array fixnum (5))          ;; var map 4 -in/output
               (simple-array single-float (1))    ;; var input effect cvgp
               (simple-array single-float (1))    ;; correction mass flow
               (simple-array single-float (1))    ;; correction temperature
               (simple-array single-float (1))    ;; design efficiency
               (simple-array single-float (1))    ;; temperature interpolation const
               (simple-array single-float (1))    ;; mass flow -output
               (simple-array single-float (1))    ;; efficiency -output
               (simple-array single-float (1))    ;; variable effect -output
               (simple-array single-float (1))    ;; enthalpy-out -output
               (simple-array single-float (1)))   ;; energy term -output
  :return-type :void
  :language :fortran)
```

Figure 10. Illustration of FORTRAN call from within Lisp.

```
              SUBROUTINE ENG2(P3,P22,XNH,T22,CVGP,WA22,ETAHC,CSHIFT)
               PARAMETER (NXP2=11,NCV2=14,NS2=2)
              COMMON /MPCVG/FX2(NCV2),F2(NXP2,NCV2,NS2),FSV2(NS2),N2(6)
               PARAMETER (NXP4=12,NCV4=14,NS4=3)
              COMMON /MPCPB/FX4(NCV4),F4(NXP4,NCV4,NS4),FSV4(NS4),N4(6)

Before        COMMON /CONST/ AQL13,AQL6,V13,V3,V4,V41,V6,V7,XIH,XIL,BSFVGP,
             * BSCVGP,CC(50),BETAHC,BETAB,BETAAB
              COMMON/ENG22D/P22D,T22D,WA22D,ETAHCD
              COMMON/ENG3D/P3D,T3D,WA3D,H3D,GM3D
              COMMON/ENG4D/P4D,T4D,WG4D,HP4D,DH4D,XNHD,WBLHTD,WBLLTD,WBLOVD


               SUBROUTINE COMPRESSOR(PIN,PIND,POUT,POUTD,TIN,TIND,TOUT,
After        *  XSPOOL,XSPOLD,WDOTID,HIN,PMAP1,PMAP2,PMAP3,MAP4,
             *  PMAP1V,PMAP2V,PMAP3V,MAP4V,CVGP,WCORR,TCORR,ETACOD,BETACOM,
             *  WDOTIN,ETACOM,CSHIFT,HOUT,EOUT)
```

Figure 11. Rehabilitated FORTRAN subroutine.

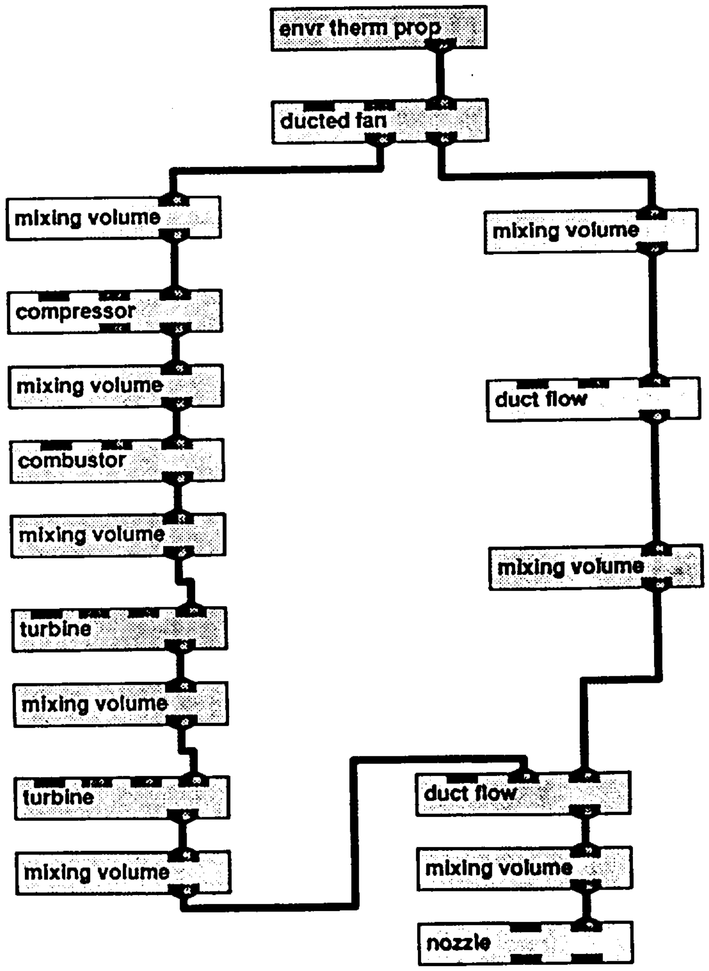


Figure 12. Sample prototype GUI for system simulation.

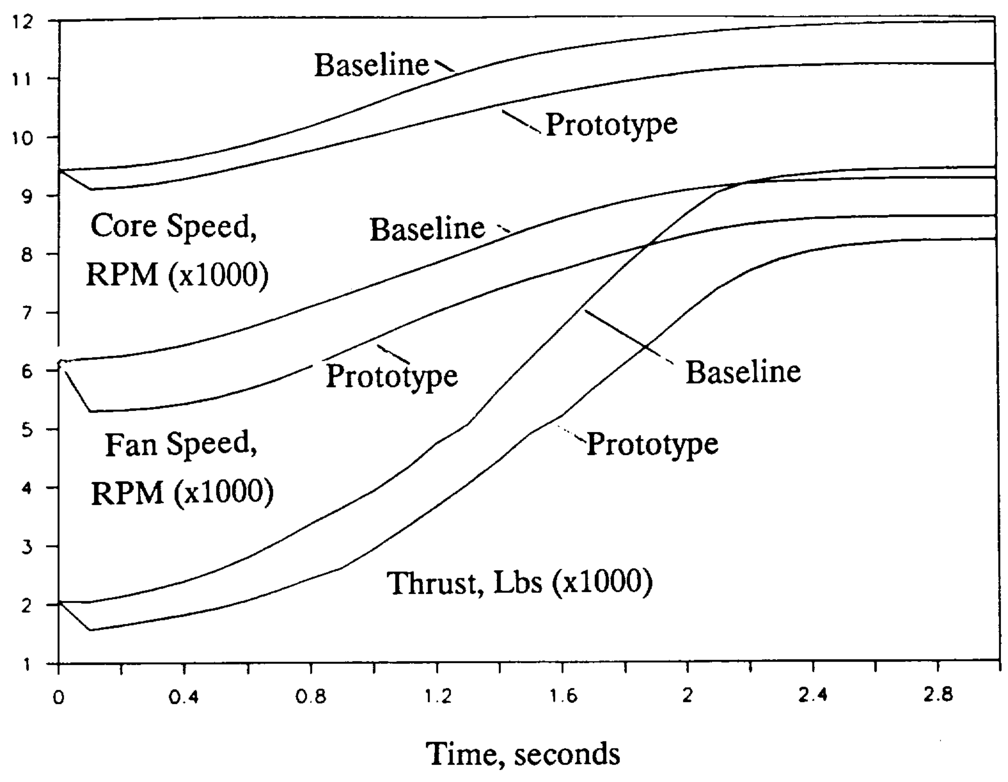


Figure 13. Comparison of Lisp and baseline calculation results.

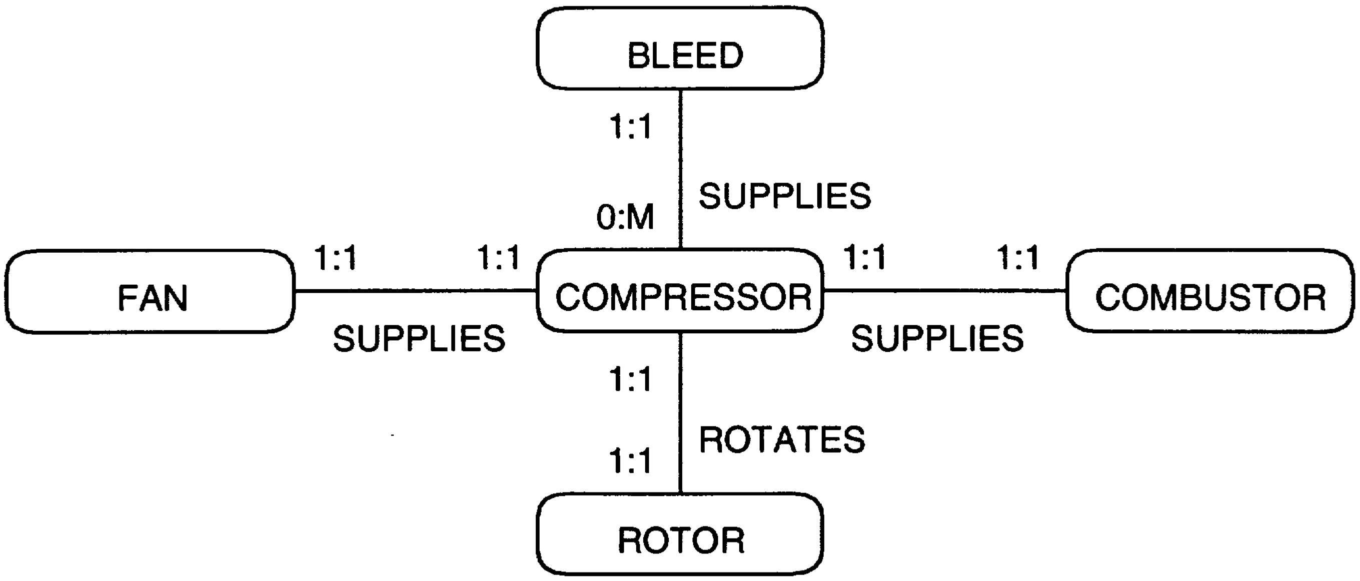


Figure 14. Objects and associations for C++ compressor.

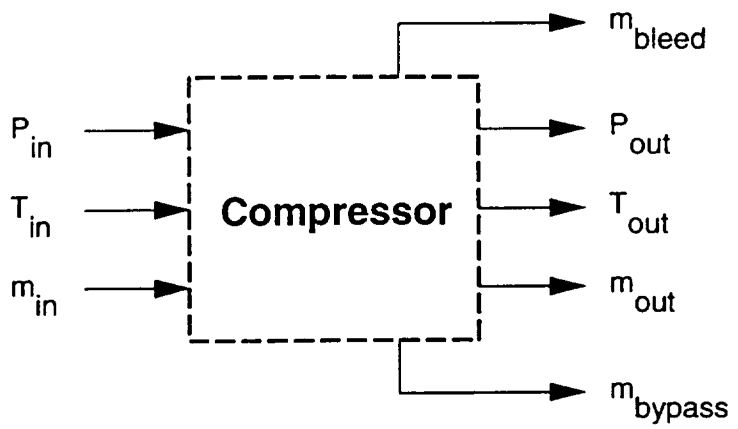


Figure 15. Fundamental component representations.

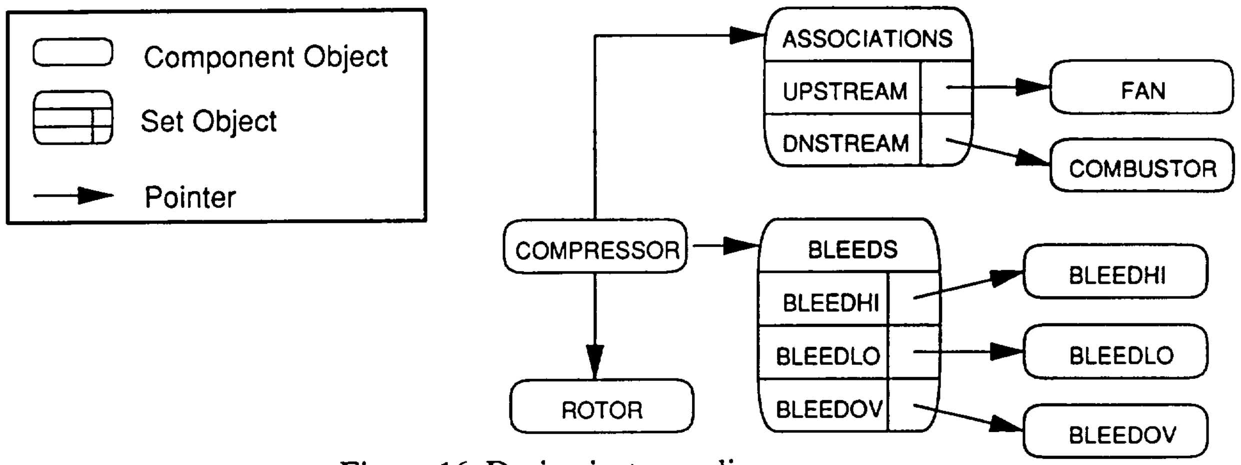


Figure 16. Design instance diagram.

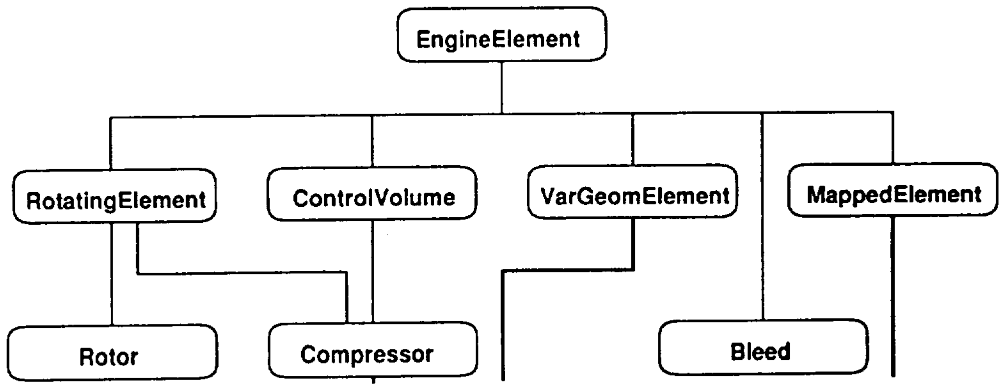


Figure 17. Simulation framework.

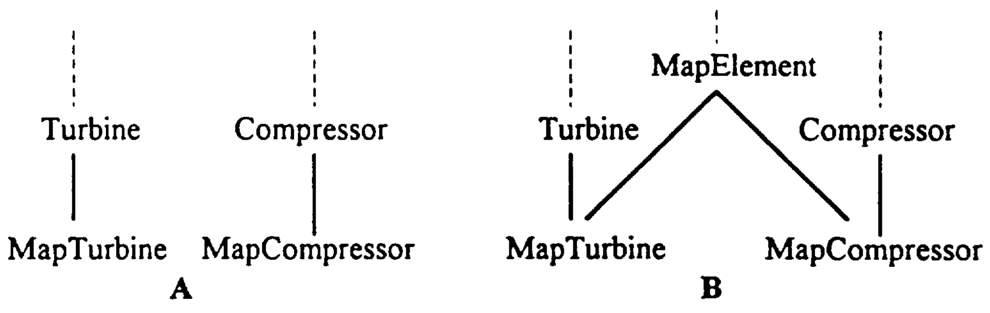


Figure 18. Multiple inheritance.

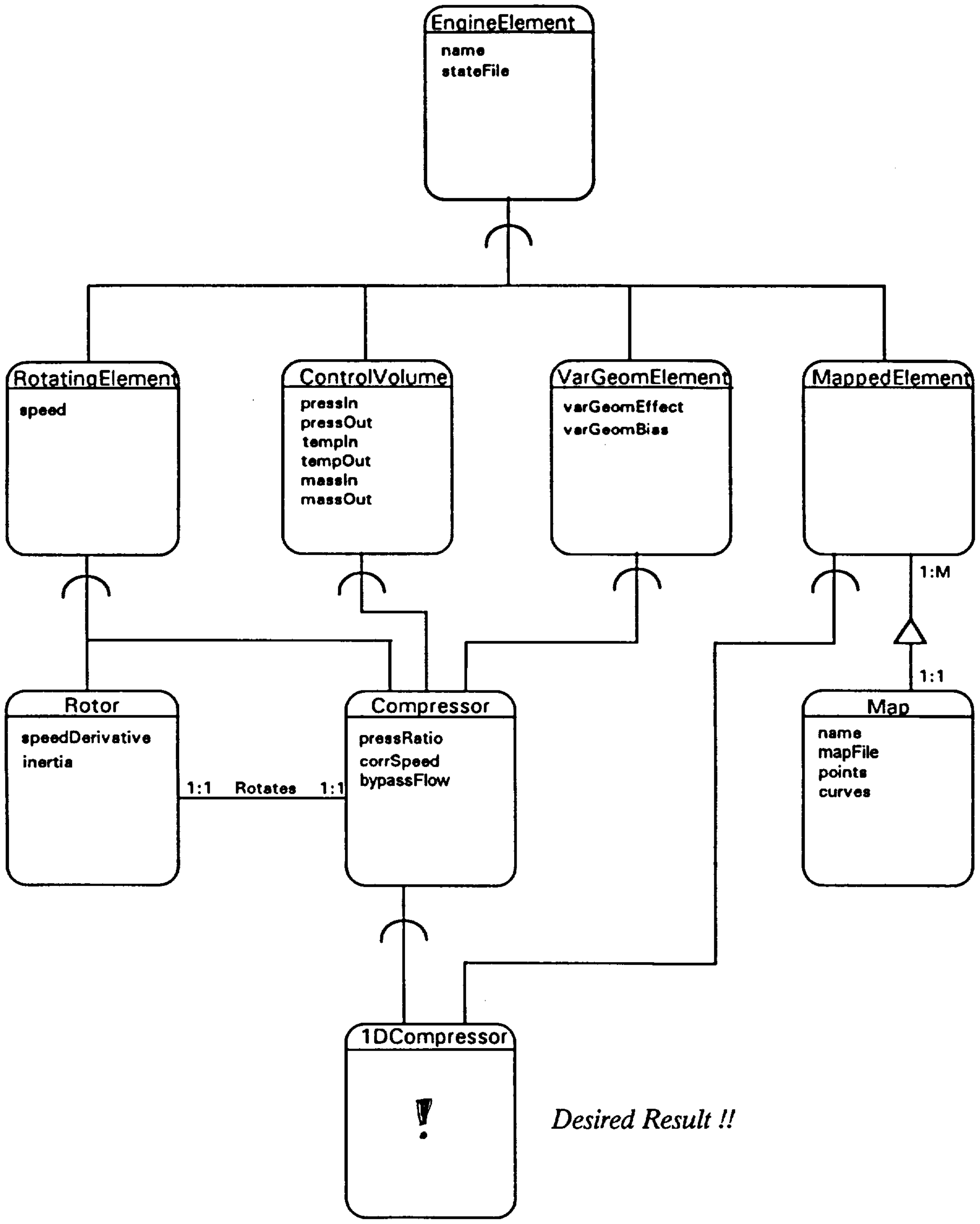


Figure 19. Map-based compressor class.

```
//
//|                              **Map**                                  |
//|    Base Class: None                                                   |
//|      Base For: None                                                   |
//|   Description: Contains the map data and handling routines           |
//|______________________________________________________________________|
//|                          **ATTRIBUTES**                               |
//|  mapFile : name of file containing map data                          |
//|  curves : number of curves on the map                                 |
//|  points : number of points on the map                                 |
//|  x : 2-D array of abcissa values                                      |
//|  y : 2-D array of ordinate values                                     |
//|  z : 1-D array of values differentiating the curves                   |
//|______________________________________________________________________|
//|                       **MEMBER FUNCTIONS**                            |
//| Map : constructor : loads map data                                    |
//| LoadMap : reads the map values from the .MAP file                     |
//| PrintMap : prints the map values to the screen                        |
//| InterpolateMap : performs bivariate interpolation on desired map      |
//| ~Map : destructor                                                     |
//|______________________________________________________________________|
//|                            **NOTES**                                  |
//| Creation Date: 12/07/92                                               |
//|   Last Update: 01/04/93                                               |
//|      Compiler: Borland Turbo C++ Version 3.1                          |
//|                SGI UNIX C++ Compiler                                  |
//|______________________________________________________________________|

//======Class Definition==================================================

class Map
{
protected:
        char *mapFile;
        int curves;
        int points;
        float x[20][20];
        float y[20][20];
        float z[20];
public:
        Map(char*,int,int);
        void LoadMap();
        void PrintMap();
        void InterpolateMap();
        float InterpolateMap(float,float);
        ~Map(){};
};

//======End Class Definition==============================================
```

C++ model is available from:
astovl@heifer.lerc.nasa.gov

Figure 20. C+ + code illustrating public and private data.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE August 1994	3. REPORT TYPE AND DATES COVERED Technical Memorandum

4. TITLE AND SUBTITLE Object-Oriented Technology for Compressor Simulation	5. FUNDING NUMBERS WU-505-68-32
6. AUTHOR(S) C.K. Drummond, G.J. Follen, and M.R. Cannon	
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-9089
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-106723 AIAA-94-3095

11. SUPPLEMENTARY NOTES
Prepared for the 30th Joint Propulsion Conference cosponsored by AIAA, ASME, SAE, and ASEE, Indianapolis, Indiana, June 27-29, 1994. C.K. Drummond and G.J. Follen, NASA Lewis Research Center; M.R. Cannon, Ohio Aerospace Institute, 22800 Cedar Point Road, Brook Park, Ohio 44142. Responsible person, C.K. Drummond, organization code 2760, (216) 433-3956.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified - Unlimited Subject Categories 07, 02 and 59	12b. DISTRIBUTION CODE

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

An object-oriented basis for interdisciplinary compressor simulation can, in principle, overcome several barriers associated the traditional structured (procedural) development approach. This paper presents the results of a research effort with the objective to explore the repercussions on design, analysis, and implementation of a compressor model in an object-oriented (OO) language, and to examine the ability of the OO system design to accommodate computational fluid dynamics (CFD) code for compressor performance prediction. Three fundamental results are that: 1. The selection of the object-oriented *language* is not the central issue; enhanced (interdisciplinary) analysis capability derives from a broader focus on object-oriented *technology*; 2. Object-oriented designs will produce more effective and reusable computer programs when the technology is applied to issues involving complex system inter-relationships (more so than when addressing the complex physics of an isolated discipline); and 3. The concept of disposable prototypes is effective for exploratory research programs, but this requires organizations to have a commensurate long-term perspective. This work also suggests that interdisciplinary simulation can be effectively accomplished (over several levels of fidelity) with a mixed-language treatment (i.e., FORTRAN-C++), reinforcing the notion the OO technology implementation into simulations is a "journey" in which the syntax can - by design - continuously evolve.

14. SUBJECT TERMS Object oriented programming; Simulation; Compressors	15. NUMBER OF PAGES 23
	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