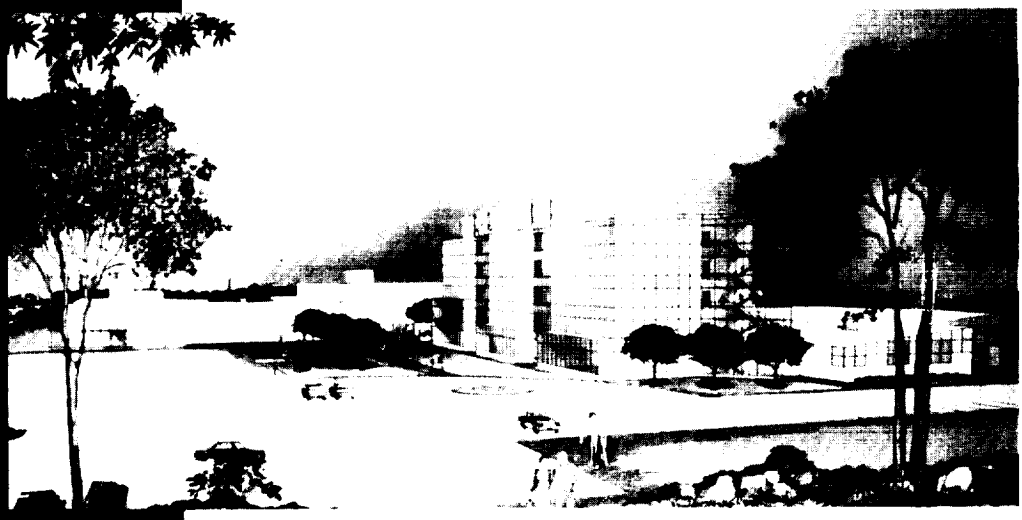
NASA-CR-200613

The University of Toledo



College of Engineering

Department of Mechanical, Industrial, and Manufacturing Engineering

Grant Report
on

Development of a Prototype Simulation Executive with Zooming in the Numerical Propulsion System Simulation

NASA Contract NCC3-207

prepared by

John A. Reed, Research Associate
Dr. Abdollah A. Afjeh, Associate Professor

Mechanical Engineering Department
University of Toledo
Toledo, Ohio 43606

July 1995

Table of Contents

List of Figures

List of Tables

1.0 Introduction

A major difficulty in designing aeropropulsion systems is that of identifying and understanding the interactions between the separate engine components and disciplines (e.g., fluid mechanics, structural mechanics, heat transfer, material properties, etc.). The traditional analysis approach is to decompose the system into separate components with the interaction between components being evaluated by the application of each of the single disciplines in a sequential manner. Here, one discipline uses information from the calculation of another discipline to determine the effects of component coupling. This approach, however, may not properly identify the consequences of these effects during the design phase, leaving the interactions to be discovered and evaluated during engine testing. This contributes to the time and cost of developing new propulsion systems as, typically, several design-build-test cycles are needed to fully identify multidisciplinary effects and reach the desired system performance.

The alternative to sequential isolated component analysis is to use multidisciplinary coupling at a more fundamental level. This approach has been made more plausible due to recent advancements in computation simulation along with application of concurrent engineering concepts. Computer simulation systems designed to provide an environment which is capable of integrating the various disciplines into a single simulation system have been proposed and are currently being developed. One such system is being developed by the Numerical Propulsion System Simulation (NPSS) project.

The NPSS project, being developed at the Interdisciplinary Technology Office at the NASA Lewis Research Center is a "numerical test cell" designed to provide for comprehensive computational design and analysis of aerospace propulsion systems. It will provide multi-disciplinary analyses on a variety of computational platforms, and a user-interface consisting of expert systems, data base management and visualization tools, to allow the designer to investigate the complex interactions inherent in these systems [1, 2].

1.1 NPSS Prototype Simulation Environment: TESS

A key component of computational multidisciplinary analysis simulation systems is the *simulation environment*, which integrates the physical sciences, computer sciences, computer systems software and computer system hardware into a unified system.

In the current work, a simulation environment for propulsion systems is developed which provides means for integrating the multitude of analysis codes, database, expert systems, etc., into a single "seamless" system, as well as presenting a user-friendly interface to the end user. This environment permits choice of analysis techniques and languages, ability to access and manage data from various sources, and interpretation of results. Additionally, it provides heterogeneous distributed computing support allowing optimal use of computational resources.

An interactive programming software system, known as the *Application Visualization System (AVS)* [3], was utilized for the development of the propulsion system simulation. The modularity of this system provides the ability to couple propulsion system components, as well as disciplines, and provides for the ability to integrate existing, well established analysis codes into the overall system simulation. This feature allows the user to customize the simulation model by inserting desired analysis codes. The prototypical simulation environment for multidisciplinary analysis, called Turbofan Engine System Simulation (TESS), which incorporates many of the characteristics of the simulation environment proposed herein, is detailed (see Figure 1.1) [4].

1.2 Zooming

One unique aspect of computational analysis, which is to be incorporated into NPSS, is the ability to carry out computations at different levels of analysis detail. Because it is expected that a detailed analysis of an entire propulsion system would be so complex and computationally intensive as to make it cost-prohibitive, a method of integrating different levels of analysis is desired. This concept, termed *zooming*, would allow physical processes, resolved from a detailed analysis, to be integrated within a system analysis performed at a lower level of detail; and conversely, to allow an engineer or scientist to "zoom in" on a particular component of the total system in order to investigate the relevant physical processes within that component. This provides a most desirable feature of interactive simulation: various design alternatives can be rapidly assessed. To support this feature, interactive graphics visualization may be incorporated within the simulation to provide the user with the ability to view the current state of the simulation through the display of the most recent simulation results.

1.3 Report Outline

This report describes the development a prototype NPSS simulation executive designed to provide support for one zooming strategy. The prototype utilizes a lumped-parameter model for transient and steady-state propulsion system simulation, with the exception of the fan component, which is zoomed to a three-dimensional level and the inlet component which is zoomed to a one dimensional model. The simulation executive provides interactive graphical control of various zooming parameters, and interactive visual monitoring of the three-dimensional flow field in the fan during the simulation.

Section 2 introduces the AVS Visualization environment. In order to more fully understand the manner in which TESS operates and the constraints placed in the development of TESS by the graphical

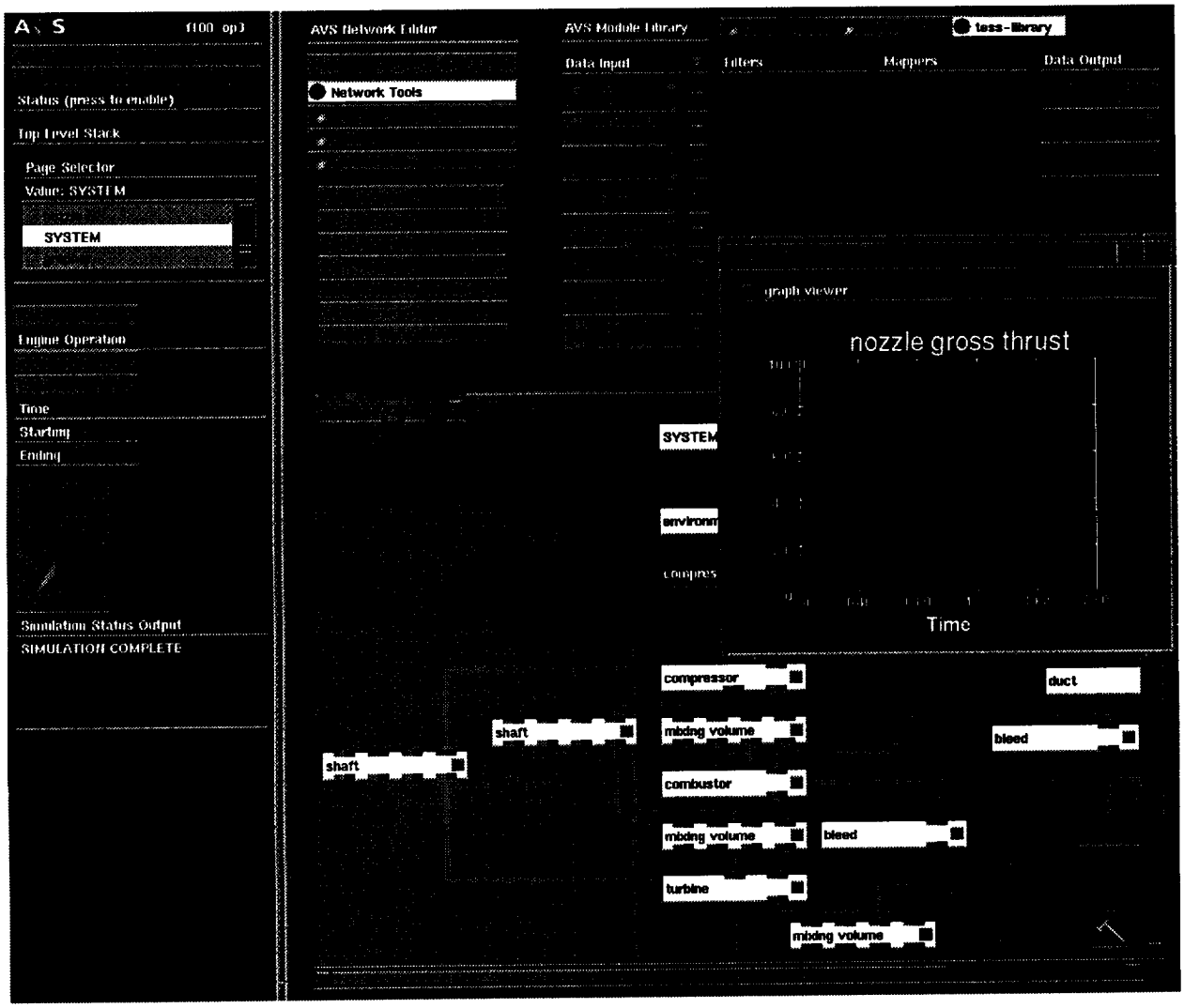


Figure 1.1 - TESS Graphical Simulation Environment

system, it is necessary to have some knowledge of the AVS system operating characteristics.

The mathematical models for each of the propulsion system components currently provided in TESS are described in Section 3. The engine components currently included in TESS are all one-dimensional, lumped-parameter models.

Sections 4 and 5 describe the operating methodology used to run the simulation and its implementation within the AVS framework. Because of the operational characteristics of AVS, certain difficulties had to be overcome in order to control the simulation. These difficulties and their solutions are discussed in Section 5.

Section 6 gives a description of each of the modules used to represent engine components within TESS. The graphical interfaces for each module in TESS are described along with a description of how the modules are to be connected.

The results of a simulation of a test engine using TESS are compared with output from another computer simulation code (DIGTEM) in Section 7.

In Section 8, the zooming concept is described in more detail, with specific application to the fan component in a turbofan propulsion system. The graphical zooming framework, comprised of TESS, a fully three-dimensional Navier-Stokes/Euler flow analysis package capable of providing detailed flow analysis of the fan component in a turbofan engine, a parallel distributed message passing package, and visualization tools, created to aid the user in monitoring the zooming process, are presented.

In Section 9, a final zooming application is presented. Here, a one-dimensional, finite-difference model, used to simulate the inlet of a jet propulsion system is zoomed from the TESS lumped-parameter model.

2.0 Application Visualization System (AVS)

In order to fully discuss the development of TESS, it is necessary to have an understanding of the AVS visualization environment. AVS is a software tool designed primarily to provide scientific and engineering users with the ability to easily analyze and view their data. It allows users to construct visualization applications by combining software components, known as *modules*, into executable *flow networks* that are used to filter the data, map it to pixel or geometry form, and render it on the display screen (see Figure 2.1).

Flow networks, or simply networks, are constructed by direct user manipulation of modules in the AVS Network Editor programming interface. Using the Network Editor, a user creates an application by selecting modules from a menu and drawing connections between them (see Figure 2.2). Data are either read in from a file or generated by a module at the top of the flow network and passed to the module(s) below. The path of data exchange is represented graphically by the connecting wires between the modules - the data are referred to as being "passed along the connection wire". In this manner, control of data exchange between modules is defined by how the modules are connected. As each module receives new input data it executes, generating new output data that is sent to the next module in the network.

The inclusion of a rich set of modules in the AVS system means that in many cases, an entire visualization application may be constructed using standard modules, removing the need to resort to traditional procedural programming. In the event that additional programming is needed, AVS allows users to create their own new modules and dynamically load them into AVS networks.

It is this extensibility of AVS that provides the ability to

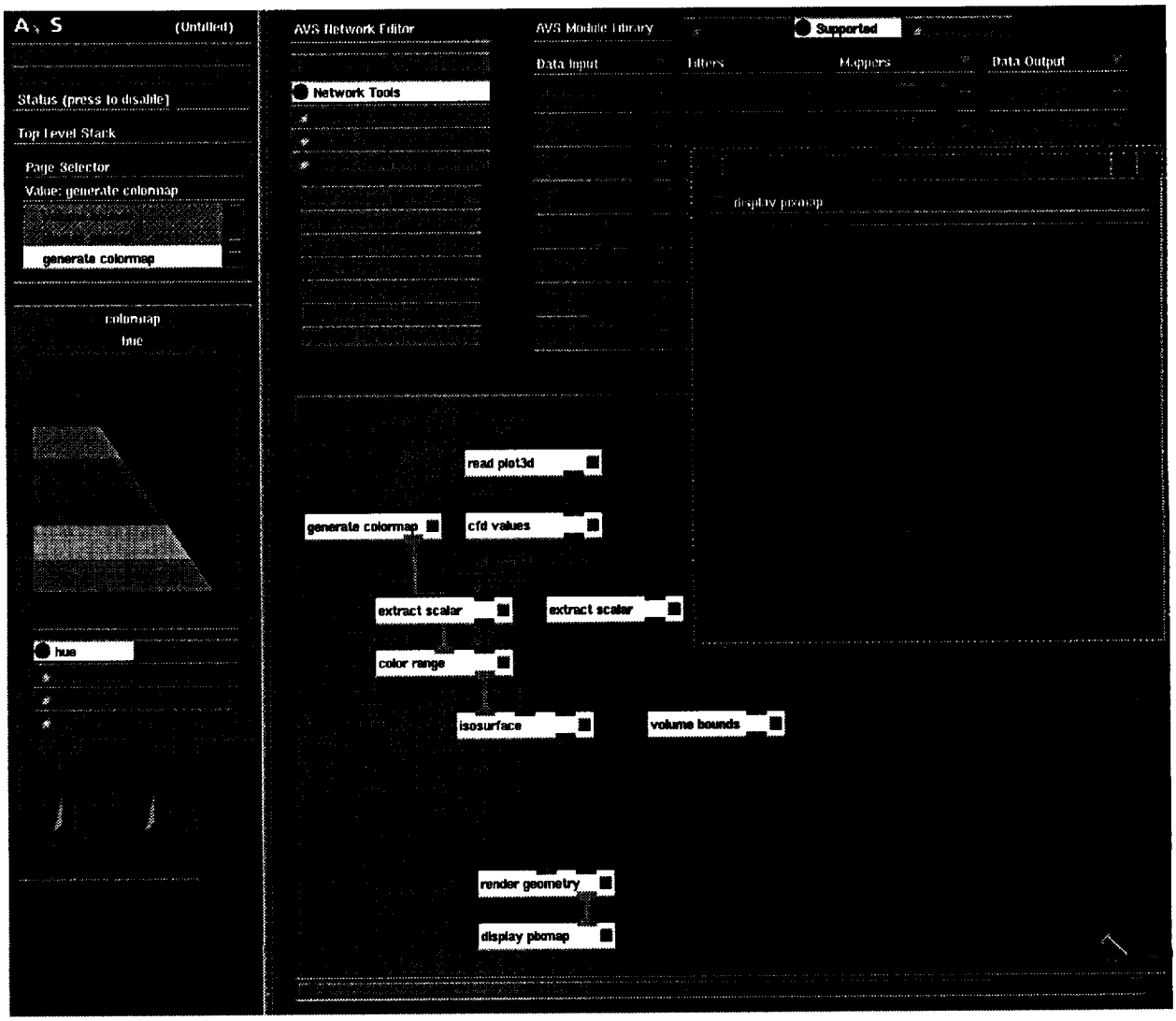


Figure 2.1 - AVS Data Visualization Environment

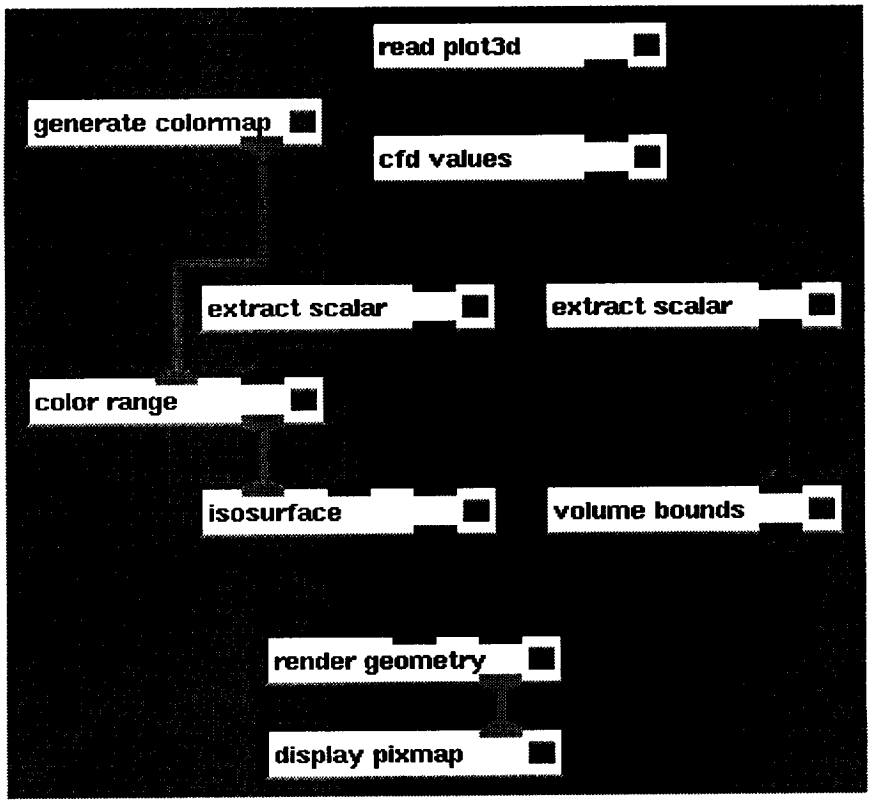


Figure 2.2 - Typical AVS Flow Visualization Network

utilize AVS as a simulation environment for the engine simulation code. Since AVS was not designed for propulsion system simulation, the visualization modules supplied with AVS are typically not of any use in constructing an engine model. But because AVS allows the user to create their new modules and load them into the system, modules representing each of the engine component objects may be created and used in developing engine models.

2.1 Modules

The module is the AVS computational unit with which applications are built. A module, written in either FORTRAN or C programming languages, is designed to be a powerful, yet easy to use, processing component. It functions by taking typed data as an input, operates on that data by executing the code in its *computation function*, generating new output data which is then sent to the next module(s) in the network.

Each module is capable of being instantiated multiple times within an application by virtue of the fact that each module executes within its own UNIX process. This allows each module to operate independently, even though modules of the same type have the same FORTRAN or C code. Thus, AVS modules are able to emulate some of the characteristics of objects in an object-oriented programming language, such as multiple instantiations of objects and data encapsulation [5], without programming in an object-oriented language.

Because AVS modules operate in separate UNIX processes, AVS provides the ability to execute modules remotely on another host of the same or differing hardware as the local machine running AVS. This provides the ability to construct efficient processing networks by running computationally intensive modules remotely on powerful hosts such as mini or supercomputers.

A module's interface to the user is the module icon, (see Figure 2.3). Each module's interface is simple and consistent and will include the following:

- A set of input data ports
- A set of input parameters
- A set of output ports

Figure 2.3 - Module Interface: The Interface Icon

2.1.1 Data Inputs

Data are input to a module through one or more input ports of the module. These input ports appear as colored bars located along the top edge of the module icon. The color of the input ports indicates the type of data the module is capable of receiving. Some of the data types supported by AVS are: field, colormap, geometry, integer, floating point, string and user-defined data. AVS allows only those modules sharing common data types to be connected when constructing a network. The input ports are classified as either REQUIRED or OPTIONAL. If a module has a REQUIRED input port, that port must be connected to another module, in order for the module to execute.

2.1.2 Input Parameters

A module's input parameters control the manner in which a module processes the input data. They provide the user with interactive control of a module's operation along with providing an interface to allow the user to enter input data. Input parameters also can be implemented so that they reflect real-time changes of certain parameters in the module.

A rich set of *control widgets* is available to control the module's input parameters. AVS includes the following types of control widgets (see Figure 2.4):

- Dials and Sliders indicate integer or floating point numbers
- Typeins allow the user to specify a character string or numeric values
- Toggles provide on/off control for various parameters
- Radio Buttons provide mutually exclusive choices to the user
- File Browsers allow a user to access system files to read, or create a file to which output is sent

2.1.3 Data Outputs

Once a module has processed its input data, it may be output through one or more module output ports. These appear as colored bars located along the bottom edge of the module icon. As with the input ports, they are color-coded to indicate what type of data the module is outputting.

2.2 Flow Networks

By using the AVS Network Editor, a mouse driven graphical interface, a user can select modules from a menu, and connect the modules together to form a network. The network can then be executed as desired and later saved to disk to be recalled when needed.

2.2.1 Network editor

The Network Editor (see Figure 2.5) is accessed from the AVS Main menu and is comprised of four windows: *Network Control Panel*, *Network Editor Menu*, *Module Palette*, and *Workspace* (see Figure 2.6).

The user constructs a network by selecting a module from the Module Palette using the mouse pointer. The pointer allows the user to focus attention and initiate action on particular portions of the display. The selected module is then dragged from the Module Palette into the Workspace. When the module is placed in the Workspace, it becomes an instance of the module listed in the Module Palette. At this time, the widgets controlling the input parameters for the module will appear in the Network Control Panel window. As described in Section 2.1.2, these widgets provide interactive control over module execution and can now be adjusted as desired. Also, the name of the module will appear in the stack list browser widget in the Network Control Panel window (see Figure 2.7).

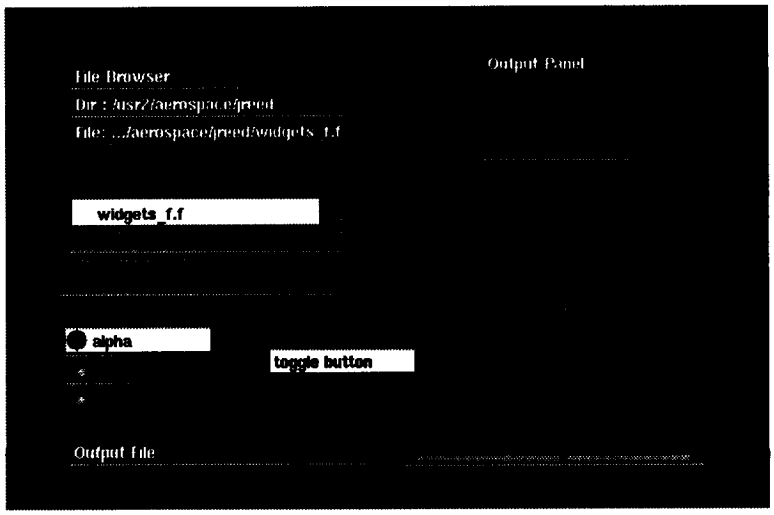


Figure 2.4 - AVS Control Widgets

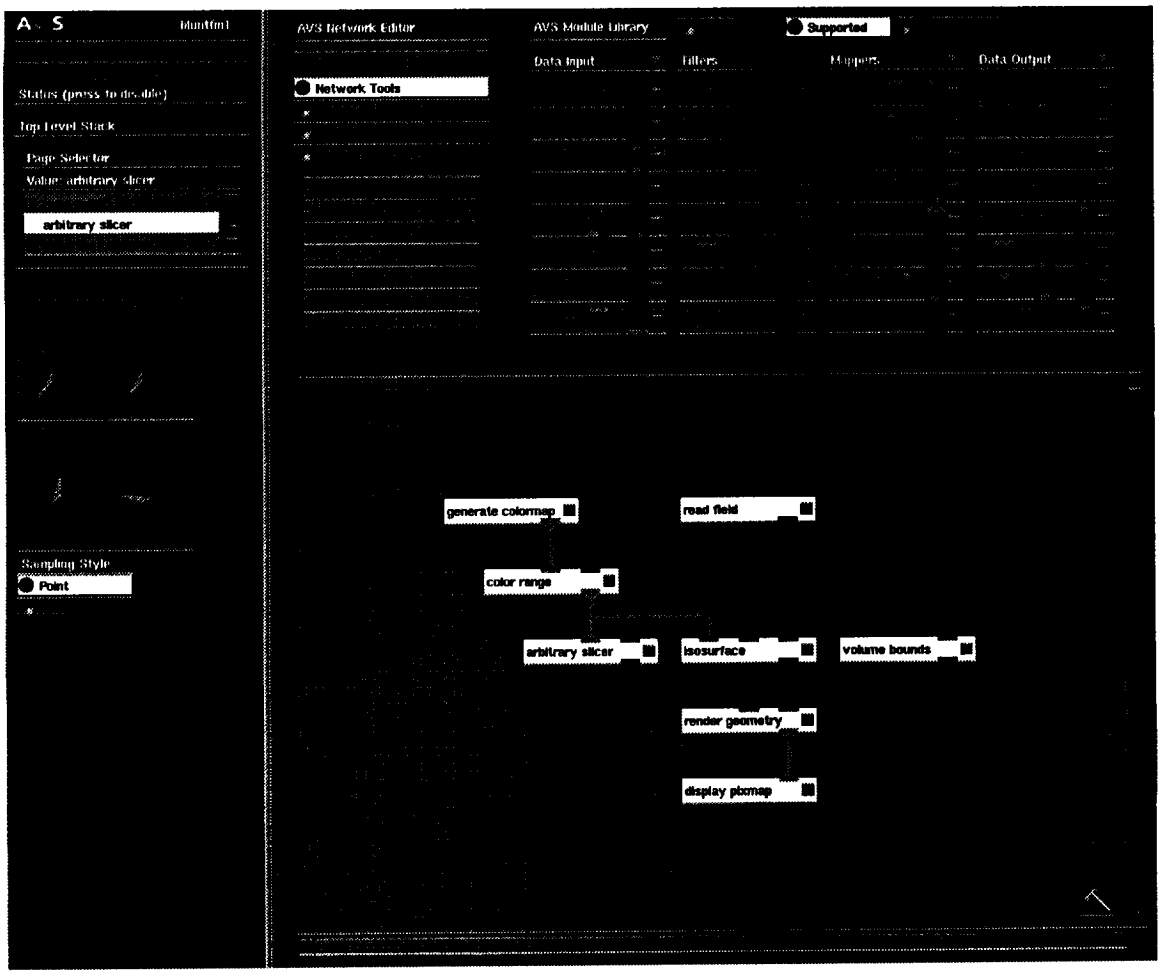


Figure 2.5 - AVS Network Editor

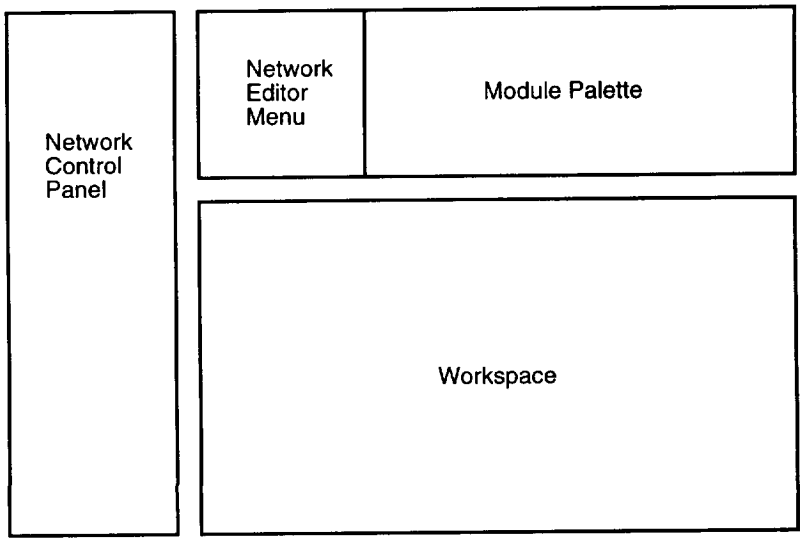


Figure 2.6 - AVS Network Editor Windows

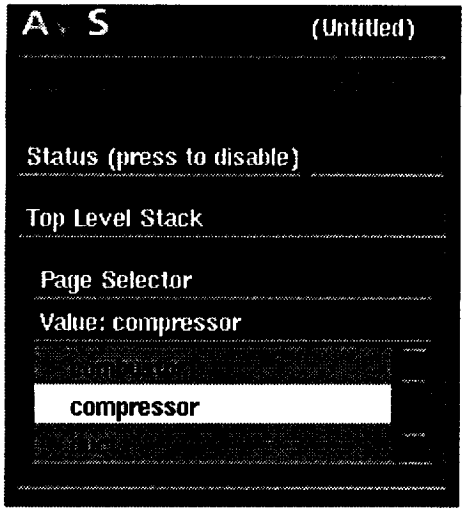


Figure 2.7 - Top Level Stack Browser

If another module is dragged from the Module Palette into the Workspace, its input parameter widgets will appear in the Network Control Panel window, replacing the previous module's widgets, and the name of the module will appear in the list browser widget. Each module has a *page* which holds the input parameter control widgets for that module. When more than one module is in the workspace, there will exist a page for each of the modules. These pages are then placed in a *stack*, much like pages of paper can be organized by placing them in a stack. The stack list browser allows the user to access a page from the stack by selecting it from the list using the mouse pointer.

The connecting lines between modules are established by using the mouse to connect the module's input and output ports. This defines the network and specifies the manner in which data flows in the application. For the engine simulation, the connecting lines represent the connection between each of the components of the engine along which data are passed. In physical terms, the connecting lines represent both the flow path for fluid through the engine and the structural connections along which mechanical energy is transmitted.

2.2.2 Flow Execution

The execution of an AVS Network is controlled by the AVS kernel, which in turn, controls the AVS *Flow Executive*. The Flow Executive determines when a module should execute based on certain criteria, and then directs it to execute. When the AVS Flow Executive is active (the user may turn it off to stop the network from executing), it determines whenever any of the module's input ports or parameters change. If the input port of a module is connected to the output port of another module, the AVS kernel marks the input port as having changed when it receives new data from the other module. A parameter is marked changed when its value has been modified by the user. In either case, the AVS Flow Executive then schedules the module to execute its computation function based on the new input data or input parameter.

The manner in which the network operates is dependent in part on the type of modules which comprise the network. There are two types of AVS modules: *Subroutines* and *Coroutines*.

- Subroutine modules are essentially passive, much like subroutines in a traditional program. When a subroutine module is instantiated, it's UNIX process *sleeps* until the Flow Executive signals it to execute. The subroutine module then takes its input data, operates on it, passes the data to the next module in the network, and returns to its dormant state.
- Unlike a subroutine module, the coroutine module's UNIX process is always active. Thus, the coroutine module can access input data, operate on the data and output it to the network on its own initiative instead of only doing so when signaled by the Flow Executive.

It is important to understand that execution of a subroutine module is completely within the control of the AVS Flow Executive. Because the execution of the modules is linked to the changing of input port data and input parameters, it is not possible to explicitly control module execution[1]. When subroutine modules are connected to form a network, each module executes only upon receiving changes in its input data. This type of operation may be thought of in terms of the *domino theory*. That is to say that once the first module in the network executes, the remainder of the modules will execute much like dominos in a line, tumbling sequentially after the first one has been pushed over. Once the process has begun, all of the modules operate on there own when triggered by the previous module.

All of the modules used in TESS are subroutine modules with the exception of the SYSTEM module, which is a coroutine module.

2.2.3 Data Flow

The AVS applications which are built by connecting modules together, function by passing data from module to module. Because the modules (normally) run within separate UNIX processes, some form of interprocess communication must be available to pass the data along from module to module. One method which allows the sharing of data among modules within different processes utilizes shared memory regions. The data from the output of a module is placed in a memory region which has been created by the AVS kernel by making UNIX system calls. Multiple modules may then access the single copy of data by making UNIX system calls.

1. Actually, AVS has made it possible to control subroutine and coroutine module execution explicitly using the Command Language Interface (CLI). The CLI provides a list of commands which can be used to control certain aspects of network operation, such as executing a module. At the time TESS was developed, this option was not available, and thus, was not implemented.

The shared memory region method of interprocess communication is the normal method which AVS uses to communicate data between processes. The AVS kernel attempts to share data when possible by placing data in the shared memory region. Pointers to the memory addresses are then passed between modules to allow the modules to access the necessary data. By sharing data between modules, memory usage is reduced and processing speed is increased because the data does not have to be copied between processes.

To better explain how data is passed between modules, consider three modules connected as shown in Figure 2.8. Figure 2.9 shows the corresponding data flow diagram for the network. M1, M2 and M3 are the modules, K is the AVS kernel, and D1 and D2 are the shared memory areas allocated for data. The AVS kernel's control communication channel (indicated by the dotted line) informs M1 that M2 requires its data. M1 then places its output data in the shared memory region (indicated by the arrows). The AVS kernel's control communication channel then informs M2 that M1 has sent its data to the shared memory region, D1. M2 is then able to access the data in the shared memory region. In this method, there is only one copy of M1's output data.

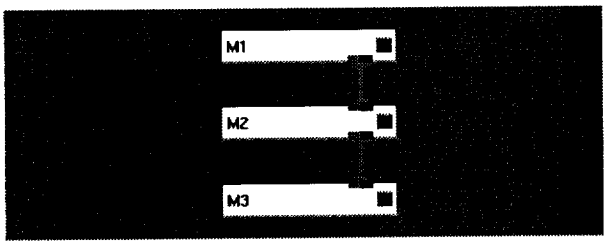


Figure 2.8 - Three Connected AVS Modules

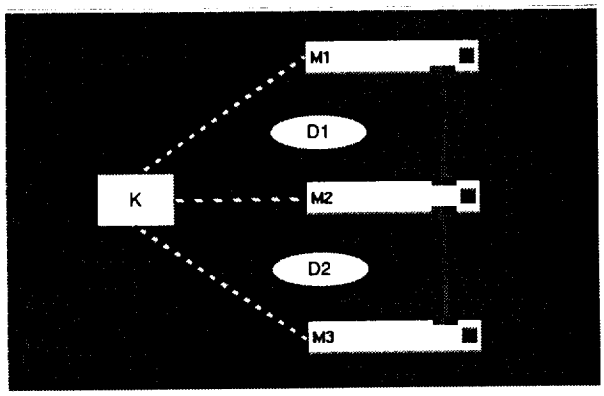


Figure 2.9 - Data Flow Between Kernel, Modules and Memory Regions

Once the data from M1 has been established in the memory region D1, M2 can copy the data into its internal arrays. This is done so that the module can operate on the data without affecting it directly. Once M2 has operated on the data to create output data, it is placed in the next memory region, D2. M3 is now notified by the kernel that it has new input data and is scheduled to execute and may access its data from D2.

2.3 TESS Data Passing Structure

As stated in Section 2.1.1, the data passed along the connecting wires is typed. AVS provides a variety of data types: field, colormap, geometry, integer, floating point, string and user-defined data. It was found in the development of the TESS code, that it would be necessary to exchange both floating point and integer values between modules. If the floating point and integer data types were utilized, this would require that each module have both integer and floating point input and output ports. Furthermore, there would be two connecting lines between connected modules: one for each of the data types. This would greatly complicate the visual representation of how the engine component modules were connected.

To solve this problem, the user-defined data type was utilized. As the name suggests, AVS allows the user to define their own data types and to use the data types for inter-module communication. In TESS, the user-defined data type is defined in the header file `tess_user_data.h`, and has the form of a C structure. The variables defined in the struct are the variables needed by the various modules available in TESS (for more information on these variables see [3]). The contents of `tess_user_data.h` are listed below:

```
typedef struct {
  int nsys;
  int nauxvar;
  float time;
  float massflow [n];
  float fuelflow [n];
  float temp [n];
  float press [n];
  float rpm [n];
  float enthalpy [n];
  float strdmass [n];
  float faratio [n];
  float energy [n];
  float volume [n];
  float massflow_deriv [n];
  float temp_deriv [n];
  float rpm_deriv [n];
  float strdmass_deriv [n];
  float thrust [n];
  float vargeom [n];
  int chkflag [n];
  } tess_user_data;
```

Each of the variables (with the exception of **nsys**, **nauxvar**, and **time**) is a vector array of length *n*. The value of *n* is dependent on the maximum number of

modules likely to be used in an engine simulation, and is used to reserve memory space. The vector indices (1, 2, 3, ...*n*) correspond to the different modules in a simulation. Data corresponding to each module is then arranged within the vector, and made available to the system, providing a structure for accessing and storing information about the state of each of the modules in the system.

For each of the memory regions which is used to store data common to the connected modules (see Section 2.2.3), the data in the memory region will be stored in the format defined in the `tess_user_data.h` file.

3.0 Engine Component Mathematical Models

The mathematical model for each engine component currently included in TESS is presented in this Section. The engine component models are one-dimensional, lumped-parameter, thermodynamic models. The unsteady forms of the continuity, momentum and energy equations are used to generate ordinary differential equations which are used to model the complete propulsion system. A detailed discussion of the development of these models is given in [4].

3.1 Gas Properties

The thermodynamic properties of air and fuel-air mixtures are calculated by considering variable specific heats and no dissociation. Curve fits of data found in [6] are used to compute specific heats, specific heat ratios, and specific enthalpies from given values of temperature and fuel-air ratio. For each engine component, the following equations are used

$$c_p = f(T, (f/a)) \tag{3-1}$$

$$R = f(f/a) \tag{3-2}$$

$$c_v = c_p - R \tag{3-3}$$

$$\gamma = \frac{c_p}{c_v} \tag{3-4}$$

$$h = f(T, (f/a)) \tag{3-5}$$

While the gas constant, R, is in general a variable when mixtures of gases are considered, according to Szuch [7], the sensitivity of R to the fuel-air ratios in a turbojet simulation could be neglected. Therefore, the gas constant of air may be used in the ideal gas law. The use of a constant value of R also prevents the occurrence of algebraic loops which require iterative solutions.

3.2 Bleed

Bleed components are used to provide turbine cooling and auxiliary drive air. Bleed air is assumed to be provided from a high pressure source (such as the exit of a compressor), and supplied to a component which is at a sufficiently low pressure so that flow in the bleed passage is choked. The following describes the equation for the bleed model.

The mass flow rate, $\dot{m}$, of fluid in the bleed passage is

$$\dot{m} = A p_{in} \sqrt{\frac{\gamma}{RT_{in}}} \left[\frac{2}{(\gamma+1)}\right]^{\frac{(\gamma+1)}{2(\gamma-1)}} \tag{3-6}$$

where p_{in} = Stagnation pressure at the bleed inlet
T_{in} = Stagnation temperature at the bleed inlet
A = Cross sectional area of the bleed passage
γ = Specific heat ratio of fluid in the bleed passage

3.3 Combustor

The combustor provides thermal energy addition to the system through the combustion of fuel. The heat addition associated with the burning of the fuel is assumed to take place in the mixing volume directly downstream of the combustor. Stagnation pressure losses are included in the combustor model. Because of the greater temperature rise through the combustor, the assumption of constant specific heat ratio based on the inlet conditions could lead to difficulties in matching steady-state conditions. For this reason, an "average" combustor temperature is computed based on inlet and exit temperatures, and used to compute an average specific heat ratio and enthalpy of the working fluid in the combustor.

The mass flow rate in the combustor is

$$\dot{m} = \left[\frac{p_{in}(p_{in} - p_{out})}{K_c T_{in}}\right] \tag{3-7}$$

where p_{in} = Stagnation pressure at the combustor inlet
p_{out} = Stagnation pressure at the combustor exit
T_{in} = Stagnation temperature at the combustor inlet
K_c = Combustor pressure loss coefficient

The specific enthalpy of the fluid in the combustor is based on the average temperature of the combustor air-fuel mixture. The average temperature is given as

$$T_{avg} = \beta T_{in} + T_{out}(1 - \beta) \tag{3-8}$$

where T_{avg} = Average stagnation temperature in the combustor
T_{out} = Combustor outlet stagnation temperature
β = Combustor interpolation constant

The combustor fuel air ratio (f/a), is computed as

$$(f/a) = (f/a)_{in} + [(f/a)_{in} + 1]\left(\frac{\dot{m}_{fuel}}{\dot{m}_{gas}}\right) \tag{3-9}$$

where $(f/a)_{in}$ =The fuel air ratio of the gases entering the combustor
$\dot{m}_{fuel}$ =The fuel mass flow rate being added to the combustor
$\dot{m}_{gas}$ =The mass flow rate of the gases entering the combustor

The enthalpy of the air-fuel mixture in the combustor, h, is then determined from curve fit data based on the average combustor temperature and the combustor fuel-air ratio:

$$h = f(T_{avg}, (f/a)) \tag{3-10}$$

The rate of energy being added to the working fluid due to the combustion of the fuel, E_{comb}, is

$$E_{comb} = \dot{m}_{fuel} \cdot \eta \cdot HVF \tag{3-11}$$

where η = Combustor efficiency
HVF = Heating value of fuel

E_{comb} is referred to as the combustor *energy term* and represents the energy flux term for the combustor due to fuel combustion. Because the combustion process is assumed to take place in the downstream mixing volume, E_{comb} is the rate of heat addition to the mixing volume due to fuel combustion. Heat addition to the mixing volume is given by the $\delta Q/dt$ term in the mixing volume dynamic energy equation. Thus, E_{comb} will be used by the mixing volume to account for the heat addition due to fuel combustion.

3.4 Compressor

Compressor performance is represented by a set of overall performance maps normalized to design point values. Baseline performance maps provide normalized inlet-corrected mass flow rate and normalized efficiency as a function of normalized pressure ratio and normalized, inlet-corrected spool speed. Shifts in normalized inlet-corrected mass flow rate, based on (un-normalized) off-schedule values of variable stator position, are also provided as a function of normalized pressure ratio and normalized inlet-corrected spool speed. Because of the greater temperature rise through the compressor, the assumption of constant specific heat ratio based on the inlet conditions could lead to difficulties in matching steady-state conditions. For this reason, an "average" compressor temperature is computed and used to compute an average specific heat ratio. The following equations describe the compressor model.

The normalized inlet-corrected mass flow rate value is obtained from the baseline compressor performance map. The map represents compressor performance with the variable geometry at nominal, scheduled position:

$$m_{base,c,n} = f(N_{c,n}, \Delta p_n) \tag{3-12}$$

where $\dot{m}_{base,c,n}$ = Baseline inlet-corrected mass flow normalized to design point
$N_{c,n}$ = Corrected spool speed normalized to design point
Δp_n = Stagnation pressure ratio normalized to design point

The normalized corrected mass flow rate which accounts for off-schedule geometry effects is obtained from the variable-geometry effects performance map:

$$\dot{m}_{var,c,n} = f(CVGP, \Delta P_n) \tag{3-13}$$

where $\dot{m}_{var,c,n}$ = Inlet-corrected mass flow for off schedule geometry, normalized to design point
$CVGP$ = Compressor variable geometry position value

The mass flow rate in the compressor may be given as

$$\dot{m}_{base,c,n}(1 + \dot{m}_{var,c,n}) \left[\dot{m}_d \sqrt{\frac{T_{in,d}}{T_{in}}}\right]\left(\frac{p_{in}}{p_{in,d}}\right) \tag{3-14}$$

where $\dot{m}_d$ = Design point value of compressor mass flow rate
T_{in} = Stagnation temperature at compressor inlet
$T_{in,d}$ = Design point stagnation temperature at compressor inlet
p_{in} = Stagnation pressure at compressor inlet
$p_{in,d}$ = Design point pressure at compressor inlet

When the normalized inlet-corrected mass flow rate at the design point, $\dot{m}_{base,c,n}$, is returned from the map, the value should be exactly 1.0. However, because the map values are determined by an interpolation routine (see [3], Appendix C), it may not be exactly 1.0. This error can be eliminated by introducing a correction coefficient, W_{corr}, which when multiplied by the returned map value will make the returned map value be exactly 1.0. The compressor mass flow rate correction coefficient is

$$W_{corr} = \frac{1}{(\dot{m}_{base,c,n})_{map@designpt}} \tag{3-15}$$

Placing the correction coefficient into Eq.(3-14) gives

$$\dot{m}_{base,c,n}(1 + \dot{m}_{var,c,n}) \left[\dot{m}_d \sqrt{\frac{T_{in,d}}{T_{in}}}\right]\left(\frac{P_{in}}{P_{in,d}}\right) \cdot W_{corr} \tag{3-16}$$

The normalized adiabatic efficiency value is obtained from the baseline compressor performance map.

$$\eta_{base,c,n} = f_{map}(N_{c,n}, \Delta P_n) \tag{3-17}$$

where $\eta_{base,c,n}$ = Adiabatic efficiency, normalized to design point

The adiabatic efficiency, η, is then computed by multiplying the normalized adiabatic efficiency, η_n, by the design adiabatic efficiency value, η_d:

$$\eta = \eta_n \cdot \eta_d \tag{3-18}$$

The stagnation temperature rise across the compressor is calculated using isentropic relationships for stagnation temperature and pressure. The stagnation temperature rise is determined using an average temperature which is computed using a temperature interpolation constant:

$$T_{avg} = \beta T_{in} + T_{out}(1-\beta) \quad (3\text{-}19)$$

where T_{avg} = Average stagnation temperature in the compressor
T_{out} = Compressor outlet stagnation temperature
T_{in} = Compressor inlet stagnation temperature
β = Compressor interpolation constant

T_{avg} is used to determine the constant specific heat of the compressor from curve fit data. With the constant specific heat known, the isentropic stagnation temperature rise is defined as

$$\left(\frac{T_{out}}{T_{in}}\right)_{ideal} = \left(\frac{p_{out}}{p_{in}}\right)^{(\gamma-1)/\gamma} \quad (3\text{-}20)$$

where p_{in} = Compressor inlet stagnation pressure
p_{out} = Compressor outlet stagnation pressure
γ = Compressor constant specific heat ratio

Defining the stagnation temperature rise parameter as

$$\left(\frac{\Delta T}{T_{in}}\right)_{ideal} = \frac{(T_{out,ideal} - T_{in})}{T_{in}} = \frac{T_{out,ideal}}{T_{in}} - 1 \quad (3\text{-}21)$$

then, Eq. (3-20) becomes

$$\left(\frac{\Delta T}{T_{in}}\right)_{ideal} = \left(\frac{p_{out}}{p_{in}}\right)^{(\gamma-1)/\gamma} - 1 \quad (3\text{-}22)$$

The stagnation temperature at the compressor outlet is then

$$T_{out} = \left[\frac{(\Delta T/T_{in})_{ideal}}{\eta} + 1\right] T_{in} \quad (3\text{-}23)$$

The compressor temperature correction coefficient is

$$T_{corr} = \frac{\Delta T/T_{in_{ideal@designpt}}}{\Delta T_d/T_{in,d_{ideal}}} \quad (3\text{-}24)$$

Placing the correction coefficient into Eq.(3-23) gives

$$T_{out} = \left[\frac{(\Delta T/T_{in})_{ideal}}{\eta \cdot T_{corr}} + 1\right] T_{in} \quad (3\text{-}25)$$

The enthalpy corresponding to the temperature at the compressor outlet, h_{out}, is determined from curve fit data based on values of stagnation temperature and the fuel-air ratio (f/a) in the compressor:

$$h_{out} = f(T_{out}, f/a) \quad (3\text{-}26)$$

The compressor functions by transmitting mechanical energy (supplied by the shaft) into kinetic energy in the fluid flow. The rate of this energy conversion is calculated as the difference in fluid energy at the inlet and exit of the compressor and is given by the following equation:

$$E_{comp} = \dot{m}(h_{out} - h_{in}) \quad (3\text{-}27)$$

where E_{comp} = The rate of energy being added to the working fluid
$\dot{m}$ = The mass flow rate of the working fluid in the compressor
h_{in} = The specific stagnation enthalpy of the working fluid at the compressor inlet
h_{out} = The specific stagnation enthalpy of the working fluid at the compressor outlet

E_{comp} is referred to as the compressor *energy term*, and represents the rate of energy being transferred from the shaft to the working fluid. This term is used to compute the torque applied to the compressor by the shaft, and is utilized in the dynamic energy balance equation given in the shaft mathematical model.

3.5 Duct

The effect of fluid momentum on the transient behavior of the engine is considered in the duct component. The model assumes the duct is adiabatic with constant area and length; and the pressure loss due to frictional effects is included. The following is the dynamic flow equation for the duct:

$$\frac{d\dot{m}}{dt} = \left(\frac{A}{L}\right)\left[p_{in} - p_{out} - \left(\frac{K_d \dot{m}^2 T_{in}}{P_{in}}\right)\right] \quad (3\text{-}28)$$

where A = Cross sectional area of the duct
L = Length of the duct
p_{in} = Stagnation pressure at the duct inlet
p_{out} = Stagnation pressure at the duct exit
K_d = Duct pressure loss coefficient
$\dot{m}$ = Fluid mass flow rate in the duct
T_{in} = Stagnation temperature at combustor inlet

3.6 Flight Conditions and Inlet (Environment)

Stagnation temperature and pressure at flight altitude are determined from standard atmospheric data tables:

$$p_{alt} = f(alt) \tag{3-29}$$

$$T_{alt} = f(alt) \tag{3-30}$$

where alt = Altitude
p_{alt} = Stagnation pressure at altitude
T_{alt} = Stagnation temperature at altitude
T_{sls} = Sea level standard ambient temperature

Although no inlet to the engine is included, simple isentropic and empirical relations are used to determine the stagnation conditions at the "inlet" exit. The stagnation temperature at the inlet exit based on the flight Mach number is computed as

$$T_{exit} = T_{alt}\left[1 + \frac{(\gamma - 1)\, M_{flight}^2}{2}\right] \tag{3-31}$$

where T_{exit} = Stagnation temperature at inlet exit
γ = Specific heat ratio of ambient air (γ =1.4)
M_{flight} = Flight Mach number

A steady-state, military specification inlet recovery characteristic is used to determine the stagnation pressure at the inlet exit:

$$\eta_{inlet} = 1 \quad \text{if} \quad (M_{flight} \le 1)$$

$$\eta_{inlet} = 1 - 0.075\,(M_{flight} - 1)^{1.35} \quad \text{if} \quad (M_{flight} \le 1)$$

$$p_{exit} = \eta_{inlet} p_{alt}\left(\frac{T_{exit}}{T_{alt}}\right)^{\gamma/(\gamma-1)} \tag{3-32}$$

where η_{inlet} = Inlet recovery characteristic
p_{exit} = Stagnation pressure at inlet exit

The enthalpy of the air at the flight altitude is determined from curve fit data based on the stagnation temperature at altitude and the fuel-air ratio (which is zero):

$$h = f(T_{alt},\ (f/a)) \tag{3-33}$$

3.7 Intercomponent Volume (Mixing Volume)

An intercomponent volume (mixing volume) is placed between engine components. In the volume, the storage of mass and energy occurs and the dynamic forms of continuity, energy and state equations are used to generate differential equations which can be solved for the stored mass, temperature and pressure. The following develops the dynamic equations stated above.

The rate of change of mass in the control volume is

$$\frac{dM}{dt} = \sum_{inlets} \dot{m} - \sum_{exits} \dot{m} \tag{3-34}$$

where M = Mass of working fluid in control volume
$\dot{m}$ = Fluid mass flow rate

The rate of change of temperature in the control volume is obtained from the first law of Thermodynamics as

$$\frac{dT}{dt} = \frac{\left[\sum_{inlets} \dot{m}h - h_{avg} \sum_{inlets} \dot{m} + \frac{\delta Q}{dt} - \frac{\delta W}{dt}\right]}{Mc_v} + \frac{(\gamma - 1)\,T}{M}\left[\sum_{inlets} \dot{m} - \sum_{exits} \dot{m}\right] \tag{3-35}$$

where T = Stagnation temperature of working fluid in control volume
$\delta Q/dt$ = Rate of energy entering control volume due to heat transfer
$\delta W/dt$ = Rate of energy leaving control volume due to all work interactions except flow work
h = Specific stagnation enthalpy of working fluid entering the control volume
h_{avg} = Specific stagnation enthalpy of working fluid in the control volume
u = Internal energy of working fluid
γ = Constant specific heat ratio of working fluid in control volume
c_v = Constant-volume specific heat of working fluid in control volume

At the (steady-state) design point, the above differential equation should be exactly zero, indicating that the mixing volume energy balance is satisfied at the design point. However, model incompatibilities and numerical inaccuracies may result in a non-zero temperature derivative value. To zero the derivative value associated with the energy balance, a correction coefficient, E_{corr} is introduced into the above equation to force it to zero

$$E_{corr} = \frac{\left[\left(h_{avg} \sum_{inlets} \dot{m}\right) - \sum_{inlets} \dot{m}h\right]}{\left[\frac{\delta Q}{dt} - \frac{\delta W}{dt}\right]} \tag{3-36}$$

where all values are design point values. If the $\delta Q/dt$ term is zero, then E_{corr} is set to 1.000. Once determined, the correction coefficient then becomes part of the model and the governing differential equation associated with the energy balance for the mixing volume becomes:

$$\frac{dT}{dt} = \frac{\left[\sum_{inlets} \dot{m}h - h_{avg}\sum_{inlets}\dot{m} + E_{corr}\left(\frac{\delta Q}{dt} - \frac{\delta W}{dt}\right)\right]}{Mc_v}$$

$$+ \frac{(\gamma-1)T}{M}\left[\sum_{inlets}\dot{m} - \sum_{exits}\dot{m}\right] \quad (3\text{-}37)$$

3.8 Nozzle

A convergent-divergent nozzle configuration is assumed for the nozzle model, with a convergent-only nozzle being considered a subset of the more general model. A relatively detailed mathematical representation of the thermodynamics is used, including the treatment of normal shocks in the divergent section. The stagnation pressure losses associated with the shocks are computed along with the gross nozzle thrust. The following equations define the nozzle model.

The ideal mass flow rate for a nozzle operating in "choked" condition (i.e. the Mach number at the nozzle throat is 1.0) can be determined from isentropic relations [8]. Pressure losses due to boundary layer effects and departure from one-dimensionality can be calibrated into the ideal mass flow rate above equation through the use of a flow coefficient, C_d, to obtain:

$$\dot{m} = p_{in}A^*C_d\sqrt{\frac{\gamma}{RT_{in}}}\left[\frac{2}{(\gamma+1)}\right]^{(\gamma+1)/2(\gamma-1)} \quad (3\text{-}38)$$

where p_{in} = Nozzle inlet stagnation pressure
T_{in} = Nozzle inlet stagnation temperature
R = Ideal gas constant
γ = Nozzle specific heat ratio (γ= 1.4 is used)
A^* = The critical nozzle cross sectional area

The nozzle gross thrust, F, may be calculated as

$$F = \dot{m}v_e + A_e(p_e - p_{out}) \quad (3\text{-}39)$$

where p_e = Static pressure at nozzle exit plane
p_{out} = Ambient static pressure
v_e = Nozzle flow exit velocity
A_e = Cross sectional area at nozzle exit plane

The compressible flow tables used in computing the nozzle operation parameters assume a specific heat ratio of 1.4. According to Szuch [7], the specific heat ratio in the tailpipe of a turbofan engine will be lower than this value. To compensate for this error when setting up the model to match the user-defined input data, correction coefficients are computed to give the desired values of gross thrust and mass flow rate for the nozzle. The correction coefficient for the gross thrust, F_{corr}, is the ratio of user-defined design gross thrust, F_d, to the calculated design gross thrust, F_c, given by the above equation:

$$F_{corr} = \frac{F_d}{F_c} \quad (3\text{-}40)$$

Once calculated, this value is used in the thrust equation at design and off-design conditions. and Eq.(3-39) is then

$$F = (\dot{m}v_e)F_{corr} + A_e(p_e - p_{out}) \quad (3\text{-}41)$$

The correction coefficient for the mass flow rate, W_{corr} is the ratio of user-defined design mass flow rate to the calculated design mass flow rate given by Eq.(3-38):

$$W_{corr} = \frac{\dot{m}_d}{\dot{m}_c} \quad (3\text{-}42)$$

Once calculated, this value is used in the mass flow equation, Eq.(3-38), at design and off-design conditions to give the nozzle mass flow rate

$$\dot{m} = W_{corr}p_{in}A^*C_d\sqrt{\frac{\gamma}{RT_{in}}}\left[\frac{2}{(\gamma+1)}\right]^{(\gamma+1)/2(\gamma-1)} \quad (3\text{-}43)$$

The exit velocity of the nozzle flow is determined using the following equation

$$v_e = C_v\left(\frac{v_x}{v_y}\right)\left[\frac{2\gamma RT_{in}}{(\gamma+1)}\right]^{1/2} \quad (3\text{-}44)$$

where v_e = Flow velocity at the nozzle exit
C_v = Nozzle velocity coefficient
v_x = Flow velocity upstream of shock
v_y = Flow velocity downstream of shock

Defining the critical cross-sectional area, A^*, as the area where the flow is sonic (M=1), the ratio of some arbitrary nozzle cross-sectional area to the critical area may be used to determine the pressure ratio across the nozzle which causes the flow to be sonic in the nozzle at the critical cross-sectional area. The pressure ratio which causes the flow to be sonic is known as the critical pressure ratio

$$\left(\frac{p_A}{p_{in}}\right)_{cr} = f\left(\frac{A}{A^*}\right) \quad (3\text{-}45)$$

where p_A = Ambient static pressure at location of A
p_{in} = Stagnation pressure at nozzle inlet
A = Arbitrary cross-sectional area of nozzle
A^* = Critical cross-sectional area of nozzle

The critical pressure, for a given area ratio, is determined from one-dimensional isentropic tables for compressible flow. Sonic flow will occur at the minimum

cross-sectional area of the nozzle which is at the throat, and Eq. (3-45) becomes

$$\left(\frac{p_{out}}{p_{in}}\right)_{cr} = f\left(\frac{A_e}{A^*}\right) \tag{3-46}$$

3.8.1 Subsonic Flow

If $(p_{out}/p_{in}) > (p_{out}/p_{in})_{cr}$ the flow is subsonic throughout the nozzle. In this case,

$$P_e = P_{out} \tag{3-47}$$

The area to which the throat area would have to be reduced to result in sonic flow (M=1) based on the pressure ratio value, may be determined from the one-dimensional isentropic compressible flow tables. If A_e is used as the arbitrary cross sectional area, then the ratio of the exit area to the critical cross-sectional area may be found

$$\left(\frac{A_e}{A^*}\right) = f^{-1}\left(\frac{p_{out}}{p_{in}}\right) \tag{3-48}$$

A^* is then determined using the following equation

$$A^* = \frac{A_e}{(A_e/A^*)} \tag{3-49}$$

The value of A^* may then be used in Eq.(3-43) to compute the mass flow rate in the nozzle. The velocity ratio term (v_x/v_y) is 1.0 since there is no shock. This value is substituted into Eq.(3-44) to compute the nozzle exit velocity.

3.8.2 Sonic Flow

If $(p_{out}/p_{in}) = (p_{out}/p_{in})_{cr}$ then the flow is sonic in the nozzle. In this case, the nozzle exit pressure is equal to the ambient pressure

$$p_e = p_{out} \tag{3-50}$$

Since $(p_{out}/p_{in}) = (p_{out}/p_{in})_{cr}$ the flow is sonic at the minimum nozzle cross-sectional area which is the throat. Thus, the throat area is the critical area for the nozzle and A_{throat} may be used in Eq.(3-43) in place of A^* to compute the mass flow rate in the nozzle. The velocity ratio term (v_x/v_y) is 1.0 since there is no shock. This value is substituted into Eq.(3-44) to compute the nozzle exit velocity.

3.8.3 Supersonic Flow

If the pressure ratio is less than the critical pressure ratio, $(p_{out}/p_{in}) < (p_{out}/p_{in})_{cr}$ the flow is supersonic in the nozzle. If the nozzle is a converging-diverging nozzle, normal shocks will likely be present in the flow and the pressure losses are accounted for. To determine if the nozzle is a converging-only or a converging-diverging nozzle, the ratio of the exit area to the throat area is compared.

Converging-Only Nozzle ($A_{exit}/A_{throat} = 1$)

For sonic flow in a converging-only nozzle, the throat area is the critical area for the nozzle and A_{throat} may be used in Eq.(3-43) in place of A^* to compute the mass flow rate in the nozzle. The velocity ratio term (v_x/v_y) is 1.0 since there is no shock. This value is substituted into Eq.(3-44) to compute the nozzle exit velocity. The nozzle exit plane stagnation pressure will be less than the ambient pressure. This value may be computed using the critical pressure ratio

$$p_e = p_{in}\left(\frac{p_{out}}{p_{in}}\right)_{cr} \tag{3-51}$$

This value may be used in Eq.(3-39) to compute the nozzle pressure drag.

Converging-Diverging Nozzle ($A_{exit}/A_{throat}) > 1$)

For supersonic flow in a converging-diverging nozzle, the throat area is the critical area for the nozzle and A_{throat} may be used in Eq.(3-43) in place of A^* to compute the mass flow rate in the nozzle. The converging-diverging nozzle, operating in the supersonic region, will likely have normal shocks present in the divergent portion of the nozzle or outside of the nozzle. To determine if the shock is outside the nozzle, the pressure ratio at which the shock is in the nozzle exit plane is given by the following equation

$$\left(\frac{p_{out}}{p_{in}}\right)_{es} = \frac{(p_y/p_x)\ (p_{s,y}/p_{s,x})}{(p_y/p_{s,x})} \tag{3-52}$$

where p_y = Stagnation pressure downstream of shock
p_x = Stagnation pressure upstream of shock
$p_{s,y}$ = Static pressure downstream of shock
$p_{s,x}$ = Static pressure upstream of shock
es = Expelled nozzle shock

The parameters given in the above equation may be determined from shock tables such as those given in reference 19.

Shock in exit plane

If $(p_{out}/p_{in}) = (p_{out}/p_{in})_{es}$, the shock is in the nozzle exit plane. The velocity ratio term (v_x/v_y) is determined from the shock tables based on the Mach number upstream of the shock

$$\frac{v_x}{v_y} = f(M_x) \tag{3-53}$$

The exit pressure will be returned to ambient pressure due to the shock, so

$$p_e = p_{out} \tag{3-54}$$

This value may be used in Eq.(3-39) to compute the nozzle pressure drag.

Shock external to nozzle

If $(p_{out}/p_{in}) < (p_{out}/p_{in})_{es}$, the shock has been expelled from the nozzle. The velocity ratio term (v_x/v_y) is determined from the shock tables based on the Mach number upstream of the shock

$$\frac{v_x}{v_y} = f(M_x) \tag{3-55}$$

This value is substituted into Eq.(3-44) to compute the nozzle exit velocity. The nozzle exit stagnation pressure may be computed using the critical pressure ratio

$$p_e = p_{in}\left(\frac{p_{out}}{p_{in}}\right)_{cr} \tag{3-56}$$

This value may be used in Eq.(3-39) to compute the nozzle pressure drag.

Shock internal to nozzle

If $(p_{out}/p_{in})_{cr} > (p_{out}/p_{in}) > (p_{out}/p_{in})_{es}$, the shock is in the divergent section of the nozzle. In this case, the nozzle exit pressure is equal to the ambient pressure

$$p_e = p_{out} \tag{3-57}$$

This value may be used in Eq.(3-39) to compute the nozzle pressure drag. The velocity ratio term (v_x/v_y) is determined from the shock tables based on the Mach number upstream of the shock

$$\frac{v_x}{v_y} = f(M_x) \tag{3-58}$$

This value is substituted into Eq.(3-44) to compute the nozzle exit velocity.

3.9 Shaft

The most significant factors in determining the transient behavior of a turbojet are the spool moments of inertia. The spool is considered the complete compressor-shaft-turbine assembly. The differential equation representing the change in spool speed is

$$\frac{dN}{dt} = \left(\frac{30}{\pi}\right)^2 \frac{J}{I} \frac{1}{N} \left[\sum E_{comp} + \sum E_{turb}\right] \tag{3-59}$$

where I = Polar moment of inertia of the spool(compressor+shaft+turbine)

J = Joulean mechanical equivalent of heat constant

N = Shaft rotational speed

ΣE_{comp} = The summation of the energy flux term, $\dot{m}\Delta h$, for each compressor attached to the shaft

ΣE_{turb} = The summation of the energy flux term, $\dot{m}\Delta h$, for each turbine attached to the shaft

At the (steady state) design point, the above differential equation should be exactly zero, indicating that the spool energy balance is satisfied at the design point. However, model incompatibilities and numerical inaccuracies may result in a non-zero spool derivative value. To zero the derivative value associated with the energy balance, a correction coefficient, E_{corr}, is introduced into the above equation to force it to zero

$$E_{corr} = \frac{\sum E_{comp@designpt}}{-\sum E_{turb@designpt}} \tag{3-60}$$

Once determined, the correction coefficient then becomes part of the model and the governing differential equation associated with the energy balance for the spool is

$$\frac{dN}{dt} = \left(\frac{30}{\pi}\right)^2 \frac{J}{I} \frac{1}{N} \left[\sum E_{comp} + (E_{corr} \cdot \sum E_{turb})\right] \tag{3-61}$$

3.10 Turbine

Turbine performance is represented by a set of overall performance maps normalized to design point values. Baseline performance maps provide normalized turbine inlet flow and normalized enthalpy drop parameters as a function of normalized pressure ratio and normalized inlet-corrected spool speed. Cooling bleed flow for the turbine is assumed to reenter the cycle at the turbine discharge, although a portion of the bleed flow is assumed to do turbine work.

The normalized turbine inlet flow parameter value is obtained from the turbine performance map.

$$(M_p)_n = f_{map}(N_{c,n}, \Delta p_n) \tag{3-62}$$

where $(M_p)_n$ = Inlet-corrected flow parameter normalized to design point

$N_{c,n}$ = Inlet-corrected spool speed normalized to design point

Δp_n = Pressure ratio normalized to design point

The inlet mass flow rate in the turbine is

$$\dot{m} = (M_p)_n \left[\dot{m}_d \left(\frac{T_{in,d}}{N_d P_{in,d}}\right)\right]\left(\frac{N p_{in}}{T_{in}}\right) \tag{3-63}$$

where p_{in} = Stagnation pressure at turbine inlet
T_{in} = Stagnation temperature at turbine inlet
N = Spool speed
$\dot{m}_d$ = Design point turbine mass flow rate
$p_{in,d}$ = Design point stagnation pressure at turbine inlet
$T_{in,d}$ = Design point stagnation temperature at turbine inlet
N_d = Design point spool speed

When the normalized inlet flow parameter at the design point, $(M_p)_n$, is returned from the map, the value should be exactly 1.0. However, because the map values are determined by an interpolation routine (see [3], Appendix C), it may not be exactly 1.0. This error can be eliminated by introducing a correction coefficient, W_{corr} which when multiplied by the returned map value will make the returned map value be exactly 1.0:

$$W_{corr} = \frac{1}{((M_p)_n)_{map@designpt}} \tag{3-64}$$

Placing the correction coefficient into Eq.(3-63) gives the mass flow rate in the turbine

$$\dot{m} = (M_p)_n \left[\dot{m}_d \left(\frac{T_{in,d}}{N_d P_{in,d}}\right)\right]\left(\frac{N p_{in}}{T_{in}}\right) \cdot W_{corr} \tag{3-65}$$

The normalized enthalpy drop parameter, $(h_p)_n$, is obtained from the turbine performance map.

$$(h_p)_n = f_{map}(N_{c,n}, \Delta p_n) \tag{3-66}$$

The enthalpy drop in the turbine, Δh, is found by the following equation,

$$\Delta h = (h_p)_n \cdot N\sqrt{T_{in}}\left[\frac{\Delta h_d}{N_d\sqrt{T_{in,d}}}\right] \tag{3-67}$$

where Δh_d = Design enthalpy drop across turbine

When the normalized enthalpy drop parameter at the design point $(h_p)_n$, is returned from the map, the value should be exactly 1.0. However, because the map values are determined by an interpolation routine (see [3], Appendix C), it may not be exactly 1.0. This error can be eliminated by introducing a correction coefficient, H_{corr} which when multiplied by the returned map value will make the returned map value be exactly 1.0,

$$H_{corr} = \frac{1}{((h_p)_n)_{map@designpt}} \tag{3-68}$$

Placing the correction coefficient into Eq.(3-67) gives

$$\Delta h = (h_p)_n \cdot H_{corr} \cdot N\sqrt{T_{in}}\left[\frac{\Delta h_d}{N_d\sqrt{T_{in,d}}}\right] \tag{3-69}$$

The rate of energy being removed from the working fluid by the turbine is

$$E_{turb} = [\dot{m}_{in} + (\dot{m}_{bleed} \cdot bldpct)]\,\Delta h \tag{3-70}$$

where E_{turb} = Rate of energy being removed from the working fluid
$\dot{m}_{inlet}$ = Mass flow rate of the working fluid at the turbine inlet
$\dot{m}_{bleed}$ = Mass flow rate of the cooling bleed-flow
$bldpct$ = Percentage of bleed flow which does turbine work

E_{turb} is the energy flux term for the turbine, and is referred to as the turbine *energy term*. It is used to compute the torque applied to the shaft by the turbine and is utilized in the energy balance equation given in the shaft mathematical model. Because the energy removal process is assumed to take place in the downstream mixing volume, E_{turb} is the rate of energy being removed from the working fluid in the mixing volume, and is given by the $\delta Q/dt$ term in the mixing volume dynamic energy equation (the $\delta Q/dt$ term will be negative). Thus, E_{turb} will be used by the mixing volume to account for the energy removed by the turbine from the working fluid.

4.0 TESS Operating Paradigm

The sequence of computational steps used to simulate the propulsion system is termed the simulation's operating paradigm. The TESS operating paradigm was derived from the DIGTEM turbofan engine simulation code [9], and consists of four steps:

1. Establish the engine system configuration. Since the engine configuration in TESS is user-defined it is not known until the simulation is started, and can only be determined after the simulation has begun. (Note: This step is implicit in the DIGTEM code because that code uses a static engine configuration.)
2. Correction coefficients are evaluated, based on user defined design-point data, to balance the engine at the design-point.
3. The steady-state (balanced) engine at the user-defined initial operating point conditions is determined.
4. The transient simulation is then begun from the converged steady-state conditions just calculated. Based on user-defined open-loop transient control schedules in the compressor, combustor, nozzle and environment components, the transient is run through the user-defined time range.

In this Section, the operating paradigm will be discussed in detail in preparation for the discussion of how it is implemented within the AVS system which is covered in Section 5.

4.1 Applying the Governing Equations

The traditional approach to building a simulation system is to reduce the problem to a set of coupled differential equations

$$\frac{dx}{dt} = f(x(t), t)$$

subject to initial conditions

$$x(t=0) = g$$

where x is a vector containing the time dependent state variables for the system. The system of equations is then solved using numerical techniques.

This approach was the fundamental approach used in TESS. As shown in Section 3, differential equations in the duct, mixing volume and shaft components of the propulsion system are used to represent the conservative laws of energy and mass. Once an engine system has been established by the user, the differential equations for each of the duct, mixing volume and shaft components are used to form the system of differential equations for the system as described above. The number of equations in the system is not fixed, but rather is arbitrary and completely dependent on the number of duct, mixing volume and shaft components within the engine model which the user has constructed. The system of differential equations describing the engine system are then used to determine the steady state conditions of the engine and to determine the engine conditions at during a transient analysis.

At the steady-state engine condition, the conservative laws are satisfied and there is no change in the state values with respect to time (i.e., the *derivative values*, d**x**/dt, are zero). Thus, for steady-state, the system of equations being solved is

$$\frac{dx}{dt} = f(x(t), t) = 0$$

In the transient analysis, x is a vector containing time-dependent state variable. The differential equations are evaluated to determine the values of the derivative terms, dx/dt, which are then used to compute the state variables, x, at the end of some discrete time step. This process is repeated throughout the transient time domain.

4.1.1 The Components Execution Sequence

In order to determine the steady-state balanced engine, or perform a transient analysis, the differential equations used by the shaft, duct, and mixing volumes, must be evaluated. These, however, cannot be evaluated directly as they are dependent on data computed by other components in the system. This input data needed by the shaft, duct, and mixing volumes, must be determined prior to applying the differential equations in these components.

Due to the methods used to apply the equations governing the operations of each of the engine components, there is a specific sequence of execution for the engine components. This sequence is comprised of the following seven steps.

Step 1 - The stored mass[1] and temperature at the physical component boundaries are set. As described in Section 3, the boundaries between the physical components are modeled by the intercomponent volumes (mixing volumes). Also, the spool speed in the shaft components, and the mass flow rate in the duct components are set. These values are considered "known" values, either as initial guesses, or as having been solved for numerically.

Step 2 - The pressure at the boundaries between the

1. The user does not actually specify the stored mass as a design point or initial operating point value. Instead, the temperature, pressure and volume are specified in the mixing volume. These are used in the ideal gas law to give the stored mass of the mixing volume. Aside from the initial computation, in which the pressure has been defined by the user, the stored mass, temperature, and volume are used to determine the pressure (see step 2). Stored mass is used as the independent variable instead of pressure because the continuity equation is used to solve for the derivative of stored mass in the mixing volume.

physical components (in the mixing volumes) are computed. The values of temperature, stored mass (defined in Step 1) and the user-defined value of volume (a constant), are used with the ideal gas law to solve for the pressure.

Step 3 - All physical engine components which are associated with the flow of the working fluid through the engine (bleed, combustor, compressor, nozzle, turbine)[2] compute their mass flow rates. Mass flow rate in the physical components is determined from either one-dimensional flow equations (bleed, combustor, nozzle), or from empirical data performance maps (compressor, turbine). In each case, the values of temperature and pressure at the component boundaries (and spool speed in rotating components), which were defined in Step 1, are used to determine the mass flow.

Step 4 - All components in the engine compute the fuel-air ratio, f/a, of the flow in the component (obviously, those components which do not deal with fluid flow, such as the shaft component, will not compute a value for f/a). The f/a ratio of the working fluid is needed, along with the temperature, to compute the enthalpy of the fluid. The fuel-air ratio in a component is dependent on the f/a and mass flow rate of the fluid entering the component at its inlet boundary and on whether the component adds fuel to the fluid flow (e.g., a combustor component).

The fact that the fuel-air ratio of a component is dependent on properties of the flow entering at its inlet is significant because it requires that before a component can compute its f/a value, those component(s) which pass fluid into the component must already have computed their values of mass flow rate and f/a. This is illustrated in Figure 4.1. Consider engine components C1, C2, and C3. C1 and C2 are components which pass the working fluid into C3. Components C1 and C2 must have already determined their f/a and mass flow rate values prior to C3 attempting to compute its f/a value as C3 needs that information to compute its own fuel-air ratio. This requires that when computing the f/a in each of the components in the system, the components must determine their f/a in a sequential order following the direction of the fluid flow through the engine.

Step 5 - All flow components compute the enthalpy of the working fluid. Enthalpy is determined from curve-fit data based on values of temperature and f/a. Temperature was defined in Step 1 and f/a defined in Step 4.

Step 6 - Compute *energy terms* in some components. Compressor, combustor, and turbine each compute energy terms (see Section 3). To compute the compressor energy terms, the f/a and enthalpy of the fluid entering the compressor and the temperature at the compressor exit, must be known. These were computed in Steps 1, 4, and 5, respectively. The combustor energy term is dependent only on the fuel mass flow rate, which is a user-defined value and is always known. The turbine energy term is dependent on the mass flow rates of the flow into the turbine and the bleed component flow which is used to cool the turbine blades. This requires that the bleed mass flow rate used to cool the turbine be determined prior to computing the turbine energy term.

Step 7 - All information necessary to evaluate the differential equations has been computed in the preceding Steps. The shaft, duct and mixing volume components can now compute the derivatives of spool speed, mass flow rate, temperature and stored mass.

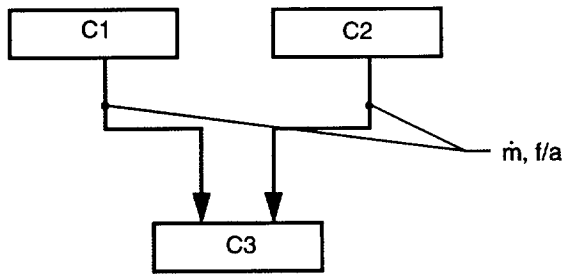


Figure 4.1 - Passing the Values of Fuel-air Ratio and Mass Flow Rate

4.1.2 "Running the Engine"

When performing an engine transient or attempting to balance the engine at the initial operating point, the differential equations used to describe the dynamics of the engine must be computed. As stated above, in order to correctly compute the fuel-air ratio, the engine components must execute in sequential order in the direction of working fluid flow through the engine.

Furthermore, using this sequential execution, it is possible to combine several of the operational steps described above. If the sequence is followed, it is possible to combine Steps 3, 4, 5, and 6 into a single computational step. If each component executes, computing its mass flow rate, f/a, enthalpy and energy terms, in a sequential order following the direction of the fluid flow through the engine, the information will be correct. This is illustrated in Figure 4.2. Consider engine components C4, C5, and C6. C4 and C5 are components which pass the working fluid into C6. Components C4 and C5 have already determined their mass flow rate, f/a, enthalpy and energy values prior to C6 attempting to compute its mass flow rate, f/a, enthalpy and energy value. Thus, when C6 executes, it will have the data necessary to carry out its computations correctly.

2. Although the duct component is considered a physical engine component, it is omitted from this list because it does not compute its mass flow rate. Mass flow rate is the independent (state) variable in the differential equation for the duct, and as such is determined numerically by the equation solver

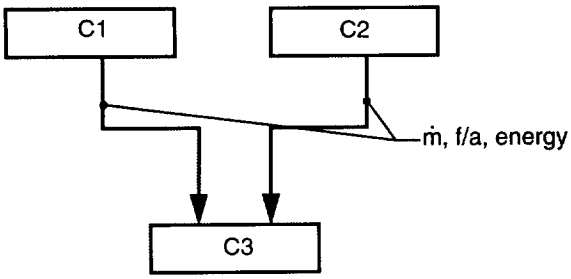


Figure 4.2 - Combined Computational Steps 3, 4, 5, and 6

The seven steps described above can be reduced to three sequential execution of components which is termed a pass through the engine.

Pass 1 - Based on known values of temperature and stored mass in each of the mixing volumes in the engine, the pressure in the mixing volume is computed (see Section 3, Intercomponent volumes). This step provides the temperature and pressure at the boundaries of the physical engine components.

Pass 2 - All engine components compute the f/a and enthalpy of the working fluid within the component. This must be done in the direction of fluid flow propagation in the engine, as the f/a ratio of a component is dependent on the flow into the component from the upstream module(s). Thus, the execution will follow the flow path lines from the first component to the last. Since mechanical components (shafts) don't deal with flow, they do not execute at this time.

Physical components, in addition to computing f/a and enthalpy of the working fluid, compute their mass flow rate excluding ducts for which the mass flow rate is a known value. Turbine and compressor components use steady-state performance maps to compute mass flow rate, using the known value of shaft speed, and the pressure and temperature at the physical component's boundaries provided by the mixing volumes as described in Pass 1. Other components use one-dimensional, isentropic relationships to determine the mass flow rate also based on the pressure and temperature at the physical component's boundaries.

Some physical components (compressor, turbine, combustor) compute energy terms at this time. These terms are used by other engine components to compute the differential equations (see Section 3) which are used to define the system.

Pass 3 - The mixing volume, shaft and duct components generate their respective derivative values. The order of execution is not specific in this step as the needed information has already been computed in the previous steps.

Mixing volumes compute the derivatives of the state values of stored mass and temperature by applying the conservation of mass and energy equations to the flow in the mixing volume component. The values of mass flow rate, enthalpy and energy source terms from each of the upstream physical components and the mass flow rate out of the mixing volume, which is the sum of the mass flow rates in the physical components at the mixing volume's exit boundary, are used to compute the derivative values. The shaft components compute the derivatives of the state value of spool speed after obtaining the energy terms (computed in Pass 2) from the compressor and turbine components which are attached at the shaft components interfaces. The duct components compute the derivative of the state value of mass flow rate based on the pressure and temperature at the component boundaries. These values are provided by the mixing volumes at the boundaries.

These three passes through the engine are collectively known as running the engine, as now all of the components have computed all necessary information about the state of the engine.

4.2 Computing Correction Coefficients

The second step in the TESS Operation Paradigm is the determination of the Correction Coefficients. It is assumed that the design values for each component in the system are available, and furthermore, that those values result in a balanced system. If for some reason (either because of poor input data or numerical inaccuracies in the computation), the design point data does not result in a balanced engine, correction coefficients are computed which cause the engine to be balanced at the steady-state design point. These values are then used in the governing equations for both on-design and off-design point operation.

The process of determining the correction coefficients is similar to running the engine, with the following exceptions:

- The design point values of temperature and pressure in the mixing volumes are already known, having been defined by the user. Thus, it is not necessary to do the operations stated in Pass 1.
- It is not necessary to evaluate the differential equations in Pass 3, as the derivative values are not needed to compute the correction coefficients.

Thus, only the computations carried out in Pass 2 are required. Once the necessary values are computed, they are compared with the user-defined design point values to determine the correction coefficients.

4.3 Determining Steady-State Engine Balance

Before proceeding to the transient analysis, TESS first determines the steady-state, balanced engine conditions at the initial operating point. As described above, the system is considered to be balanced at steady-state when the state variables for the system (contained in the x vector) are no longer changing with time:

$$\frac{dx}{dt} = f(x((t), t)) = 0$$

The state variable values at the initial operating point are input data which is supplied by the user. These are the initial operating point values of temperature and pressure in the mixing volumes, spool speed in the shafts and mass flow rate in the physical components. The state variable values defined by the user are then used to run the engine and determine the derivative values. If all of the derivatives are within a user-defined tolerance value of zero, the system is considered to be balanced at the initial operating point. If, however, the derivatives are not near enough to zero, a numerical balancing routine is applied to determine the state values for the system which force the derivatives to zero. TESS currently utilizes two numerical techniques to achieve steady-state balance of the engine: the Newton-Raphson method and a Fourth-order Runge-Kutta method. These methods and their implementation in TESS are described in Appendix B of [4].

4.4 Transient Engine Analysis

Once the engine has been balanced at the initial operating point, the transient analysis may begin. As described above, the system of differential equations

$$\frac{dx}{dt} = f(x(t), t)$$

are evaluated to determine the derivative values, dx/dt, which are then used to compute the state variables, x, at the end of some discrete time step. In the transient, the engine is run, in time, away from its initial operating point by using open-loop engine controls (e.g, combustor fuel flow rate). These controls, which are defined by the user using transient control schedules (see Figure 4.3), provide linear, parametric control of certain engine variables. (See Appendix D of [4] for more information). The simulation is continued until the user-defined transient end time is reached.

TESS currently includes four numerical methods for solving the system of equations during the transient: Improved Euler, Fourth-order Runge-Kutta, Adams, and Gears method. The latter two are part of a ordinary differential equation solver package developed by Livermore Laboratories (see Appendix B of [4] for more information).

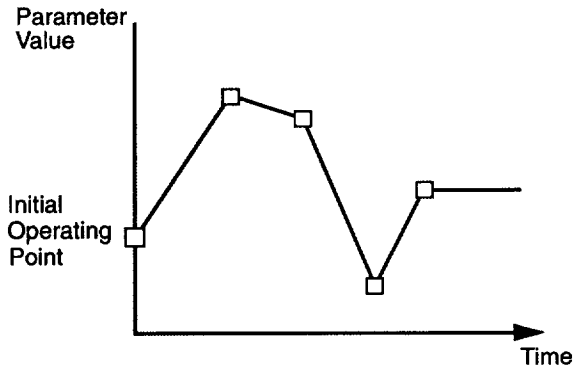


Figure 4.3 - Graphical representation of Transient Control Schedule

5.0 Implementation of Operating Paradigm Within AVS Framework

In this Section, the methods of implementing the propulsion system analysis code operating paradigm (described in Section 4) within the graphical user interface provided by AVS is discussed. Two possible techniques to implement the AVS system as the Graphical User Interface (GUI) for the engine code were considered.

In the first case, AVS would be used only as a graphical pre/post processor to the propulsion system analysis code. The propulsion system analysis code would exist as a separate code hidden from the user beneath the GUI. The user would use the AVS graphical interface to define the engine system and input data for the components in the model. That data would then be sent to the propulsion system analysis code, and the simulation would be executed. Any output from the code would be made available to the AVS system to provide the user with a graphical representation of the output data.

This approach is a traditional method used to make batch engine codes easier to operate. However, to provide for arbitrary engine configurations, the engine analysis code must be capable of handling multiple instantiations of component objects and encapsulating the data for each object. This can be handled in a number of different ways. The most efficient is to use of an object-oriented programming language which by its very nature provides the capability for creating multiple instances of an object which encapsulate their data. Another method is to put each instance of an object in a separate UNIX process. A third, less elegant and more complex method, as far as programming, is a code translator. In this method, a program uses the input data and engine configuration provided by the GUI to write a code which reflects that engine system model. Objects are instanced by creating copies of their programming code in different subroutines (see reference 12) into a file. The file is then compiled into an executable file which may later be executed.

In the second technique for implementing the AVS system as the GUI for the propulsion system analysis code, the engine analysis code is integrated with the AVS system. The propulsion system analysis code and the GUI provided by AVS are coupled so that the operation of each of the codes interacts with the other. The engine analysis code for each component object is placed within an AVS module along with the code necessary to allow it to function within AVS. The user utilizes the AVS graphical interface to define the engine system and input data for the components in the model. Data input from the user to the engine code and data output from the engine code to the user is fully integrated with the graphical tools provided by AVS.

Because the AVS system creates instances of objects which run within separate processes, it provides, by design, the capability of creating multiple instances of an object which encapsulate their data. There is no need to use an object-oriented language - the relatively more common non-object-oriented languages of FORTRAN and C may be used; and, there is no need to employ a code translator. For this reason, the second technique, utilizing full integration of the propulsion system code within AVS, was used in developing TESS.

However, because the engine code is fully integrated within AVS, which does not provide complete control over all aspects of its operation, this creates difficulties in implementing the operating paradigm of the propulsion system analysis code. This includes difficulties in the areas of:

- Scheduling module execution
- Passing messages to modules
- Determining how the system is connected
- Integrating data among several modules

The solutions to these problems are discussed in this Section.

5.1 System Execution Control

One of the more difficult aspects of integrating the propulsion system analysis code within the framework of the AVS system was the inability to explicitly control execution of the AVS modules within the engine network according to the operating paradigm. As described in Section 2, AVS was developed primarily to take data, manipulate them and render them visually to the screen. In most image processing applications, the data are created and then manipulated in a sequential order to generate output data which is displayed. This type of process generally requires that data flow from top to bottom (see Figure 5.1). Once the first module (i.e. the one at the top of the Workspace) in the network has executed, each downstream module will execute successively as it receives new input data from the upstream module.

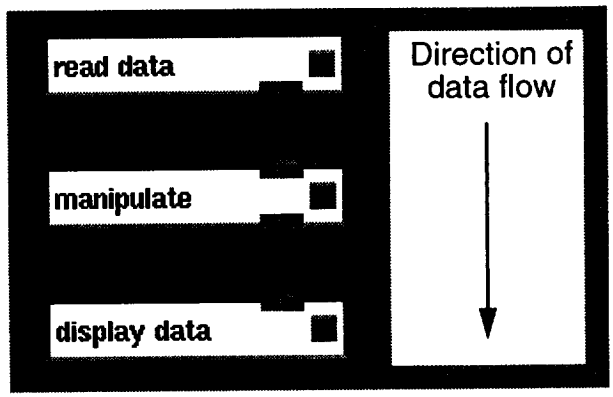


Figure 5.1 - Flow of Data in an AVS Visualization Application

However, if an application requires that certain modules execute prior to other modules and they are not connected in sequence, a problem exists. This is a

feature inherent in modeling turbine systems. Consider the core section of a gas-turbine propulsion system as shown in Figure 5.2.

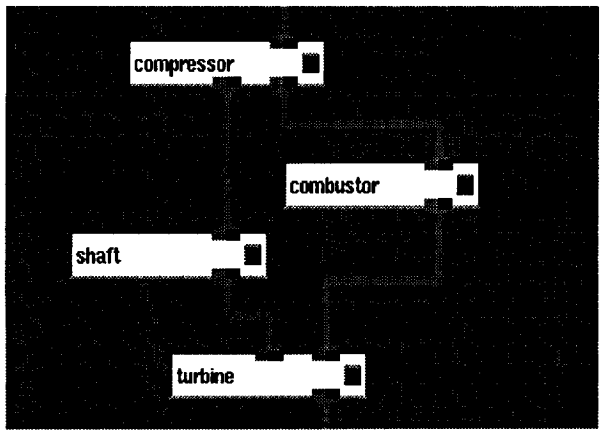


Figure 5.2 - Representation of Turbine Propulsion System Core Section

In the mathematical model representing compressor-shaft-turbine assembly, which is known as the spool, the differential form of the energy conservation balance for the spool is (Eq. 3-61):

$$\frac{dN}{dt} = \left(\frac{30}{\pi}\right)^2 \frac{J}{I} \frac{1}{N} [\sum E_{comp} + (E_{corr} \cdot \sum E_{turb})]$$

To evaluate this equation requires that the compressor energy term (E_{comp}) and the turbine energy term (E_{turb}) be known. These values can only be known if the compressor module and turbine module both execute prior to the shaft module. However, in the AVS system shown in Figure 5.2, the modules will execute in the following sequence: compressor first, then the combustor[1], shaft and finally the turbine. Clearly, when the shaft component computed the derivative value of the spool speed (dN/dt), the result would be erroneous, as the turbine energy term would not yet be computed.

For developing a propulsion system simulator in which the engine configuration being modeled is static, it is not difficult to write a simulation program so that engine computations occur in the correct order. For the above example, it would be very easy to, say, call the compressor and turbine subroutines to compute their respective energy terms before calling the shaft subroutine to compute the speed derivative.

For a propulsion system simulator in which the engine configuration is not known until the simulation begins executing, the solution is not quite as simple. For the simulation to execute properly, it is necessary that some controller exist which can direct the simulation by executing the correct code at the appropriate time. Furthermore, in a simulation where the components in the engine are treated as generic objects, there will possibly exist multiple instances of an engine component. Each instance of an object will have the same code and operate identically, but will have different data associated with it.

A relatively simple solution to the problems encountered in a dynamic system simulation with multiple instances is to use an object-oriented programming (OOP) language. This would allow multiple instances of common pieces of code (objects) and provides the ability to schedule the execution of code correctly. One of the reasons for using AVS was that it allowed having multiple copies of pieces of code (much like multiple instances of objects in an OOP code) without using an object oriented code. Unlike an OOP language, however, with AVS there is no convenient method to pass messages to a specific module and direct it to perform the required operation. For example, in an OOP language, the system controller could send a message to the instance of the compressor object called LPC, telling it to compute the mass flow rate and return that value simply by executing a message such as LPC *getmassflow*. Using AVS it is not possible to send such a simple message directly.

In lieu of being able to pass messages directly to modules within AVS, a different approach was applied to the problem. Instead of a calling routine directing an object to perform a certain function, each of the engine component modules are designed to perform certain computational operations when directed by a system controller module (SYSTEM). The difference being that the SYSTEM module will send a message to perform some function to the *entire* network of modules. Those modules which are programmed to respond to the message sent by the SYSTEM module will do so, and those which are not designed to respond to the message will simply ignore the message. In this way, the modules themselves have some built in intelligence about when to execute, and the system controller simply sends the correct messages in the order as determined by the operating paradigm described in Section 4.

The advantage of this scheme is that it utilizes the way in which AVS was designed to operate. If the message from the SYSTEM module is sent along the connecting wires connecting each module, then all modules in the network will receive the message. Utilizing a coroutine module as the system controller

1. The order of execution for the shaft and combustor modules is as follows: Once the compressor has finished executing, the combustor and shaft would receive new input data from the compressor. The AVS Flow Executive would then schedule them both to execute. The order of execution is dependent on the position of the modules in the Workspace. As stated above, module execution is from the top-down in sequence along the data connecting line(s). For modules which have a common upstream module, the module which is nearer the top of the Workspace will generally execute first. In this case, it would be the combustor. It may also be noticed that if the shaft finished operating prior to the combustor, the turbine could possibly begin operating as its input (from the shaft) would have been changed. Thus, the turbine would execute without the data from the combustor an produce erroneous results. This possibility is discussed in section 5.2.1

module provides the ability to send a message to the network and then wait until all of the modules in the network have responded to that message.[2]

5.1.1 TESS Message Passing Under AVS

The problem of sending messages to the various objects in the system in the correct sequence was solved by creating a system controller object which passed integer values to all of the modules in the system. However, there still existed a problem of sending the actual messages. It was decided to use integer values to represent the message being sent to the engine network.[3] The integer values do not represent a single message, but rather are used to trigger a set of computations in certain engine components. The system controller module (SYSTEM) sends an integer value (which is contained in the NSYS variable) to each module in the network, to control the sequence of operations in the simulation. Different NSYS values are sent depending on what function is need to be performed. Each type of module (compressor, turbine, duct, etc.) responds to the different NSYS values according to its specific programming. In this way it is possible to control which modules execute, in what order they execute, and what function they perform.

This method, while not object-oriented, retains the object-oriented concept of encapsulation. The module still remains a "black-box" which responds to messages sent to it and returns the desired values. The difference now is that instead of text-base messages, the messages are in the form of integer numbers.

5.2 Operating Paradigm Implementation Under AVS

The operating paradigm used by TESS was described in Section 4. The following sections describe how the various engine operations are controlled through the passing of the various messages (NSYS values) to the engine network. Table 5.1 lists the NSYS values used to control engine operation in the TESS modules and a short description of the corresponding engine operation.

5.2.1 Engine System Connectivity (NSYS values 0, 1, and 9)

One of the more difficult aspects of utilizing AVS to control the graphical construction of an engine model

NSYS Values	Engine Operation
0	Establish Data Struct Pointers
1	List All Engine Components
2	Define Design Pt Data to the System
3	Compute Correction Coefficients (except shaft)
4	Compute Shaft Correction Coefficients
5	Third part of Pass through the Engine
6	Second part of Pass through the Engine
7	First part of Pass through the Engine
8	Define Operating Pt Data to the System
9	Determine Upstream Components
10	Define Operating Pt Data to the System

Table 5.1 - NSYS Values and Engine Operations

was how to determine the *connectivity* between engine component modules (i.e. how the engine component modules are connected to one another). This information was needed because engine components must be able to access information computed by other engine components. For example, in order for the shaft module to apply an energy balance, it must determine the energy terms from the compressor and turbine which are attached to the shaft. This requires that the shaft "know" to which compressor and turbine component it is connected

An obvious method to determine the connectivity of the engine model would be to have the user use the mouse pointer on the display screen to "draw" lines between the modules, (see Figure 5.3). These lines would indicate either physical connections (such as between a shaft and a compressor), or paths of fluid flow (such as between a compressor and combustor) between engine components. Once drawn, the lines would establish which module is connected to which other module and the system would be defined.

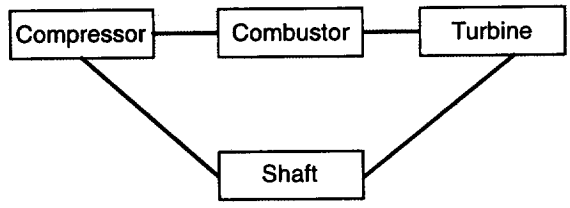


Figure 5.3 - Connecting Lines Between Components

In most graphical user interfaces, the process of drawing the lines would involve having the user use a

2. A module executes even if the message the SYSTEM module has sent is one which the module does not respond to. This is necessary to ensure that the data being sent along the connecting wires is available to the next module in the system. If the module did not execute, the downstream modules would have no input data and the simulation would be disrupted.

3. Text strings could have been used as the message format. In an earlier development phase of TESS, only integers were being passed between modules, so integers were used as the message format.

mouse pointer to select modules to be connected. The location of the modules selected by the mouse pointer on the screen would be determined by the windowing system. This information could then be compared to the known location of each of the engine component objects on the screen, to determine which of the objects the user had selected. In this way, the selected module and the modules it was connected to could be established and a connecting line could be drawn.

In AVS, the above process is also used: modules on the screen are connected by selecting a module with the mouse pointer and "dragging" the pointer to the module to be connected. This causes a line to be drawn between the modules to show that they are connected. Unfortunately, the information used to draw the line is not accessible by the user or the programmer, as it is hidden within the AVS software. Thus, it is not possible to determine which modules have been selected and how they are connected[4]. Another approach, which takes advantage of AVS's operational characteristics, was used instead to determine the system connectivity. The connectivity of a engine model can be defined by three sets of information: 1) a list of each of the components in the simulation, 2) for each component in the system, a list of the components which are connected to its input ports (i.e., it's upstream components), and 3) for each component in the system, a list of the components which are connected to its output ports (i.e., it's downstream components). The process of obtaining this information involves directing each component to perform specific actions by sending each component a series of NSYS values as described below.

NSYS value of 0 - Once an engine model has been configured (see Figure 5.4 for an example network), and the user begins the simulation, the SYSTEM module immediately begins running and sends an NSYS value of 0 to the network. This is necessary to allow the data passing mechanism to initialize itself. Within each module, AVS utilizes integer pointers to return the address of the data struct within memory for each of the input and output ports on the module. Without this, it is possible that some modules would return null pointers which would cause errors in the initialization process. The NSYS value of 0 is used since this value triggers no action in any of the modules.

NSYS value of 1 - Upon completion of execution of each module in the network, the SYSTEM module then opens the file comp.list.no. This file, when completed, will contain a list of all the components in the network (except the SYSTEM-END module). The SYSTEM module is always the first module to list itself in comp.list.no and lists the following: its component number (always 1), component name (system), component type (syst), component code (00), and solver flag (0). These terms are described below:

- The component number is a unique number which identifies each of the components connected in the network.
- The component name is a unique, user-defined, character label (10 characters maximum). It is used to distinguish multiple instances of a module (e.g. to distinguish between the high and low pressure turbines, they could be given the name HPT and LPT, respectively).
- The component type is a four character label used to identify a module's type.
- The component code is a 4 digit integer value also used to identify a module's type.
- The solver flag defines whether the component generates derivative terms. Zero (0) is false, one (1) is true. Mixing volume, shaft, and duct modules generate derivative terms so their solver flag value is 1; all other modules are zero. (See Figure 5.5).

After SYSTEM has completed writing to comp.list.no, it sends an NSYS value of 1 to the network. Each module in the network then responds with the following action:

1. It reads the file comp.list.no to determine the number of components already listed in the file.
2. Upon reaching the end of the file, the component determines its component number and appends its component number, name, type, and code to the comp.list.no file.

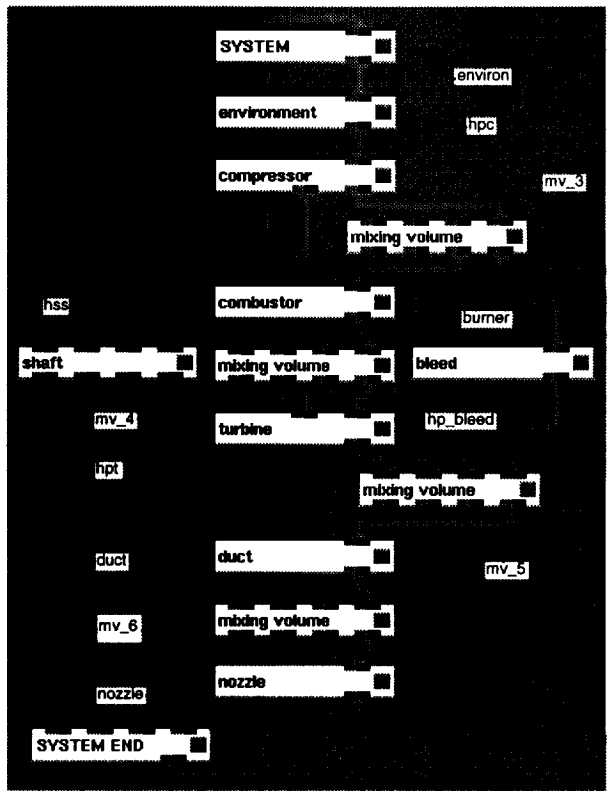


Figure 5.4 - Example Propulsion System Network

4. It is now possible to determine the connectivity of the modules using the AVS Command Language Interface using the *net_show* command. At the time TESS was developed, this option was not available and thus was not implemented.

For example, if the DUCT component shown in the example network of Figure 5.4, was executing in response to an NSYS value of 1, it would read the comp.list.no file to determine its component number. Figure 5.6 shows what comp.list.no would look like at that time. DUCT would determine that ten of the components in the network had already executed and listed themselves in comp.list.no, therefore the DUCT component number would be 11. It would then write it's component number (11), name (duct), type (duct), component code (77) and solver code flag (1) in the file.

MODULE	TYPE	CODE	FLAG
System	syst	0	0
Environment	envr	11	0
Compressor	comp	22	0
Mixing Volume	mxvl	33	1
Combustor	comb	44	0
Shaft	shft	55	1
Turbine	turb	66	0
Duct	duct	77	1
Bleed	bled	88	0
Nozzle	nozz	101	0

Figure 5.5 - TESS Module Types and Codes

3. The component sets the value of its component number to the variable NCOMP which is a static variable and will be retained by the module after it has completed executing.
4. The component sends the NSYS value of 1 to the downstream component.

After the network has completed executing, the comp.list.no file contains a description of each module in the network as shown in Figure 5.7.

NSYS value of 9 - Next, the SYSTEM module opens the file comp.list.up and writes a header to the file. This file, when completed, will contain a list of the upstream component(s) connected to each of the components in the network (except the SYSTEM-END module). The SYSTEM component then sets the NAUXVAL variable equal to its component number (which is always 1). SYSTEM then sends an NSYS value of 9 to the network to which each module in the network responds with the following action:

1. It receives input data from the upstream module(s) and extracts the component number of the upstream component from the NAUXVAL variable. The component number, name, type, and upstream component number are appended to the comp.list.up file. If the component has multiple inputs, input is checked for each of its upstream components and they are also appended to the comp.list.up file.

COMP. NO.	COMP. NAME	COMP. TYPE	COMP. CODE	FLAG
1	system	syst	0	0
2	environ	envr	11	0
3	hpc	comp	22	0
4	hss	shft	55	1
5	mv_3	mxvl	33	1
6	hp_bleed	bld	88	0
7	burner	comb	44	0
8	mv_4	mxvl	33	1
9	hpt	turb	66	0
10	mv_5	mxvl	33	1

Figure 5.6 - Contents Of comp.list.no file As Read By DUCT Component

COMP. NO.	COMP. NAME	COMP. TYPE	COMP. CODE	FLAG
1	system	syst	0	0
2	environ	envr	11	0
3	hpc	comp	22	0
4	hss	shft	55	1
5	mv_3	mxvl	33	1
6	hp_bleed	bld	88	0
7	burner	comb	44	0
8	mv_4	mxvl	33	1
9	hpt	turb	66	0
10	mv_5	mxvl	33	1
11	duct	duct	77	1
12	mv_6	mxvl	33	1
13	nozzle	nozz	101	0

Figure 5.7 - Contents of comp.list.no File After Completion

For example, if the DUCT component shown in the example network, was executing in response to an NSYS value of 9, it would extract the component number of the upstream component (which in this case is 10 - the component number of the MV_6 component). It would then write it's component number (11), name (duct), type (duct), and the upstream component number (10) in the file. Figure 5.8 shows what the comp.list.up file would look like after this was completed.

2. The component sets the value of its upstream component number to the variable NCOMPUP which is a static variable and will be retained by the module after it has completed executing. If the component

has multiple inputs the values of the upstream component numbers are stored in the NCOMPUP array which is a static array and will be retained by the module after it has completed executing.

3. The component sets the NAUXVAL value to its component number and sends it to the downstream component(s). Once the comp.list.no and comp.list.up files have been established, the SYSTEM module opens the file comp.list.dn and writes a header to the file. For each component listed in comp.list.no, the comp.list.up file is searched for any components having the same upstream component number.[5] These are then appended to the comp.list.dn file. This file, then contains a list of the downstream component(s) connected to each of the components in the network (except the SYSTEM-END module). Figure 5.9 shows the completed comp.list.dn file for the example network.

The connectivity of the engine network is now defined. By using these three files, the upstream and downstream components of any component in the network may be determined.

COMP. NO.	COMP. NAME	COMP. TYPE	UPSTREAM COMP.
2	environ	envr	1
3	hpc	comp	2
4	hss	shft	3
5	mv_3	mxvl	3
6	hp_bleed	bld	5
7	burner	comb	5
8	mv_4	mxvl	7
9	hpt	turb	8
9	hpt	turb	4
10	mv_5	mxvl	6
11	duct	duct	10

Figure 5.8 - Contents of comp.list.up file After DUCT Component Has Executed

5.2.2 Initializing System with Design Point Data (NSYS value 2)

The next step in the operating paradigm is calculation of the corrections coefficients for the engine model. Before computing these values, however, the user-defined design point data must be available throughout the system. Each component in the network, in response to a NSYS value of 2 sent by the SYSTEM module, writes its design point values to the data arrays corresponding to the component, which are then passed to the downstream components. This is necessary because other components must be able to access the design point values of adjacent components. For example, a compressor component must have the design values of pressure at its boundaries in order to compute its temperature correction coefficient (TCORR). This information is provided by the mixing volume components connected both upstream and downstream of the compressor.

5.2.3 Computing Correction Coefficients (NSYS values 3 and 4)

The SYSTEM module then sends a NSYS value of 3 to the network. All components except the shaft components now compute their correction coefficients.[6] As described above, the shaft component requires input data from the compressor and turbine, and because the modules in the network will operate sequentially, the shaft would normally operate before the turbine. Thus, the shaft must be made to wait until the turbine has executed and therefore, does not respond to a NSYS value of 3.

COMP. NO.	COMP. NAME	DWNSTRM NO.	DWNSTRM COMP.	DWNSTRM COMP.
1	system	2	environ	11
2	environ	3	hpc	22
3	hpc	4	hss	55
3	hpc	5	mv_3	33
4	hss	9	hpt	66
5	mv_3	6	hp_bleed	88
5	mv_3	7	burner	44
6	hp_bleed	10	mv_5	33
7	burner	8	mv_4	33
8	mv_4	9	hpt	66
9	hpt	10	mv_5	33
10	mv_5	11	duct	77
11	duct	12	mv_6	33
12	mv_6	13	nozzle	101

Figure 5.9 - Contents of Completed comp.list.dn File

After all components have responded to the NSYS value of 3, the SYSTEM module sends an NSYS value of 4 to the network. All shaft components now compute their correction coefficients.

5. The process of determining the downstream component list comp.list.dn is done entirely by the SYSTEM module; there are no NSYS values sent to the network to perform this task

6. Not all components compute correction coefficients. Those which do not compute correction coefficients still respond to the NSYS value of 3 and perform other necessary calculations. See Section 6 for a description of how each module responds to this message

5.2.4 Initializing System with Initial Operating Point Data (NSYS values 10 and 8)

The next step in the operating paradigm is balancing of the engine to steady-state conditions at the initial operating point. Before beginning the balancing, however, the user-defined initial operating point data must be defined throughout the system. This is achieved in two steps. In the first step, mixing volume and shaft components in the network, in response to a NSYS value of 10 sent by the SYSTEM module, write their initial operating point values to the data arrays corresponding to the component, which are then passed to the downstream components.

In the second step, the SYSTEM module sends an NSYS value of 8 to the network. Each of the components in the network then respond by computing the initial operating point values of mass flow rate and fuel-air ratio.

5.2.5 "Running The Engine" (NSYS values 7, 5, and 6)

With the preceding steps completed, the system is now fully defined at the initial operating point. The next step in the operating paradigm is to determine if these initial conditions reflect a balanced engine condition. In order to determine if the engine is balanced, the differential equations used to model the mixing volume, shaft, and duct are evaluated. If all derivative values are (nearly) zero, then the engine is balanced.

As described in Section 4, to determine the derivative values, the engine is "run". This process is comprised of three passes through the engine. These passes are implemented by having the SYSTEM module send three NSYS values to the network in order to determine the derivatives. The NSYS values of 7, 5, and 6 are sent to the network to perform the computations as described in Passes 1, 2, and 3, respectively.

In response to an NSYS value of 7, mixing volume components compute the pressure in their control volume based on known values of temperature and stored mass. At this time, the mass flow rate in any bleed components is also computed. As described in Section 4, computation of mass flow rates is performed on Pass 2. However, this departure from the operating paradigm is required in order to properly implement the bleed-cooled turbine components within AVS. To better explain this need, consider the network shown in Figure 5.10.

Cooling bleed flow for the turbine reenters the cycle in the mixing volume at the turbine exit. Because the bleed flow goes into the mixing volume below the turbine, the turbine and bleed components will be in parallel flow paths. Thus, because of the operational characteristics of module execution under AVS, the turbine may execute, before the bleed has computed its mass flow rate. In order to correctly compute the turbine energy term, the turbine module must be able to get the correct bleed mass flow rate value. This problem is solved by running the bleed at the same time the pressures in the mixing volumes are computed. Because the bleed component only requires that its input boundary values be defined in order to compute mass flow rate, it is only necessary that the mixing volume upstream of the bleed execute prior to the bleed. Since it is necessary that components execute in sequence in the direction of fluid flow propagation, this requirement is always satisfied.

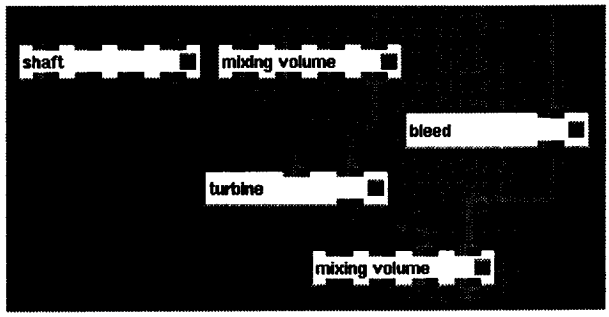


Figure 5.10 - Bleed-cooled Turbine Implementation

In response to an NSYS value of 5, all engine components compute the f/a and enthalpy of the working fluid within the component. Physical components, in addition to computing f/a and enthalpy of the working fluid, compute their mass flow rate. Some physical components (compressor, turbine, combustor) compute energy terms at this time.

In response to an NSYS value of 6, the mixing volume, shaft and duct components generate their respective derivative values.

5.2.6 Balancing Engine to Steady-state Conditions at Initial Operating Point

The balancing method used to achieve a steady-state balanced engine is selected by the user. Currently, TESS provides two iterative numerical methods to determine a steady-state balanced engine. These methods are the Newton-Raphson method and the Fourth-order Runge-Kutta method. At this point in the simulation, the SYSTEM module calls the appropriate subroutine to control the process of determining a balanced engine.

The iterative process used by both of these numerical solvers consists of determining the derivatives of the system by "running the engine" and then checking to see if the derivatives have been reduced to zero.[7] If the derivatives are not zero, the independent variable in each differential equation is improved upon, returned to its respective engine component and the engine is run again to determine the new derivative values. This process is repeated until the derivatives are all reduced to zero.

7. TESS does not actually check if the derivatives have been reduced to zero but rather uses another value known as the error value. See Appendix B of [4] for a detailed description on the steady-state solvers used in TESS and the definition of the error value.

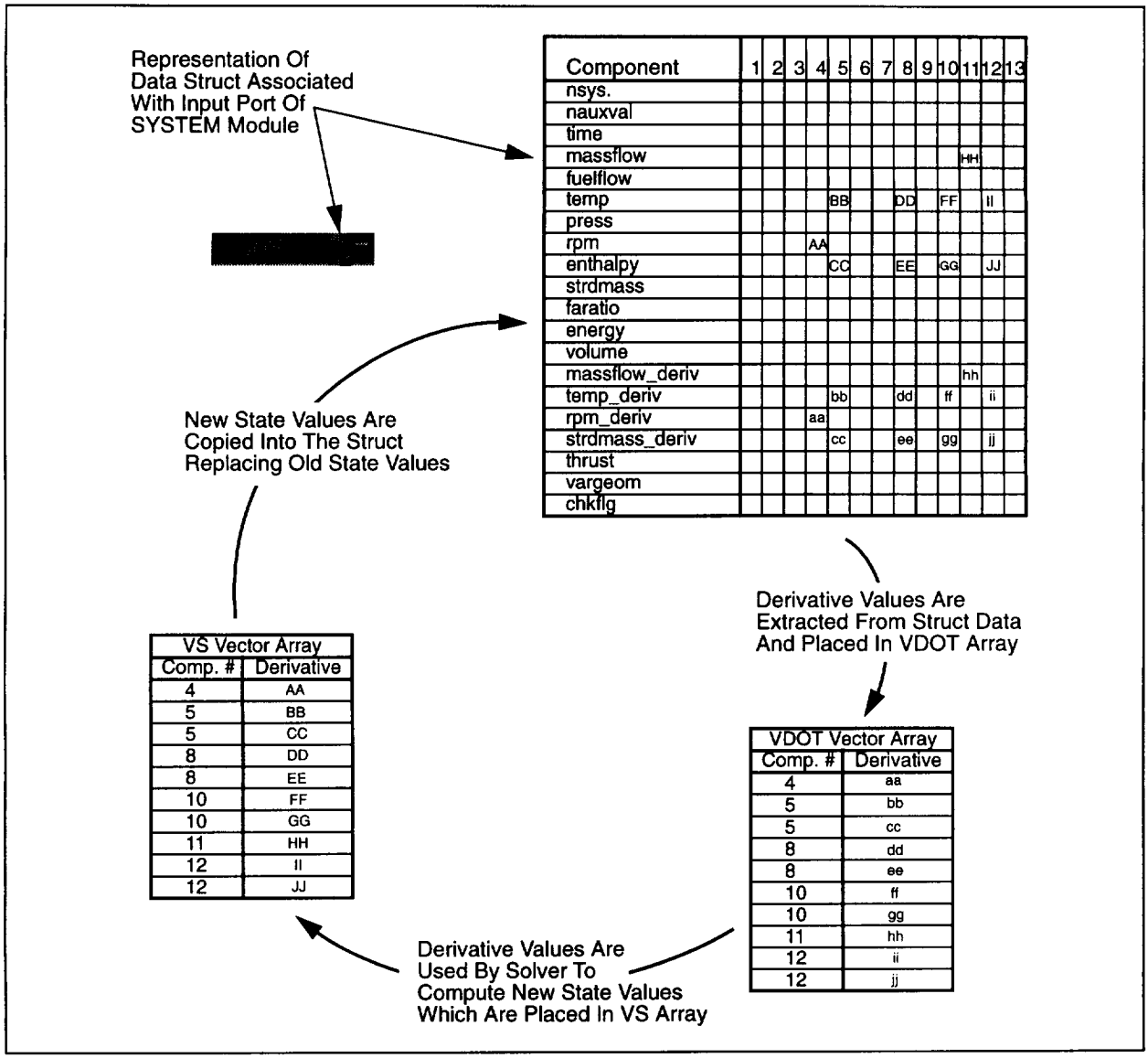


Figure 5.11 - Extraction and Replacement of VDOT and VS Values

In order to apply this iterative scheme within the framework of an arbitrary propulsion system simulator requires that the numerical solver be able to access the state derivative values computed by the mixing volumes, shafts and ducts in the engine configuration, and once new state values have been computed, to place those values back in the correct engine components. In TESS, the process of getting the state derivative values from the engine components is described below.

Once the engine has been "run", the solver must determine which of the components in the engine network have compute state derivative values. This is done by searching the comp.list.no file for components with a 1 in the solver flag column. For each component with a 1 in the solver flag column, the solver then uses the component number to access the data for that component contained in the data struct associated with the SYSTEM module's input port. The solver then checks the component type to determine if the component is a duct, a shaft, or a mixing volume. If it is a shaft, the solver accesses the **rpm_deriv** data array to get the derivative of the spool speed for that particular shaft component; if it is a duct, the solver accesses the **massflow_deriv** data array to get the derivative of the mass flow rate for that particular duct component; if it is a mixing volume, the solver accesses the **temp_deriv** and **strdmass_deriv** data arrays to get the derivative of the temperature and stored mass for that particular mixing volume component.

The derivative values are extracted from the struct in the order they are read from the comp.list.no and placed into the VDOT vector array. This array is then used by

the solver to compute new values of the state values which are placed in the VS vector array. These values are then placed back in the data struct associated with the SYSTEM module's input port in the order that they were extracted. This is done by searching the comp.list.no file for components with a 1 in the solver flag column. For each component with a 1 in the solver flag column, the solver uses the component number to replace the data for that component with the newly computed data contained in the VS vector array. If the component is a shaft, the solver replaces the **rpm** value with the new value of rpm for that particular shaft component; if it is a duct, the solver replaces the **massflow** value with the new value of mass flow rate for that particular duct component; if it is a mixing volume, the solver replaces the **temp** and **strdmass** values with the new values of temperature and stored mass for that particular mixing volume component. This process of extracting the derivative values, solving for the state values, and replacing them in the data struct is illustrated in Figure 5.11.

5.2.7 Transient Analysis of Engine

The final step in the operating paradigm is the transient analysis. This procedure is similar to the steady-state balancing operation except that the governing equations now include time as an independent variable. The differential equations are integrated numerically to determine the state of the engine at each time step in the transient.

As with the steady state balancing method, the user has a choice of numerical methods with which to solve the set of differential equations. Currently, TESS provides four iterative numerical methods to determine a steady-state balanced engine. These methods are: Fourth-order Runge-Kutta method, Improved Euler method, Adams-Moulton method and the Gear method. Depending on the user's selection, the SYSTEM module calls the appropriate subroutine to control the process of running the transient.

The iterative process used by all of these numerical solvers consists of determining the derivatives of the system by "running the engine" and then using the derivatives to estimate the state values at the end of some discrete time step This process is repeated over the entire user-defined time range.

5.3 Modules With Multiple Input Ports

Modules with more than one input port present certain difficulties in handling their execution and data passing.

5.3.1 Module Execution Control

For a module which has more than one input port, a mechanism is needed to control when the module begins to execute it's computation function. As described in Section 2, a subroutine module will execute automatically when its input changes. In a module having more than one input port it is possible that the module will execute it's computation function before all of the input ports have received new input data. The computation function will then execute using erroneous input data. To illustrate this, consider the following portion of a network shown in Figure 5.12.

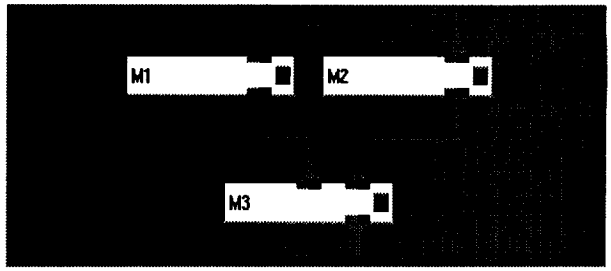


Figure 5.12 - Multiple Input Port Module

Module M1 receives input from some upstream module and executes, manipulating its input data and sending output to module M3. Module M2 also receives input from some upstream module and executes, manipulating its input data and sending output to M3. If M1 completes executing before M2, M3 will receive new input data from M1 before receiving new input data from M2. M3, in response to new input from M1 will begin executing before M3 has received the new data from M2. (Note: despite the fact that M2 has not yet run, there will still be input data available to the input port. However, it will be data output from a previous execution of M2, and thus will possibly be incorrect). M3 will then use old data previously sent by M2. Thus, the data sent to the computation function will be incomplete or incorrect and the output from M3 will likely be in error. To resolve this problem, a mechanism was developed which would allow the module to execute its computation function only when the module had received all of its input data.

Each input port in a TESS module is identified by an array of integer variable pointers, INDATA(n) (where n is the number of input ports). Input ports which have not been connected to another module return an INDATA value of zero (a NULL pointer). During execution of the network, each input port is associated with its own set of input data which can be accessed by the INDATA pointer associated with that input port. For each input port, the variable INPUT(n) is set equal to the NSYS value contained in that input port's data.

INPUT(1) = NSYS for input data accessed by INDATA(1) pointer.
INPUT(2) = NSYS for input data accessed by INDATA(2) pointer.
. .
. .
INPUT(n) = NSYS for input data accessed by INDATA(n) pointer.

In order for a module to execute properly, all of the data it receives from upstream modules must have been computed by modules executing in response to the same NSYS value issued by the SYSTEM module (see section 5.1.1). Consider the modules given in Figure 5.12. If M1 executes in response to an NSYS value of 3, M3 will receive input data from M1 which contains the NSYS value of 3. If M2 has not yet responded to the NSYS value of 3, the input data for M3 from M2 will be data generated in response to a previous NSYS value (which would be NSYS = 2). M3 would only receive all of its correct input if both M1 and M2 had executed in response to the NSYS value of 3. M3 would "know" that both M1 and M2 had both executed in response to NSYS = 3, by the fact that it would receive and NSYS value of 3 from both of these modules.

Thus, a module may execute when all of the NSYS values are identical. As stated above, the INPUT(n) values contain the NSYS values for input data associated with each input port. So when all of the INPUT values agree, then all of the NSYS values agree and the module may execute. The criteria for module execution may be stated:

- A module having more than one input port may execute only when all of the INPUT(n) values are identical

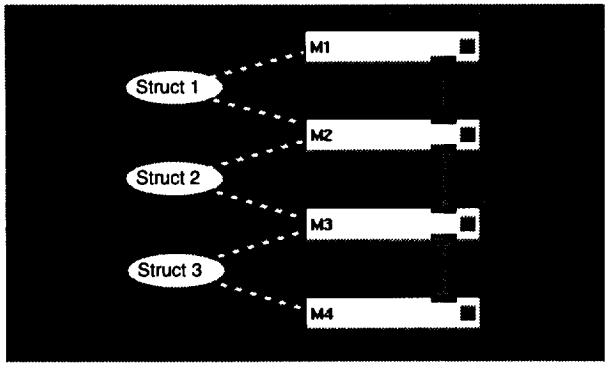


Figure 5.13 - Module Network Connected in Linear Fashion

5.3.2 Data Integration

As described in Section 2, a struct is used to store the data which is being shared between connected modules in a system. For a system where the modules are connected in linear fashion such as those shown in Figure 5.13, the transfer of data between modules is straight-forward, as there is only a single data struct located between each component.

The data in Struct 1 is copied into module M2 (indicated by the dashed lines), operated on, and copied to Struct 2. The process is repeated on down the network. A linear network such as this poses no problems in updating values in the data struct. However, if the network diverges into one or more parallel execution branches which then converge, the module at the convergent point will have multiple input data sets. These multiple input data sets will contain both common and different data which will have to be correctly integrated into a single data set before it can be copied into the data struct associated with its output port(s). To better illustrate this, consider the network shown in Figure 5.14. The solid lines between modules indicate the connections which AVS has drawn, and the dotted lines indicate the connections between modules and the data structs.

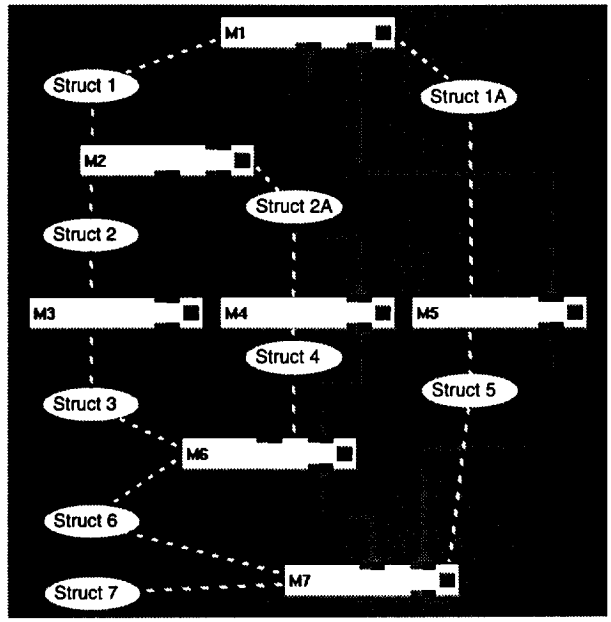


Figure 5.14 - Module Network Connected in Parallel Fashion.

As can be seen from the figure, the engine network splits into two parallel branches downstream of module M1. Modules M2, M3, M4 and M6 are in one branch while module M5 is in the other. The two branches converge at module M7. Furthermore, another parallel branch is formed downstream of M2. Modules M3 and M4 are in the separate branches and which then converge at module M6.

To better explain how the data is passed through the network and how parallel data paths are handled, consider the following explanation of the execution of each of the modules in the network. The status of the data in each of the structs is represented by a table (see Table 5.2)

Module	1	2	3	4	5	6	7
Updated	X		X		X	X	

Table 5.2 - Representation of Data Struct

This table indicates whether a module has executed

and generated new data in response to the current NSYS value sent by the SYSTEM.[8] An X is placed in the *Updated* row to indicate that data output of the module contains updated information. Modules without an X still have data associated with them, but it is from computations in response to a previous NSYS value.

In the network shown in Figure 5.14, module M1 executes first, and copies its output data to the memory regions associated with each of its two output ports. As stated above, during module instantiations, each module creates a data struct for each of its input and output ports. Thus M1 has two separate data structs for each of it's output ports: Struct 1 and Struct 1A. Although there are two separate structs, each contain identical information. In actuality, a single output port could have been used and multiple connections run from it to downstream modules. In TESS, however, the decision was made to have a single connection at each input or output port. This was done to better model certain components. For instance, the compressor module has two output ports, one to the shaft and one to the downstream mixing volume. Both data sets associated with each output, however, contain the same information. The data in Struct 1 and 1A is described by Table 5.3.

Module	1	2	3	4	5	6	7
Updated	X						

Table 5.3 - Representation of Data Struct 1 and Struct 1A

Similarly, module M2 executes and copies the data from Struct 1 into it's internal arrays, computes any new data, and copies the output data to the data struct associated with its two output ports: Struct 2 and Struct 2A. The data in Struct 2 and Struct 2A are then described by Table 5.4.

Module	1	2	3	4	5	6	7
Updated	X	X					

Table 5.4 - Representation of Data Struct 2 and Struct 2A

Modules M3 and M4 execute next. M3 copies the data from Struct 2 and outputs the data to Struct 3. M4 copies the data from Struct 2A and outputs the data to Struct 4. The data in Struct 3 and Struct 4 are described by Table 5.5 and Table 5.6, respectively.

Module	1	2	3	4	5	6	7
Updated	X	X	X				

Table 5.5 - Representation of Data Struct 3

8. Recall that the SYSTEM module sends an NSYS value to the entire network and then waits until each of the modules has executed in response to that particular NSYS value.

Module	1	2	3	4	5	6	7
Updated	X	X		X			

Table 5.6 - Representation of Data Struct 4

Notice that the data structs are now different due to the fact that M3 and M4 are in parallel paths. The output of modules M3 and M4 are both connected to the input ports of M6. Before the input data can be copied into Struct 6, the data sets from Struct 3 and Struct 4 must be integrated.

In looking at the incoming data from the perspective of M6, it "sees" that both structs contain updated information about modules M1 and M2. This means that both structs contain the same information about these modules. It also sees that neither struct has any updated information about modules M5, M6, and M7. This also means that both structs contain the same information about these modules as these modules have not yet executed.

The difference in the data exists with modules M3 and M4. Module M6 sees that the data in Struct 3 contains updated information about module M3. On the other hand, the data in Struct 4 contains old (non-updated) information about M3. Module M6 would want to ignore the information about M3 contained in Struct 4, as it was old information, and use only the information about module M3 contained in Struct 3.

Similarly, M6 would want to ignore the information about module M4 contained in Struct 3, as it was old information, and use only the data about module M4 contained in Struct 4. Applying the above strategy, M6 would executes and copy the data from Struct 3 and Struct 4 into it's internal arrays, computes any new data, integrate the two sets of data and copy the output data to the data struct associated with its output port: Struct 6. The data in Struct 6 is then described by Table 5.7.

Module	1	2	3	4	5	6	7
Updated	X	X	X	X		X	

Table 5.7 - Representation of Data Struct 6

When module M5, contained in the other branch of the network, executed it would copy the data from Struct 1A into it's internal arrays, computes any new data, and copy the output data to the data struct associated with its output ports: Struct 5. The data in Struct 5 is then described by Table 5.8.

Module	1	2	3	4	5	6	7
Updated	X				X		

Table 5.8 - Representation of Data Struct 5

The last module in the network, M7, finally executes. Since it receives input from two upstream modules, it will have to integrate the input data as was done in module M6, using data from Struct 6 and Struct 5. In looking at Tables 5.6 and 5.7, which represent those structs, M7 would "see" that both structs contain updated information about module M1; that Struct 6 contains updated data on modules M2, M3, M4, M6, and Struct 5 contained updated data on module M5. M7 upon executing, would copy the data from the two structs, integrate the two sets of data and copy the output data to the data struct associated with its output port: Struct 7. The data in Struct 7 is then described by Table 5.9.

Module	1	2	3	4	5	6	7
Updated	X	X	X	X	X	X	X

Table 5.9 - Representation of Data Struct 7

As can be seen from Table 5.8, the data in Struct 7 contains a completely updated set of data describing the system. In general, once every module in the network has executed, the last module (i.e., the module at the bottom of the network) will have a complete description of the data from all of the modules in the network. In TESS, the SYSTEM-END module is always the last module in a network. Its output port is connected to the input port of the SYSTEM module. Thus, the data struct associated with the output port of the SYSTEM-END module is also the struct associated with the input port for the SYSTEM module. This ensures that the SYSTEM module always receives completely updated data.

The thought process expressed above was implemented as the methodology used to combine data in modules having more than one input data port. An integer variable, CHKFLAG, is included in the data struct for each module in the network. When each module executes, it sets its CHKFLAG value to 1 (The SYSTEM module resets the CHKFLAG values to zero after each network execution). The process can be expressed as follows:

- In comparing multiple sets of input data for a given module, if one of the sets has a CHKFLAG value of 1, that data has been updated and can be used. For the case when none of the sets has a CHKFLAG value, this means that module has not yet run and any value can be used, as all the values are identical.

6.0 Description of TESS Modules Execution

This Section presents a detailed description of each of the modules currently supplied in TESS. For each module, the following information is supplied:

- Input and output port connection requirements
- Input parameter requirements

6.1 SYSTEM

Module Description - The SYSTEM module controls the simulation execution.

Input and Output Ports - The SYSTEM module has a single input port and a single output port. The output port must be connected to the environment module and the upstream port must be connected to the SYSTEM-END module's output port. This creates a data connection loop from the SYSTEM-END module to the SYSTEM module. Figure 6.1 illustrates the SYSTEM and SYSTEM-END combination. IMPORTANT: There must only be one instance of the SYSTEM module.

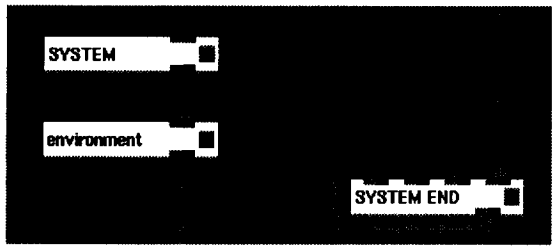


Figure 6.1 - SYSTEM Module Shown Connected In Typical Fashion

Input Parameters - The following are the user-controllable parameters displayed in the SYSTEM module's control panel (see Figure 6.2):

- run — Toggle button used to start and stop the simulation
- steady state — Toggle button used to control visibility of pop up windows containing steady state balancing methods and their control parameters
- transient — Toggle button used to control visibility of pop up windows containing transient solver methods and their control parameters
- starting — The simulation starting time, (sec)
- ending — The simulation ending time, (sec)

Two widgets are also displayed in the SYSTEM control panel to provide the user with feedback as the simulation is running:

- elapsed time — Dial which indicates the elapsed simulation time
- Simulation Status — The Simulation Status Output displays various messages indicating the action that the simulation is performing or one that has been completed

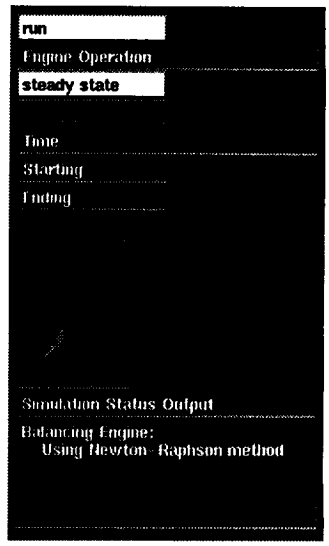


Figure 6.2 - SYSTEM Module Control Panel

Module Execution - As described in the TESS Message Passing Under AVS section of Section 5, each module responds to certain NSYS values sent by the SYSTEM module. The order in which the NSYS values are sent by the SYSTEM is described in the Operating Paradigm Implementation Under AVS section of Section 5. The flowchart shown in Figure 6.3 provides an overview of the SYSTEM modules execution process. A detailed description of each step in the process follows the flow chart.

- SYSTEM sends an NSYS value of 0 to the network to establish connections between each module. This is necessary so that all pointers to connected input and output ports are initialized. Failure to do this can result in null pointers in modules which are connected to multiple upstream modules. This will cause the initialization scheme (see Section 5.2.1) to fail and produce multiple listings of a component in the comp.list.no file.
- Execution control passes into a continuous loop which waits until the user presses the 'run' button. An integer flag, RUN, which corresponds to the value of the 'run' toggle button is used to control when the simulation is to execute (initially, the toggle is off and RUN is zero). When the user presses the 'run' button (setting RUN to 1), an AVSCOROUT_INPUT call is used to get the current values of all of the input parameters for all widgets controlled by the SYSTEM module.
- As stated above, the SYSTEM module provides pop-up windows to display information about the steady

state and transient solvers. Based on the values of the integer flags STDYST and TRANS which correspond to the value of the 'steady state' and 'transient' toggle buttons, respectively, the SYSTEM module then either hides or shows these pop-up windows. Also, the AVSMODIFY_PARAMETER command is used to update the 'Elapsed Time' dial widget so that its minimum and maximum values correspond to the values listed in the 'starting time' and 'ending time' widgets.

- The SYSTEM module's data arrays are initialized to zero and those values copied to the data set associated with the SYSTEM's output port which is accessed by the integer pointer OUTDATA.
- The AVSMODIFY_PARAMETER command is issued to display a message identifying the current version of TESS in the Simulation Status Output widget.
- The simulation time is set to -1. This is done to indicate that the system has not yet been balanced.
- SYSTEM creates file comp.list.no and writes a five line header to the file. It then sends subroutine PRESET the NSYS value of 1. Subroutine PRESET is used whenever an NSYS value is sent to the modules in the network. It first calls subroutine GETPOINTER to get the pointer (INDATA) which accesses the data set associated with the SYSTEM module's input port. This call also returns the current value of RUN. If RUN is zero, the user has stopped the simulation (the 'run' toggle button has been turned off which sets RUN to zero) and subroutine PRESET returns. Next the data arrays associated with the input port are copied into internal data arrays, all of the values in the CHKFLAG array are reset to zero (see Section 5.2.2 for a discussion of the CHKFLAG value and its significance), and the internal data arrays are copied to the data set associated with the SYSTEM's output port. Then the AVS command COROUT_OUTPUT is executed. This command sends data from the SYSTEM module's output port to the next downstream module. This will then cause each module in the network to execute. Because the SYSTEM module is a coroutine, and as such runs asynchronously, it will begin to execute its next command while the other modules are executing. To prevent this the AVSCOROUT_EXEC call is executed. This call causes the SYSTEM module to wait until each module in the network has run. When the AVSCOROUT_EXEC call returns, subroutine PRESET returns to the calling program.
- SYSTEM creates file comp.list.up and writes a five line header to the file and sets the NAUXVAL value to be equal to its component number (which is always 1). It then sends an NSYS value of 9 to the modules in the network by calling subroutine PRESET. Subroutine PRESET returns the current value of RUN which is checked to see if the user has stopped the simulation (0=stop, 1=run). If the user has stopped the simulation, a "SIMULATION HALTED" message is displayed in the Simulation

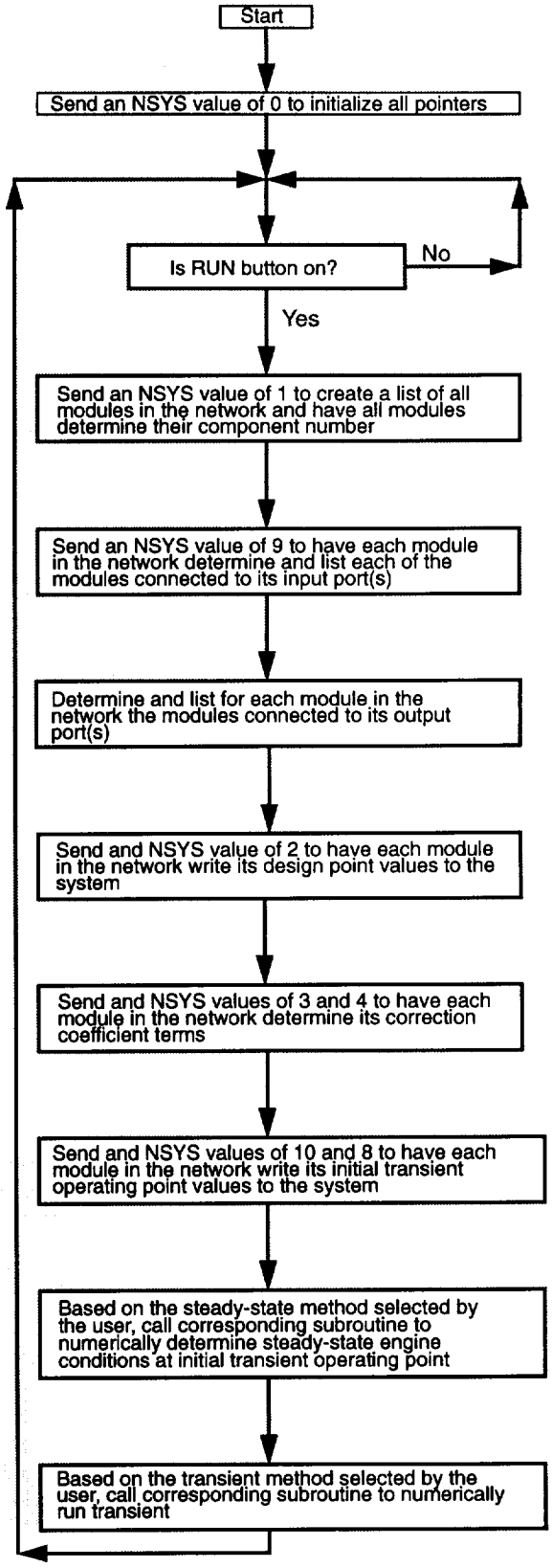


Figure 6.3 - Flowchart Describing SYSTEM module Control Process

Status Output widget.

- SYSTEM creates file comp.list.dn, writes a five line header to the file, determines the downstream module(s) for each module in the network and writes those relationships to the file.
- SYSTEM sends an NSYS value of 2 to the modules in the network by calling subroutine PRESET. Subroutine PRESET returns the current value of RUN which is checked to see if the user has stopped the simulation (0=stop, 1=run). If the user has stopped the simulation, a "SIMULATION HALTED" message is displayed in the Simulation Status Output widget.
- SYSTEM begins the correction computing phase of operation and a "Computing Correction Coefficients" message is displayed in the Simulation Status Output widget. SYSTEM then sends an NSYS value of 3 to the modules in the network by calling subroutine PRESET. Subroutine PRESET returns the current value of RUN which is checked to see if the user has stopped the simulation (0=stop, 1=run). If the user has stopped the simulation, a "SIMULATION HALTED" message is displayed in the Simulation Status Output widget.
- SYSTEM then sends an NSYS value of 4 to the modules in the network by calling subroutine PRESET. Subroutine PRESET returns the current value of RUN which is checked to see if the user has stopped the simulation (0=stop, 1=run). If the user has stopped the simulation, a "SIMULATION HALTED" message is displayed in the Simulation Status Output widget.
- SYSTEM begins the operating point balancing phase of operation and a "Writing Operating Point Data" message is displayed in the Simulation Status Output widget. SYSTEM then sends an NSYS value of 10 to the modules in the network by calling subroutine PRESET. Subroutine PRESET returns the current value of RUN which is checked to see if the user has stopped the simulation (0=stop, 1=run). If the user has stopped the simulation, a "SIMULATION HALTED" message is displayed in the Simulation Status Output widget.
- SYSTEM then sends an NSYS value of 8 to the modules in the network by calling subroutine PRESET. Subroutine PRESET returns the current value of RUN which is checked to see if the user has stopped the simulation (0=stop, 1=run). If the user has stopped the simulation, a "SIMULATION HALTED" message is displayed in the Simulation Status Output widget.
- SYSTEM now attempts to balance the engine at the operating point. It first determine the number of differential equations in the system by calling subroutine GETNUMEQN. This subroutine checks the comp.list.no file for components with a 1 in the solver flag column and sums the number of differential equations (shafts and ducts have a single differential equation while mixing volumes have two).
- SYSTEM next determines which balancing method the user has selected. The list of available steady state balancing methods and their control parameters are contained in two pop-up windows whose visibility is controlled by the 'steady-state' toggle button in the SYSTEM control panel. When the 'steady-state' button is highlighted (i.e., it is on), two small pop-up windows appear (see Figure 6.4). The left window contains the available selected steady state balancing methods. Currently there are two balancing methods available in TESS. These are the Newton-Raphson iterative method and the fourth-order Runge-Kutta method. These are presented as radio buttons which provide the user with mutually exclusive selection of a method. The control parameters associated with that method are then displayed in the right hand pop-up window. Selecting a different method will display different control parameters.
- Using the variable SELECT1, which contains the string indicating the user's selected steady state balancing method, SYSTEM calls the appropriate subroutine and passes the control parameters as arguments. These subroutines then execute until the system has been balanced. For a detailed explanation of the steady state balancing methods, see Appendix B of [4].
- SYSTEM then determines which transient solver the user has selected. The list of available steady state balancing methods and their control parameters are contained in two pop-up windows whose visibility is

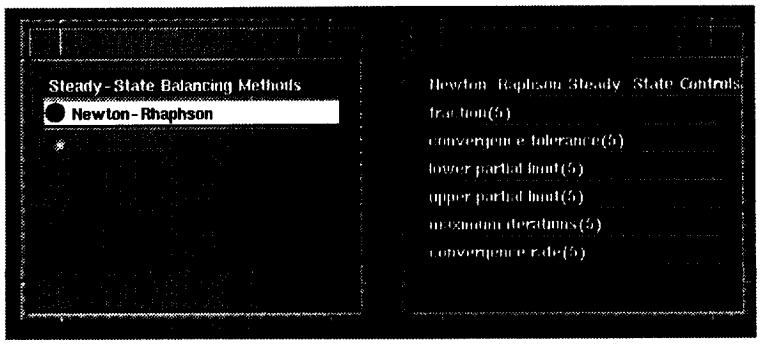


Figure 6.4 - SYSTEM Module Steady-state Methods Pop-up Windows

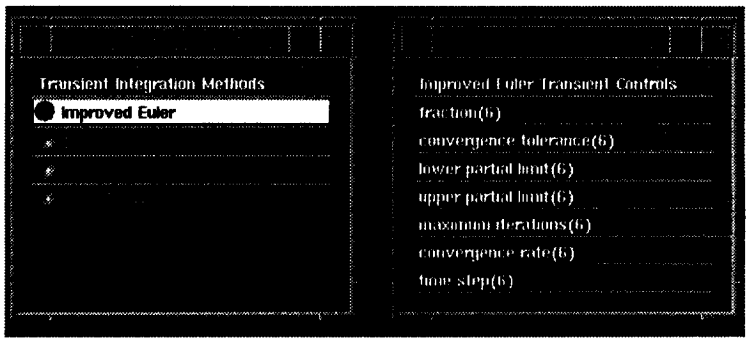


Figure 6.5 - SYSTEM Module Transient Methods Pop-up Windows

controlled by the 'transient' toggle button in the SYSTEM control panel. When the 'transient' button is highlighted (i.e., it is on), two small pop-up windows appear (see Figure 6.5). The left window contains the available transient solvers. Currently there are four solvers available in TESS. These are the Modified Euler, Fourth-Order Runge-Kutta, Adams and Gear's Stiff method. These are presented as radio buttons which provide the user with mutually exclusive selection of a method. The control parameters associated with the selected solver are then displayed in the right hand pop-up window. Selecting a different method will display different control parameters.

- Using the variable SELECT2, which contains the string indicating the user's selected transient solver, SYSTEM calls the appropriate subroutine and passes the control parameters as arguments. These subroutines then execute until the system has completed the simulation. For a detailed explanation of the transient solvers, see Appendix B of [4].

Once the simulation has been complete, a "SIMULATION COMPLETE" message is displayed in the Simulation Status Output widget and the 'run' toggle button is turned off (not highlighted). Execution control then goes to the beginning of the loop and waits until the user presses the 'run' button at which time the simulation is started again.

6.2 SYSTEM-END

Module Description - The SYSTEM-END module is used to combine data from any module whose output ports exit to the atmosphere and send it to the SYSTEM module. These would typically be nozzles and overboard bleed components.

Input and Output Ports - The SYSTEM-END module has four input ports and a single output port. The input ports are to be connected to any module whose output ports exit to the atmosphere. The output port MUST be connected to the input port of the SYSTEM module. The right-hand input port is a REQUIRED port and must be connected in order for the module to execute. Additional connections may be made to any of the remaining input ports. Figure 6.6 illustrates the SYSTEM-END module connected in typical fashion.

IMPORTANT: There must only be one instance of the SYSTEM-END module.

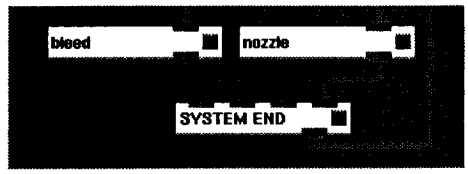


Figure 6.6 - SYSTEM-END Module Shown Connected In Typical Fashion.

Input Parameters - There are no input parameters for the SYSTEM-END module and, therefore, it has no control panel.

Module Execution - The SYSTEM-END module does not respond to any particular NSYS values. It executes when it receives new input from the upstream modules. Its only purpose is to combine data from multiple input data sets *into a single* data set which is then sent to the SYSTEM module.

6.3 Bleed

Module Description - The bleed module represents a bleed duct. It is typically used to provide cooling air to a bleed-cooled turbine, or to provide customer air.

Input and Output Ports - The bleed module has a single input port and a single output port allowing connection to one upstream mixing volume component and one downstream mixing volume component. If the bleed is to provide customer or overboard bleed flow, its output port may be connected to the SYSTEM-END module. Since there is only a single input port, it is the REQUIRED port and must be connected in order for the module to execute. Figure 6.7 illustrates bleed module connected in typical fashion.

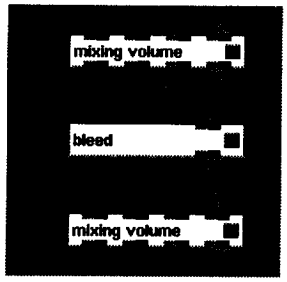


Figure 6.7 - Bleed Module Shown Connected In Typical Fashion

Input Parameters - The following are the user-controllable parameters displayed in the bleed module's control panel (see Figure 6.8):

- name — A unique, user-defined, 10 character string which is used to identify the instance of bleed module
- mass flow rate — The design point bleed mass flow rate, (lbm/sec)

Figure 6.8 - Bleed Module Control Panel

Module Execution - As described in the TESS Message Passing Under AVS section of Section 5, each module responds to certain NSYS values sent by the SYSTEM module. The following describes the bleed module's responses to various NSYS values:

NSYS = 1

In response to an NSYS value of 1, the bleed determines its unique component number (NCOMP). It then appends this information to the comp.list.no file along with its instance name, component type (bled), component code (0088), and solver flag (0).

NSYS = 9

In response to an NSYS value of 9, the bleed determines the component which is connected to its input port and appends it to the comp.list.up file. The NAUXVAL value received at the input port is the component number of the upstream component. This component number is stored as NUPCOMP, and saved to be used later in the simulation to access information about the upstream component.

NSYS = 2

In response to an NSYS value of 2, the bleed writes its user-defined design point value of mass flow rate (WBLEDD) to the massflow data array.

NSYS = 3

In response to an NSYS value of 3, the bleed computes the design point value of the specific heat ratio for the working fluid. The bleed module first determines the design point temperature (TIND), pressure (PIND), enthalpy (HIND) and fuel-air ratio (FARIND) for the upstream mixing volume. A call to subroutine PROCOMNPSS returns the design specific heat ratio (GMIND), which is saved for later use. The enthalpy and faratio data arrays for the bleed are then updated using HIND, and FARIND, respectively.

NSYS = 8

In response to an NSYS value of 8, the bleed module computes the operating point value of the specific heat ratio for the working fluid, and the operating point value of mass flow rate. The module first determines the operating point temperature (TINOP), pressure (PINOP) and fuel-air ratio (FARINOP) for the upstream mixing volume. A call to subroutine PROCOMNPSS returns the operating point specific heat ratio (GMINOP). Subroutine BLEED is then called to determine the operating point mass flow rate, WBLEEDOP. The massflow and faratio data arrays for the bleed are then updated using WBLEEDOP and FARINOP, respectively.

NSYS = 7

In response to an NSYS value of 7, the bleed computes the value of the specific heat ratio for the working fluid, and the mass flow rate. The module first determines the temperature (TIN), pressure (PIN) and fuel-air ratio (FARIN) for the upstream mixing volume. A call to subroutine PROCOMNPSS returns the specific heat ratio (GMIN) and enthalpy (HIN). Subroutine BLEED is then called to determine the mass flow rate, WBLEED. The massflow, enthalpy and faratio data arrays for the bleed are then updated using WBLEED, HIN, and FARIN, respectively.

6.4 Combustor

Module Description - The combustor module adds thermal energy to the working fluid in the engine through combustion of jet fuel (assumed to be JP-4). Fuel flow rate is controlled through the use of a user-defined fuel transient control schedule.

Input and Output Ports - The combustor module has a single input port and a single output port allowing connection to one upstream mixing volume component and one downstream mixing volume component. Since there is only a single input port, it is the REQUIRED port and must be connected in order for the module to execute. Figure 6.9 illustrates a combustor module connected in typical fashion.

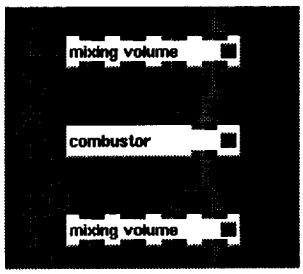


Figure 6.9 - Combustor Module Shown Connected In Typical Fashion

Input Parameters - The following are the user-controllable parameters displayed in the combustor module's control panel (see Figure 6.10):

- name — A unique, user-defined, 10 character string which is used to identify the instance of combustor module
- mass flow rate — The design point combustor mass flow rate, (lbm/sec)
- efficiency — The design point combustor efficiency, (dimensionless)
- fuel flow rate — The design point combustor fuel flow rate, (lbm/sec)
- fuel flow rate-op — The operating point combustor fuel flow rate, (lbm/sec)
- transient control — Toggle button to control visibility of pop-up windows containing schedulestransient control schedule parameters for fuel flow rate

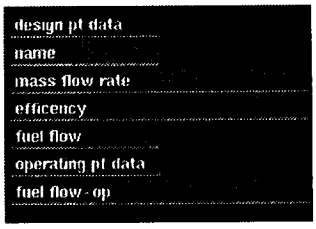


Figure 6.10 - Combustor Module Control Panel

Module Execution - As described in the TESS Message Passing Under AVS section of Section 5, each module responds to certain NSYS values sent by the SYSTEM module. The following describes the combustor module's responses to various NSYS values:

NSYS = 1

In response to an NSYS value of 1, the combustor determines its unique component number (NCOMP). It then appends this information to the comp.list.no file along with its instance name, component type (comb), component code (0044), and solver flag (0).

NSYS = 9

In response to an NSYS value of 9, the combustor determines the component which is connected to its input port and appends it to the comp.list.up file. The NSYS value received at the input port is the component number of the upstream component. This component number is defined as NUPCOMP and saved for later use.

NSYS =2

In response to an NSYS value of 2, the combustor writes the design point values of mass flow rate (WDOTID) and fuel flow rate (WFD) to the massflow and fuelflow data arrays, respectively.

NSYS = 3

In response to an NSYS value of 3, the combustor first determines the temperature (TIND), pressure (PIND) and fuel-air ratio (FARIND) values at its inlet from the upstream mixing volume component. The module then determines the component number (NDNCOMP) for the mixing volume which is connected to its output (downstream) ports. Using this information, the design point values of temperature (TOUTD) and pressure (POUTD) at its exit are obtained from the downstream mixing volume. TIND, PIND, TOUTD, POUTD, NUPCOMP and NDNCOMP are saved to be used later in the simulation.

Subroutine SETCOMBUSTOR is then called to determine the design values of enthalpy (HOUTD) and the temperature interpolation constant (BETACMB) which is saved to be used in subsequent computations. Also, the design point energy term, EOUTD, which represents the rate of heat energy addition in the combustor due to combustion of the fuel is computed. The design point fuel air ratio of the combustor (FARD) is then computed. The massflow, fuelflow, enthalpy, faratio, and energy data array values for the combustor are updated using the WDOTID, WFD, HOUTD, FARD and EOUTD, respectively.

NSYS = 8

In response to an NSYS value of 8, the combustor obtains the operating point temperature (TINOP), pressure (PINOP) and fuel-air ratio (FARINOP) values for the upstream mixing volume. Next, operating point values of temperature (TOUTOP) and pressure (POUTOP) from the downstream mixing volume is obtained.

Subroutine COMBUSTOR is then called to determine the operating point value of mass flow rate in the combustor (WDOTOP). The operating point fuel air ratio of the combustor (FAROP) is then computed. The massflow and faratio data array values for the combustor are updated using the WDOTOP and FAROP, respectively.

NSYS = 5

In response to an NSYS value of 5, the combustor

first determines the temperature (TIN), pressure (PIN) and fuel-air ratio (FARIN) values for the upstream component, and the values of temperature (TOUT) and pressure (POUT) from the downstream mixing volume.

The fuel flow rate value during the transient is dependent on the fuel flow transient control schedule defined by the user. The fuel flow schedule allows the user to define the fuel flow rate at four time values during the transient. For a more detailed description of defining transient control schedules, see Appendix D of [4].

Based on the current simulation time (TIME), the value of fuel mass flow rate (WFUEL) is computed. Subroutine COMBUSTOR then computes the mass flow rate (WDOTIN), enthalpy (HOUT) and energy term (EOUT) for the combustor. The fuel air ratio of the combustor (FAR) is then computed. These values are then used to update the massflow, fuelflow, enthalpy, faratio, and energy data arrays for the combustor.

6.5 Compressor

Module Description - The compressor module represents a variable-stator compressor used to increase the kinetic energy of the working fluid in the compressor by transmitting the mechanical energy from a shaft to the fluid.

Input and Output Ports - The compressor module has one input port and two output ports. The single input port is to be connected to a mixing volume module. The left-hand output port is to be connected to a shaft module and the right-hand output port is to be connected to a mixing volume. The input port is a REQUIRED ports and must be connected in order for the module to execute. Figure 6-11 illustrates a compressor module connected in typical fashion.

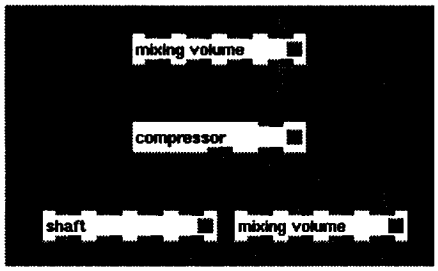


Figure 6.11 - Compressor Module Shown Connected In Typical Fashion

Input Parameters - The following are the user-controllable parameters displayed in the compressor module's control panel (see Figure 6-12):

- name — A unique, user-defined, 10 character string which is used to identify the instance of the compressor.
- mass flow rate — The design point compressor mass flow rate, (lbm/sec)
- efficiency — The design point compressor efficiency, (dimensionless)
- stator angle — The design point compressor variable-stator angle, (deg)
- stator angle bias — The design point compressor variable-stator angle bias, (deg)
- base performance map filename — The name (and path) of the file containing baseline performance data for the compressor
- variable performance map filename — The name (and path) of the file containing baseline performance data performance for the variable stator
- mass flow rate-op — The operating point compressor mass flow rate, (lbm/sec)
- transient control — Toggle button to control visibility of pop-up windows containing schedulestransient control schedule parameters for variable stator angle

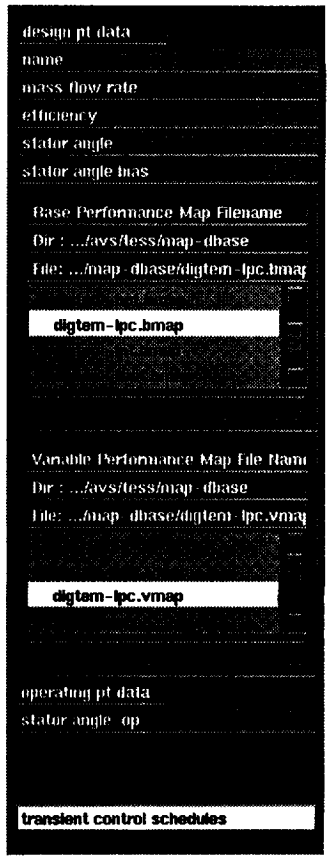


Figure 6.12 - Compressor Module Control Panel

Module Execution - As described in the TESS Message Passing Under AVS section of Section 5, each module responds to certain NSYS values sent by the SYSTEM module. The following describes the compressor modules responses to various NSYS values:

NSYS = 1

In response to an NSYS value of 1, the compressor determines and saves its unique component number (NCOMP). It then appends this information to the comp.list.no file along with its instance name, component type (comp), component code (0022), and solver flag (0).

NSYS = 9

In response to an NSYS value of 9, the compressor determines the components which are connected to its input port and appends it to the comp.list.up file. The NAUXVAL value received at the input port is the component number of the upstream component. This component number is stored as NUPCOMP and saved to be used later in the simulation to access information about the upstream component.

NSYS = 2

In response to an NSYS value of 2, the compressor writes its user-defined design point value of mass flow rate (WDOTID) to the massflow data array.

NSYS = 3

In response to an NSYS value of 3, the compressor computes the correction coefficients WCORR and TCORR, in the following procedure.

The design point temperature (TIND), pressure (PIND), enthalpy (HIND) and fuel-air ratio (FARIND) are obtained from the upstream mixing volume. The compressor then determines which output port is connected to the shaft and which is connected to the mixing volume. Although the user should connect the left-hand output port to the shaft and the right-hand output port to the mixing volume, this check allows the module to operate correctly even if the user has connected the modules improperly. The component number for the mixing volume is NDNCOMP1 and the shaft component number is NDNCOMP2. Both are saved to be used in subsequent calculations. The design point temperature (TOUTD) and pressure (POUTD) are then obtained from the downstream mixing volume and the design point spool speed (XSPOLD) is obtained from the shaft.

Subroutine GETGEMAP reads in performance data for the compressor using the string BFILENAME which contains the filename (and path) of the file which the user has selected using the base map file browser in the compressor module control panel. This map contains baseline performance data for the compressor, which has been normalized to the compressor design point. This data is saved to be used when performance data is required.

Subroutine SEARCHMAP is then called to interpolate the design point value of mass flow rate (WARM) and efficiency (ETAMAP) from the baseline performance map data. ETAMAP is then multiplied by the user-defined design efficiency value (ETAD) to give the compressor efficiency (ETACMM). Next, the bias of the stator geometry is then accounted for by subtracting the bias (BIAS) from the design stator geometry value (CVGPD).

Subroutine GETGEMAP is called again, this time reading in the variable stator performance data using the string VFILENAME which contains the filename (and path) of the file which the user has selected using the variable map file browser in the compressor module control panel. This map contains the effects of off-schedule stator settings on the compressor mass flow rate, which have been normalized to the compressor design point. This data is saved to be used when performance data is required.

Subroutine SEARCHMAP is then called to interpolate the design point change in baseline mass flow rate (CSHIFT), due to variable geometry effects form the variable map data. This is then used to determine the mass flow correction coefficient (WCORR) which is saved for use in subsequent computations. The temperature interpolation constant (BETACOM) is computed next (see mathematical models Section on compressor for more information) and the temperature correction coefficient (TCORR) determined and saved for later use.

The design values of enthalpy (HOUTD), energy term (EOUTD), stator variable geometry (CVGPD) and fuel-air ratio (FARIND) are placed in the enthalpy, energy, vargeom and faratio data arrays, respectively.

NSYS = 8

In response to an NSYS value of 8, the compressor first obtains the operating point temperature (TINOP), pressure (PINOP) and fuel-air ratio (FARINOP) from the upstream mixing volume. The operating point temperature (TOUTOP) and pressure (POUTOP) are then obtained from the downstream mixing volume and the operating point spool speed (XSPOLOP) is obtained from the shaft.

The point to be located on the performance map, XCOM and YCOM, are computed next. XCOM represents the operating point pressure ratio across the compressor (POUTOP/PINOP) normalized to the design point pressure ratio (POUTD/PIND). YCOM represents the ratio of operating point corrected spool speed (XSPOOLOP/$\sqrt{\text{TIN}}$) normalized to the corrected design spool speed (XSPOLD/$\sqrt{\text{TIND}}$).

Subroutine SEARCHMAP is then called to interpolate the design point value of mass flow rate (WARMOP) and efficiency (ETAMAPOP) from the baseline performance map data at the desired point. ETAMAPOP is then multiplied by the user-defined design efficiency value (ETAD) to give the compressor efficiency (ETACOM). Next, the bias of the stator geometry is then accounted for by subtracting the bias (BIAS) from the operating point stator geometry value (CVGPOP).

Subroutine SEARCHMAP is then called a second time to interpolate the operating point change in baseline mass flow rate (CHSIFT), due to variable geometry effects from the variable map data. This is then used to determine the operating point mass flow rate in the

compressor (WDOTOP).

The operating point values of mass flow rate (WDOTOP), stator variable and fuel-air ratio (FARINOP) are placed in the massflow and faratio data arrays, respectively.

NSYS = 5

In response to an NSYS value of 5, the compressor obtains the temperature (TIN), pressure (PIN), enthalpy (HIN) and fuel-air ratio (FARIN) from the upstream mixing volume. The temperature (TOUT) and pressure (POUT) are then obtained from the downstream mixing volume and spool speed (XSPOOL) is obtained from the shaft.

The point to be located on the performance map, XCOM and YCOM, are computed next. XCOM represents the pressure ratio across the compressor (POUT/PIN) normalized to the design point pressure ratio (POUTD/PIND). YCOM represents the ratio of the corrected spool speed ($XSPOOL/\sqrt{TIN}$) normalized to the corrected design spool speed ($XSPOLD/\sqrt{TIND}$).

Subroutine SEARCHMAP is then called to interpolate the design point value of mass flow rate (WARM) and efficiency (ETAMAP) from the baseline performance map data at the desired point. ETAMAP is then multiplied by the user-defined design efficiency value (ETAD) to give the compressor efficiency (ETACOM).

Next, the bias of the stator geometry is then accounted for. Stator variable geometry values during the transient are dependent on the stator variable geometry transient control schedule defined by the user. The variable stator schedule allows the user to define the stator angle at four time values during the transient. For a more detailed description of defining transient control schedules, see Appendix D of [4].

Based on the current simulation time (TIME), the value of stator variable geometry (CVGP) is computed. Subroutine SEARCHMAP is then called to interpolate the change in baseline mass flow rate (CHSIFT), due to variable geometry effects from the variable map data. This is then used to determine the mass flow rate in the compressor (WDOTIN).

Next, the enthalpy at the compressor exit (HOUT) is computed by subroutine PROCOMNPSS. The energy term (EOUT) is then computed and the mass flow rate (WDOTIN), enthalpy (HOUT), energy term (EOUT), stator variable geometry (CVGP) and fuel-air ratio (FARIN) are placed in the massflow, enthalpy, energy, vargeom and faratio data arrays, respectively.

6.6 Duct

Module Description - The duct module represents a constant area, adiabatic duct with friction. Stagnation pressure losses are included, as are the effect of the momentum of the working fluid. The dynamic form of linear momentum equation is used to solve for the duct value mass flow rate (see Section 3).

Input and Output Ports - The duct module has a single input port and a single output port allowing connection to one upstream mixing volume component and one downstream mixing volume component. Since there is only a single input port, it is the REQUIRED port and must be connected in order for the module to execute. Figure 6-13 illustrates the duct module connected in typical fashion.

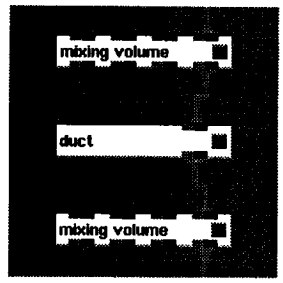


Figure 6.13 - Duct Module Shown Connected In Typical Fashion

Input Parameters - The following are the user-controllable parameters displayed in the duct module's control panel (see Figure 6-14):

- name — A unique, user-defined, 10 character string which is used to identify the instance of duct module
- mass flow rate — The design point duct mass flow rate, (lbm/sec)
- reactance — The value of the ducts ratio of area to length, (ft2/ft)
- mass flow rate-op — The operating point duct mass flow rate, (lbm/sec)

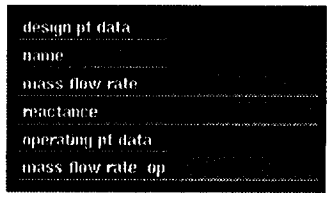


Figure 6.14 - Duct Module Control Panel

Module Execution - As described in the TESS Message Passing Under AVS section of Section 5, each module responds to certain NSYS values sent by the SYSTEM module. The following describes duct module's responses to various NSYS values:

NSYS = 1

In response to an NSYS value of 1, the duct determines its unique component number (NCOMP). It then appends this information to the comp.list.no file along with its instance name, component type (duct), component code (0077), and solver flag (1).

NSYS = 9

In response to an NSYS value of 9, the duct determines the component which is connected to its

input port and appends it to the comp.list.up file. The NAUXVAL value received at the input port is the component number of the upstream component. This component number is stored as NUPCOMP, and saved to be used later in the simulation to access information about the upstream component.

NSYS = 2

In response to an NSYS value of 2, the duct writes its user-defined design point value of mass flow rate (WDOTID) to the massflow data array.

NSYS = 3

In response to an NSYS value of 3, the duct determines the component number of the mixing volume connected to its output port (NDNCOMP).

NSYS = 8

In response to an NSYS value of 8, the duct computes the values of the constants CDUCT1 and CDUCT2. These values are saved and used in subsequent calculations. The duct module obtains the operating point temperature (TINOP), pressure (PINOP), fuel-air ratio (FARINOP) and volume (VOLOP) of the upstream mixing volume, and the operating point pressure (POUTOP) from the downstream mixing volume.

A call to subroutine DUCT returns the constants CDUCT1 and CDUCT2. CUDCT1 is the product of the constant, gc, and the user-defined duct reactance value (AQL). CDUCT2 is the pressure loss constant, K (see Section 3).

The faratio data array for the duct is then updated using FARINOP and the massflow data array is updated using the user-defined operating point mass flow rate value WDOTOP.

NSYS = 6

In response to an NSYS value of 6, the duct computes the derivative mass flow rate value, DWDOTN. The duct module obtains the temperature (TIN), pressure (PIN), enthalpy (HIN) and fuel-air ratio (FARINOP) of the upstream mixing volume, and the pressure (POUT) from the downstream mixing volume.

The stored mass and mass flow rate of the duct is needed to compute the mass flow derivative, DWDOTN. In the DIGTEM model, the stored mass (WSTOR) of the duct is associated with the upstream mixing volume and the mass flow rate in the duct (WDOTIN), has been computed by the solver. After obtaining these values, subroutine DCTINT computes and returns DWDOTN.

The enthalpy, faratio and massflow_deriv data arrays for the duct are then updated using HIN, FARIN and DWDOTN respectively.

6.7 Environment

Module Description - The environment module defines the surrounding environment for the engine based on user-defined schedules for altitude and Mach number. It also serves as a simple inlet model, computing the flight conditions (stagnation pressure and temperature) at the module exit.

Input and Output Ports - The environment module has a single input port and a single output port. The input port MUST be connected to the SYSTEM module. The output port is connected to a physical engine component (typically a compressor). Since there is only a single input port, it is the REQUIRED port and must be connected in order for the module to execute. Figure 6-15 illustrates an environment module connected in typical fashion.

IMPORTANT: There must only be one instance of the environment module.

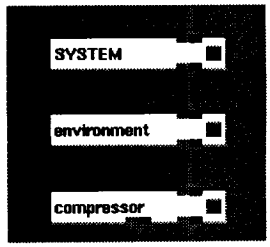


Figure 6.15 - Environment Module Shown Connected In Typical Fashion

Input Parameters - The following are the user-controllable parameters displayed in the environment module's control panel (see Figure 6-16):

• name	A unique, user-defined, 10 character string which is used to identify the instance of environment module
• altitude	The design point altitude for engine operation, (ft)
• mach no	The design point Mach number for engine operation, (dimensionless)
• SL temp	Sea level temperature, (°R)
• altitude-op	The operating point altitude for engine operation, (ft)
• mach no-op	The operating point Mach number for engine operation, (dimensionless)
• transient control	Toggle button to control visibility of pop-up windows containing schedulestransient control schedule parameters for altitude and Mach number

Module Execution - As described in the TESS Message Passing Under AVS section of Section 5, each module responds to certain NSYS values sent by the SYSTEM module. The following describes the environment module's responses to various NSYS values:

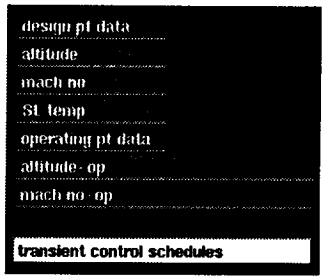


Figure 6.16 - Environment Module Control Panel

NSYS = 1

In response to an NSYS value of 1, the environment determines its unique component number (NCOMP). It then appends this information to the comp.list.no file along with its instance name, component type (envr), component code (0011), and solver flag (0).

NSYS = 9

In response to an NSYS value of 9, the environment determines the component which is connected to its input port (this should always the SYSTEM module) and appends it to the comp.list.up file. The NAUXVAL value received at the input port is the component number of the upstream component. This component number is defined as NUPCOMP.

NSYS = 2

In response to an NSYS value of 2, the environment computes the design values of stagnation temperature (TENVD) and pressure (PENVD) based on the design values of altitude (ALTD) and Mach number (XMND) by calling subroutine FLCOND. These values are then used to update the temp and press data arrays, respectively. At this point, TENVD, PENVD, XMND, and ALTD are written to file comp.envr.val.

NSYS = 3

In response to an NSYS value of 3, the environment module computes the operating point value of enthalpy (HENVOP) and updates the enthalpy data array.

NSYS = 8

In response to an NSYS value of 8, the environment module computes conditions at the operating altitude and Mach number. Subroutine FLCOND returns the stagnation values of temperature (TENVOP) and pressure (PENVOP). The operating point stagnation enthalpy (HENVOP) is computed by subroutine PROCOMNPSS, and the temp, press, and enthalpy data arrays are then updated.

NSYS = 5

In response to an NSYS value of 5, the environment computes the values of stagnation temperature (TENV) and pressure (PENV) based on the values of altitude (ALT) and Mach number (XMN). Altitude and Mach number values during the transient are dependent on the altitude and Mach number transient control schedules defined by the user. The altitude schedule allows the user to define the altitude at four time values during the transient. Similarly, the Mach number schedule allows the Mach number to be defined at four times during the transient. For a detailed description of defining transient control schedules, see Appendix D of [4].

Based on the current simulation time (TIME), the values of altitude (ALT) and Mach number(XMN) are computed. Subroutine FLCOND then computes the stagnation temperature (TENV) and pressure (PENV), and subroutine PROCOMNPSS then computes the stagnation enthalpy (HENV). These values are then used to update the temp, press and enthalpy data arrays, respectively. Also, the values of TENV, PENV, XMN, and ALT are then written to file comp.envr.val.

6.8 Mixing Volume

Module Description - The mixing volume module represents a control volume which defines the state conditions at the interface between connecting engine components. It is a mathematical entity which uses the dynamic forms of continuity, energy and state equations to solve for the values of stored mass, temperature and pressure in the volume (see Section 3).

Input and Output Ports - The mixing volume module has four input ports and four output ports allowing connection to four different upstream components and four different downstream components. The right-hand input port is a REQUIRED port and must be connected in order for the module to execute. Additional connections may be made to any of the remaining input ports. Figure 6-17 illustrates a mixing volume module connected in typical fashion.

IMPORTANT: The mixing volume is designed to be placed at the interface of connecting physical engine components.

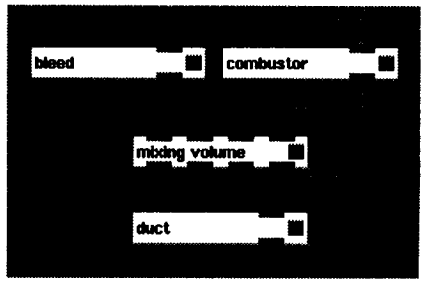


Figure 6.17 - Mixing Volume Module Shown Connected In Typical Fashion

Input Parameters - The following are the user-controllable parameters displayed in the mixing volume module's control panel (see Figure 6-18):

- name — A unique, user-defined, 10 character string which is used to identify the instance of mixing volume
- temperature — The design point stagnation temperature of the working fluid in the mixing volume, (° R)
- pressure — The design point stagnation temperature of the working fluid in the mixing volume, (Psia)
- volume — The component volume, (in3)
- temperature-op — The initial operating point stagnation temperature of the working fluid in the mixing volume, (° R)
- pressure-op — The initial operating point stagnation temperature of the working fluid in the mixing volume, (Psia)

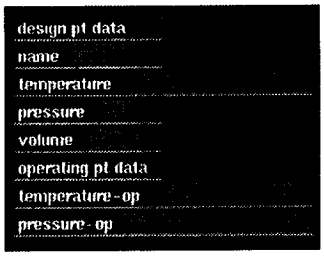


Figure 6.18 - Mixing Volume Module Control Panel

Module Execution - As described in the TESS Message Passing Under AVS section of Section 5, each module responds to certain NSYS values sent by the SYSTEM module. The following describes the mixing volume responses to various NSYS values:

NSYS = 1

In response to an NSYS value of 1, the mixing volume determines its unique component number (NCOMP). It then appends this information to the comp.list.no file along with its instance name, component type (mxvl), component code (0033), and solver flag (1).

NSYS = 9

In response to an NSYS value of 9, the mixing volume determines the components which are connected to its input ports and appends them to the comp.list.up file. There may be up to 4 (upstream) components connected to the mixing volume. The NAUXVAL values received at each input port are the component numbers of the upstream components. These component numbers are placed in the NUPCOMP array and saved to be used later in the simulation to access information about the upstream components. In the event that an input port is unconnected, its NSYS value will be zero. This will cause the corresponding NUPCOMP array term to be zero. Subsequently, any NUPCOMP array term which is zero must be ignored.

NSYS = 2

In response to an NSYS value of 2, the mixing volume writes its user-defined design point values of temperature, pressure and volume to the temp, press, and volume data arrays.

NSYS = 3

In response to an NSYS value of 3, the mixing volume computes the correction coefficient ECORR, in the following procedure.

The mixing volume module first determines the enthalpy, fuel-air ratio, and mass flow rate values for each of the upstream components. These values are stored in the HIND, FARIND, and WDOTID arrays respectively. A call to subroutine SETPREMIX returns the design fuel-air ratio (FARMVD) and enthalpy (HMVD) of the mixing volume.

The module then determines which components are connected to its output (downstream) ports. The component numbers for these components are placed in the NDNCOMP array and saved to be used later in the simulation to access information about the downstream components. Using this information, the mass flow rate values for the downstream components, WDOTOUTD, are obtained.

Next, the module determines if any of the upstream components are turbines or fuel-adders (combustors). Turbines remove heat and convert it to rotational energy, while fuel-adders add heat through combustion. The DIGTEM model assumes that heat addition and removal for turbines and fuel-adders takes place in the mixing volume just downstream of the component. Therefore, the mixing volume must account for these energy additions. The dynamic energy equation for the mixing volume includes the term, $\delta Q/dt$, which is used to account for the rate of heat addition ($\delta Q/dt > 0$) and rate of heat removal ($\delta Q/dt < 0$). To provide these heat addition/removal terms to the mixing volume, the turbines and fuel-adders compute their $\delta Q/dt$ terms and place them in energy data arrays (see the discussion on Turbines and Combustors in this Section).

The mixing volume obtains the energy values for the turbines and fuel-adders and these terms are placed in the ENGTERM array. Subroutine SETMIXVOL is then called to determine ECORR. The ECORR value is saved and used in subsequent computations for the mixing volume. The stored mass of the control volume, WSTOR, is then computed by subroutine VOLUMENPSS. Finally, the enthalpy, strdmass, and faratio data arrays for the mixing volume are then updated for HMVD, WSTOR, and FARMVD.

NSYS = 10

In response to an NSYS value of 10, the mixing volume writes its user-defined operating point values of temperature and pressure to the temp and press data arrays.

NSYS = 8

In response to an NSYS value of 8, the mixing volume determines the operating point values of fuel-air ratio, and mass flow rate for each of the upstream components. These are stored in the FAROP, and WDOTOP arrays, respectively. A call to subroutine SETPREMIX returns the operating point fuel-air ratio (FARMVOP) and enthalpy (HMVOP) of the mixing volume, and the faratio and enthalpy data arrays are then updated.

NSYS = 7

In response to an NSYS value of 7, the mixing volume computes pressure in the mixing volume using the ideal gas law. The temperature (TMV), stored mass (WSTOR) and volume (VMVD) of the mixing volume are obtained from the temp, strdmass, and volume data arrays. These values of temperature, and stored mass have already been updated by the solver. These values are used in the ideal gas law equation to determine the pressure in the mixing volume, PMV. The press data array for the mixing volume is then updated

NSYS = 5

In response to an NSYS value of 5, the mixing volume determines values of fuel-air ratio, and mass flow rate values for each of the upstream components. These values are stored in the FARIN, and WDOTIN arrays respectively. The fuel-air ratio and enthalpy of the mixing volume are then computed so that they may be available for the downstream module.

The module gets the temperature (TMV) and stored mass(WSTOR) values for the mixing volume from the temp and strdmass data arrays and they are used in the call to subroutine SETPREMIX. The fuel-air ratio (FARMV) and enthalpy (HMV) of the mixing volume are returned by this call and used to update the faratio and enthalpy data arrays, respectively.

NSYS = 6

In response to an NSYS value of 6, the mixing volume computes the enthalpy, fuel-air ratio, temperature derivative, and stored-mass derivative values in the following procedure.

The enthalpy, fuel-air ratio, fuel mass flow rate and mass flow rate values for each of the upstream components is first determined. These values are stored in the HIN, FARIN, WFL, and WDOTIN arrays respectively. The temperature (TMV), pressure (PMV), and stored mass (WSTOR) are retrieved from the data arrays. The call to subroutine SETPREMIX returns the fuel-air ratio (FARMV) and enthalpy (HMV) of the mixing volume.

The mass flow rate values for the downstream components, WDOTOUT, are obtained. Any energy terms from upstream fuel-adders or turbines are placed in the ENGTERM array. Subroutine MIXVOL is then called to determine the stored mass derivative (DWMV) and the energy derivative term, DEMV. DEMV is sent to subroutine VOLINT, which determines the temperature derivative term, DTMV.

The enthalpy, faratio, temp_deriv and strdmass_deriv data array values for the mixing volume are then updated.

6.9 Nozzle

Module Description - The nozzle module represents a converging-diverging or converging-only nozzle. Nozzle throat and exit areas may be adjusted during the transient by user-defined schedules.

Input and Output Ports - The nozzle module has a single input port which is to be to an upstream mixing volume component. It is assumed that the nozzle exits to the atmosphere, therefore the nozzle output port MUST be connected to the SYSTEM-END module. Since there is only a single input port, it is the REQUIRED port and must be connected in order for the module to execute. Figure 6-19 illustrates a nozzle module connected in typical fashion.

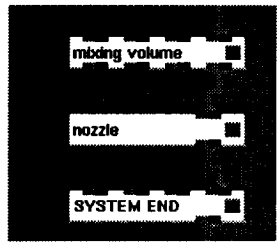


Figure 6.19 - Nozzle Module Shown Connected In Typical Fashion

Input Parameters - The following are the user-controllable parameters displayed in the nozzle module's control panel (see Figure 6-20):

• name	A unique, user-defined, 10 character string which is used to identify the instance of nozzle module
• mass flow rate	The design point nozzle mass flow rate, (lbm/sec)
• throat area	The design point nozzle throat cross-sectional area, (in2)
• exit area	The design point nozzle exit cross-sectional area, (in2)
• drag coefficient	The design point nozzle coefficient of drag, (dimensionless)
• velocity coefficient	The design point nozzle velocity coefficient, (dimensionless)
• gross thrust	The design point nozzle gross thrust, (lbf)
• throat area-op	The operating point nozzle throat cross-sectional area, (in2)
• exit area-op	The operating point nozzle exit cross-sectional area, (in2)
• transient control	Toggle button to control visibility

of pop-up windows containing schedulestransient control schedule parameters for throat area and exit area

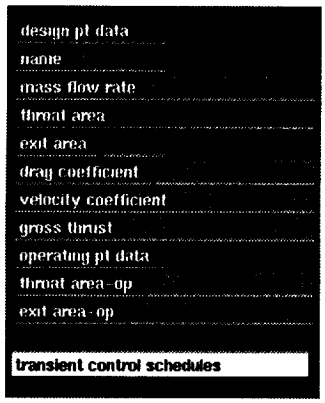


Figure 6.20 - Nozzle Module Control Panel

Module Execution - As described in the TESS Message Passing Under AVS section of Section 5, each module responds to certain NSYS values sent by the SYSTEM module. The following describes the nozzle module's responses to various NSYS values:

NSYS = 1

In response to an NSYS value of 1, the nozzle determines its unique component number (NCOMP). It then appends this information to the comp.list.no file along with its instance name, component type (nozz), component code (0101), and solver flag (0).

NSYS = 9

In response to an NSYS value less than 1000, the nozzle determines the component which is connected to its input port and appends it to the comp.list.up file. The NAUXVAL value received at the input port is the component number of the upstream component. This component number is defined as NUPCOMP.

NSYS = 2

In response to an NSYS value which is greater than zero and of 2, the nozzle writes the design point value of mass flow rate (WDOTID) to the massflow data array.

NSYS = 3

In response to an NSYS value of 3, the nozzle module computes the thrust correction coefficient (FCORR) and the mass flow correction coefficient (WCORR) based on the design point values of throat cross-sectional area (ATHROD) and exit cross-sectional area (AEXITD).

The module first obtains the design point fuel-air ratio (FARIND), temperature (TIND) and pressure (PIND) from the upstream mixing volume. In order to compute the correction coefficients, the nozzle must determine the design point temperature and pressure at its outlet. Since the nozzle exits to the atmosphere, these values are the temperature and pressure defined by the environment component and are listed in the comp.envr.val file. The nozzle module reads that file to obtain the exit design temperature (TOUTD) and pressure (POUTD) and flight Mach number (XMND), and then calls subroutine SETNOZZLE to compute WCORR and FCORR. The faratio data array is then updated using the FARIND value.

NSYS = 8

In response to an NSYS value of 8, the nozzle module computes the operating point values of mass flow rate and gross thrust based on the operating point values of throat cross-sectional area (ATHROP) and exit cross-sectional area (AEXITOP).

The module first obtains the operating point fuel-air ratio (FAROP), temperature (TINOP) and pressure (PINOP) from the upstream mixing volume. The exit operating point temperature (TOUTOP) and pressure (POUTOP) and flight Mach number (XMNOP) are read from the comp.envr.val file and then used in the call to subroutine NOZZLE to compute mass flow rate (WDOTOP). FAROP and WDOTOP are then used to update the faratio and massflow data arrays, respectively.

NSYS = 5

In response to an NSYS value of 5, the nozzle computes the values of mass flow rate and gross thrust based on the values of throat cross-sectional area (ATHROAT) and exit cross-sectional area (AEXIT). Throat and exit area values during the transient are dependent on the throat and exit area transient control schedules defined by the user. The throat area schedule allows the user to define the throat area at four time values during the transient. Similarly, the exit area schedule allows the exit area to be defined at four times during the transient. For a more detailed description of defining transient control schedules, see Appendix D of [4].

First the nozzle module obtains the fuel-air ratio (FARIN), temperature (TIN) and pressure (PIN) from the upstream mixing volume. The exit temperature (TOUT) and pressure (POUT) and flight Mach number (XMN) are read from the comp.envr.val file.

Based on the current simulation time (TIME), the values of throat area (ATHROAT) and exit area (AEXIT) are computed. Subroutine NOZZLE is then called to compute the nozzle gross thrust (FNOZOP) and mass flow rate (WDOTOP). FARINOP, WDOTOP and FNOZOP are then used to update the faratio, massflow, and thrust data arrays, respectively.

6.10 Shaft

Module Description

The shaft module represents a rotating shaft used to transmit rotational energy between compressor and

turbine components. It uses the dynamic form of the angular momentum equation to solve for the rotor speed (rpm) of the shaft (see Section 3).

Input and Output Ports - The shaft module has four input ports and four output ports allowing connection to four different upstream components (typically compressors) and four different downstream components (typically turbines). The right-hand input port is a REQUIRED port and must be connected in order for the module to execute. Additional connections may be made to any of the remaining input ports. Figure 6-21 illustrates a shaft module connected in typical fashion.

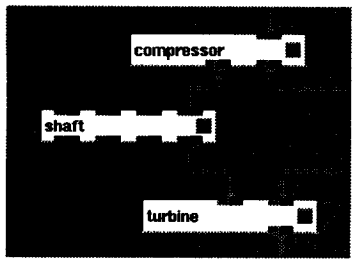


Figure 6.21 - Shaft Module Shown Connected In Typical Fashion

Multiple input and output ports in the shaft module give the ability to connect multiple components to a common shaft. This is particularly useful in modeling a split fan where information about the hub and tip sections are known. Two compressor components may be used to model the hub and tip. The multiple input ports on the shaft module allow both compressors to be connected to the shaft while having a single connection to the turbine.

Input Parameters - The following are the user-controllable parameters displayed in the shaft module's control panel (see Figure 6-22):

- name — A unique, user-defined, 10 character string which is used to identify the instance of the shaft
- moment inertia — The design point polar moment of inertia value for the compressor(s)-shaft-turbine(s) assembly, (lbf-in-sec2)
- spool speed — The design point shaft speed, (rpm)
- spool speed-op — The operating point shaft speed, (rpm)

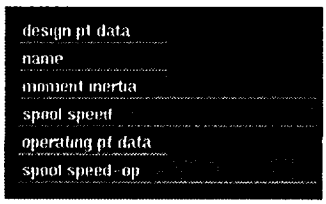


Figure 6.22 - Shaft Module Control Panel

Module Execution - As described in the TESS Message Passing Under AVS section of Section 5, each module responds to certain NSYS values sent by the SYSTEM module. The following describes the shaft modules responses to various NSYS values:

NSYS = 1

In response to an NSYS value of 1, the shaft determines its unique component number (NCOMP). It then appends this information to the comp.list.no file along with its instance name, component type (shft), component code (0055), and solver flag (1).

NSYS = 9

In response to an NSYS value of 9, the shaft determines the components which are connected to its input ports and appends them to the comp.list.up file. There may be up to 4 (upstream) components connected to the shaft. The NAUXVAL values received at each input port are the component numbers of the upstream components. These component numbers are placed in the NUPCOMP array and saved to be used later in the simulation to access information about the upstream components.

In the event that an input port is unconnected, its NSYS value will be zero. This will cause the corresponding NUPCOMP array term to be zero. Subsequently, any NUPCOMP array term which is zero must be ignored.

NSYS = 2

In response to an NSYS value of 2, the shaft writes its user-defined design point value of spool speed (XSPOOL) to the rpm data array.

NSYS = 4

In response to an NSYS value of 4, the shaft computes the correction coefficient ECORR, in the following procedure. The shaft module first obtains the energy term for each of the upstream components, which are typically compressors. These values are stored in the ECOMD array. The module then determines which components (typically turbines) are connected to its output (downstream) ports. The component numbers for these components are placed in the NDNCOMP array and saved to be used later in the simulation to access information about the downstream components. Using this information, the energy term for each of the downstream components are obtained and placed in the ETURD array. Subroutine SETSHAFT is then called to compute ECORR which is saved for use in subsequent calculations.

NSYS = 10

In response to an NSYS value of 10, the shaft writes its user-defined operating point value of spool speed to the rpm data array.

NSYS = 6

In response to an NSYS value of 6, the shaft computes the value of the spool speed derivative, DXSPL. The module first obtains the energy term for each of the upstream components. These values are stored in the ECOM array. Next, the energy term for each of the downstream components are obtained and placed in the ETUR array. The current value of the shaft spool speed (XSPL) is obtained from the rpm data array and used in the call to subroutine SHAFT to compute DXSPL. The rpm_deriv data array value for the shaft is then updated.

6.11 Turbine

Module Description - The turbine module represents a bleed-cooled turbine used to convert kinetic energy into mechanical energy.

Input and Output Ports - The turbine module has two input ports and one output port. The left-hand input port is to be connected to a shaft module and the right-hand input port is to be connected to a mixing volume. The single output port is to be connected to a mixing volume module. Both input ports are a REQUIRED port and must be connected in order for the module to execute.

IMPORTANT: Because the turbine module incorporates bleed cooling flow, a bleed module must be present and correctly connected. The bleed module used to provide cooling flow must be connected to the mixing volume connected to the output port of the turbine. This is illustrated in Figure 6-23.

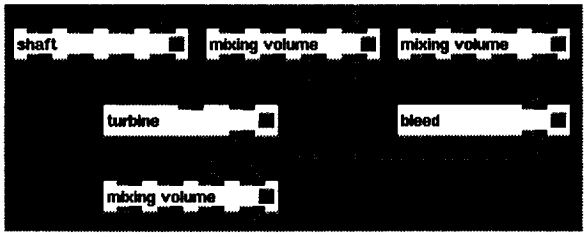


Figure 6.23 - Turbine Module Shown Connected In Typical Fashion

Input Parameters - The following are the user-controllable parameters displayed in the turbine module's control panel (see Figure 6-24):

- name — A unique, user-defined, 10 character string which is used to identify the instance of the turbine.
- mass flow rate — The design point turbine mass flow rate, (lbm/sec)
- enthalpy drop — The design point enthalpy drop, (Btu/lbm)
- bleed work pct — The design point percentage bleed flow doing work on the turbine, (%)
- base performance map — The name (and path) of the file containing baseline performance data for filenamethe turbine

Module Execution - As described in the TESS Message Passing Under AVS section of Section 5, each module responds to certain NSYS values sent by the SYSTEM module. The following describes the turbine modules responses to various NSYS values:

Figure 6.24 - Turbine Module Control Panel

NSYS = 1

In response to an NSYS value of 1, the turbine determines its unique component number (NCOMP). It then appends this information to the comp.list.no file along with its instance name, component type (turb), component code (0066), and solver flag (0).

NSYS = 9

In response to an NSYS value of 9, the turbine determines the components which are connected to its input ports and appends them to the comp.list.up file. The NAUXVAL values received at each input port are the component numbers of the upstream components. These component numbers are placed in the NUPCOMP array and saved to be used later in the simulation to access information about the upstream components.

NSYS = 2

In response to an NSYS value of 2, the turbine writes its user-defined design point value of mass flow rate (WDOTID) to the massflow data array.

NSYS = 3

In response to an NSYS value of 3, the turbine computes the correction coefficient WCORR, in the following procedure.

The turbine module first determines which input port is connected to the shaft and which is connected to the mixing volume. Although the user should connect the left-hand input port to the shaft and the right-hand input port to the mixing volume, this check allows the module to operate correctly even if the user has connected the

modules improperly. The component number for the mixing volume is NUPCOMP1 and the shaft component number is NUPCOMP2. Both are saved to be used in subsequent calculations.

The design point temperature (TIND) and pressure (PIND) are obtained from the upstream mixing volume and the design point spool speed (XSPOLD) is obtained from the shaft. The turbine then determines the component number of the mixing volume connected to its output port (NDNCOMP). The design point temperature (TOUTD) and pressure (POUTD) are then obtained from the downstream mixing volume.

To determine the design bleed cooling mass flow rate across the turbine it is necessary to determine which bleed(s) is connected to the mixing volume connected to the turbine's output port. The file comp.list.dn (see Section 5.1.1) is searched to determine if there are any bleeds which have a downstream component number identical to NDNCOMP, which is the component number of the mixing volume component connected to the output port of the turbine. For each of the bleeds that satisfy this requirement, the design mass flow rate of the bleed is obtained and summed to give the total design bleed flow used to cool the turbines (WBLEDD). This means that WBLEDD is the sum of all bleed mass flow entering the mixing volume and the bleed percentage work value will be applied to this total sum.

Subroutine GETGEMAP reads in performance data for the turbine using the string BFILENAME which contains the filename (and path) of the file which the user has selected using the file browser in the turbine module control panel. This map contains baseline performance data for the turbine, which has been normalized to the turbine design point. Subroutine SEARCHMAP is called to determine the design point value of mass flow rate (WPD) and enthalpy drop (HPD). The mass flow correction coefficient, WCORR and the enthalpy drop correction coefficient, HCORR, are computed and the values saved for use in subsequent computations. Also, the design point energy term, EOUTD, which represents the rate of heat energy removal by the turbine ($\delta Q/dt < 0$), is computed based on the design point values of enthalpy drop (DHD), mass flow rate (WDOTID) and the design bleed cooling flow (WBLEDD).

The enthalpy and fuel-air ratio in the turbine, (HTURB) and (FARTURB), respectively, are set equal to the enthalpy and fuel-air ratio of the upstream mixing volume and those values along with EOUTD are placed in the enthalpy, faratio and energy data arrays for the turbine.

NSYS = 8

In response to an NSYS value of 8, the turbine determines the operating point values of fuel-air ratio, and mass flow rate.

The operating point bleed cooling mass flow rate across the turbine it is obtained and summed to give the total operating bleed flow used to cool the turbines (WBLEEDOP). The operating point temperature (TINOP) and pressure (PINOP) are then obtained from the upstream mixing volume and the operating point spool speed (XSPOOLOP) is obtained from the shaft. The operating point temperature (TOUTOP) and pressure (POUTOP) are then obtained from the downstream mixing volume.

The points to be located on the performance map, XTUR and YTUR, are computed next. XTUR represents the operating point pressure ratio across the turbine (POUTOP/PINOP) normalized to the design point pressure ratio (POUTD/PIND). YTUR represents the ratio of corrected operating point spool speed (XSPOOLOP/sqrt(TINOP)) normalized to the corrected design spool speed (XSPOLD/sqrt(TIND)). Subroutine SEARCHMAP is called to determine the value of mass flow rate (WP) based on the input values of XTUR and YTUR. This values is then used to determine the operating point value mass flow rate (WDOTOP).

The fuel-air ratio in the turbine (FARTURB) is set equal to the fuel-air ratio of the upstream mixing volume and is placed in the faratio data array for the turbine. Also, the mass flow rate (WDOTOP) is placed in the massflow data array.

NSYS = 5

In response to an NSYS value of 5, the turbine determines the values of fuel-air ratio, and mass flow rate. The bleed cooling mass flow rate across the turbine it is obtained and summed to give the total bleed flow used to cool the turbines (WBLEED). The temperature (TIN) and pressure (PIN) are then obtained from the upstream mixing volume and the spool speed (XSPOOL) is obtained from the shaft. The temperature (TOUT) and pressure (POUT) are then obtained from the downstream mixing volume.

The points to be located on the performance map, XTUR and YTUR, are computed next. XTUR represents the pressure ratio across the turbine (POUT/PIN) normalized to the design point pressure ratio (POUTD/PIND). YTUR represents the ratio of corrected spool speed (XSPOOL/sqrt(TIN)) normalized to the corrected design spool speed (XSPOLD/sqrt(TIND))

Subroutine SEARCHMAP is called to determine the value of mass flow rate (WP) and enthalpy drop (HP) based on the input values of XTUR and YTUR. These values are used to determine the values of enthalpy drop, (DH), mass flow rate (WDOTIN) and energy term, EOUT.

The enthalpy and fuel-air ratio in the turbine, (HTURB) and (FARTUR), respectively, are set equal to the enthalpy and fuel-air ratio of the upstream mixing volume and those values along with EOUT are placed in the enthalpy, faratio and energy data arrays for the turbine.

7.0 Simulation Comparison

In order to verify the TESS code, a test engine model based on the DIGTEM test case (see reference 8), was constructed and analyzed. This Section discusses the development of the TESS equivalent of the DIGTEM test engine and the difficulties encountered.

The DIGTEM test case represents a two-spool, two stream augmented turbofan engine. Figure 7.1 shows a schematic representation of that engine. Figure 7.2 depicts the DIGTEM analytical model which represents the test engine.

7.1 DIGTEM Fan Model Incompatibility with TESS

In using TESS to model the DIGTEM test engine, it was found that the fan model used in DIGTEM did not fit the object-based format used in TESS. To illustrate this problem, consider the DIGTEM fan model as shown in Figure 7.3.

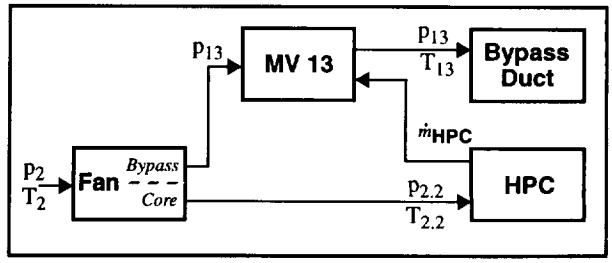


Figure 7.3 - DIGTEM Fan Model

The air mass flow into the fan is split into the bypass and core sections. The core flow passes into the high-pressure compressor (HPC), and the bypass flow goes into the Bypass duct. Notice that in the DIGTEM model,

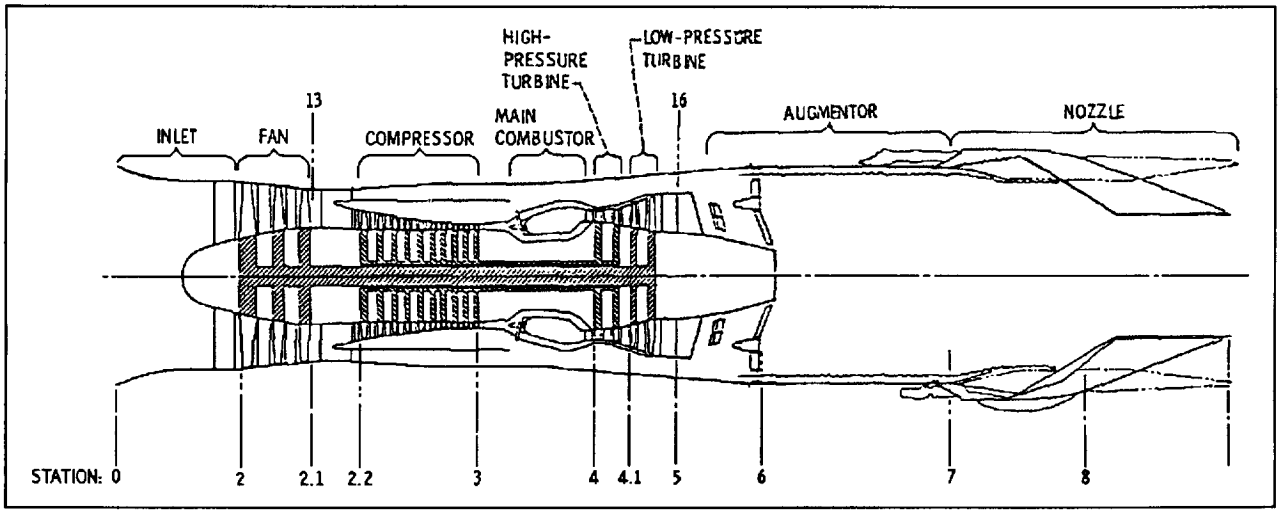


Figure 7.1 - Schematic Representation of Test Engine

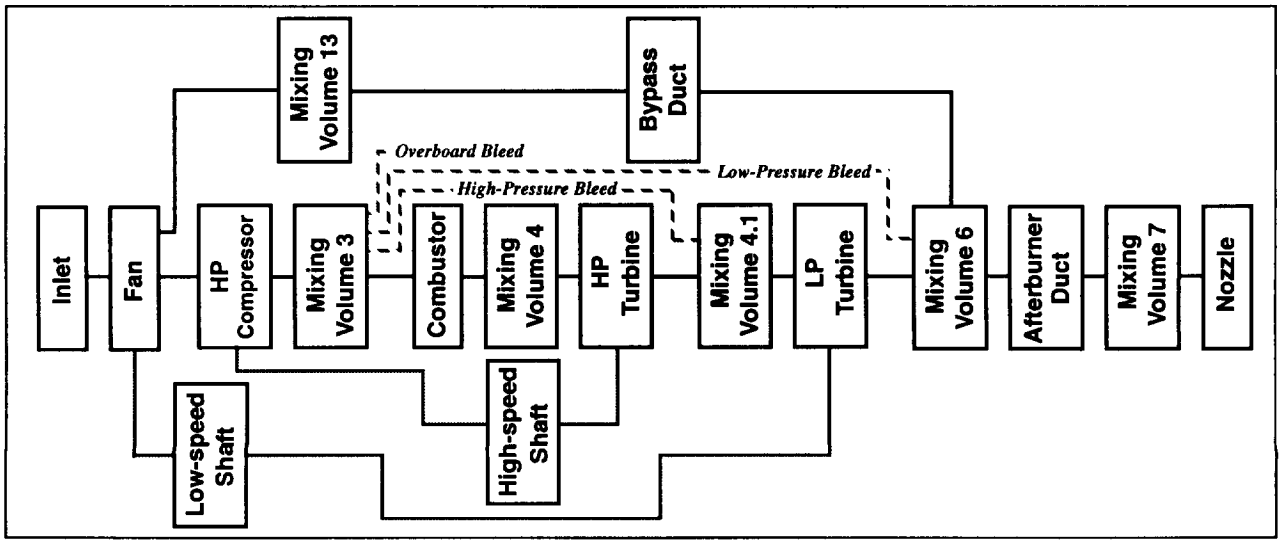


Figure 7.2 - Analytical Model of Test Engine

no mixing volume is placed between the Fan and HPC to provide an exit temperature and pressure for the core section with which to determine the mass flow rate value from the performance map. Instead, the DIGTEM fan model uses the bypass pressure ratio (p_{13}/p_2) and the fan shaft speed to determine the mass flow rate though the entire fan from the performance map. It also determines the pressure ratio across the Fan core ($p_{2.2}/p_2$) from the map data at the same time. This value is then used to determine the temperature at the exit of the Fan core ($T_{2.2}$) using isentropic relations. $T_{2.2}$ and $p_{2.2}$ are then used by the HPC to determine its mass flow rate ($\dot{m}_{HPC}$) from the HPC performance map. The HPC mass flow rate represents the mass flow rate through the core of the Fan and can then be used with the total mass flow rate value through the fan to determine the mass flow rate in the Bypass Duct:

$$\dot{m}_{Bypass} = \dot{m}_{Fan,\,total} - \dot{m}_{HPC}$$

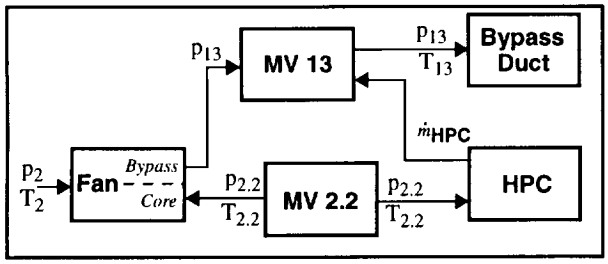


Figure 7.4 - Revised DIGTEM Fan Model

This is the only instance in the DIGTEM model where isentropic relations are used instead of a mixing volume to provide temperature and pressure values. To provide consistent use of a mixing volume to determine temperature and pressure at component interfaces, TESS requires that a mixing volume be used between all physical flow components. Thus, the TESS implementation of the DIGTEM fan section shown in Figure 7.3 will have a mixing volume between the Fan and the HPC, as shown in Figure 7.4.

Because all other physical components have only a single upstream mixing volume and a single downstream mixing volume, the fan shown in this configuration would not be consistent with the other components. It was decided that the generality and consistency-of-use of each of the components were of paramount importance in TESS, thus, the DIGTEM fan model was reconfigured to be able to use the more general compressor model provided in TESS. Two possible solutions were considered to solve the problem of modeling the fan.

In the first approach, the bypass and core performance data was "smeared" into a single set of performance data representing overall fan performance. A single compressor component could then be used to model the fan. This, however, meant that there would only be a single mixing volume between the fan and the HPC, and the fan and the bypass duct (see Figure 7.5), and thus only single values of temperature and pressure for both the bypass and core sections.

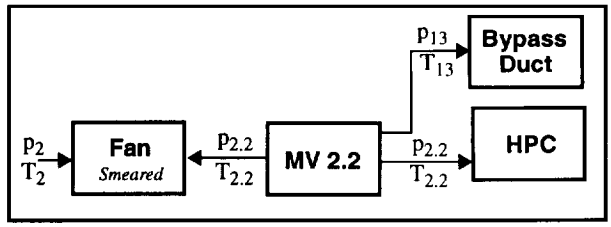


Figure 7.5 - TESS "Smeared-Fan" Configuration.

Comparing the DIGTEM test output values of temperature and pressure in mixing volumes 2.2 and 13, it was found that the values are relatively close, so this approach could be considered in this case. For engine models where this is suspected not to be the case, the second approach may be considered.

In the second approach, the fan is split into two separate components, with the core and bypass sections each being modeled by a single compressor (see Figure 7.6)[1]. Using this approach, the conditions at the exits of the core and bypass are independent, unlike that of the first approach.

In order to implement the split-fan approach, the efficiency, pressure ratio, and mass flow rate for both the core and bypass were needed create separate performance maps. The DIGTEM fan performance data provided the efficiency and pressure ratio for each, but the mass flow rate data was for the entire mass flow rate through the fan (i.e., bypass + core).

To aid in matching the operating point conditions in order to compare the TESS results with DIGTEM, the mass flow data was scaled so that the mass flow rates in the bypass and core matched the corresponding DIGTEM values at the low-power operating point (Operating point 3) and the design point (Operating point 1). In this manner, two separate performance data maps were created.

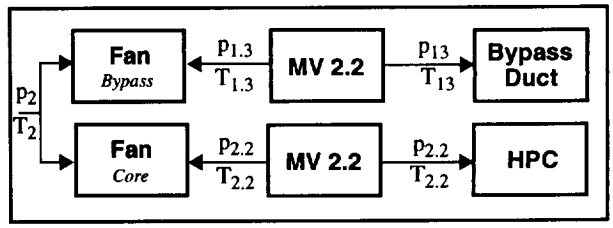


Figure 7.6 - TESS "Split-Fan" Configuration

7.2 TESS Test Engine Configurations

Two TESS engine configurations corresponding to the DIGTEM test engine case were created. These

1. Although not indicated in this figure, both the fan core and fan bypass compressor components are connected to the same shaft. This is accomplished by the fact that the shaft module allows multiple input (and output) connections.

correspond to the Smeared-Fan configuration model (shown in Figure 7.7) and the Split-Fan configuration model (shown in Figure 7.8).

7.3 Results of the Test Engine Simulations

The test case used in each of the three engine models represents a three second transient. The engine is simulated as being on a static test stand (i.e. sea level conditions and a Mach number of zero). The lowest engine power operating point is used as the initial operating point for the transient. The engine is then accelerated to the design operating point in the first two seconds of the transient and then held at that point for the remainder of the transient. Engine acceleration is controlled by the Combustor Fuel Mass Flow Rate control schedule, which is depicted in Figure 7.9. The variable geometry controls for each of the compressors fan (represented as LPC) and HPC are controlled by their respective control schedules shown in Figure 7.10. The nozzle throat and exit areas are held constant at their initial operating point values and the altitude and Mach number of the engine are zero.

Each of the engine models were run to a balanced condition at the initial operating point using the Newton-Raphson method according to the control parameter values shown in Table 7.1. Upon convergence, the transient was begun using the Improved Euler numerical method. The control parameters for the Improved Euler method are shown in Table 7.2

Some of the results of the transient are plotted in the Figures 7.11 (a) through (e). Shown are: Nozzle Gross Thrust, Low-Speed Spool speeds, High-Speed Spool speeds, Combustor pressure and, High Pressure Turbine inlet temperature, as functions of time.

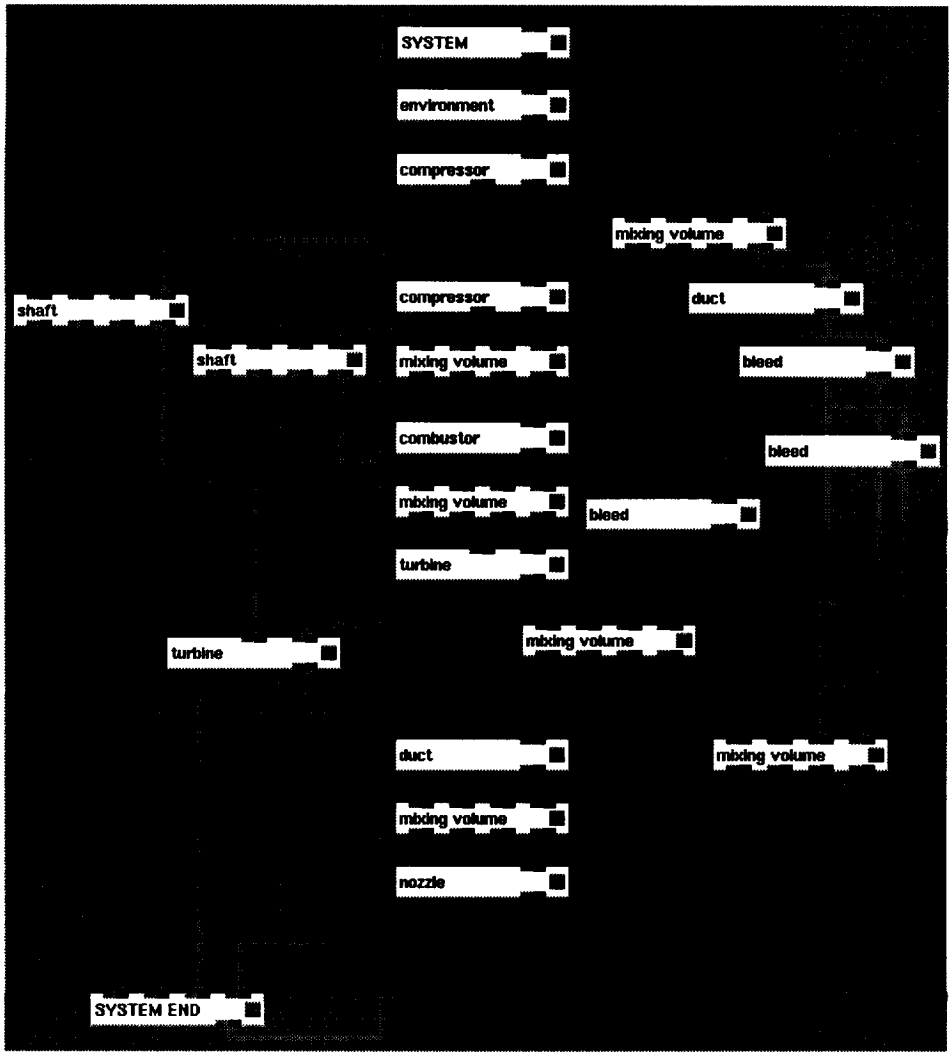


Figure 7.7 - TESS Smeared-fan Test Engine

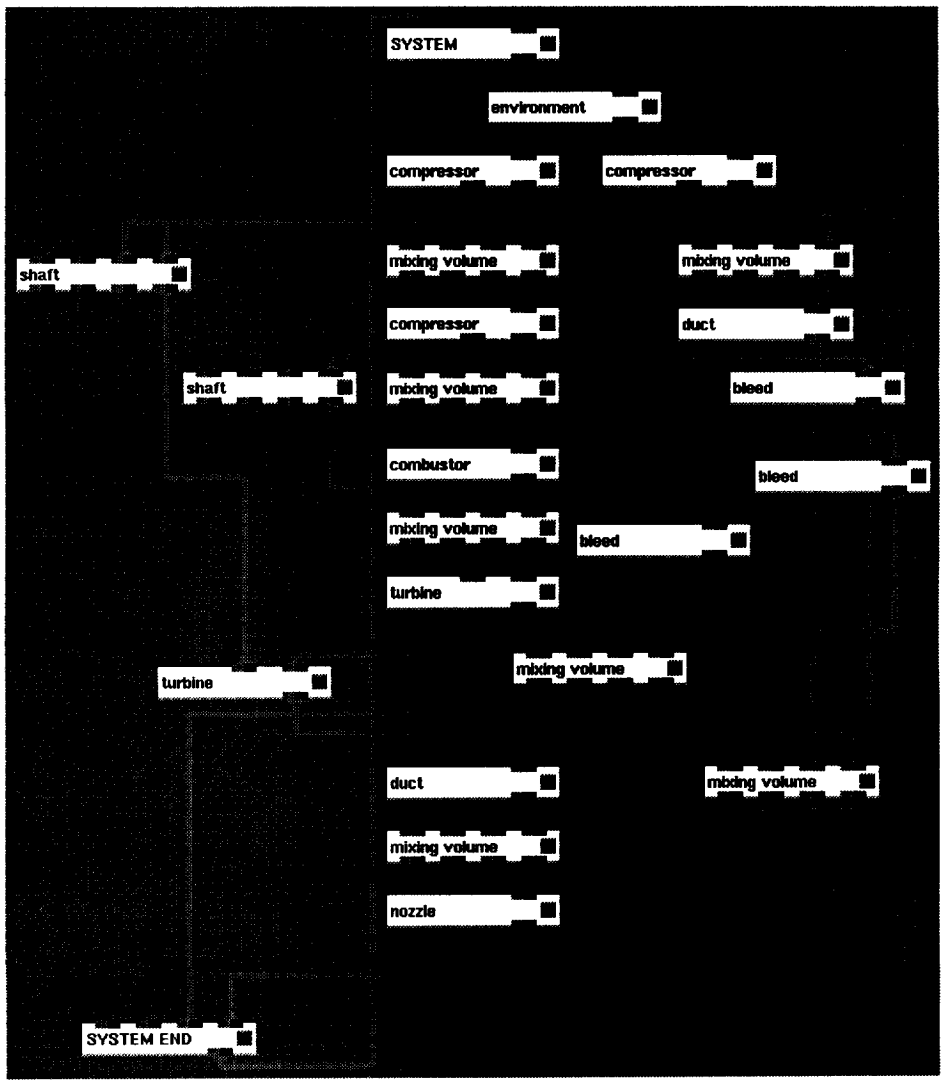


Figure 7.8 - TESS Split-fan Test Engine

Parameter	Value
Fraction (FRAC)	0.25
Convergence Tolerance	0.0005
Lower Partial Limit	0.001
Upper Partial Limit	0.01
Maximum Iterations	50
Convergence Rate (TOLPCG)	0.7

Table 7.1 - Newton-Raphson Control Parameters.

Parameter	Value
Fraction (FRAC)	0.25
Convergence Tolerance	0.0005
Lower Partial Limit	0.001
Upper Partial Limit	0.01
Maximum Iterations	50
Convergence Rate (TOLPCG)	0.7
Time Step (Δ t)	0.1 sec

Table 7.2 - Improved Euler Control Parameters.

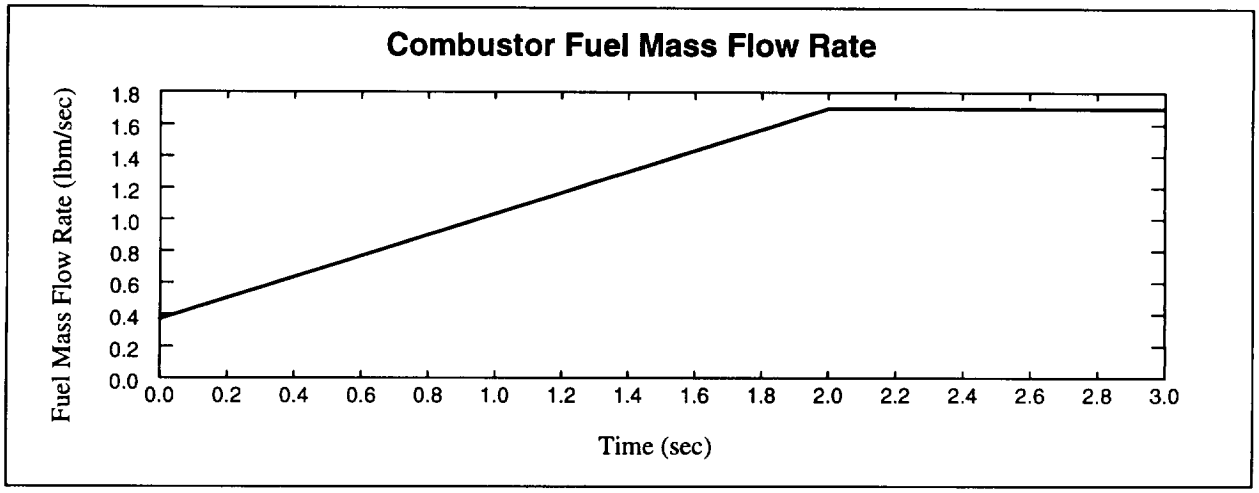


Figure 7.9 - Combustor Fuel Mass Flow Rate Control Schedule

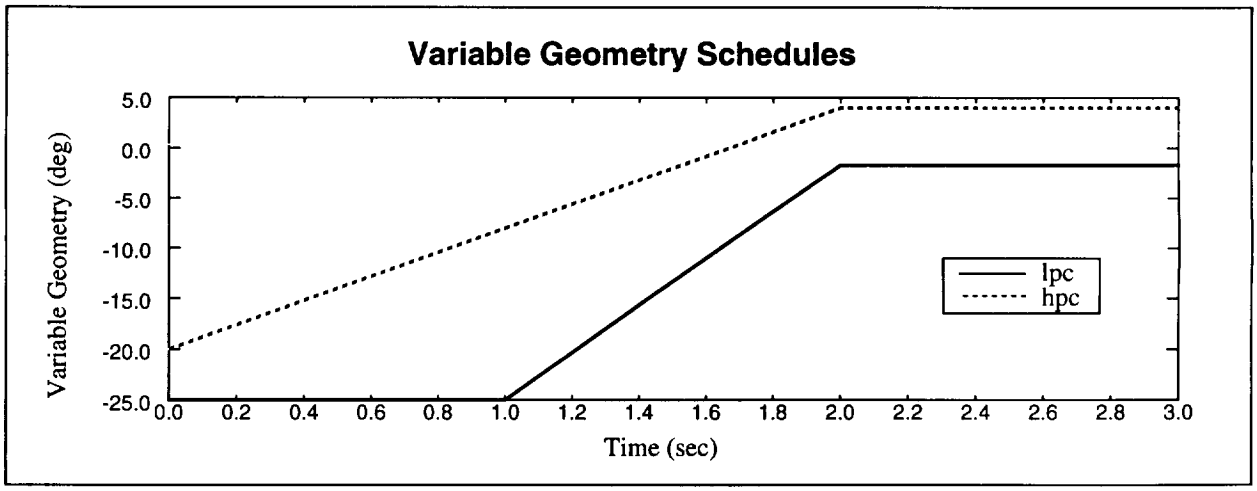


Figure 7.10 - Compressor Variable Geometry Control Schedules

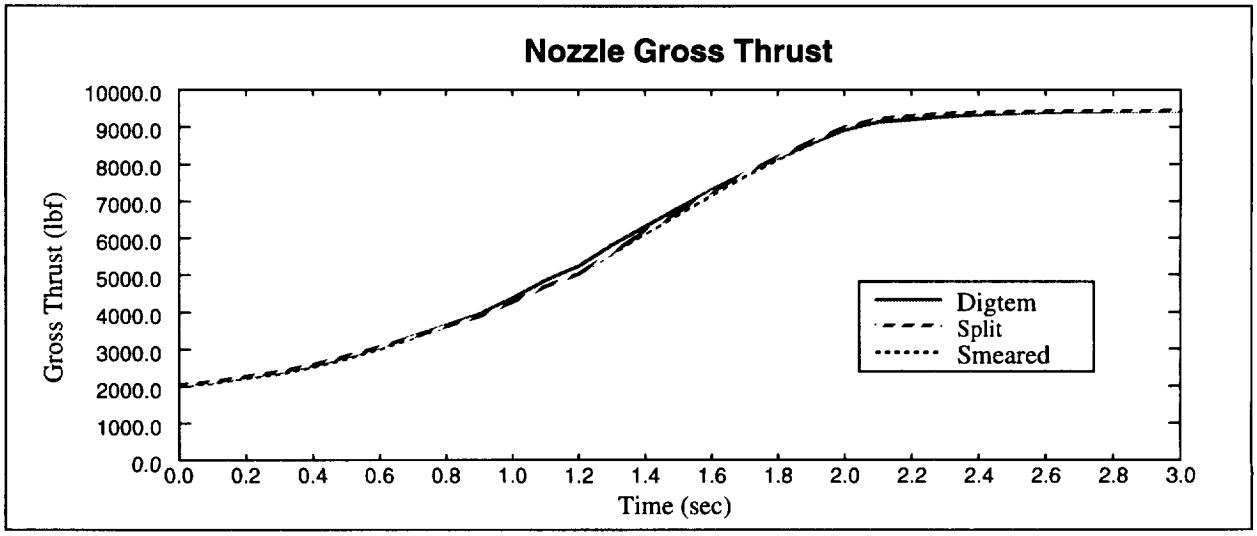


Figure 7.11 (a) - Nozzle Gross Thrust Transient Plot

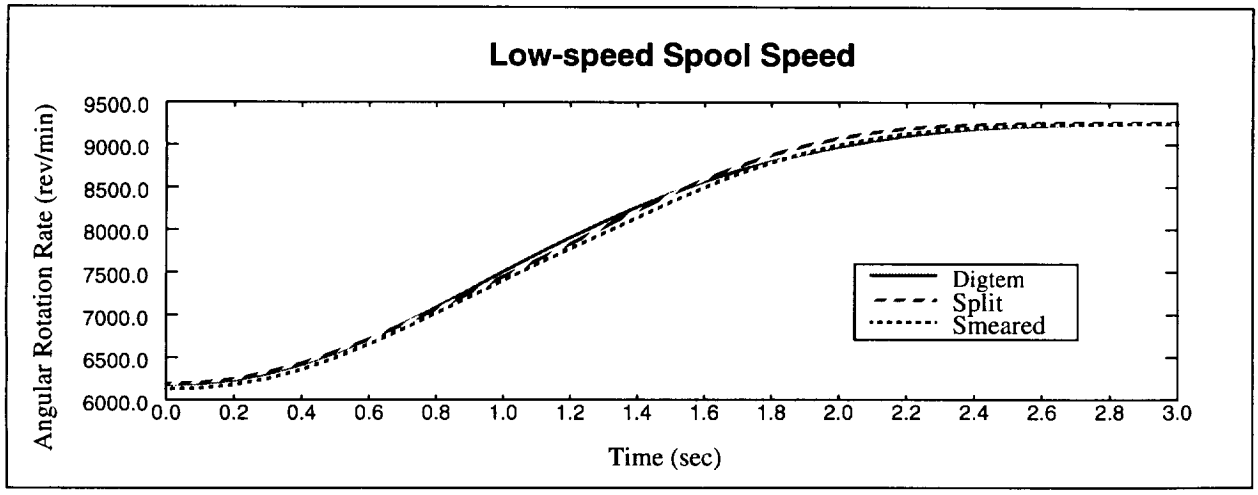


Figure 7.11 (b) -Low-speed Spool Transient Plot

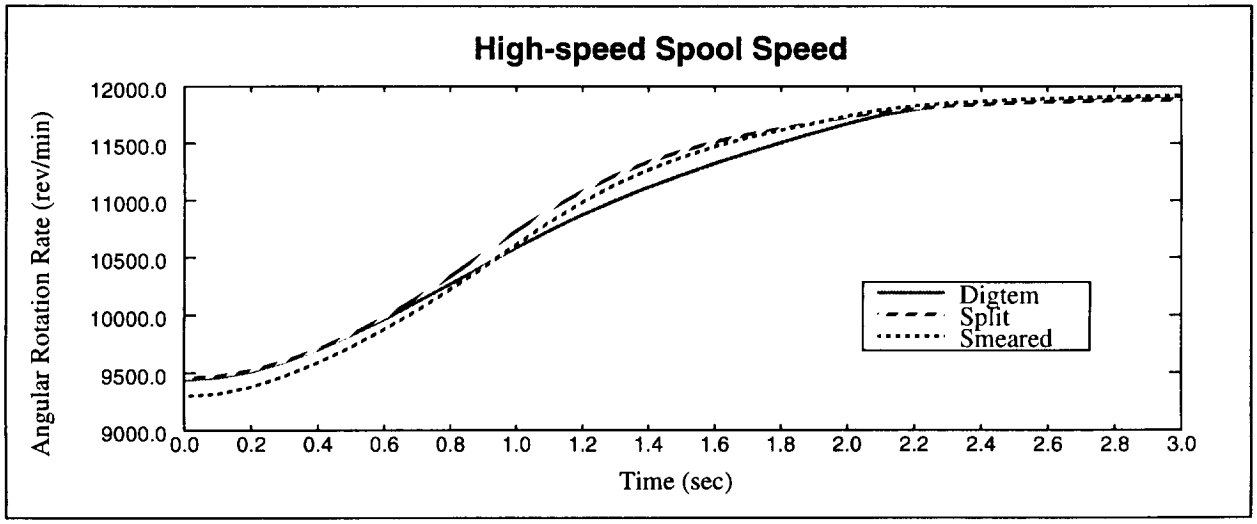


Figure 7.11 (c) - High-speed Spool Transient Plot

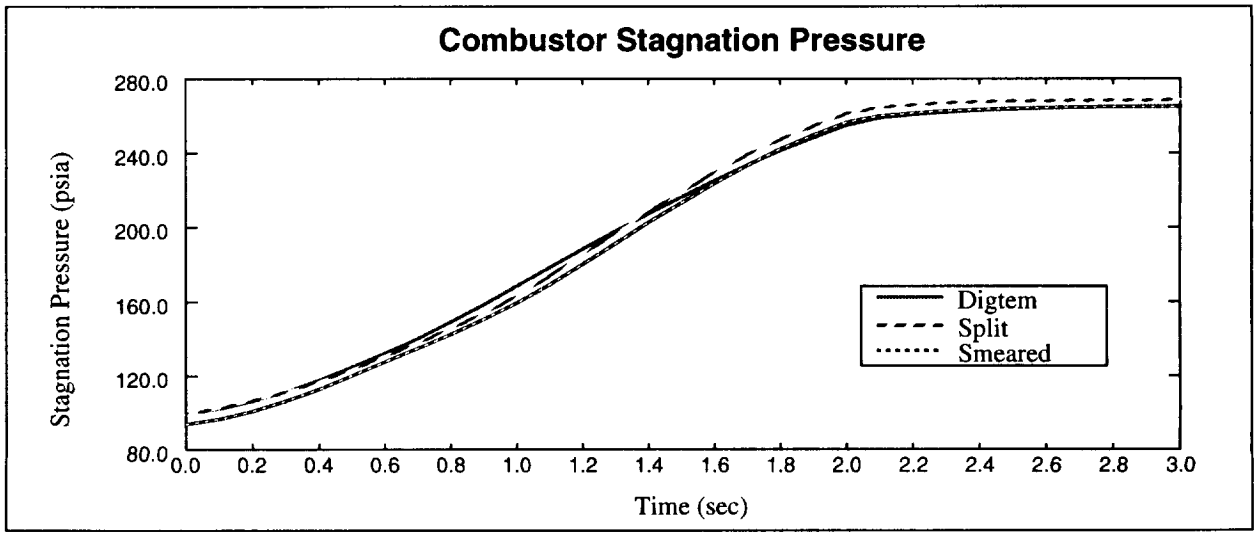


Figure 7.11 (d) - Combustor Stagnation Pressure Transient Plot

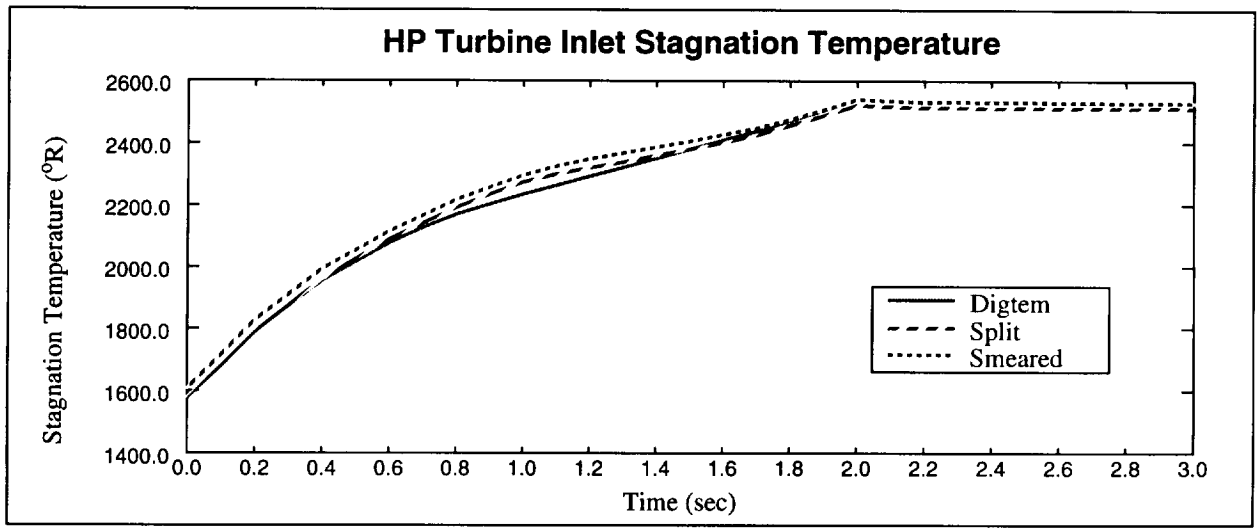


Figure 7.11 (e) -Turbine Inlet Stagnation Temperature Transient Plot

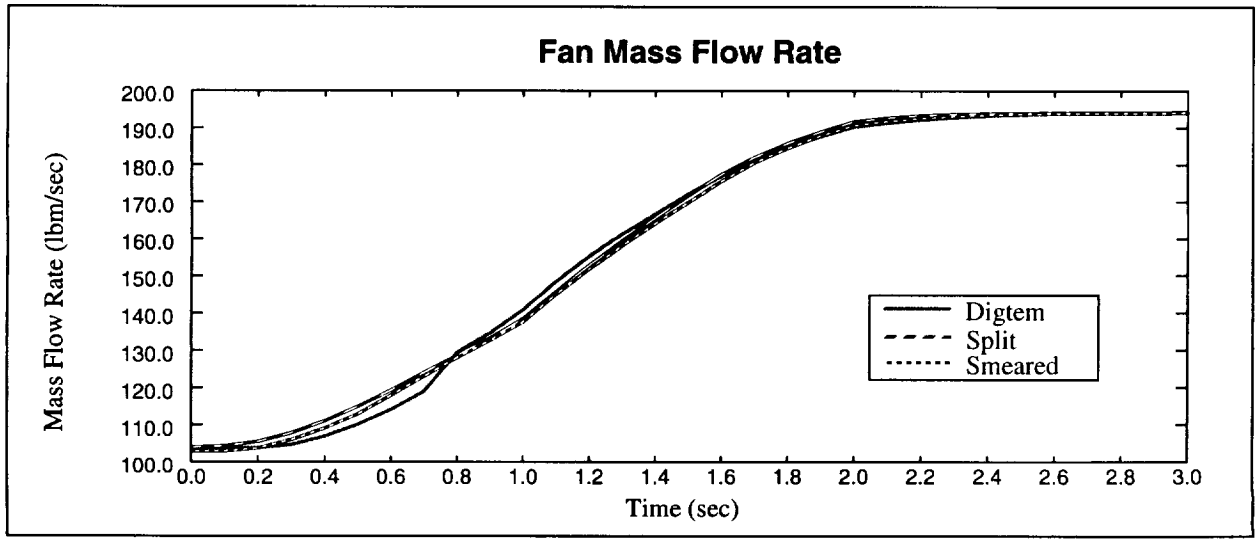


Figure 7.12 - Comparison of Fan Mass Flow Rates

The results indicate that while the values for each of the three different models do not agree precisely, they are very close. The reason for the discrepancies is due to the difficulty in matching the mass flow rates in the LPC (fan). Figure 7.12 shows a plot of the mass flow rate in the fan for each of the different models. As can be seen, neither the smeared nor the split fan models match identically with the DIGTEM model. The discrepancies introduced by the different mass flow rate values propagate downstream through the engine causing differences in mass flow rates in other components, which in turn affects the other engine state values.

8.0 Zooming

In this section, a prototype implementation of the *zooming* concept, which is to be utilized in the Numerical Propulsion System Simulation (NPSS) project, is described. Zooming allows for codes which model at different levels of fidelity to be integrated within a single simulation providing the user with the ability to "zoom in" and investigate relevant physical processes occurring in an engine component. Due to the computationally intensive nature of higher fidelity codes, and the methods utilized to carry out the zooming, parallel and distributed programming techniques are utilized.

This section describes the development of a framework which allows a higher-fidelity flow-solver, such as a 2-D axi-symmetric or fully 3-D Euler/Navier-Stokes solver, to be integrated within the low-fidelity NPSS propulsion system simulator. The low-fidelity system simulator used is the Turbofan Engine System Simulator (TESS) described in the previous sections of this report, and the high-fidelity flow solver is the Advanced Ducted Propfan Analysis Code (ADPAC) [10]. The PVM parallel distributed message passing package [11] is used to handle the communication and distribution of data between the ADPAC simulations placed on a heterogeneous network of workstations.

8.1 Introduction

Traditionally, propulsion system simulations have utilized mathematical models based on estimated or empirically determined component characteristics to represent the complete "baseline" engine model. Typically, this is a one-dimensional flow path model which is used to determine engine system performance, system dynamics and provide controls modeling. The operational characteristics of the individual system components are supplied in the form of "performance maps", which along with design geometry information and dynamic information, can be used to calculate engine thrust and weight, as well as engine performance over a wide range of operating conditions for both steady-state and transient simulations. However, to provide descriptions of the physical processes occurring in an engine component beyond that supplied by a performance map, or to create data for cases when performance data are not available, requires the use of a higher fidelity component simulation.

Recent advances in simulation methods and computational power have made possible such analysis models. These models are more useful since they provide the ability to simulate both component and inter-component flow details during the cycle, providing for a more direct accounting and analysis of interactions among engine components. By integrating a higher fidelity component simulation within a low-fidelity system simulation, it is possible to "zoom in" on the relevant flow physics occurring in the component. Such a scheme would allow a designer to quickly and efficiently determine the effects of component design changes on the operation of both a component as well as the entire propulsion system.

The current research is focused on "zooming in" on the fan component of an aircraft propulsion system. This is depicted in Figure 8.1. Here the fan component of a one-dimensional "baseline" engine model has been "zoomed" to a three-dimensional analysis. This type of detailed analysis would be performed, for example, to study the effects of a new fan blade design, sudden inlet distortions, tip treatment modifications, etc. on engine response.

Implementation of the zooming concept is difficult, due mainly to the inability to accurately resolve high-fidelity flow fields from a single value as supplied from the low-fidelity system simulator. In order for the zooming to be accurate, the upstream and downstream boundary values (which are single valued), must be extrapolated to define a suitable three-dimensional distribution of field variables such that when integrated over, the original single-valued boundary conditions are recovered.

To illustrate this, consider the "zooming" simulation for the fan component as depicted in Figure 8.2. Here the one-dimensional fan component provides the inlet boundary values of stagnation pressure (p_0), stagnation temperature (T_0), and Mach number (Ma); and the exit boundary value of static pressure (p). These single-valued boundary conditions are then extrapolated to appropriate three-dimensional field distributions which are applied as boundary conditions to the high-fidelity solution domain. These boundary conditions are subsequently used to determine the flow field solutions which satisfy the three-dimensional Navier-Stokes or Euler equations for the given domain. Next, the flow field variables are integrated to determine the space-averaged values of mass-flow rate, stagnation pressure ratio, and stagnation temperature ratio. The averaged stagnation pressure ratio value is then compared with the stagnation pressure ratio defined across the one-dimensional model. If the two ratios match, then the extrapolated field distributions are proved to be suitable representations and the averaged values may be used in the one-dimensional simulation. Typically, however, the space-averaged stagnation pressure ratio will not initially match the low-fidelity simulator pressure ratio, and the three-dimensional boundary condition representations must be redefined and the above process repeated until the necessary values match.

Initially, an iterative approach was used to attempt to match the boundary conditions. However, it was found that this is extremely difficult, requiring many iterations to match the boundary conditions and subsequently causing a significant increase in the simulation time. Furthermore, due to the numerical methods used to solve the system of equations governing the engine operation, using an iterative method sometimes caused the system to enter into an oscillatory mode where convergence could not be achieved.

Another possible solution method to the problem of matching the boundary values is to parallelize the high-fidelity simulations. This is accomplished by distributing several identical copies of the high-fidelity flow solver across a collection of machines. The host machine (i.e., the machine running the low-fidelity simulation) passes the necessary parameters to the flow solvers on the remote machines and the solvers are then run in parallel using slightly different boundary values. Upon completion, each of the individual flow solutions are integrated to determine the single-valued flow parameters (as described above), and these values are returned to the host machine. This process creates a single-curve performance map which can be used to match the flow parameters impressed by the low-fidelity simulator.

By taking advantage of the parallel nature of this method, it is possible to reduce the simulation time to approximately that of a single high-fidelity simulation. Furthermore, the distributive nature of the method is naturally suited to the application of computationally intensive high-fidelity flow solvers, as these can be distributed on the appropriate high-performance computer platform such as a super-computer.

8.2 Zooming Framework

The zooming strategy which utilizes the parallelization of the high-fidelity flow solvers described in the previous section has been implemented in a prototypical zooming framework consisting of the following computer codes and systems:

- TESS - A propulsion system simulator running within the *Application Visualization System.*
- ADPAC - A fully three-dimensional Navier-Stokes/Euler flow analysis package capable of providing detailed flow analysis of the fan component in a turbofan engine.
- PVM - A parallel distributed message-passing package.

This section presents an overview of TESS, ADPAC and PVM, and describes the prototype zooming framework formed by the combination of these systems.

8.2.1 Turbofan Engine System Simulator (TESS)

TESS is an object-based, one-dimensional, transient, thermodynamic aircraft engine simulator which runs under the Application Visualization System (AVS). This integrated system provides the graphical user interface

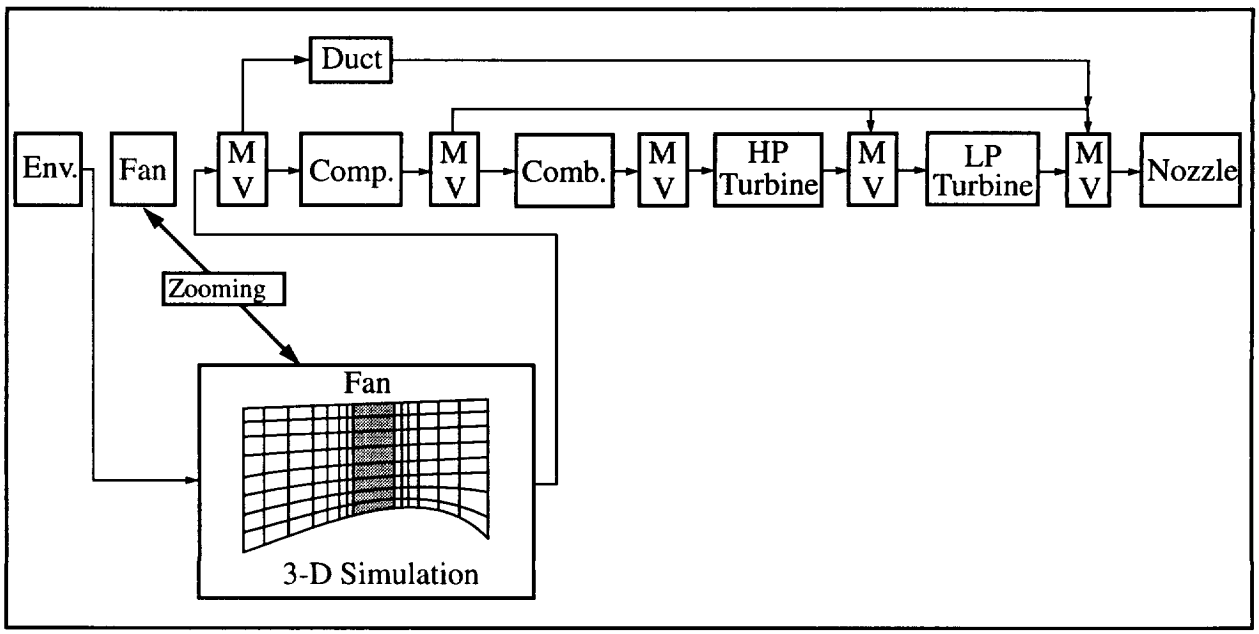


Figure 8.1 - A data flow network representing "zooming" on the fan component

and operating environment for graphical construction of arbitrary engine configurations, selecting and controlling steady-state and transient engine operation, and graphical display of simulations results.

Engine components (e.g., compressor, turbine, duct, etc.) are represented graphically as AVS module icons, or simply modules, and are initially located in the AVS Module Library area. Engine component modules are selected by "dragging" the desired module from the library area into the Workspace area. When a module is brought into the Workspace, a control panel for that component is displayed in the left hand window. Here the operational characteristics of the component are defined by the user (e.g., the component name, mass flow rate, design point performance data). This process is repeated for each of the desired engine components to be included in the engine model. Once these modules have been arranged in the Workspace, the user uses the mouse to connect the modules to establish the physical connections of the engine.

Once all of the components have been connected and their input data entered, the user can select the length of time for the transient, and define how the governing equations are to be solved numerically for both the steady-state and transient portions of the simulation. Currently, for steady-state solutions, the user may choose either Newton-Raphson or Fourth-order Runge-Kutta methods. For transient solutions, the user may choose either Modified Euler, Fourth-order Runge-Kutta, Adams, or Gear methods. When simulation execution is begun, TESS first attempts to balance the engine at the initial operating point using the steady-state balancing method. Once the engine is balanced, the transient is begun and proceeds up to the number of simulation seconds defined by the user.

8.2.2 Advanced Ducted Propfan Analysis Code (ADPAC)

The high-fidelity flow solver program used in the current research to model the operation of the fan component is the Advanced Ducted Propfan Analysis Code (ADPAC). ADPAC is a three-dimensional Euler/ Navier-Stokes numerical analysis tool developed to study high-speed ducted propfan aircraft propulsions systems. The program utilizes a three-dimensional, time-marching numerical procedure along with a flexible, coupled 2-D/3-D multiple block geometric grid representation to predict the flowfield in and around the fan.

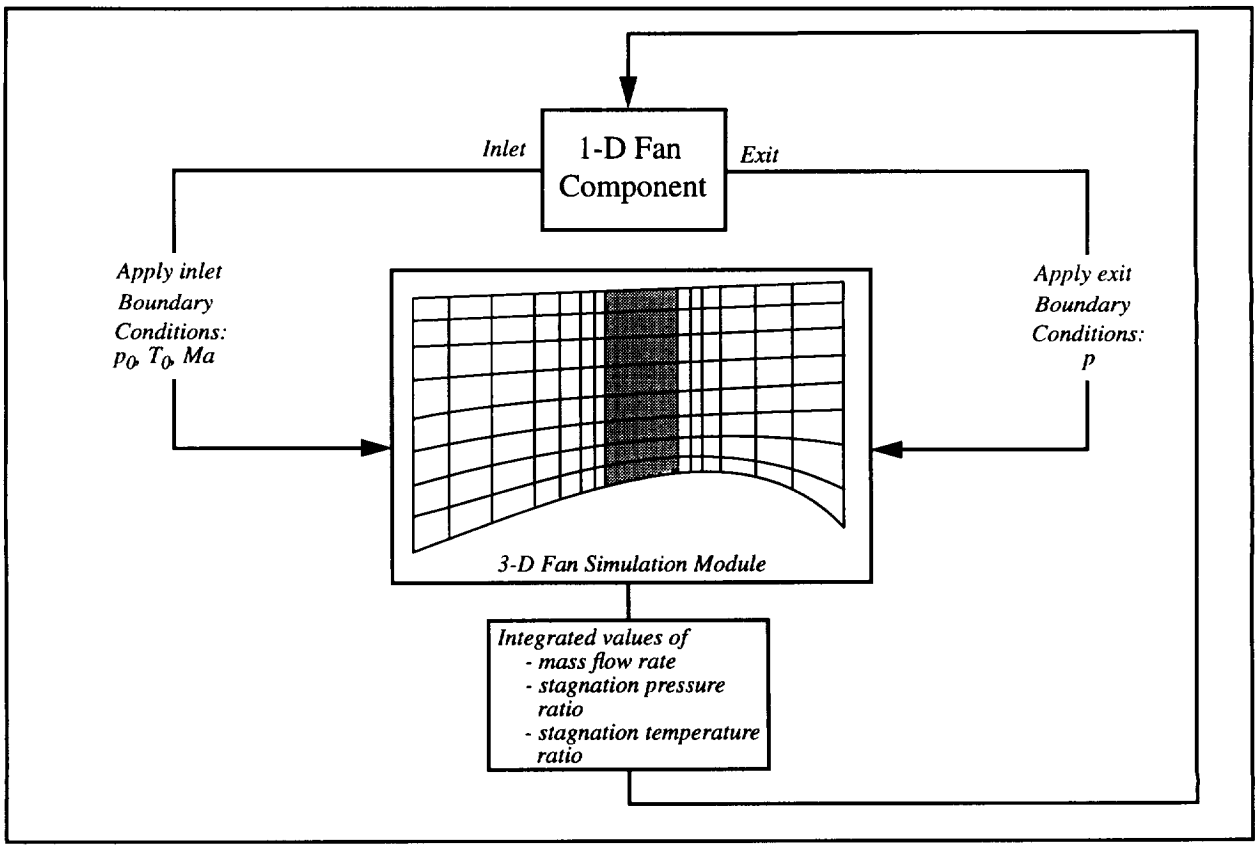


Figure 8.2 - Representation of boundary value extrapolation, interpolation, and matching

8.2.3 Parallel Virtual Machine (PVM)

The Parallel Virtual Machine (PVM) is a software system that permits a network of heterogeneous Unix computers to be used as a single large parallel computer. Using PVM, a user-defined collection of different computers, known as the *virtual machine*, is used to provide aggregate power for solving large computational problems.

The PVM system is composed of a daemon which resides on all of the computers making up the virtual machine, and a library of PVM interface routines which supply user-callable routines. These functions, along with the PVM daemon, allow a PVM application on one computer to automatically start up *tasks* (computational processes) on other computers in the virtual machine and communicate data to and from the task by using message-passing constructs.

8.3 Prototype Zooming System

A prototype zooming system has been constructed by combining TESS, ADPAC and PVM. This system is defined by two suites of codes: the first, residing on the local (host) machine, runs AVS and TESS; the second, residing on each of the remote machines, includes Unix shell scripts, FORTRAN code and the ADPAC executable. Additionally, PVM is resident on each machine which is to be used in distributing the zooming system.

A new TESS engine component module, *fan Multi-ADPAC*, was created to provide the user interface and functionality for the zooming system. In general, the module performs the following actions:

- Handles the basic data transfer between itself and the other engine component modules according to the TESS data passing paradigm
- Establishes and configures the PVM connections between the local and remote machines
- Spawns the remote tasks using PVM
- Controls the data transfer between the ADPAC simulations and TESS

To utilize the *fan Multi-ADPAC* module in a TESS engine simulation, the user need only start PVM on a local machine and drag the *fan Multi-ADPAC* module into an engine model. Defining the ADPAC control parameters and the remote machines on which to spawn the ADPAC simulations is accomplished through the TESS graphical user interface, which creates an AVS pop-up window containing the necessary input data widgets (see Figure 8.3).

Upon starting the TESS simulation data connections to the remote machines are established automatically by the *fan Multi-ADPAC* module by dynamically configuring the PVM virtual machine to include the user-defined list of remote machines. During the system's attempt to balance the engine at the initial operating point, fan performance data will be needed for calculations in TESS, and the *fan Multi-ADPAC* module will initiate the "zooming" process to compute the necessary fan flow data.

Figure 8.3 - fan Multi-ADPAC module control panels

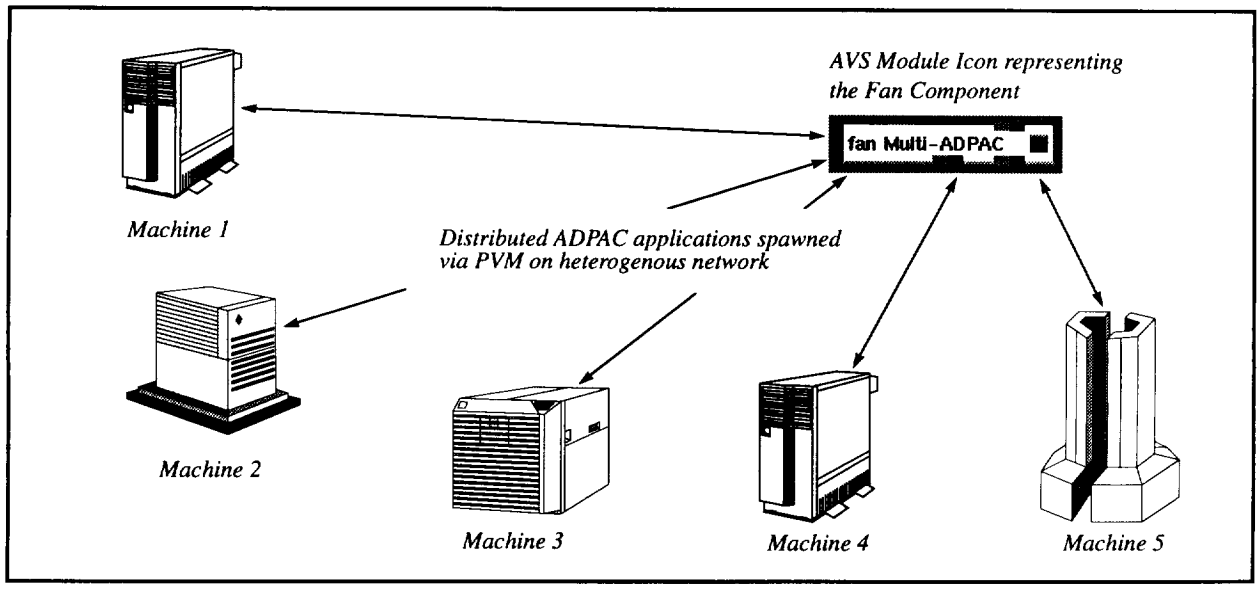


Figure 8.4 - Parallel distribution of ADPAC simulations on heterogeneous network

To begin the zooming process, the *fan Multi-ADPAC* code enters a loop which establishes a PVM connection to each of the remote machines, spawns a remote slave program called `adpacslave`, and broadcasts the boundary condition parameters from TESS to `adpacslave` (see Figure 8.4). The *fan Multi-ADPAC* module then begins a second loop to wait for the simulation results. Since all of the ADPAC simulation results must be returned to the *fan Multi-ADPAC* module before they can be used in TESS, a blocking receive function was utilized for simplicity. Because each of the spawned `adpacslave` tasks have unique data associated with them, the receive function checks each returned message for the appropriate task identifier (tid) and message tag (msgtag). This is necessary so that the space-averaged results may correctly match the corresponding boundary conditions.

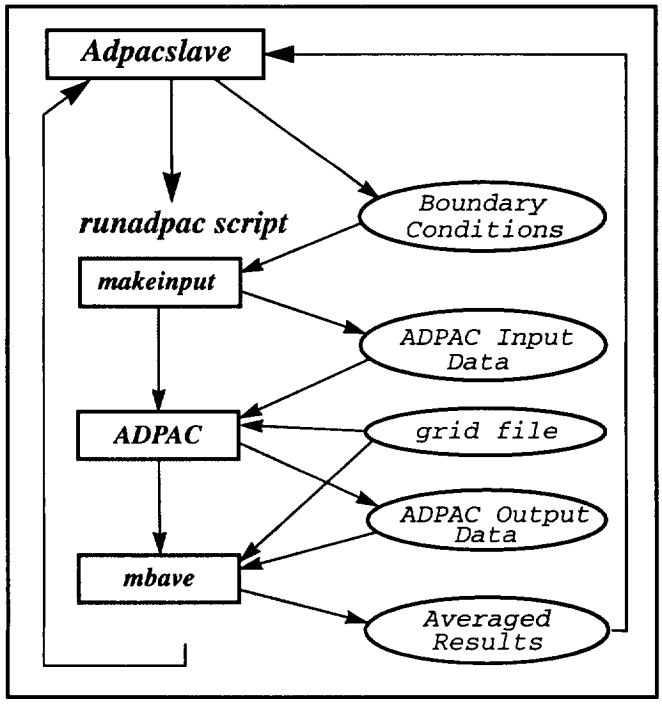


Figure 8.5 - Remote code suite flow chart

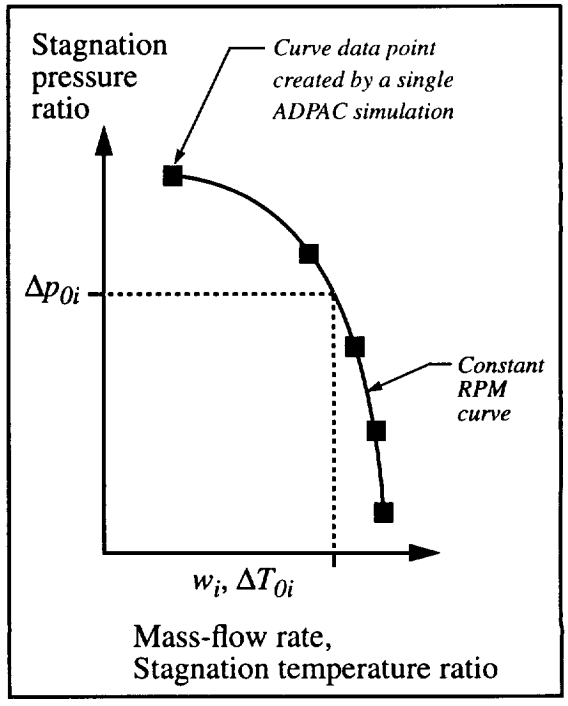


Figure 8.6 - Map curve created by zooming

The operation of the remote code suite is presented in the flowchart shown in Figure 8.5. When executed, `adpacslave` first writes the boundary conditions to a file, then performs a system call to execute a Unix shell script called `runadpac`. This script executes several FORTRAN executable programs:

- `makeinput` uses the boundary condition data to create an ADPAC input data file.
- `ADPAC` computes the fan flowfield solution using the ADPAC input data and grid data representing the fan component physical geometry.
- `mbave`, a multi-block averaging program, integrates the three-dimensional flow solution to give the single (space-averaged) flow values which are needed in the TESS system simulation.

After `mbave` has finished executing and output the space-averaged results to a file, the shell script exits and process control returns to `adpacslave` which takes the space-averaged values and returns them, via PVM, to the *fan Multi-ADPAC* module.

Once all of the remote tasks have run, the results are collected in the *fan Multi-ADPAC* module. This data creates a single curve where each of the data points on the curve represents the space-averaged variables computed by a single ADPAC simulation (see Figure 8.7). These data are then interpolated to match the stagnation pressure ratio across the fan, impressed by the TESS simulation, to determine the stagnation temperature ratio and mass flow rate. These values are then used by TESS to continue the complete propulsion system simulation.

The above process is repeated each time fan performance data is needed by TESS. To reduce overall simulation time, the space-averaged values are retained and used to create a performance map. Before running flow solutions, this data is checked to see if the current operating conditions are within the data range; if so, the data is interpolated and used in the system simulation. In this manner, the simulation time may be significantly reduced. This also has the added benefit of creating a performance map which can be used in subsequent non-zooming TESS simulations.

8.4 Results and Conclusions

To demonstrate the utility of this zooming simulation system, the fan component of the NASA/GE Energy Efficient Engine was zoomed and the steady-state operation of the engine studied. The zooming framework was successfully applied on a heterogenous system of Unix workstations connected over both local and wide-area networks. The local machine used was an SGI Iris 4D/440VGX computer located at the University of Toledo and the remote machines were IBM RS6000 computers located at NASA's Lewis Advanced Cluster Environment (LACE).

The model predictions were found not to significantly deviate from those of experimentally-based simulations. Results from the model simulations suggest that the developed strategy is adequate for modeling the inter-component interactions. However, the effectiveness of this method is dependent on many factors which affect the time needed for the simulation and the amount of computer resources needed. The following list several of the parameters which were found to influence the effectiveness of the zooming strategy:

- *initial flow start-up solutions*. This parameter affects the number of iterations of high fidelity solver needed to converge, which in turn increases the simulation time and increases the load on computer resources.
- *number of points per curve*. This parameter affects the ability of the system simulation to converge to steady-state or a transient time step. The accuracy of the "performance curve" interpolation, used to match the boundary conditions, is dependent on the number of ADPAC flow solutions being spawned. By supplying more points per curve, the accuracy is increased without a significant increase in overall simulation time. However, amount of computational resources needed is significantly increased.
- *numerical solver used by low-fidelity simulation*. The different numerical solvers offered in TESS were found to have very different effects on simulation time and resource use. This mainly manifested itself in the number of "zooming" calls which were needed to attain convergence, but also in the length of time the solver needed to achieve convergence.

9.0 LAPIN/TESS Zooming

The objective of this work was to evaluate the development and operation of a low-level zooming simulation by coupling the Large Perturbation Inlet (LAPIN) code with the TESS engine simulation environment running under the Application Visualization System (AVS). The low-level zooming is due to the fact that LAPIN, which utilizes a one-dimensional finite-difference simulation, models at a higher-fidelity than the TESS engine system, which uses a lumped-parameter approach. This provides the ability to determine axial distributions of flow field variables in the inlet without modeling the complete engine as one-dimensional.

To provide for a more efficient and flexible computing system, the capability for heterogeneous distribution of the LAPIN code has been incorporated. This allows LAPIN to be located on a remote computer platform and connected with the TESS system operating on a local computer host. The Parallel Virtual Machine (PVM) parallel distributed message passing package is used to handle the communication and distribution of data between LAPIN and TESS.

9.1 LAPIN/TESS Framework

The first step in integrating LAPIN with TESS was to create a new TESS engine component to represent a jet propulsion system inlet component. As with all engine components in the TESS system, this component is graphically represented in the AVS system by an AVS module icon (see Figure 9.1). This new module, called *Inlet/LAPIN* is located on the local host machine and contains the source for the LAPIN module which is loaded into the AVS system. This module provides a user interface for the LAPIN code as well as the standard AVS module icon which is used to graphically integrate LAPIN into an engine simulation. This module handles the data transfer between itself and the other engine component modules, establishes and configures the PVM connections between the local and remote machine, spawns the remote LAPIN task using PVM, and controls the data transfer between LAPIN and TESS.

Figure 9.1: Inlet/ LAPIN AVS Module Icon

To provide the most flexibility, a suite of codes (see Figure 9.2) was utilized to run LAPIN on the remote computer. In this manner, the LAPIN source code does not need to be modified in order to integrate LAPIN with TESS. The first program, *mkinput-lapin*, constructs the LAPIN input file using data from `lapin-bcfile.dat`, Then *LAPIN* executes, reading data from the input file and outputting results to `2lapin.out` which is read by the third program, *converge*. This program determines the converged inlet flow field values which are to be returned to TESS. The *slave-lapin* program controls the execution of the other three programs and handles the PVM communications with *Inlet/ LAPIN* receiving input data for *mkinput-lapin* and returning results from *converge*.

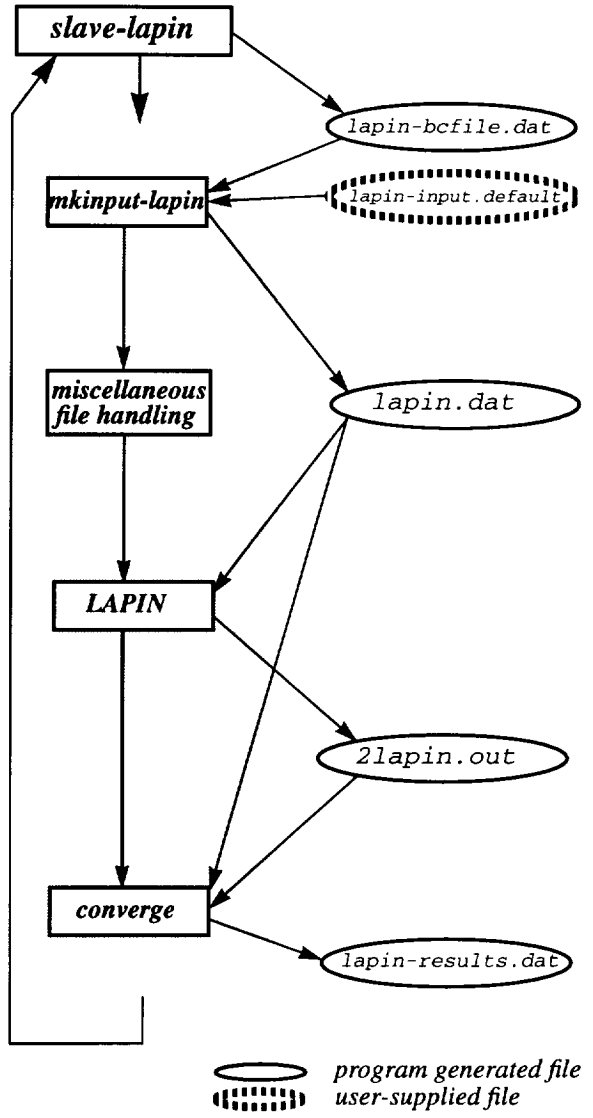


Figure 9.2: Slave-LAPIN Flow Chart

9.2 Integration Difficulties

There arose problems in integrating LAPIN within TESS as a result of differences in the simulation models used by LAPIN and TESS. The fact that LAPIN requires a mass flow down-stream boundary condition (BC) causes considerable difficulty in integrating LAPIN into TESS. TESS was designed to use stagnation (total) temperature (T_T) and stagnation (total) pressure (P_T) as the BC's of a component and to use these BC's to compute the mass flow rate in a component. In a normal TESS simulation, these stagnation temperature and stagnation pressure BC's are defined by the mixing volumes (or environment) located at the inlet and exit boundaries of each component.

For example in a compressor component (see Figure 9.3), the upstream stagnation pressure BC is supplied by the mixing volume located at the compressor inlet and the downstream

stagnation pressure BC is supplied by the mixing volume located at the compressor exit. These BC's are used to compute the stagnation pressure ratio across the component which is used to determining the mass flow rate in the compressor using a performance map. In turn, the mass flow rates of a component are used by a mixing volume to compute the dynamic mass flow changes between components.

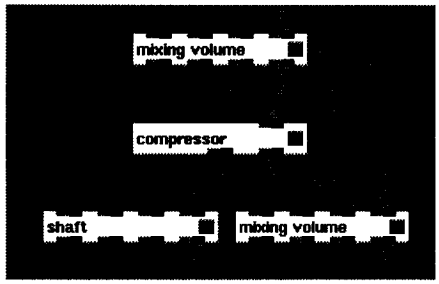


Figure 9.3: Typical Compressor set-up

The LAPIN code does not fit this model entirely: it *does* require inlet conditions of Mach number, static pressure and static temperature[1] which can be supplied from the TESS environment component, but LAPIN does not utilize pressure or temperature conditions at the exit boundary condition. Instead, LAPIN utilizes a mass flow rate value as the exit boundary condition. Furthermore, since the inlet exit mass flow rate is defined, there is no need to compute the mass flow rate in the inlet.

TESS operates by running the engine components in a sequential order which is derived from the path of fluid flow through the engine. Thus, the inlet/LAPIN component runs before any component connected to it's exit. For example, consider the system shown in Figure 9.4.

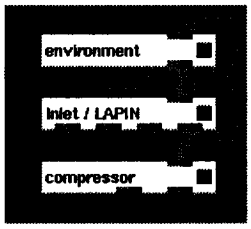


Figure 9.4: Inlet/LAPIN set-up

The environment component would run first, passing T_T, P_T, and Mach values to the Inlet/LAPIN code. Next, the LAPIN component would execute. However, at this point the compressor has not computed the mass flow rate value needed for the LAPIN down-stream BC. In order for the down-stream BC value of mass flow rate to be available, the compressor must execute first. However, in order for the compressor to run, it needs the T_T, P_T values from the inlet; which are not available if the compressor runs before the inlet/LAPIN component.

In the current implementation, it is assumed that the inlet exit values of T_T and P_T are unchanged from its inlet values of T_T and P_T. **This assumes that there are no losses due to shocks or viscous effects in the inlet**. The inlet/LAPIN component then runs first, but only passes the T_T and P_T values to the compressor, The compressor then runs to determine the mass flow rate in the compressor, and finally the inlet/LAPIN component is run to determine the inlet performance data.

9.3 Results

A LAPIN inlet model for the EEE engine was constructed and incorporated into an TESS network representing the EEE engine. The engine simulation was then run for a variety of transient engine operations ranging from take-off to cruise conditions with nominal results.

1. TESS provides stagnation values, but the static conditions can be derived from the stagnation conditions using the Mach number.

References

[1] Claus, R.W., Evans, A.L., Lytle, J.K., and Nichols, L.D. "Numerical Propulsion System Simulation," *Computing Systems in Engineering*, **2**, 4 (Apr. 1991), 357-364.

[2] Claus, R.W., Evans, A.L., Follen, G.J. "Multidisciplinary Propulsion Simulation Using NPSS," *4th AIAA/USAF/NASA/OAI Symposium on Multi-disciplinary Analysis and Optimization*, Cleveland, Ohio, Sep. 21-23, 1992.

[3] Advanced Visual Systems Inc. *AVS Developer's Guide* (Release 4.0), Part number: 320-0013-02, Rev. B, Advanced Visual Systems Inc., Waltham Mass,. May 1992.

[4] Reed, J.A. "Development of an Interactive Graphical Aircraft Propulsion System Simulator," Master of Science Thesis, University of Toledo, August 1993.

[5] G. Booch, "Object-Oriented Design: With Applications," Benjamin/Cummings Publishing Company, Inc., 1991.

[6] J. H. Keenan and J. Kays, "Gas Tables," John Wiley & Sons, Inc., 1948

[7] J. R. Szuch, "HYDES - A Generalized Hybrid Computer Program for Studying Turbojet or Turbofan Engine Dynamics," NASA TM X-3014, 1974.

[8] A. H. Shapiro, "The Dynamics and Thermodynamics of Compressible Fluid Flow: Vol I," Ronald Press Company, 1953

[9] C. J. Daniele, S. M. Krosel, J. R. Szuch, and E. J. Westerkamp, "Digital Computer Program for Generating Dynamic Turbofan Engine Models (DIGTEM)," NASA TM 83446, 1983.

[10] Hall, E.J., Delaney, R.A., and Bettner, J.L. "Investigation of Advanced Counterrotation Blade Configuration Concepts for High Speed Turboprop Systems, Task 5 - Unsteady Counterrotation Ducted Propfan Analysis Computer Program User's Manual," NASA CR-187125, Jan. 1993.

[11] Sunderam, V.S. "PVM: A Framework for Parallel Distributed Computing," *Journal of Concurrency: Practice and Experience*, **2** (4), (Dec. 1990), 315-339.