Creare

Engineering Research and Development

September 23, 2002

Dr. Jonathan B. Ransom
National Aeronautics and Space Administration
Langley Research Center
Attn: Mail Stop 240
Contract NAS1-01085
Hampton, VA 23681-2199

Subject: Fiscal Year Report
"Algorithms and Object-Oriented Software for Distributed Physics-Based Modeling"
NASA Contract No. NAS1-01085

Dear Jonathan:

Enclosed is the second fiscal year report which is being submitted in lieu of the fifteenth monthly progress report for the project titled "Algorithms and Object-Oriented software for Distributed Physics-Based Modeling" which Creare is performing for the National Aeronautics and Space Administration. This report covers work conducted during the period from 20 September 2001, to 19 September 2002. The next report is due October 29.

Also enclosed is the current version of our distributed simulation library. As was the case in previous transmittals, this is a working prototype but is not considered suitable for production calculations at this time. The code is transmitted with the understanding that it will not be released outside of NASA.

As of September 18, we have spent approximately \$219,839 out of the total funding amount of \$462,000.

We are looking forward to presenting a summary of this work to you and your colleagues on October 31. In the meantime, please contact me if you have any questions.

Sincerely,

Marc

Marc A. Kenton
Principal Investigator

8728.5/lam

Enclosure: Second Fiscal Year Report
Distributed Simulation Library (CD-ROM)

cc: Contract File
Jerry Bieszczad, Creare Inc.
Chip Audette, Creare Inc.
Robert Kline-Schoder, Creare Inc.
Contracting Officer, M/S 126
New Technology Representative, M/S 212
Center for Aerospace Information (CASI)

ALGORITHMS AND OBJECT-ORIENTED SOFTWARE FOR DISTRIBUTED PHYSICS-BASED MODELING

FISCAL YEAR REPORT
PERIOD: September 20, 2001–September 19, 2002

CONTRACT NAS1-01085
CREARE PROJECT 8728

Marc A. Kenton
Principal Investigator

Jerry Bieszczad
Engineer

Chip Audette
Engineer

Creare Incorporated
Hanover, NH 03755

September 23, 2002
8728

Creare

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 Project Motivation and Goals

Developing detailed computer simulations of aerospace vehicles and other engineered systems is difficult. As models of subsystems are combined to form a complete model of the overall system, there appears to be a virtually inevitable, geometric growth in the complexity of the resulting computer code, the difficulty of overcoming numerical instabilities, and the effort required to eliminate software bugs and perform quality assurance. Further, the complete model can be so unwieldy that uncertainty analysis and optimization efforts are impaired.

The project seeks to develop methods to address these issues. The goals are to reduce model development time, increase accuracy (e.g., by allowing the integration of multi-disciplinary models), facilitate collaboration by geographically-distributed groups of engineers, support uncertainty analysis and optimization, reduce hardware costs, and increase execution speeds. These problems are the subject of considerable contemporary research (e.g., Biedron et al., 1999; Heath and Dick, 2000).

All of these goals would be addressed if complicated system models with intricate interconnections and strong internal feedback mechanisms could be partitioned into subsystems that were executed semi-independently. Each subsystem would be modeled in detail by separate (possibly geographically-isolated) groups of engineers and then executed on a separate processor of a computer network. The development and implementation of such an approach is the subject of this project. The principal deliverables from the project will be mathematical techniques to allow the linking of the separate subsystem models, an object-oriented methodology for developing the models, and a distributed simulation software library that facilitates the process. The library is intended for use on ordinary personal computers connected by a local area network, as found in most engineering organizations, purpose-built "Beowulf" networks, as well as computers connected by the Internet.

The potential payoff of this technology is very large, in terms of hardware and software development costs, as well as design capabilities. Hardware costs would be reduced by allowing networked processors to be efficiently used for detailed simulation of complicated aerospace systems. Software costs would be reduced by a divide-and-conquer approach that directly attacks complexity, supports interdisciplinary modeling, promotes object-oriented techniques, and allows efficient re-use of component or subsystem models developed on earlier projects. Design and analysis efforts are facilitated by integrated uncertainty analysis and optimization capabilities.

1.2 Overview of Software Library

The primary focus of the project is on lumped parameter simulation models comprised of coupled algebraic equations (AEs) and ordinary differential equations (ODEs), and not, for example, single discipline finite element models (although some work has been devoted to the latter as discussed below). If the overall set of ODEs is to be divided into separate subsystem models, these submodels must be integrated with respect to time in a way that provides computationally-stable results in the face of strong feedbacks and complicated interactions between the subsystems. Stable integration is accomplished in our scheme by automatically generating simplified versions of each subsystem model from the detailed model created by its analysts. These simplified models are distributed to the other processors of the network where they serve as sufficiently accurate stand-ins for the associated detailed models for short periods of time. Thus, if we break an entire system into N subsystems, then each of N processors in a network could solve one detailed subsystem model and N-1 simplified models for the other subsystems. The latter provide the required feedbacks necessary for the stable integration of the detailed subsystem model. The simplified models are kept accurate by periodically updating them to account for changes in the nominal operating state or discontinuities.

The software library is described in much more detail in Section 5.

1.3 ACCOMPLISHMENTS IN FY02

The project began in June, 2001. In the first complete fiscal year of the project, we made substantial progress, and the following accomplishments are noted:

- We developed improved, more general methods for generating simplified subsystem models for systems comprised of ODEs, ODEs and AEs, and just AEs. In particular, a block diagonalization process was developed to substantially improve the generality and numerical robustness of the model simplification process.
- We evaluated the use of sophisticated control theory techniques for creating more reliable simplified subsystem models.
- We refined the preliminary software architecture developed in FY01.
- We developed an efficient and highly flexible means for exchanging information between the separate processors using CORBA.
- We updated the software library to incorporate many of these advances.
- We tested the software library on several models of increasing difficulty.
- We performed tests in which we simulated aerospace systems on separate computers connected by both a local area network as well as the Internet.

These efforts are described in more detail in the remainder of the report.

2 METHODS FOR GENERATING SIMPLIFIED SUBSYSTEM MODELS AND ASSESSING THEIR PERFORMANCE

2.1 AVAILABLE METHODS FOR GENERATING SIMPLIFIED MODELS

Simplified subsystem models need to be developed periodically as a simulation proceeds and then distributed over the network to the various processors. Frequent updating may be necessary when a detailed subsystem model is highly nonlinear or when discontinuities occur, since in either case the simplified model may become inaccurate relatively quickly.

Ideally, the simplified models will be generated "on the fly" by the software library and their development will require little effort of the model developer other than that necessary to formulate his or her model in the first place. If this and the other project goals can be accomplished, the developer of a new subsystem model could basically ignore the software engineering complications posed by interactions with other subsystem models and focus entirely on the physics of their individual subsystem.

One possible method for developing simplified models is to use Response Surface methodology. This technique has long been used to develop simplified versions of models to support optimization and uncertainty analysis. The generation of Response Surfaces usually requires the execution of a large number of detailed model simulations, often defined by experimental design techniques. Adoption of this technique would require extensive preprocessing of the models, which would compromise the goal of providing a transparent environment for software development.

We have chosen instead to utilize modified versions of techniques developed for generating simplified models for control system implementation. In this section, we introduce these techniques and develop a framework for assessing their costs and benefits. Later sections will develop the chosen methods in more detail.

2.2 THE ROLES OF LINEARIZATION, DIAGONALIZATION, AND MODEL ORDER REDUCTION

Let us first consider how a simplified model can be developed when a model consists only of ODEs. The original detailed subsystem model is assumed to be of the form:

$$\dot{x}_i = f_i(x,u) \qquad \text{i=1,...n} \tag{2-1}$$

In this equation, x represents the values of the states, and u represents the various inputs to the model. Note that included in the inputs are the states of other subsystem models to which this model is attached, as well as explicit functions of time. The rates of change f are, in general, nonlinear and may possess discontinuities. Subsystem models encountered in practice will also contain coupled sets of algebraic equations. However, as shown in the next section, these can be eliminated, so we incur no loss of generality in assuming that the model contains only ODEs as indicated in Equation (2-1).

To generate a simplified model at a particular time, we first linearize around the vectors of current values of the state variables and input parameters, $\boldsymbol{x_o}$ and $\boldsymbol{u_o}$, respectively:

$$\delta \dot{x}_i = \left.\frac{\partial f_i}{\partial x_k}\right|_{x_o,u_o} \delta x_k + \left.\frac{\partial f_i}{\partial u_k}\right|_{x_o,u_o} \delta u_k + f_i^o \tag{2-2}$$

In Equation (2-2) and elsewhere in this report, summation is implied over repeated indices (in this case, k). Matrices and vectors are denoted by quantities in bold. It is convenient to drop the δ notation and write this in the standard form:

$$\dot{\boldsymbol{x}} = \boldsymbol{Jx} + \boldsymbol{Bu} + \boldsymbol{f_o} \tag{2-3}$$

We also allow for the possibility that the desired outputs of the subsystem model z are not necessarily the states themselves, but algebraic functions of the states and inputs:

Creare

$$z = \boldsymbol{h}(x,u) \tag{2-4}$$

When we linearize the output equations, we obtain:

$$z = \boldsymbol{C}x + \boldsymbol{D}u + z_o \tag{2-5}$$

The various matrices shown are all partial derivatives of the underlying equations. For example, the elements of the Jacobian matrix $\boldsymbol{J}$ are given by:

$$J_{ik} = \frac{\partial f_i}{\partial x_k} \tag{2-6}$$

In the controls literature, the Jacobian is often called the "system" matrix, usually denoted A. Similarly, the elements of matrix $\boldsymbol{C}$ are given by:

$$C_{ik} = \frac{\partial h_i}{\partial x_k} \tag{2-7}$$

and so on for the other matrices. The set of Equations (2-3) and (2-5) already constitutes a simplified version of the subsystem model, of course. However, it is desirable to simplify the model further, to minimize the amount of information that must be distributed over the network and the computational burden that is placed on each of the processors executing the simplified model. One way to do this is to diagonalize the system matrix $\boldsymbol{A}$ using a similarity transformation. If a matrix $\boldsymbol{S}$ can be found such that:

$$\boldsymbol{S}\,\boldsymbol{J}\,\boldsymbol{S}^{-1} = \boldsymbol{\Lambda} \tag{2-8}$$

where Λ is a diagonal matrix (whose elements are the eigenvalues of $\boldsymbol{A}$), we then define a different ("hat") coordinate system by:

$$\hat{\boldsymbol{x}} = \boldsymbol{S}\boldsymbol{x} \tag{2-9}$$

We also define:

$$\hat{\boldsymbol{B}} = \boldsymbol{B}\boldsymbol{S}^{-1} \tag{2-10}$$

$$\hat{\boldsymbol{f}}_o = \boldsymbol{S}\boldsymbol{f}_o \tag{2-11}$$

$$\hat{\boldsymbol{C}} = \boldsymbol{C}\boldsymbol{S}^{-1} \tag{2-12}$$

In terms of these, Equations (2-3) and (2-5) become:

$$\frac{d\hat{x}}{dt} = \Lambda \hat{x} + \hat{B} u + \hat{f}_o \tag{2-13}$$

$$z = \hat{C} \hat{x} + Du + z_o \tag{2-14}$$

In the following development, we will take the liberty of suppressing the "^" notation on the states x, the matrices B, and C, and the vector f_o. This creates no real ambiguity since we always analyze the subsystem behavior in transformed coordinate systems.

The simplified model represented by Equations (2-13) and (2-14) is already easier to deal with since the size of the diagonalized system matrix Λ is much smaller than the size of J. This makes it much easier to transmit the subsystem model over the network that connects the various processors and to integrate the models in time. Just as importantly, the diagonalization process also decouples each of the states from each other, making it possible to write closed-form expressions for the time-dependent values of the states in terms of the various inputs. If in doing so we determine that some states have little effect on the outputs of interest, they can be eliminated from the set of equations since they are, by definition, not needed for the calculation of other states that might be important. This makes it possible to further reduce the size of the simplified model, a process termed *model order reduction.*

Thus, linearization, diagonalization, and model order reduction can all be used to develop a simplified version of a subsystem model. While model order reduction need not necessarily be preceded by an explicit diagonalization step, there are good reasons for doing so, as will be discussed below.

2.3 BLOCK VERSUS STRICT DIAGONALIZATION

The preceding discussion assumed that the Jacobian matrix could be strictly diagonalized using a similarity transformation. Diagonalization has the highly useful feature that it completely decouples the states of a model, allowing the contribution of each state to the model outputs to be separately quantified. States which do not appreciably affect the model outputs can then be eliminated from the model, resulting in a smaller model. Unfortunately, strict diagonalization is not always possible or desirable.

2.3.1 Strict Diagonalization

The Jacobian matrix, J in Equation (2-3), can often be completely diagonalized by a similarity transformation as shown above. Such a transformation can be defined by a matrix S, whose rows are composed of the left eigenvectors of J, and a matrix S^{-1} whose columns are composed of the right eigenvectors of J. When the set of right eigenvectors and the set of left eigenvectors are biorthogonal, this similarity transformation yields a diagonal matrix, Λ, whose diagonal elements are the eigenvalues of J:

$$S\,J\,S^{-1} = \Lambda = diag(\lambda_1, \lambda_2, \ldots, \lambda_N) = \begin{bmatrix} \lambda_1 & & 0 \\ & \ddots & \\ 0 & & \lambda_N \end{bmatrix} \tag{2-15}$$

When all the eigenvalues of a matrix are different, it can be shown that this diagonalization process can be performed, at least in principle (i.e., with infinite precision arithmetic). More generally, a matrix is diagonalizable by a similarity transformation when all of its eigenvalues are *nondefective* (Golub and Van Loan, 1983). An eigenvalue is said to be defective when its algebraic multiplicity (i.e., the number of times λ_i is repeated) exceeds its geometric multiplicity (i.e., the number of linearly independent eigenvectors associated with λ_i). In other words, J is nondefective (and therefore diagonalizable) whenever its eigenvalues are all distinct, or when each of the repeated eigenvalues may be associated with a linearly independent eigenvector.

We had originally anticipated that the occurrence of defective matrices associated with physical models would be rare. However, we have found relatively simple physical models (e.g., conservation of mass balance equations for a mixing tank) which produce defective Jacobian matrices. Model order reduction using the strict diagonalization procedure is not possible in these cases. Less serious drawbacks to the use of common algorithms for strict diagonalization include the need to support complex number arithmetic (since real models can have complex eigenvalues) and the possibility that highly ill-conditioned transformation matrices S and S^{-1} will result. Consequently, it was judged necessary to devise a more robust algorithm for transforming the Jacobian matrix into a form amenable to model simplification.

2.3.2 Block Diagonalization

Defective Jacobian matrices cannot be transformed into a form that is strictly diagonal using a similarity transformation. However, according to theory, such matrices can always be put into the *Jordan canonical form* in which *Jordan blocks* appear along the diagonal of the transformed matrix, and all other entries are zero. A Jordan block of order 3 (for example), corresponding to a particular repeated eigenvalue λ, has the form:

$$\begin{bmatrix} \lambda & 1 & 0 \\ 0 & \lambda & 1 \\ 0 & 0 & \lambda \end{bmatrix} \tag{2-16}$$

As illustrated, the Jordan block consists of a matrix in which the eigenvalue appears along the diagonal, and unity appears in the matrix elements just to one side. Unfortunately, computing the Jordan canonical form can present serious numerical problems and we chose not to pursue this approach (Golub and Van Loan, 1983).

To overcome the inability, even in theory, to always strictly diagonalize Jacobian matrices, while also avoiding the numerical problems associated with generating the Jordan canonical form, we have instead implemented an algorithm based on *block diagonalization* of the Jacobian matrix. Block diagonalization can be expressed as a similarity transformation:

Creare

$$S\ J\ S^{-1} = B = diag(B_1, B_2, \ldots B_n) = \begin{bmatrix} B_1 & & 0 \\ & \ddots & \\ 0 & & B_n \end{bmatrix} \qquad (2\text{-}17)$$

where each B_i represents a square n_i-x-n_i submatrix. As in the case of Jordan blocks, the submatrices have their associated eigenvalue along the diagonal, but in this case entries will generally fill the entire upper triangular portion of the submatrix. When a block diagonalized matrix is generated, the treatment is equivalent to the previous algorithm if the order n_i of each block B_i is equal to one. That is, each state associated with a 1x1 block is completely decoupled from the remaining states and may be considered independently for the purpose of model order reduction. However, all states associated with a block of rank greater than 1 are coupled to the other states of that block.

2.3.3 Algorithm for Block Diagonalization

Bavely and Stewart (1979) developed an algorithm for the block diagonalization of a matrix using a similarity transformation as described in Equation (2-17). This algorithm has several very attractive features:

1. It is applicable to defective Jacobian matrices.
2. The initial step of the algorithm (reduction to quasi-triangular form) is based on the Schur decomposition, a much more numerically robust algorithm than other similarity transformations, e.g., using left and right eigenvectors or reduction to Jordan canonical form.
3. It produces quite well-conditioned transformation matrices S^{-1} and S, by grouping both equal and nearly equal eigenvalues into the block diagonal structure when necessary.
4. It uses only real matrix arithmetic since complex eigenvalues are stored in an equivalent fashion in real-valued, 2x2 diagonal blocks.
5. When repeated, but not defective eigenvalues exist, the transformation matrices generated by the algorithm are orthogonal and the transformed matrix is strictly diagonal. This was not the case with the previous algorithm used in the software library.

We also evaluated a variation of the algorithm for block diagonalization (available in the free software package SCILAB) which uses complex arithmetic. The use of complex matrix arithmetic has the potential benefit of allowing further reduction in the block size of matrices with complex eigenvalues, since such eigenvalues may be stored in 1x1 rather than 2x2 diagonal blocks (if an eigenvalue of a real matrix is complex, then another eigenvalue of that matrix is the first eigenvalue's complex conjugate). However, the use of complex numbers requires twice the memory space, involves more than twice the computational complexity, and introduces complex-valued rounding errors into the calculation of real-valued model outputs. Further, we expect that in the model order reduction process, if a state associated with a complex valued eigenvalue contributes significantly to the subsystem model outputs, the state associated with the complex conjugate of this eigenvalue will as well. If so, both states must be included anyway, and there is little advantage to considering them separately.

2.4 Approximate Effect of Subsystem Model Complexity on Network Loading and Execution Speed

To assess the costs and benefits of model order reduction, and to let us eventually compare the various schemes available for carrying this out, it is helpful to consider the demands that various schemes might place on computer time and network bandwidth.

As discussed earlier, there are several potential benefits in using the distributed modeling software library. Many of these address what might be termed software engineering problems. In that regard, the library should:

- Allow subsystem modelers working in different technical disciplines to easily integrate their separate models into a complete system model. If the design goals are realized, little additional effort will be required of the subsystem modeler beyond creating his individual model in the first place. Importantly, stable and accurate integration of the complete system model should be achievable more or less automatically, allowing subsystem modelers to focus on their technical discipline, rather than difficult areas of numerical analysis and software engineering.
- Permit geographically distributed sets of subsystem modelers to interact transparently.
- Ease the quality assurance burden, by allowing such efforts to focus on ensuring the integrity of the individual subsystem models on a model-by-model basis, rather than having to treat the entire code as a whole.
- Support uncertainty analysis on complex system models.
- Support optimization of complex system models.

In the end, these are probably the most important benefits of the library. Software developers creating complex system models usually discover that integrating many relatively simple submodels can be very time-consuming and error-prone when the individual models interact in ways that are complex, subtle, and often numerically "stiff."

Nevertheless, it is also possible that gains in execution speed will also result from harnessing the power of large computer networks, and this would be another advantage of this approach. Conversely, if the software library results in a huge drop in execution speed, it will have considerably less appeal to developers. Similarly, if the network bandwidth required to transmit simplified models is too high, the system may not be practicable. For these reasons, it is useful to qualitatively consider the effect of distributed simulation on network loading and execution speed.

2.4.1 Assumed Model Characteristics

For simplicity, we consider a dynamic system model that has been partitioned into m subsystem models, each of roughly equal complexity. We assume that we can utilize m

processors to solve these models, i.e. one processor is used for each subsystem model. Assuming for the moment that the simplified versions of the submodels that are exchanged between processors are relatively small, we neglect the times required to transmit the simplified models between the various processors compared to the time required to develop them in the first place.

It should be noted that we have already made a key assumption: that no single slow-running subsystem model delays the execution of the model as a whole. This may require considerable care when partitioning the overall system model into subsystems.

In general, a particular subsystem model can consist of ordinary differential equations (ODEs) and algebraic equations. The latter are used in the computation of the rates-of-change of the former. For example, the gas pressure might be calculated algebraically from the temperature and mass and then used to compute the rate of change of the mass. Other algebraic equations may be present that relate the values of the internal variables defined by ODEs and algebraic equations to the key subsystem model outputs needed by the other subsystem models. To make the following discussion simpler, we assume that the effort required to evaluate the algebraic equations is included in the effort required to calculate the rates-of-change of the ODE-defined states on which they depend. If this is done, and if we make the further assumption that the number of key outputs is much smaller than the number of internal states, we make little error in considering the model as consisting entirely of ODEs. In practice, the mapping between outputs and states is usually trivial, i.e., certain ODE states or algebraic variables are themselves the key model outputs, so no additional effort is required to evaluate the outputs in the detailed model given the values of the states.

Let us now define:

n	= number of states in each detailed subsystem model
n'	= number of states in a simplified subsystem model (after model order reduction)
f	= average number of floating point operations required per state in a detailed model
f'	= average number of floating point operations required per state in a simplified model
j	= average number of operations required to compute one term of the Jacobian matrix
q	= average number of nonzero terms in Jacobian matrix per rate (number of nonzero matrix elements per row)
p	= number of time steps between updates of simplified models ("model update interval")
r	= number of floating point operations that can be performed per second
u	= number of inputs
o	= number of outputs

With no loss in generality, we can let $r = 1$. In other words, time is measured in the units of the time required to perform a single floating point operation on the processor.

2.4.2 Requirements on Network Bandwidth of Distributed Simulation

One way to compare the effectiveness of different schemes for developing simplified subsystem models is to perform an approximate analysis of the network bandwidth required to transmit such a model over the network. We consider four different scenarios:

1. The full linearized model is transmitted.
2. The full linearized model is transmitted using a sparse matrix scheme in which only the nonzero elements of large matrices are transmitted, together with integers describing the row and column numbers of these elements.
3. The diagonalized model is transmitted with no model order reduction.
4. A diagonalized and model order reduced model is transmitted.

For simplicity, we assume that the system matrix can be essentially fully diagonalized, i.e., there are relatively few 2x2 or larger sub-blocks. The approximate number of variables that must be transmitted to communicate the linearized model, Equations (2-3) and (2-5) is then shown in Table 2-1.

Table 2-1. Approximate Size of Simplified Models Developed and Transmitted Using Different Techniques

Quantity (matrix or vector)	Full linearized model	Sparse version of full model	Diagonalized model	Diagonalized, reduced order model
$\boldsymbol{A}$	n^2	$3nq$	n	n'
$\boldsymbol{B}$	nu	nu	nu	$n'u$
f_o	n	N	n	n'
$\boldsymbol{C}$	no	no	no	$n'o$
$\boldsymbol{D}$	ou	ou	ou	ou
z_o	o	o	o	o

While the number of nonzero elements of the $\boldsymbol{B}$, $\boldsymbol{C}$, and $\boldsymbol{D}$ matrices may be somewhat overstated in Table 2-1 (full matrices were assumed), this will not cause much error if the number of inputs and outputs is small compared to the number of states, as appears reasonable.

We can evaluate the implications of these estimates using crude assumptions on the complexity of subsystem models. We believe that the use of the software library that is likely to be most practicable is when a large number of relatively simple subsystem models are being coupled. Thus, a reasonable set of guesses might be that a representative subsystem model has 50 states, of which 10 are needed in a particular update interval to provide a reasonably accurate approximation of subsystem behavior. If we assume that this system is coupled to its neighbors using 5 inputs and outputs, and that the average state variable's rate-of-change f is a function of 5 other states, we have:

$$\begin{gathered} n = 50 \\ n' = 10 \\ q = 5 \\ o = 5 \\ u = 5 \end{gathered}$$

From these estimates (or, perhaps more accurately, guesses), we obtain the results shown in Table 2-2.

Table 2-2. Relative Size of Simplified Models for a Particular Set of Assumptions

Method Used to Develop Model	Number of variables that must be transmitted
Full linearized model	3080
Sparse version of full model	1330
Diagonalized model	630
Diagonalized, reduced order model	150

While these specific assumptions are highly arguable and will no doubt vary widely from application to application, we can generally conclude that diagonalization by itself offers huge benefits in reducing the model size, and that an additional model order reduction step after block diagonalization offers further significant benefits.

2.4.3 Effect of Distributed Simulation on Execution Speed

We now consider the impact on execution speed of simulating a system by distributing the subsystem models to various processors. In the most favorable situation, the number of operations required to solve the complete model is just the effort required to individually integrate each state. Note that this is only the case when the ODEs are not numerically stiff, and is probably relatively rare. In the more typical situation where stiff ODEs are present, an iterative technique is generally required to achieve stable and accurate integration, and much more time would be required to model the system on a single processor. All else being equal, in such a case the multiple processor approach using the distributed modeling library would look <u>much</u> more attractive than is shown below.

a. <u>Entire System Model Executed on a Single Processor Without Using the Distributed Software Library</u>

Given the assumption that all equations can be integrated explicitly, the time t_1 required to integrate the equations on one processor is essentially just that required to calculate the rates at each time step multiplied by the number of time steps in an update interval:

$$t_1 = nfmp$$

b. <u>Each Subsystem Model is Executed in a Distributed Fashion on a Separate Processor</u>

When the distributed modeling library is used, each processor must accomplish several tasks, which we consider separately below:

(1) Calculation of Jacobian Matrix

Any practicable implementation of the distributed modeling system will utilize an explicit, hard-coded representation of the terms of the Jacobian matrix. These are developed by preprocessing the subsystem model's source code with an external program such as ADIFOR or ADIC, and the time required to do this preprocessing is not included in the execution time. Once the Jacobian has been encoded, the time required to calculate the Jacobian matrix at the beginning of each update interval is:

$$njq$$

(2) Diagonalization of Jacobian Matrix

The software library uses block diagonalization to decouple the individual states to facilitate overall model simplification and model order reduction. According to Bavely and Stewart (1979), the time required to accomplish this is generally upper-bounded by:

$$n^4/12$$

In some cases, this grossly overestimates the time required. For example, in the optimistic case where all the eigenvalues are real and can be deflated, the time required is <u>much</u> smaller, only:

$$n^3/2$$

According to Varga (1993), a reasonable "maximal" estimate is:

$$15n^3$$

In what follows, we shall conservatively use the lesser of $15n^3$ and $n^4/12$. We use the minimum of these two estimates since for modest *n*, the Bavely and Stewart estimate is actually smaller than that of Varga even though the former varies as a higher power of *n*.

(3) Model Order Reduction

After the Jacobian matrix has been diagonalized, typically just a few operations are necessary to decide whether each state in the transformed coordinate system should be kept. To avoid introducing more parameters, we assume that this is about the same as the number of operations required to integrate each simplified model equation, or:

$$f'$$

Transformation of the system equations into the coordinate system that block-diagonalizes the states requires several matrix multiplications, Equations (2-10)–(2-12):

$$\sim 3n'^2$$

(4) Integration of Simplified Models

The time required to integrate all the simplified models over the update interval is simply:

$$pmn'f'$$

(5) Integration of One Detailed Subsystem Model

The time required to integrate the single detailed subsystem model that is resident on a given processor is:

$$pnf$$

Thus the total time required per processor to integrate the system for one update interval is:

$$t_m = \min(5n^3, n^4/12) + njq + f' + 3n'^2 + pmn'f + pnf$$

Assuming that all the processors run simultaneously, this is also the total clock time required to perform the calculation.

To more clearly see the implications of these expressions, we make the following assumptions:

1. The number of processors m is much larger than 1.
2. The number of floating point operations required to evaluate the rate-of-change of a state in the detailed model f is much larger than that required to evaluate the same rate-of-change in the simplified model, f'. This is nearly certainly going to be the case, since the simplified model states have been diagonalized and are essentially trivial to evaluate.
3. The amount of work required to calculate all the rates-of-change of a simplified model is much less than that required by its detailed counterpart:
$$n'f' << nf$$
4. Each state of the detailed model depends on only a few of the other states so that the number of terms in the Jacobian matrix is:
$$q << n$$
5. The number of operations required to evaluate a term in the Jacobian matrix is smaller than or comparable to the number of operations required to evaluate the corresponding state:
$$j \leq f$$

Given these assumptions, the ratio of the time required to solve the system in a distributed fashion on multiple processors compared to <u>the most optimistic situation</u> for solving on one processor is given by:

$$\frac{t_m}{t_1} \sim \frac{\min\left(\frac{n^3}{12}, 15n^2\right)}{pfm} + \frac{n'f'}{nf} + \frac{1}{m} \qquad (2\text{-}18)$$

If the above ratio is less than 1, it represents the clock time savings obtained by using the distributed processing library. What this ratio will be in a given case is difficult to estimate, since this obviously depends on the number of subsystem models, how complex they are, how long a simplified model can be used before it is updated, etc. Based on the previous discussion, the last two terms in Equation (2-1) are likely to be very small, so the relative cost primarily hinges on the first term, i.e., how long the block-diagonalization process takes. This clearly depends very strongly on the complexity of the original subsystem models, i.e., n.

Extending the logic introduced earlier for assessing network bandwidth, a reasonable set of guesses might be:

$$\begin{aligned} n &= 50 \\ n' &= 10 \\ m &= 10 \\ p &= 1000 \\ f &= 50 \\ f' &= 5 \end{aligned}$$

From these values we obtain from Equation (2-18):

$$\frac{t_1}{t_m} \sim \frac{1}{10} \tag{2-19}$$

The validity of this numerical estimate is certainly arguable, but several <u>qualitative</u> conclusions can be drawn from Equation (2-1):

1. The cost of block-diagonalizing the Jacobian matrix of each subsystem model can easily dominate the effort required to implement the distributed computing technique. Whether this actually occurs depends primarily on the size of the subsystem models (i.e., the number of differential equations) and their complexity.
2. The distributed processing method will be most cost-efficient if the number of states in the individual subsystem models is not too large, the rate of change of each state is complicated to evaluate, and the simplified models can be used for a relatively long duration before they are replaced.
3. The cost of distributed computing could be reduced if more efficient means for diagonalizing the Jacobian can be found. For example, if the Jacobian matrix changes slowly between model updates, perturbation methods might be employed, analogous to techniques available in the literature for computing the effect of small changes in a matrix on its eigenvectors.

The preceding qualitative analysis neglects any time required to transmit the simplified models over the network. This may be reasonable in many cases, but there is some evidence from our testing on very low quality (dial-up) lines that the "start-up" costs associated with each transmission can be substantial. Thus, a quantitative estimate of the time required to perform distributed computation would require a more thorough analysis.

2.4.4 Implications of Choice of Model Order Reduction Schemes on Execution Speed

Various schemes for model order reduction have been developed other than the procedure based on block diagonalization that is currently implemented in the software library. One appealing scheme, discussed further in Section 4, involves basing model order reduction on a "balanced implementation" of the model. The balancing step requires the calculation of the so-called observability and controllability Grammians of the system matrices. Given certain assumptions, this scheme offers explicit control of the reduced order model's accuracy.

These techniques are generally used off-line, often with relatively simple system models, rather than for "on the fly" calculation of simplified versions of relatively complicated system models as we need. Based on the cost-benefit analysis presented above, these other approaches will not be very appealing if they require much longer computational times than is necessary for block diagonalization. What might be practicable is to sequentially apply these more advanced techniques to the relatively small blocks of a block-diagonalized matrix, assuming that these extra steps require little additional cost. Such a sequential approach has been proposed by Varga (1993, 1995).

2.5 CONCLUSIONS

The most important reasons to develop distributed simulation techniques are to improve the efficiency of the software <u>development</u> process, not to gain improvements in <u>execution</u> speed. Nevertheless, if distributed simulation techniques are to become widely used, they must be efficient enough that they do not present too large a burden on network bandwidth or individual processor speed.

Linearization followed by block diagonalization provides a powerful technique for developing simplified subsystem models. Even without a subsequent MOR step, the diagonalized models can be transferred relatively efficiently across the network and will present a relatively small computational burden on the other processors. However, the cost of block diagonalization is relatively large for complex models having a large number of states. Thus, the distributed software library will be most efficient during execution when the overall system consists of a relatively large number of models of moderate complexity whose linearized versions do not require too frequent updating. Also, for a given level of model complexity, methods that can extend the time required between block diagonalization (such as perturbation techniques) could be beneficial.

Sophisticated model order reduction schemes have been presented in the literature, e.g., those discussed in Section 4 that involve calculating the Grammians of the system matrices. If they require computer time that is much larger than that required for block diagonalization, these may not be practicable for this application.

3 REFORMULATING SUBSYSTEM MODELS TO FACILITATE DIAGONALIZATION AND MODEL ORDER REDUCTION

In the preceding section, a method for decoupling the various states was presented. This block diagonalization process is relatively time-consuming and can present mathematical difficulties. In this section, several means are presented for reformulating the model to improve the efficiency of this process.

3.1 Elimination of States Represented by Pure Integrals From the System Matrix

Equations that represent pure integrals, rather than ordinary differential equations, often appear in dynamical models. For example, the position of an object might be calculated by integrating its velocity. Such a calculation would be included if another subsystem model, such as a controller, needed the object's position. However, if no other variable (such as a force) in the same subsystem model depends on position, then the column of the system (Jacobian) matrix that defines the dependence of the various state rates-of-change on the position will contain only zeros. In such cases, the row and column of the matrix corresponding to the position variable could be removed prior to diagonalization.

While we have always recognized that states calculated by pure integrals have zero eigenvalues, early experience did not suggest that their inclusion would lead to any problems. However, in the case of some of the recent test models, formally removing such variables prior to the diagonalization process appeared to improve the robustness and efficiency of the diagonalization process. Moreover, after removing pure integrals from one model, the diagonalized matrix in some cases was found to no longer contain sub-blocks larger than 2x2, potentially leading to more compact simplified models. Finally, reducing the dimension n of the matrix will have a large effect on the time ($\sim 15n^3$) required for block diagonalization.

To ensure that all pure integrals are removed, an iterative approach is needed. Iteration is required since a given state variable might only be needed for calculating another pure integral. In such cases, both variables can be made pure integrals so long as they are subsequently calculated in the proper order. It can be shown that the proper order of integration is the same as the order in which the pure integrals are removed during the iteration process.

In practice, however, a single pass through the pure integral removal process may be the best choice, i.e., removing only those states that do not appear in any other state's rate of change. If an iterative process is used, the pure integrals become interdependent, and it does not appear possible to calculate closed-form solutions for them. This complicates model order reduction.

At present, a single-pass and an iterative scheme for removing pure integrals are both available in the software library. Further testing is planned to fully understand the costs and benefits of removing such states.

3.2 Elimination of Algebraic Variables From a Coupled Set of AEs and DAEs

In the previous year's progress report, an algorithm was described for the transformation of models consisting of differential and algebraic equations. The main drawback of this algorithm was that the NxN size of the Jacobian matrix that must be diagonalized for transformation was equal to the sum of the number of differential equations and the number of algebraic equations. An alternative algorithm has been developed where the dimension of the Jacobian matrix that must be diagonalized is equal to the number of differential equations only. As a result, the computational complexity of the algorithm is greatly reduced for models with a large number of algebraic variables. This simplification is obtained by formally eliminating the algebraic equations from the system variables.

To facilitate the analysis, we write the linearization of a model containing only ODEs, Equations (2-3) and (2-5), in a slightly different fashion that emphasizes the nature of the various matrices:

$$\frac{d\delta\mathbf{x}}{dt} = \mathbf{J}_\mathbf{x}^\mathbf{f}\delta\mathbf{x} + \mathbf{J}_\mathbf{u}^\mathbf{f}\delta\mathbf{u} + \mathbf{f}^o \tag{3-1}$$

$$\delta\mathbf{z} = \mathbf{J}_\mathbf{x}^h\delta\mathbf{x} + \mathbf{J}_\mathbf{u}^h\delta\mathbf{u} + \mathbf{z}^o \tag{3-2}$$

In this scheme, terms such as J_x^f denote the partial derivatives of the state rate-of-change functions **f** with respect to the states **x**. To develop a revised algorithm for systems comprised of both ODEs and AEs, we assume that a particular subsystem model has been written in the following form:

$$\frac{d\mathbf{x}}{dt} = \mathbf{f}(\mathbf{x},\mathbf{y},\mathbf{u}) \tag{3-3}$$

$$\mathbf{g}(\mathbf{x},\mathbf{y},\mathbf{u}) = \mathbf{0} \tag{3-4}$$

Again, **x** are the differential equation-defined state variables, **y** are the algebraic variables, and **u** are the input variables of the subsystem. As before, a set of outputs variables **z** that serve as inputs to other subsystem models are defined as nonlinear functions of **x**, **y**, and **u:**

$$\mathbf{z} = \mathbf{h}(\mathbf{x},\mathbf{y},\mathbf{u}) \tag{3-5}$$

We again assume that the time interval over which a simplified model will be used is sufficiently small that linearization of the overall model is valid. Again suppressing the "δ" notation for changes in variables around the linearization point, we have:

$$\frac{d\mathbf{x}}{dt} = \mathbf{J}_\mathbf{x}^\mathbf{f}\mathbf{x} + \mathbf{J}_\mathbf{y}^\mathbf{f}\mathbf{y} + \mathbf{J}_\mathbf{u}^\mathbf{f}\mathbf{u} + \mathbf{f}^o \tag{3-6}$$

$$\mathbf{J}_\mathbf{x}^\mathbf{g}\mathbf{x} + \mathbf{J}_\mathbf{y}^\mathbf{g}\mathbf{y} + \mathbf{J}_\mathbf{u}^\mathbf{g}\mathbf{u} = \mathbf{0} \tag{3-7}$$

$$\mathbf{z} = \mathbf{J}_\mathbf{x}^\mathbf{h}\mathbf{x} + \mathbf{J}_\mathbf{y}^\mathbf{h}\mathbf{y} + \mathbf{J}_\mathbf{u}^\mathbf{h}\mathbf{u} + \mathbf{z}^o \tag{3-8}$$

Solving for **y** in Equation (3-7) yields:

$$\mathbf{y} = -(\mathbf{J}_\mathbf{y}^\mathbf{g})^{-1}(\mathbf{J}_\mathbf{x}^\mathbf{g}\mathbf{x} + \mathbf{J}_\mathbf{u}^\mathbf{g}\mathbf{u}) \tag{3-9}$$

Substituting Equation (3-9) into Equations (3-6) and (3-11):

$$\frac{d\mathbf{x}}{dt} = \mathbf{J}_\mathbf{x}^\mathbf{f}\mathbf{x} + \mathbf{J}_\mathbf{y}^\mathbf{f}[-(\mathbf{J}_\mathbf{y}^\mathbf{g})^{-1}(\mathbf{J}_\mathbf{x}^\mathbf{g}\mathbf{x} + \mathbf{J}_\mathbf{u}^\mathbf{g}\mathbf{u})] + \mathbf{J}_\mathbf{u}^\mathbf{f}\mathbf{u} + \mathbf{f}^o \tag{3-10}$$

$$\mathbf{z} = \mathbf{J}_x^h \mathbf{x} + \mathbf{J}_y^h[-(\mathbf{J}_y^g)^{-1}(\mathbf{J}_x^g \mathbf{x} + \mathbf{J}_u^g \mathbf{u})] + \mathbf{J}_u^h \mathbf{u} + \mathbf{z}^o \tag{3-11}$$

Rearranging yields:

$$\frac{d\mathbf{x}}{dt} = (\mathbf{J}_x^f - \mathbf{J}_y^f(\mathbf{J}_y^g)^{-1}\mathbf{J}_x^g)\mathbf{x} + (\mathbf{J}_u^f - \mathbf{J}_y^f(\mathbf{J}_y^g)^{-1}\mathbf{J}_u^g)\mathbf{u} + \mathbf{f}^o \tag{3-12}$$

$$\mathbf{z} = (\mathbf{J}_x^h - \mathbf{J}_y^h(\mathbf{J}_y^g)^{-1}\mathbf{J}_x^g)\mathbf{x} + (\mathbf{J}_u^h - \mathbf{J}_y^h(\mathbf{J}_y^g)^{-1}\mathbf{J}_u^g)\mathbf{u} + \mathbf{z}^o \tag{3-13}$$

Note that except for the alteration of the matrices multiplying $\mathbf{x}$ and $\mathbf{u}$, Equations (3-6) and (3-12) and Equations (3-7) and (3-13) are functionally equivalent. Consequently, to eliminate the AEs to form a system comprised solely of ODEs, we make the following substitutions:

$$\mathbf{J}_x^f := (\mathbf{J}_x^f - \mathbf{J}_y^f(\mathbf{J}_y^g)^{-1}\mathbf{J}_x^g) \tag{3-14}$$

$$\mathbf{J}_u^f := (\mathbf{J}_u^f - \mathbf{J}_y^f(\mathbf{J}_y^g)^{-1}\mathbf{J}_u^g) \tag{3-15}$$

$$\mathbf{J}_x^h := (\mathbf{J}_x^h - \mathbf{J}_y^h(\mathbf{J}_y^g)^{-1}\mathbf{J}_x^g) \tag{3-16}$$

$$\mathbf{J}_u^h := (\mathbf{J}_u^h - \mathbf{J}_y^h(\mathbf{J}_y^g)^{-1}\mathbf{J}_u^g) \tag{3-17}$$

This is roughly analogous to the "superelement" used to eliminate "internal" variables from large sets of AEs presented in the next section.

After this preprocessing step, the same algorithm used for the solution of subsystem models consisting only of ODEs can then be used without modification for the solution of DAE subsystem models. Furthermore, by incorporating the contribution of changes in the algebraic variable to changes in the outputs before diagonalization, the NxN size of the Jacobian matrix that must be diagonalized for transformation is equal to number of differential states only. Since the diagonalization process has N^3 or N^4 complexity, the computational cost is reduced significantly for models with a large number of algebraic equations compared to the previous scheme in which the entire system matrix containing both differential- and algebraically-defined matrix elements was diagonalized.

3.3 Elimination of Internal Algebraic Variables in Sets of AEs

We have previously discussed model order reduction of ODEs or coupled sets of AEs and ODEs. While not the main thrust of this project, we have also considered steady-state models consisting only of large sets of algebraic equations, such as are encountered in large, multi-disciplinary finite element models. In this case, model order reduction is much simpler and, for linear models, exact.

Assume that a particular subsystem model consists solely of a set of nonlinear algebraic equations written in the following form:

$$\mathbf{g}(\mathbf{y},\mathbf{u}) = \mathbf{0} \tag{3-18}$$

where

$$\mathbf{y} = \begin{bmatrix} \mathbf{y}_b \\ \mathbf{y}_i \end{bmatrix} \tag{3-19}$$

In Equation (3-18), **y** denotes the solution of a set of algebraic equations, given a set of inputs **u** supplied to the subsystem model from other subsystems. Equation (3-19) denotes a partitioning of the set of solution variables **y** into two subsets: "internal variables" denoted as $\mathbf{y}_i$ and "boundary variables" denoted as $\mathbf{y}_b$. The boundary variables $\mathbf{y}_b$ represent output variables of this subsystem that are of particular interest. Explicitly defining key outputs in this way allows us to distinguish the important results of a model from internal details that are important only in that they affect these results. In particular, the set of boundary output variables $\mathbf{y}_b$ includes the interface variables between subsystem models, so that the boundary outputs $\mathbf{y}_b$ from one model are fed into another model as input parameters (i.e., elements of **u**) to that model.

We now proceed to develop a simplified model framework for the solution of sets of algebraic equations. The problem may be stated as follows: given a set of values for input variables **u** and a set of nonlinear functions $\mathbf{g}(\mathbf{y},\mathbf{u})$, find $\mathbf{y}^*$ such that $\mathbf{g}(\mathbf{y}^*,\mathbf{u}) = \mathbf{0}$. Numerical techniques for the solution of sets of nonlinear algebraic equation are typically based on some variation of Newton's method. This iterative algorithm may be summarized by the following steps:

1. Given a set of initial guesses for solution variables $\mathbf{y}^i$ and values for input variables **u**, calculate the residual **R** defined by:

$$\mathbf{g}(\mathbf{y}^i,\mathbf{u}) = \mathbf{R}^i \tag{3-20}$$

Note that if $\mathbf{R}^i = 0$, then $\mathbf{y}^* = \mathbf{y}^i$.

2. A first order Taylor series expansion about $\mathbf{y}^i$ yields the linear approximation:

$$\mathbf{g}(\mathbf{y}^i + \delta\mathbf{y}^i,\mathbf{u}) \approx \mathbf{R}^i + \mathbf{J}^{yi}\delta\mathbf{y}^i \tag{3-21}$$

where $\mathbf{J}^{yi}$ is the Jacobian matrix whose (m,n)th element is given by:

$$J^y_{m,n} \equiv \frac{\partial g_m(y,u)}{\partial y_n} \tag{3-22}$$

3. We wish to find a correction $\delta\mathbf{y}^i$ such that $\mathbf{g}(\mathbf{y}^i + \delta\mathbf{y}^i,\mathbf{u}) = \mathbf{0}$. The preceding approximation can be used to solve for $\delta\mathbf{y}^i$ yielding:

$$\delta\mathbf{y}^i = -(\mathbf{J}^{yi})^{-1}\mathbf{R}^i \tag{3-23}$$

4. The estimates for the solution variables are then updated as:

$$\mathbf{y}^{i+1} = \mathbf{y}^i + \boldsymbol{\delta}\mathbf{y}^i \tag{3-24}$$

Since $\mathbf{g}$ is nonlinear, an iterative approach is necessary, and the algorithm returns to step 1, with $\mathbf{y}^{i+1} \mapsto \mathbf{y}^i$ until some convergence criteria are met (e.g., $\|\mathbf{R}^i\| < \varepsilon_R$ and $\|\delta\mathbf{y}^i\| < \varepsilon_{\delta y}$).

For the purpose of Newton's method iterations, we now consider linearization of the AEs that comprise our model around some values of $\mathbf{y}$ and $\mathbf{u}$:

$$\mathbf{J}^{\mathbf{y}}\boldsymbol{\delta}\mathbf{y} + \mathbf{J}^{\mathbf{u}}\boldsymbol{\delta}\mathbf{u} + \mathbf{g}^o = \mathbf{0} \tag{3-25}$$

where

$$J^y_{m,n} \equiv \frac{\partial g_m(y,u)}{\partial y_n} \tag{3-26}$$

$$J^u_{m,n} \equiv \frac{\partial g_m(y,u)}{\partial u_n} \tag{3-27}$$

$$\mathbf{g}^o \equiv \mathbf{g}(\mathbf{y}^o, \mathbf{u}^o) \tag{3-28}$$

During the iterative numerical solution of a distributed multi-subsystem model, a subsystem will be represented by its linearized counterpart. Assuming that the model consists of m algebraic equations and p inputs, the model is represented by an (m x m) matrix $\mathbf{J}^{\mathbf{y}}$, an (m x p) matrix $\mathbf{J}^{\mathbf{u}}$, and an (m x 1) vector $\mathbf{g}^o$. In addition, calculation of $\boldsymbol{\delta}\mathbf{y}$ requires an m x 1 vector $\mathbf{y}^o$ and calculation of $\boldsymbol{\delta}\mathbf{u}$ requires a p x 1 vector $\mathbf{u}^o$. Thus, the number of elements that must be transmitted over the network at each iteration is $m(m+p+2)+p$. In general, we assume $m >> p$ so this number may be approximated as m^2. Obviously, such a requirement may be infeasible since the number of solution variables m can grow very large for a complicated model.

A more attractive option utilizes the concept of a "superelement," which exploits the partitioning of solution variables $\mathbf{y}$ into internal variables $\mathbf{y}_\mathrm{i}$ and boundary variables $\mathbf{y}_\mathrm{b}$. Likewise, the algebraic equations are partitioned by assigning each input and output variable to a unique equation. As a result, the linearized model may be partitioned as:

$$\begin{bmatrix} \mathbf{J}^{\mathbf{y}}_{\mathbf{bb}} & \mathbf{J}^{\mathbf{y}}_{\mathbf{bi}} \\ \mathbf{J}^{\mathbf{y}}_{\mathbf{ib}} & \mathbf{J}^{\mathbf{y}}_{\mathbf{ii}} \end{bmatrix} \begin{bmatrix} \boldsymbol{\delta}\mathbf{y}_\mathbf{b} \\ \boldsymbol{\delta}\mathbf{y}_\mathbf{i} \end{bmatrix} + \begin{bmatrix} \mathbf{J}^{\mathbf{u}}_{\mathbf{b}} \\ \mathbf{J}^{\mathbf{u}}_{\mathbf{i}} \end{bmatrix} \boldsymbol{\delta}\mathbf{u} + \begin{bmatrix} \mathbf{g}^o_\mathbf{b} \\ \mathbf{g}^o_\mathbf{i} \end{bmatrix} = \mathbf{0} \tag{3-29}$$

Suppressing "δ", the first "super row" of this superelement matrix yields:

$$\mathbf{J}^{y}_{bb}\mathbf{y}_{b} + \mathbf{J}^{y}_{bi}\mathbf{y}_{i} + \mathbf{J}^{u}_{b}\mathbf{u} + \mathbf{g}^{o}_{b} = \mathbf{0} \tag{3-30}$$

The second "super row" of the superelement matrix yields:

$$\mathbf{J}^{y}_{ib}\mathbf{y}_{b} + \mathbf{J}^{y}_{ii}\mathbf{y}_{i} + \mathbf{J}^{u}_{i}\mathbf{u} + \mathbf{g}^{o}_{i} = \mathbf{0} \tag{3-31}$$

Solving the second super row for $\boldsymbol{\delta}\mathbf{y}_{i}$ gives:

$$\mathbf{y}_{i} = -(\mathbf{J}^{y}_{ii})^{-1}[\mathbf{J}^{y}_{ib}\mathbf{y}_{b} + \mathbf{J}^{u}_{i}\mathbf{u} + \mathbf{g}^{o}_{i}] \tag{3-32}$$

Substituting this expression for $\mathbf{y}_{i}$ into the first super row and rearranging gives:

$$(\mathbf{J}^{y}_{bb} - \mathbf{J}^{y}_{bi}\mathbf{J}^{y\,-1}_{ii}\mathbf{J}^{y}_{ib})\mathbf{y}_{b} + (\mathbf{J}^{u}_{b} - \mathbf{J}^{y}_{bi}\mathbf{J}^{y\,-1}_{ii}\mathbf{J}^{u}_{i})\mathbf{u} + (\mathbf{g}^{o}_{b} - \mathbf{J}^{y}_{bi}\mathbf{J}^{y\,-1}_{ii}\mathbf{g}^{o}_{i}) = \mathbf{0} \tag{3-33}$$

or

$$\hat{\mathbf{J}}^{y}\mathbf{y}_{b} + \hat{\mathbf{J}}^{u}\boldsymbol{\delta}\mathbf{u} + \hat{\mathbf{g}}^{o} = \mathbf{0} \tag{3-34}$$

where

$$\hat{\mathbf{J}}^{y} = \mathbf{J}^{y}_{bb} - \mathbf{J}^{y}_{bi}\mathbf{J}^{y\,-1}_{ii}\mathbf{J}^{y}_{ib} \tag{3-35}$$

$$\hat{\mathbf{J}}^{u} = \mathbf{J}^{u}_{b} - \mathbf{J}^{y}_{bi}\mathbf{J}^{y\,-1}_{ii}\mathbf{J}^{u}_{i} \tag{3-36}$$

$$\hat{\mathbf{g}}^{o} = \mathbf{g}^{o}_{b} - \mathbf{J}^{y}_{bi}\mathbf{J}^{y\,-1}_{ii}\mathbf{g}^{o}_{i} \tag{3-37}$$

Thus, assuming the number of boundary variables is b, the equivalent superelement model consists of a b x b matrix $\hat{\mathbf{J}}^{y}$, a b x p matrix $\hat{\mathbf{J}}^{u}$, and a b x 1 vector $\hat{\mathbf{g}}^{o}$. In addition, calculation of $\mathbf{y}_{b}$ requires a b x 1 vector $\mathbf{y}^{o}_{b}$ and calculation of $\mathbf{u}$ requires a p x 1 vector $\mathbf{u}^{o}$. Thus, the number of elements that must be transmitted over the network at each iteration is $b(b+p+2)+p$. In general, we assume $m \sim p$ so the total number of elements required is on the order of b^2. Since $b << m$, the relative size of the transmitted model can be reduced by several orders of magnitude for large models. The computational burden placed on the remote processors when solving the simplified model is also drastically reduced.

3.4 Conclusions

When applying diagonalization and model order reduction techniques, real subsystem models present various complications. For example, such models can contain:

- States that are represented by pure integrals over time, rather than ordinary differential equations.
- Coupled sets of algebraic and differential equations.

To prevent these complications from unnecessarily reducing the efficiency and numerical robustness of diagonalization and model order reduction, preprocessing steps have been developed to simplify the model prior to diagonalization.

While not the main thrust of this project, we have also considered subsystem models containing only algebraic equations. For such models, we adapted the "superelement" method used in finite element analysis that eliminates all internal variables that cannot be directly observed by external subsystems. This provides a very effective and comparatively simple method for developing simplified versions of a subsystem model.

4 MODEL ORDER REDUCTION OF BLOCK DIAGONALIZED SETS OF ORDINARY DIFFERENTIAL EQUATIONS

4.1 Introduction

To recap the preceding discussion, after linearization of a subsystem model's equations, we can formally eliminate any algebraically-defined variables y using the methods presented above. After these steps have been taken, the subsystem model can be written between discontinuities (if present):

$$\dot{x} = Ax + Bu + F_o \tag{4-1}$$

$$\dot{z} = Cx + Du + W_o \tag{4-2}$$

where A is the NxN system or Jacobian matrix. We now turn to model order reduction methods for reducing the size of these matrices.

4.2 Inapplicability of Textbook MOR Strategies

Model order reduction has been studied very extensively, and disparate techniques for accomplishing this are available in the literature. Textbooks often demonstrate the development of a coordinate transformation that eliminates states that do not contribute even slightly to the output variables (Hendricks et al., 2000; Brogan, 1991). This form of the system is called the "Kalman Observable Canonical Form."

To reduce the system to the Kalman Observable Canonical Form, one seeks a coordinate transformation X that transforms the variables x and the matrix C shown in Equations (4-1) and (4-2) as follows:

$$\hat{x} = Xx \tag{4-3}$$

$$\hat{C} \equiv CX^{-1} = \begin{bmatrix} C_o \\ \mathbf{0} \end{bmatrix} \tag{4-4}$$

If the dimension of C_o is $p < n$, where n is the dimension of the original system, then we don't need the transformed states $x_{p+1},...,x_n$ for calculating the outputs. If X can also be chosen in such a way that:

$$\hat{A} \equiv XAX^{-1} = \begin{bmatrix} A_o & \mathbf{0} \\ A_{21} & A_{no} \end{bmatrix} \tag{4-5}$$

then these states are also not needed for calculating the observable states $x_1,...x_p$. In this case we can dispense with them altogether, leading to a system with p states that can calculate all the outputs.

Methods for finding the transformation matrix X are provided in textbooks. This scheme initially appears highly advantageous since it does not appear to require that any assumptions be made on the future values of the inputs u. However, the examples given in textbooks are somewhat contrived, and numerical experiments indicate that when applied to the real models encountered in practice, we will rarely be able to eliminate states using such methods. That is, each state of the model will usually contribute at least somewhat to the outputs. In this regard, Moore (1981) has noted that "arbitrarily small perturbations in an uncontrollable [or unobservable] model may make the subspace technically proper [e.g., there no longer are strictly unobservable states]... There may well exist, however, a lower order model which has **effectively** [emphasis added] the same impulse response matrix. [Thus] there is a gap between minimal realization theory and the problem of finding a lower order approximation, which we shall refer to as the 'model reduction problem'."

We must therefore turn to other techniques to try to effectively address the model reduction problem.

4.3 "Rate-Based" Versus "Importance-Based" MOR Strategies

There are two basic techniques for eliminating diagonalized states from a state-space model such as Equation (4-1). First, we can usually eliminate some states because they effectively act as algebraic rather than differential equations over the time frame of interest. This could be the case because they have large, negative eigenvalues and thus adjust so quickly to any change in inputs that we can regard the adjustment transient as effectively instantaneous. We can eliminate other ODEs because their eigenvalues are very small and they thus act as pure integrals. Eliminating all such states may result in very substantial reductions in model complexity.

To simplify the model still further, we must look at how significantly the remaining ODE-defined states contribute to the behavior of the subsystem, as manifested in their affect on the outputs z. If a given state does not affect any of the outputs at all, then it can be eliminated. However, we will rarely find very many transformed states that have no effect at all on the outputs, and less straightforward techniques must therefore be employed.

In order to judge how important a given state is, we look at how important it is, i.e., how much the state affects the various outputs. *Just for this purpose,* we could assume that the various inputs u to the model are relatively constant over this model update interval. While this assumption is hard to rigorously justify, it has so far proven quite effective for distinguishing unimportant states and has the inestimable advantage of allowing us to explicitly calculate the value of each state at the end of the update interval. This then enables us to eliminate those states that do not contribute significantly (relative to other states) to the model outputs. In this scheme, the values of the inputs have generally been set to their average value over the preceding update interval. We have termed this a "state-centric" approach to MOR.

In view of the potentially risky assumption of constant input values over the update interval, an alternative "output-centric" approach has also been formulated. In its extreme form, this scheme essentially eliminates the whole notion of states represented by differential equations. Instead, each output of a subsystem model is viewed as an explicit time-integral over the various inputs to the model. These integrals are obtained by substituting exact solutions to the underlying linearized and diagonalized model in the expressions used to calculate the outputs. As documented in previous reports, the kth output can be written:

$$z_k(t) = \sum_{inputs,m} \sum_{states,i} \frac{\partial h_k}{\partial x_i} e^{\lambda_i t} \left(\int_o^t e^{-\lambda_i \tau} \frac{\partial f_i}{\partial u_m} u_m(\tau) d\tau + \int_o^t e^{-\lambda_i \tau} \overset{o}{f}_i \, d\tau \right) + \frac{\partial h_k}{\partial u_m} u_m + z_k^o \tag{4-6}$$

In this case, we assumed for simplicity that the system matrix could be strictly diagonalized by a coordinate transformation.

This formulation has one key advantage: model order reduction does not require that any assumption be made about the future behavior of the inputs $\boldsymbol{u}$. Instead, one considers a particular input, say u_m, in Equation (4-6) and simply eliminates those terms in the sum over states which are relatively unimportant compared to other terms (arising from other states) involving the same input.

Unfortunately, since the terms from a given state are generally represented in many of the output integrals, straightforward implementations of this scheme result in the transmission of highly redundant information. Thus, a quite possibly fatal disadvantage of this formulation is that it potentially requires the transmission of either relatively large arrays of redundant coefficients (e.g., λ's and $\frac{\partial f}{\partial u}$'s) or a "database" of coefficients along with indices that define which terms are used in each output integral.

If the assumption of constant inputs in the state-centric approach proves too crude, it may be most useful to combine the two techniques by continuing to integrate states as in the state-centric approach, but using the output-centric approach to decide whether a state should be included or not. In other words, one includes a state in the reduced order model if <u>any</u> of the output calculations needs it to properly represent the effect of a given input u on that output. This process makes no particular assumptions about the values of the inputs that will be encountered over the next update interval.

If any of the importance-based techniques are used for accomplishing MOR, we must be able to solve the subsystem's equations. This is necessary so that the effect of each input on the various outputs can be estimated. Such solutions are helpful in any case for obtaining qualitative insights into the states' behaviors. We address this in the next subsection.

4.4 Calculating the Values of the States at the End of the Update Interval to Support MOR

Obtaining solutions to the state equations is very simple when the Jacobian matrix, $\boldsymbol{A}$ in Equation (4-1), can be strictly diagonalized. In this case the individual state equations are completely decoupled from each other and can be solved individually. Unfortunately, as discussed previously, fully diagonalizing the Jacobian matrix, so that only single numbers appear on the diagonal of the transformed matrix, is not always possible. Even in cases where strict diagonalization is possible, this procedure may not be desirable for numerical reasons.

For this reason, we have adopted a block diagonalization scheme in which the diagonal of the Jacobian matrix can contain not only discrete eigenvalues (for states where strict diagonalization is possible), but also blocks of dimension 2x2 or greater. This necessitates that the various MOR schemes be generalized to handle these possibilities. The values of the states at the end of the update interval are calculated as follows for each of these types of diagonal entries.

4.4.1 Single Real Value

When the diagonal element is only a single real value, the corresponding state equation takes the very simple form:

$$\dot{x}_i = \lambda x_i + \frac{\partial f_i}{\partial u_k} u_k(t) + f_i^o \tag{4-7}$$

$$x_i(0) = 0$$

As before, $\boldsymbol{u}(t)$ is used to represent the various external inputs to the state equation, which are for the most part the outputs of the other subsystem models, and we have suppressed the "^" notation that indicates that a transformed coordinate system is used and the "δ" notation that denotes that the state variables have been linearized about a particular state vector. The solution of this equation at the end of the update interval can be readily shown to be:

$$x_i(t) = e^{\lambda t} \int_o^t e^{-\lambda s} \left(\frac{\partial f_i}{\partial u_k} u_k(s) + f_i^o\right) ds \tag{4-8}$$

For model order reduction based on importance, we currently assume in the software library that the values of the inputs are constant, e.g., the average values calculated during the previous update interval. As noted above, this assumption is not strictly necessary, and other, more general methods can also be considered. In any event, this assumption is only made to rank the importance of the various states; the actual reduced order system solution does, of course, take into account the actual time-dependence of the inputs.

Given this assumption, after a change in variables the solution reduces simply to:

$$x_i(t) = \left(\frac{\partial f_i}{\partial u_k} u_{ko} + f_i^o\right)\int_o^t e^{\lambda s} ds = \frac{e^{\lambda t} - 1}{\lambda}\left(\frac{\partial f_i}{\partial u_k} u_{ko} + f_i^o\right) \tag{4-9}$$

The special case with $\lambda = 0$ is encountered fairly frequently. For example, in dynamics models where output quantities such as position are obtained by integrating the velocities, if none of the other equations depend on the velocity, the corresponding eigenvalue will be zero. For this case, the value of the state is obtained from an explicit integral over time (or by merely multiplying by the time interval if the inputs are constant):

$$x_i(t) = \int_o^t \left(\frac{\partial f_i}{\partial u_k} u_k(s) + f_i^o\right) ds \tag{4-10}$$

4.4.2 A Pair of Complex Conjugate Eigenvalues

As discussed above, an additional advantage of the block diagonalization procedure is that it involves only real arithmetic. For cases where the system matrix would ordinarily return a pair of complex conjugate eigenvalues, the algorithm returns a 2x2 block in that location. To be specific, let the two "real" eigenvalues be given by:

$$\lambda = \alpha \pm \beta i \tag{4-11}$$

where α and β are, respectively, the real and imaginary parts of the pair of eigenvalues. One can show that an additional transformation applied to the 2x2 block produced by the Bavely and Stewart block diagonalization algorithm will reduce that block to the canonical form:

$$T = \begin{pmatrix} \alpha & \beta \\ -\beta & \alpha \end{pmatrix} \tag{4-12}$$

For a constant value for the associated input vector $\underline{u}$, the solution of these two states can be written:

$$\underline{x}(t) = \left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{u}} \underline{u}_o + \boldsymbol{f}^o\right)\int_o^t \boldsymbol{e}^{Ts} \boldsymbol{ds} \tag{4-13}$$

where the term in the integral is now a <u>matrix</u> exponential rather than the conventional exponential that appeared in the very similar-looking Equation (4-9). The quantities $\boldsymbol{x}(t)$ and $\boldsymbol{f}$ are two-dimensional vectors. The matrix exponential is defined in terms of a power series expansion:

Greare

$$\exp(T) = 1 + T + \frac{T^2}{2!} + \frac{T^3}{3!} + \ldots \tag{4-14}$$

Evaluating the matrix exponential is, in general, both time-consuming and numerically problematical (Moler and Van Loan, 1979). However, for the special form of the matrix T shown in Equation (4-12), calculating the matrix exponential is quite simple (Hogan, 1994). Let us decompose T as follows:

$$T = \Lambda_s + \Lambda_a \tag{4-15}$$

where the symmetric part is given by:

$$\Lambda_s = \begin{pmatrix} \alpha & 0 \\ 0 & \alpha \end{pmatrix} \tag{4-16}$$

and the asymmetric part is:

$$\Lambda_a = \begin{pmatrix} 0 & \beta \\ -\beta & 0 \end{pmatrix} \tag{4-17}$$

The symmetric part is proportional to the identity matrix, so these two matrices commute. In such a case, it can be shown that the usual scalar property holds true:

$$\exp(T) = \exp(\Lambda_s)\exp(\Lambda_a) \tag{4-18}$$

By employing the series expansion Equation (4-14), one can readily show that:

$$\exp(\Lambda_s) = \begin{pmatrix} \exp(\alpha) & 0 \\ 0 & \exp(\alpha) \end{pmatrix} \tag{4-19}$$

and

$$\exp(\Lambda_a) = \begin{pmatrix} \cos(\beta) & \sin(\beta) \\ -\sin(\beta) & \cos(\beta) \end{pmatrix} \tag{4-20}$$

Equations (4-19) and (4-20) are substituted into Equation (4-13) and their product is integrated term-by-term. The result is a 2x2 matrix:

$$e^{\alpha t}\begin{pmatrix} \frac{\alpha\cos(\beta t) + \beta\sin(\beta t)}{\alpha^2+\beta^2} - \frac{\alpha}{\alpha^2+\beta^2} & \frac{\alpha\sin(\beta t) - \beta\cos(\beta t)}{\alpha^2+\beta^2} + \frac{\beta}{\alpha^2+\beta^2} \\ \frac{-\alpha\sin(\beta t) + \beta\cos(\beta t)}{\alpha^2+\beta^2} - \frac{\beta}{\alpha^2+\beta^2} & \frac{\alpha\cos(\beta t) + \beta\sin(\beta t)}{\alpha^2+\beta^2} - \frac{\alpha}{\alpha^2+\beta^2} \end{pmatrix} \tag{4-21}$$

To obtain the values of the two states at the end of the update interval, this 2x2 matrix is substituted into Equation (4-13):

$$
\begin{aligned}
&\left(x_1(t) \quad x_2(t)\right) = e^{\alpha t}\left(\frac{\partial f_1}{\partial u_k}u_k + f_1^o \quad \frac{\partial f_2}{\partial u_k}u_k + f_2^o\right) * \\
&\begin{pmatrix}
\frac{\alpha\cos(\beta t)+\beta\sin(\beta t)}{\alpha^2+\beta^2} - \frac{\alpha}{\alpha^2+\beta^2} & \frac{\alpha\sin(\beta t)-\beta\cos(\beta t)}{\alpha^2+\beta^2} + \frac{\beta}{\alpha^2+\beta^2} \\
\frac{-\alpha\sin(\beta t)+\beta\cos(\beta t)}{\alpha^2+\beta^2} - \frac{\beta}{\alpha^2+\beta^2} & \frac{\alpha\cos(\beta t)+\beta\sin(\beta t)}{\alpha^2+\beta^2} - \frac{\alpha}{\alpha^2+\beta^2}
\end{pmatrix}
\end{aligned}
\tag{4-22}
$$

This result could also have been obtained by initially retaining the complex form for each of the eigenvalues, performing the integration, and then converting the result to the equivalent 2x2 real form.

4.4.3 Upper Triangular Block Matrix of Order NxN

The last type of matrix that can be returned on the diagonal by the block diagonalization algorithm is an NxN upper triangular matrix. This always occurs when the subsystem model possesses *N* degenerate (equal valued) eigenvalues for which independent eigenvectors cannot be defined (defective states). Such blocks also appear in practice when the eigenvalues, although slightly different, are not sufficiently different that they can be treated separately without introducing a numerically ill-conditioned similarity transformation.

Computing the values of the states at the end of the update interval is done in a very similar fashion to that just discussed. The values of the coupled states at the end of the update interval are again given by Equation (4-13). The matrix exponential is also calculated using a similar procedure (Tomasi, 2000): as in the case of the 2x2 complex conjugate eigenvalues, the sub-block matrix is decomposed as follows.

$$\boldsymbol{T} = \boldsymbol{\Lambda}_s + \boldsymbol{U} \tag{4-23}$$

Here Λ_s is similar to what was obtained in the 2x2 case, namely a NxN diagonal matrix with (we assume) essentially equal matrix elements λ (the degenerate eigenvalue) along the diagonal. U is a strictly upper-triangular matrix (no entry on the diagonal or below) that results from the Schur decomposition used in the Bavely-Stewart block diagonalization process. The matrices Λ_s and U obviously commute, so the only new task is to evaluate $\exp(U)$. A strictly upper triangular NxN matrix U can readily be shown to be *nilpotent* of order *N*, that is:

$$\boldsymbol{U}^n = 0 \tag{4-24}$$

Thus, the power series expansion (4-14) only contains terms up to order *n*-1:

$$\exp(\boldsymbol{U}) = 1 + \sum_{j=1}^{n-1}\frac{\boldsymbol{U}^j}{j!} \tag{4-25}$$

Substituting into Equation (4-13) and integrating term-by-term yields:

$$x(t) = \frac{f_o + u_o \frac{\partial f}{\partial u}}{\lambda} \left(\sum_{j=0}^{n-1} \left(e^{\lambda t} U^j t^j \sum_{m=0}^{j} \frac{(-1)^m}{(j-m)!(\lambda t)^m} - \frac{U^j (-1)^j}{(\lambda)^j} \right) \right) \tag{4-26}$$

or, equivalently:

$$x(t) = \frac{f_o + u_o \frac{\partial f}{\partial u}}{\lambda} \left(\sum_{j=0}^{n-1} U^j \left(e^{\lambda t} t^j \sum_{m=0}^{j} \frac{(-1)^m}{(j-m)!(\lambda t)^m} - \frac{(-1)^j}{(\lambda)^j} \right) \right) \tag{4-27}$$

Note that this reduces to the usual result for the 1x1 (completely diagonalized) case, for which the only term is j=m=0.

For the special case where $\lambda_s = 0$, which is encountered fairly often, we obtain:

$$x(t) = \left(f_o + u_o \frac{\partial f}{\partial u} \right) \sum_{j=0}^{n-1} \frac{U^j t^{j+1}}{(j+1)!} \tag{4-28}$$

4.4.4 Variables Defined by Pure Integrals Over Time

"States" that can be expressed as explicit, "pure" integrals over time can be removed from the set of ODE-defined states x since no other state equation depends on them. If we perform a "one-pass" removal of such variables (see Section 3), their rates-of-change can be written:

$$\dot{y} = Ha + Kx + y_o \tag{4-29}$$

As discussed in Section 3, if a "one-pass" removal is performed, the y do not depend on each other, and we have assumed that here.

As in the case of the other types of states, the values of the y at the end of the update interval are needed to support Model Order Reduction. These values can be calculated by simply integrating each of the terms on the right-hand side of Equation (4-29) over time. This can be done quite simply given the explicit or series formulas developed for each of the types of the states x, Equations (4-9), (4-22), and (4-27). This would not be possible if an iterative, multiple-pass algorithm was used to eliminate more of the pure integrals.

4.5 Model Order Reduction

4.5.1 Implications of the Form of the Subsystem Models on the Optimal MOR Strategy

Our ultimate aim is to find the simplest possible model that will adequately represent the detailed behavior of the subsystem over a limited time period called the update interval. The structure of the subsystem models affects the most effective strategy for model order reduction.

The software library is intended to integrate in a distributed fashion subsystems that are internally complicated but that connect to neighboring subsystems via a relatively few number of "boundary variables." If the number of boundary variables is too high, such a partitioning is unlikely to be effective. Based on this consideration and the discussion provided in Section 2, the following assumptions are considered reasonable:

1. The number of inputs u in any subsystem is small compared to the total number of internal states n.
2. The number of outputs o, which are also inputs to the other subsystem models, is also small compared to the number of internal states.
3. The compiler and computer hardware are such that they can take dot products and multiply by zero very quickly.
4. The number of subsystems in the overall system is large as is the number of available processors.

Some implications of these assumptions are:

1. The "output-centric" approach mentioned earlier requires either enormous redundancy in the subsystem model (since each output generally will involve the same state-related information needed by one or more of the other outputs), or the transmission of a large number of pointers to a "central database" of state-related data. Given assumptions (2) and (3) above, if we need to retain a state at all (because it is needed in at least one output), it would be just as efficient to integrate it by itself ("state-centric approach") and include the state in every output.
2. If a state is kept, assumption (1) suggests that there is little motivation to attempt to eliminate its dependence on any of the inputs (i.e., taking advantage of "uncontrollability"). Doing so would save little time and could compound the problem of accurately dealing with unexpected large changes in inputs.
3. Based on assumption (4), we should not necessarily reject model order reduction techniques just because they require a relatively large computational effort. If a substantial effort is required to generate a simplified subsystem model, this could be repaid if the model is used on a large number of other processors. Thus, spending extra time to reduce the complexity of a subsystem model could be worthwhile.

States can be eliminated from the model, either because they don't act as differential equations or because they are not "important." We first consider the former.

4.5.2 Rate-Based Schemes for Eliminating States Based On Their Individual Time-Dependence

a. 1x1 states

When the eigenvalue of a state that has been completely diagonalized (i.e., is represented by a 1x1 block) is very small, its rate of change no longer depends on itself to a significant degree, and the state can be integrated directly as shown in Equation (4-10). Thus, for a given output given by $\boldsymbol{h(x,u)}$, we can form pure integrals over time of the form:

$$\sum_{inputs\,u} \int_0^t \sum_{states\,with\,\lambda \to 0} \frac{\partial h}{\partial x} \frac{\partial f}{\partial u} u(s)ds \tag{4-30}$$

Grouping the terms in this way can be advantageous because the summation over the states collapses all such states involving a given output into a single term for each of the inputs *u*:

$$\int_0^t \sum_{inputs\,i} a_i\, u_i(s)ds \tag{4-31}$$

Because of assumption (2), there probably will be many more such defective states than there are outputs *z*. For this reason, it appears to be advantageous to combine all such states affecting a given output, rather than including them in the set of "pure integral states" that were eliminated prior to diagonalization, although this is another possibility. Thus, after performing this initial MOR step, the number of states has been reduced and the values of the outputs are given by:

$$z = \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}} \boldsymbol{x} + \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{u}} \boldsymbol{u} + z_o + \int_o^t \boldsymbol{a}\, \boldsymbol{u}(s)ds \tag{4-32}$$

To review, the terms on the right-hand side of the Equation (4-32) represent, respectively:

1. The dependence of the output on the retained (ODE) states $\boldsymbol{x}$.
2. The "direct dependence" of the output on the inputs $\boldsymbol{u}$.
3. A constant, around which the output function was linearized.
4. The dependence of the output on the time integral of inputs that arose from states with very small eigenvalues that were eliminated in accordance with Equation (4-31).

Now consider the case where an eigenvalue is very large and negative. In this case, the associated state will very quickly respond to a change in its inputs by settling down to a quasi-equilibrium value given by taking the limit $\lambda t \to -\infty$ in Equation (4-9):

$$x_i(t) = \frac{-1}{\lambda_i} \left(\frac{\partial f_i}{\partial u_k} u_k(t) + f_i^o \right) \tag{4-33}$$

(no sum on *i*)

In these cases, the modeler typically does not care about the precise dynamical path the system follows toward equilibrium (if he or she did care about fast dynamics, this would be "known" to the software library by the small value specified for the time step). Therefore, we can simply substitute the equilibrium value of the state rather than solving its ODE. This allows us to eliminate the state by combining its terms with the terms shown in Equation (4-32) that define the direct dependence of the output on the inputs. So:

$$\frac{\partial z}{\partial u} := \frac{\partial z}{\partial u} - \frac{\partial z}{\partial x_i}\frac{1}{\lambda_i}\frac{\partial f_i}{\partial u} \tag{4-34}$$

The constant term associated with a given output z also is combined with the z_o terms in Equation (4-32) in the same way:

$$z_o := z_o - \frac{\partial z}{\partial x}\frac{f_i^o}{\lambda} \tag{4-35}$$

A side benefit of this procedure is that it may also justify simple integration techniques for the states that are retained as ODEs: the very "stiff" states that could otherwise require special treatment will have been eliminated. More sophisticated integration schemes might still be desirable for accuracy (so long as the accuracy achieved is not rendered moot by the approximations involved in generating the simplified models), but stability will be guaranteed, even when using simple Euler integration.

b. 2x2 eigenvalues

Consider the case where the on-diagonal element of a 2x2 block, which denotes the real part of the conjugate eigenvalues, has a large, negative value. We know from the complex counterpart of the 2x2 real form that transients caused by input parameter changes will very quickly damp out, just as in the 1x1 case. The quasi-steady-state solution is the 2x2 counterpart of Equation (4-33):

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = -\Lambda^{-1}\left[\begin{pmatrix} \dfrac{\partial f_1}{\partial u} \\ \dfrac{\partial f_2}{\partial u} \end{pmatrix}\underline{u}(t) + \begin{pmatrix} f_1^o \\ f_2^o \end{pmatrix}\right] \tag{4-36}$$

The inverse of the 2x2 eigenvalue matrix Λ is easily evaluated, yielding:

$$\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \frac{-1}{\alpha^2+\beta^2}\begin{pmatrix} \alpha & -\beta \\ \beta & \alpha \end{pmatrix}\left[\begin{pmatrix} \dfrac{\partial f_1}{\partial u} \\ \dfrac{\partial f_2}{\partial u} \end{pmatrix}\underline{u}(t) + \begin{pmatrix} f_1^o \\ f_2^o \end{pmatrix}\right] \tag{4-37}$$

The states represented by Equation (4-37) can be eliminated from Equation (4-32) by combining their effect on the outputs with the direct dependence and constant terms as was done in the 1x1 case.

c. NxN block associated with defective states

An NxN block will generally have nearly the same eigenvalues down the diagonal, since this is the reason that the Bavely-Stewart algorithm will form a block in the first place. If this value is very large and negative, the entire group of states can be represented adequately as algebraic equations, which can be formally eliminated in a manner similar to that used for other algebraic equations. First, the quasi-steady-state solution of the states represented by the block is given by setting the rates to zero:

$$x = -(\Lambda + U)^{-1}(f_o + \frac{\partial f}{\partial u} u) \tag{4-38}$$

where $(\Lambda + U)^{-1}$ is the inverse of the NxN block shown in Equation (4-23). Then the direct dependence terms in Equation (4-32) that prescribe how the outputs depend on parameters are altered as follows:

$$\frac{\partial z}{\partial u} := \frac{\partial z}{\partial u} - \frac{\partial z}{\partial x}(\Lambda + U)^{-1}\frac{\partial f}{\partial u} \tag{4-39}$$

Due to the special form of $\Lambda + U$, its inverse is upper triangular and can be efficiently evaluated by back-substitution. The constant term in Equation (4-38) is incorporated as:

$$z_o := z_o - \frac{\partial z}{\partial x}(\Lambda + U)^{-1} f_o \tag{4-40}$$

If we compare Equations (4-39) and (4-40) with (4-34) and (4-35), we see that as in the 2x2 case, the matrix inverse of the NxN block has the same role as the reciprocal of a single eigenvalue.

In principle, cases where the eigenvalues are zero can also be eliminated by combining all the various blocks in which this is the case into one explicit integral using Equation (4-28). This would be useful only if numerous NxN blocks with zero eigenvalues were present; to date, this has not been encountered.

4.5.3 MOR Strategy for Selecting States That Should Be Kept Based on Their Relative Importance

The previous discussion has detailed how states may be eliminated based on their associated time-dependent behavior. After this has been done, further simplification of the subsystem model may be desirable. Taking further measures to eliminate states is somewhat more difficult, because we now must find a way to eliminate states defined by true ODEs that do contribute to the various outputs, but in some sense do not contribute significantly. This is more difficult for several reasons, not least of which is the fact that we must define "significant" in a meaningful fashion.

Possibilities that could be investigated are:

1. Purely diagonal states: We could check terms that define the contribution of a given parameter to an output one by one. We would retain any state when its contribution is at least a user-specified percent of the total contribution (i.e., we would set a flag indicating that the state will be retained in the subsystem model because at least one output needs it). The closed form solution Equation (4-9) would be used to compare the contributions of the various states.

2. 2x2 and NxN: Use the closed form solutions, e.g., Equations (4-22) and (4-27), to integrate to the end of the update interval, assuming a constant value for the inputs to that state. The latter could, for example, be based on the average over the last update interval. As noted previously, this scheme may not be accurate if the input values assumed vary radically from those previously encountered. All states contained in a block would be retained if any of the states in its block are needed.

The benefits to be gained by considering the relative importance of the various states no doubt depends on the nature of the model, the need for accuracy, and how loose a criterion is used to define "importance." The software library should probably provide the user an option to use these techniques or not, since they may lead to a less accurate result if the inputs vary strongly from one update interval to another. Loss of accuracy may trigger the *simulation manager* to issue a model update; if this occurs too frequently, execution time could actually be increased.

Implementation and testing of the "state-centric" elimination, using the average input values for the inputs over the preceding update interval has proven surprisingly accurate and effective. Implementation of the other techniques, including treatment of 2x2 and NxN blocks, will occur early in the next fiscal year.

4.6 ADVANCED MOR TECHNIQUES

As has been discussed at length, the biggest problem with eliminating states based on their relative contribution to the outputs is that we don't reliably know what their inputs will be over the coming update interval. Any techniques that could be brought to bear on this difficult problem would be very worthwhile.

Over the past two decades, a very large body of work has appeared in the control systems literature on so-called "balancing transformation" methods for model order reduction (Dullerud and Paganini, 2000). Such methods are unique in that they not only provide reduced order models, but, if certain assumptions are met, they also provide rigorous error-bounds on the output of the approximate models. The theory behind such methods is relatively complicated, but the underlying principle is that a coordinate transformation can be defined so that the "observability" and "controllability" is distributed in a desired manner among the transformed states. One then chooses the states which are both most affected by the inputs and have the largest influence on the outputs and ignores the rest of the states.

This method, while very appealing on theoretical grounds, may not be applicable in our application. For one thing, the method assumes that the system is stable and is observed for an infinite time. If these conditions are satisfied, relatively uncontrollable states (i.e., those that depend weakly on the inputs), while they certainly can affect the outputs early, will eventually die away. This is not the case in our application, since even completely uncontrollable states will contribute terms to the outputs that drop exponentially over the update interval in accordance with their associated eigenvalue.

Nevertheless, generalizations of this approach to finite time situations may prove possible, and we will investigate these further in the coming fiscal year. Diagonalization, as implemented in the software library, is a very useful starting point for applying such methods since:

- Diagonalization allows the unstable parts of the model, characterized by positive real eigenvalues, to be partitioned from the stable parts. MOR using balancing methods could then be performed on the latter.
- The calculation of the Grammians, which involves calculating the matrix exponential of the Jacobian, is much more easily performed on the diagonalized system than on the full matrix, as was outlined above.

4.7 CONCLUSIONS

To support model order reduction of block-diagonalized system models, we have developed closed-form solutions for the state variables in terms of the inputs. These solutions were obtained for the three types of system equations defined by the block diagonalization algorithm: 1x1 blocks (strictly diagonalized states), 2x2 blocks resulting from pairs of states with complex conjugate eigenvalues, and NxN blocks representing groups of states with degenerate or nearly-degenerate eigenvalues.

MOR based on the individual time-dependence of each set of states, as determined by the real part of the associated eigenvalue, is relatively straightforward and does not require that assumptions be made about the behavior of the inputs over the upcoming time interval. MOR based on the relative importance of the various states on the outputs can also be done if reasonable assumptions can be made about the inputs. More advanced MOR techniques, based on balancing transformations, hold promise but require generalization if they are to be used in this application.

5 ARCHITECTURE OF SOFTWARE LIBRARY

This section describes the object-oriented structure and functionality of the classes that comprise the distributed simulation library, as well as the CORBA-based network communication protocol implemented for carrying out remotely distributed simulations.

5.1 OVERVIEW OF CLASS STRUCTURE AND FUNCTIONALITY

The software library being developed in this project is designed to support the object-oriented modeling and distributed simulation of aerospace systems. An object-oriented class structure is used that allows these complex systems to be partitioned into relatively independent objects. Partitioning the system in this manner supports the re-use of model objects developed on previous projects and attacks overall model complexity by allowing the subsystem models to be developed and validated independently.

We have designed and implemented a class structure for developing models using this object-oriented approach in the C++ programming language. The modeling classes are classified into two categories. The first type, which include the classes *Detailed_Model* and *Simple_Model*, focus primarily on localized (i.e., non-distributed) model development and simulation of a single subsystem. These localized model classes are designed to support a bottom-up modeling approach where a relatively small number of engineers initially develop and validate a model of a particular subsystem of an aerospace vehicle. By encapsulating these subsystem models as components in a common framework, these classes facilitate subsequent large-scale simulation of the entire aerospace vehicle. The second type of modeling classes, which include the classes *Simulation_Process* and *Simulation_Manager*, implement the necessary logic and communication protocols for combining these localized subsystem models into a geographically distributed simulation of a complex process.

In summary, the *Detailed_Model* class encapsulates the user-defined mathematical model of a single subsystem (e.g., a fuel pump) described by a set of ordinary differential and algebraic equations. Periodically during a simulation, the *Detailed_Model* class generates simplified versions of itself (through numerical linearization, diagonalization and order reduction of its equations), resulting in an instance of a *Simple_Model*. This *Simple_Model* serves as a valid surrogate for predicting the behavior of the rigorous *Detailed_Model* over short periods of simulation time. During a simulation, each of the *N* subsystem models is encapsulated as a *Detailed_Model* and coupled with *Simple_Model* versions of the other *N-1* subsystems in a *Simulation_Process*. As a result, each *Simulation_Process* represents a complete model of all subsystems in the overall process. In addition, the *Simulation_Process* also maintains a local *Simple_Model* of its *Detailed_Model* for determining when a new updated version of its *Simple_Model* is required. The overall simulation is coordinated by a single instance of a *Simulation_Manager*, which communicates the updated *Simple_Model*s among the various *Simulation_Process*es as necessary.

The interaction of these four classes during a distributed simulation is depicted schematically Figure 5-1 for a process with three subsystems (labeled *A*, *B*, and *C*). Subsystem *A* is encapsulated as *Detailed_Model A* in *Simulation_Process A* along with *Simple_Model* versions of subsystems *B* and *C*. *Simulation_Process* A then integrates *Detailed_Model A* with *Simple_Model*s *B* and *C* simultaneously. At each intermediate time step during the simulation, *Simulation_Process A* compares the outputs calculated by *Detailed_Model A* with the outputs calculated by the current local *Simple_Model A*. If the error exceeds some prespecified criteria, the *Simulation_Process* notifies the *Simulation_Manager* that *Simple_Model A* must be updated in all *Simulation_Process*es. In a likewise manner, *Detailed_Model B* is integrated with *Simple_Models* of *A* and *C*, and *Detailed_Model C* is integrated with *Simple_Models* of *A* and *B* in their respective *Simulation_Process*es. The *Simulation_Manager* coordinates the overall simulation by using a CORBA-based protocol to issue command invocations and communicate *Simple_Model*s among the various *Simulation_Processes.*

Typically, each *Simulation_Process* and the *Simulation_Manager* will run on its own computer (or processor) which may reside anywhere on the local intranet or on the Internet. However, it is entirely possible that some or even all of the *Simulation_Process*es and the *Simulation_Manager* run on the same computer.

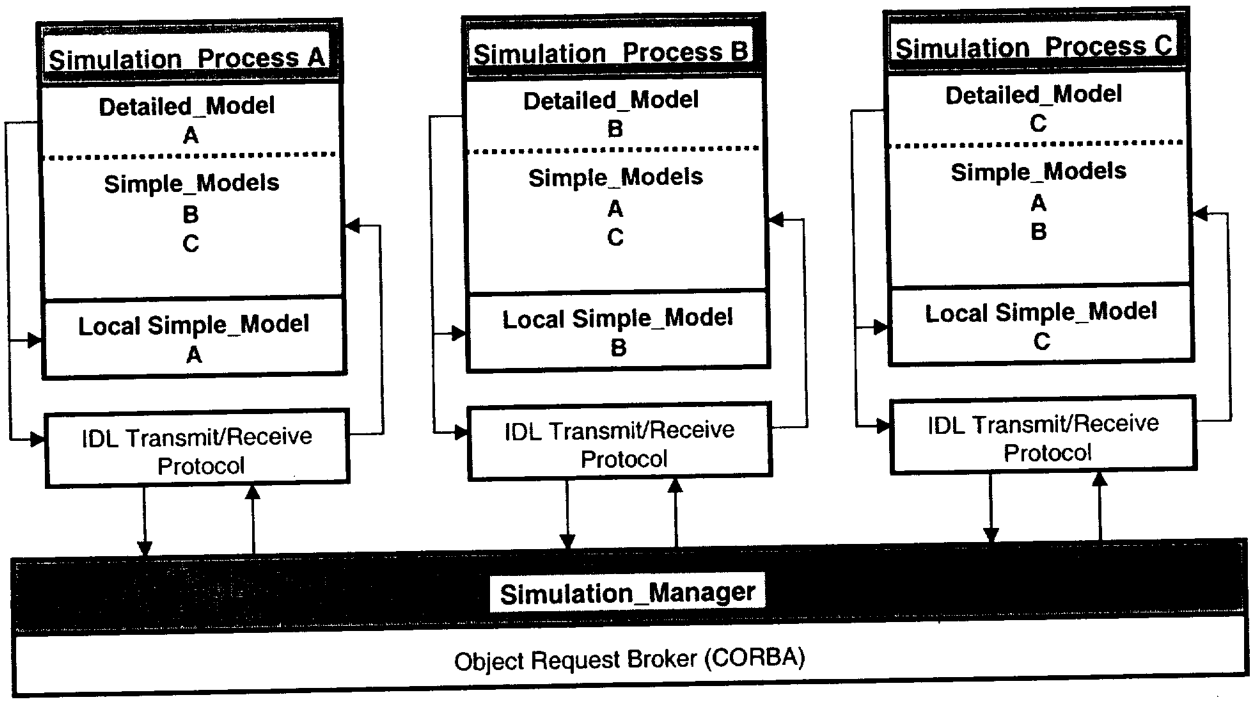

Figure 5-1. Distributed Interaction of Object-Oriented Modeling Classes During a Simulation.

5.2 PRIMARY ACTIONS EXECUTED DURING DISTRIBUTED SIMULATION

Before describing the classes in the distributed simulation software library in greater detail, the primary actions executed during a distributed simulation will first be described. These actions may be structured into four sequential stages: (1) distributed object initialization, (2) model initialization, (3) model simulation, and (4) simulation termination. As each of these stages are described below, it may be helpful to refer to Figure 5-1.

Stage-1: Distributed Object Initialization

1. Before the simulation is run, a CORBA-based *Naming Service* daemon is started on a networked computer with a commonly known IP address. The *Naming Service* daemon is a standard program provided with CORBA ORB software libraries. It provides a central location for storing the addressable object reference of each *Simulation_Process* and the *Simulation_Manager* in the simulation, which allows the various objects to locate one another across the network.
2. An instance of a *Simulation_Process* is created on each computer hosting a simulation of a single subsystem. The *Simulation_Process* creates an instance of a *Detailed_Model* that represents the user-defined model of this subsystem. The *Simulation_Process* then locates the *Naming Service* on the prespecified IP address and registers its textual name and addressable object reference with the *Naming Service* so that the *Simulation_Manager* can locate it.

3. An instance of the *Simulation_Manager* is then created on a single computer. It also locates the *Naming Service* on the prespecified IP address and registers its textual name and addressable object reference. The *Simulation_Manager* then uses the *Naming Service* to resolve the addressable object reference of each *Simulation_Process*. The *Simulation_Manager* then initiates contact with each *Simulation_Process*. When contacted, each *Simulation_Process* uses the *Naming Service* to resolve the addressable object reference of the *Simulation_Manager*. Two-way communication is thus made possible between each *Simulation_Process* and the *Simulation_Manager*.

Stage-2: Model Initialization

4. Once all *Simulation_Process*es have been located, the *Simulation_Manager* requests that all *Simulation_Process*es initialize the state variables of their *Detailed_Models* to their initial conditions.
5. After initialization, each *Simulation_Process* then requests that its *Detailed_Model* generate the data required to instantiate a *Simple_Model*. An overview of this process is illustrated in Figure 5-2. The *Simulation_Process* then transmits this data to the *Simulation_Manager*.
6. Once the *Simulation_Manager* receives a complete set of *Simple_Model* data structures, it transmits this set among all *Simulation_Processes*.
7. When a *Simulation_Process* receives the set of *Simple_Model* data structures, it creates a *Simple_Model* version of each remote subsystem, as well as a *Simple_Model* of its local subsystem.

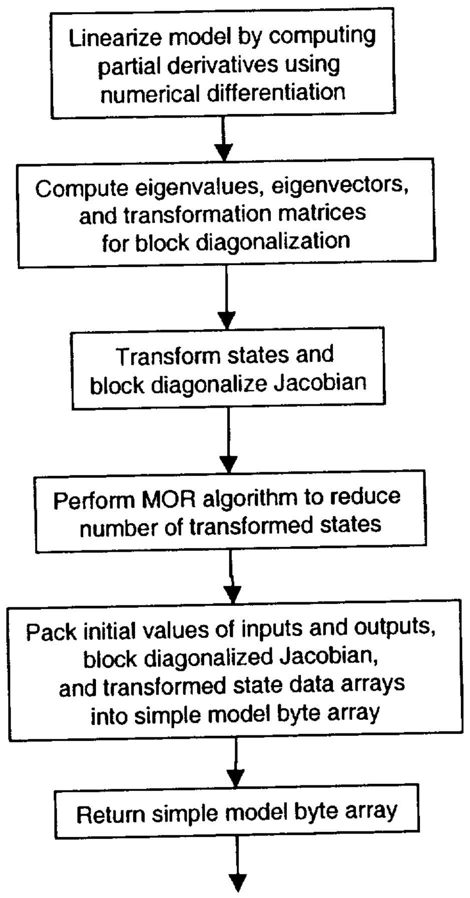
Linearize model by computing partial derivatives using numerical differentiation
Compute eigenvalues, eigenvectors, and transformation matrices for block diagonalization
Transform states and block diagonalize Jacobian
Perform MOR algorithm to reduce number of transformed states
Pack initial values of inputs and outputs, block diagonalized Jacobian, and transformed state data arrays into simple model byte array
Return simple model byte array

Figure 5-2. Detailed_Model:: generate_simple_model(...) function.

Stage-3: Model Integration

8. The *Simulation_Manager* then requests that each *Simulation_Process* begin simultaneous numerical integration of its *Detailed_Model* with its *Simple_Models* up to the next reporting interval. Note that each *Simulation_Process* is allowed to proceed at its own integration rate during the simulation.

9. After every intermediate integration time step, each *Simulation_Process* verifies the validity of its current local *Simple_Model* by comparing the outputs calculated by the *Simple_Model* with the outputs calculated by its *Detailed_Model*. If a specified error criterion is exceeded, the *Simulation_Process* halts integration and reports the time point at which its local *Simple_Model* was determined to be invalid. This process is illustrated in Figure 5-3.

10. If one or more *Simulation_Process*es do not reach the end of the reporting interval, the *Simulation_Manager* requests that all *Simulation_Process*es reset their state variables to the values at the beginning of the reporting interval, and only simulate as far as the first time point at which a *Simple_Model* became invalid. At that time point, new *Simple_Models* are generated and distributed among the *Simulation_Process*. The updated *Simple_Model*s are used for the remainder of the reporting interval (or at least until another becomes invalid). This process is illustrated in Figure 5-4.

11. When all *Simulation_Process*es reach the end of the reporting interval successfully, the calculated results are reported to a data file, and the algorithm returns to step 8 until the final simulation time is reached.

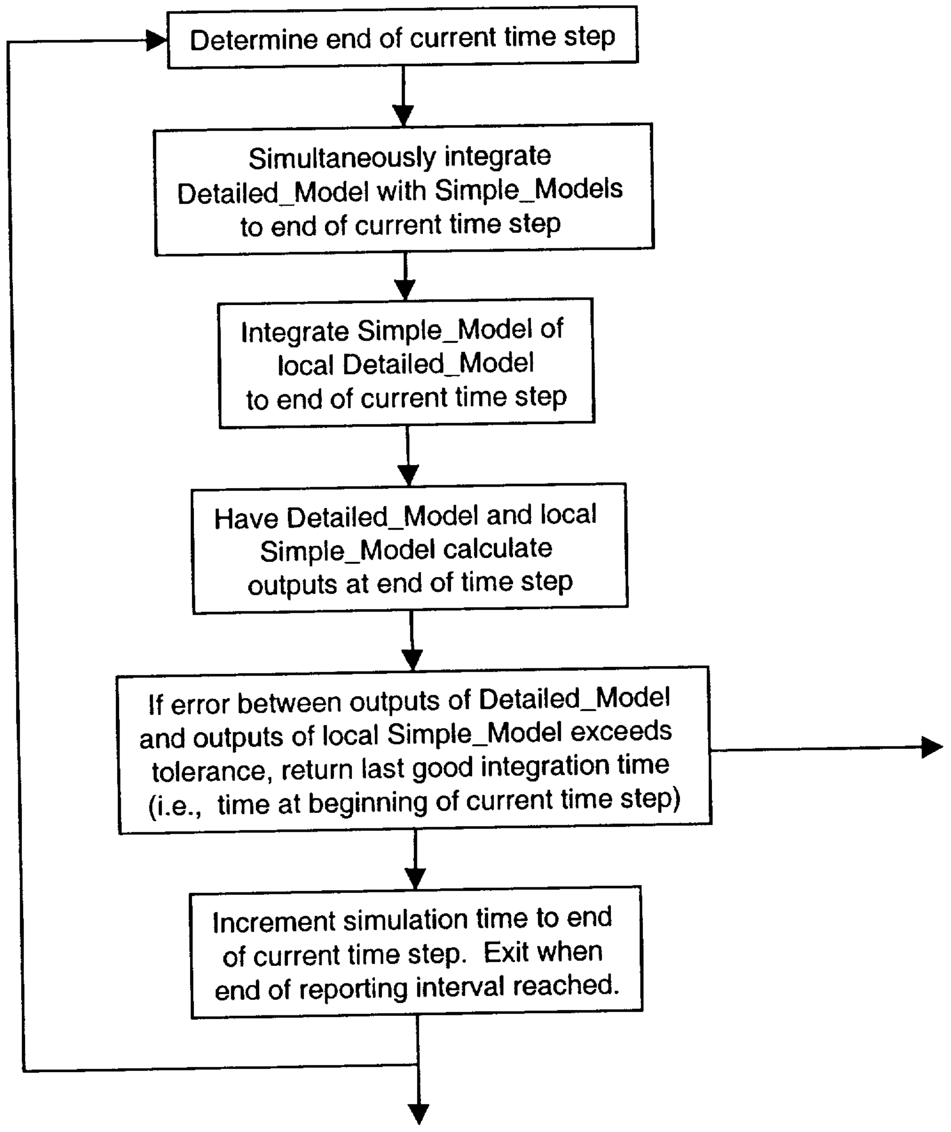
Determine end of current time step
Simultaneously integrate Detailed_Model with Simple_Models to end of current time step
Integrate Simple_Model of local Detailed_Model to end of current time step
Have Detailed_Model and local Simple_Model calculate outputs at end of time step
If error between outputs of Detailed_Model and outputs of local Simple_Model exceeds tolerance, return last good integration time (i.e., time at beginning of current time step)
Increment simulation time to end of current time step. Exit when end of reporting interval reached.

Figure 5-3. Simulation_Process:: integrate(...) function.

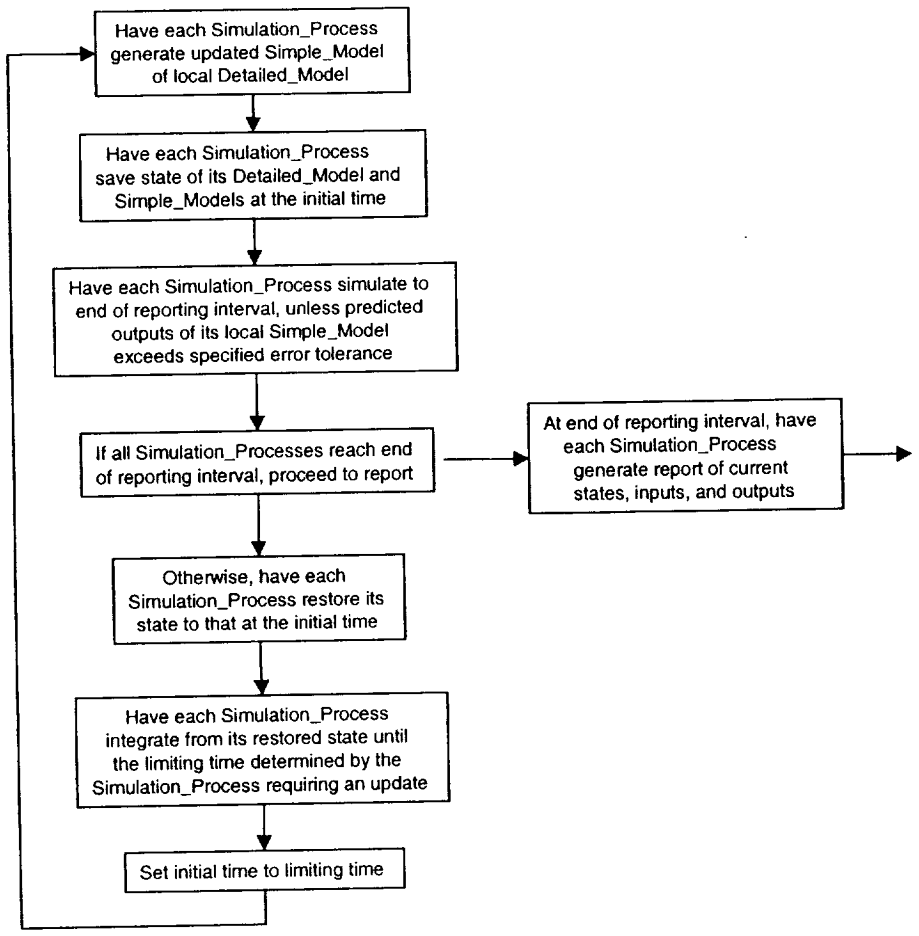
Have each Simulation_Process generate updated Simple_Model of local Detailed_Model
Have each Simulation_Process save state of its Detailed_Model and Simple_Models at the initial time
Have each Simulation_Process simulate to end of reporting interval, unless predicted outputs of its local Simple_Model exceeds specified error tolerance
If all Simulation_Processes reach end of reporting interval, proceed to report
At end of reporting interval, have each Simulation_Process generate report of current states, inputs, and outputs
Otherwise, have each Simulation_Process restore its state to that at the initial time
Have each Simulation_Process integrate from its restored state until the limiting time determined by the Simulation_Process requiring an update
Set initial time to limiting time

Figure 5-4. Simulation_Manager:: integrate(...) function.

Stage-4: Simulation Termination

12. Once the final simulation time is reached, the *Simulation_Manager* requests that all *Simulation_Process*es transmit their results as a data file. The *Simulation_Manager* can then issue a command for all *Simulation_Process*es to shut down, or they can remain active for another subsequent simulation.

5.3 OBJECT-ORIENTED MODELING CLASSES

The object-oriented structure of the four modeling classes, *Detailed_Model*, *Simple_Model*, *Simulation_Process*, and *Simulation_Manager*, that make implementation of this logic possible are now described.

The *Detailed_Model* class is summarized in Table 5-1. This class encapsulates the mathematical model of a single subsystem described by a set of ordinary differential equations and algebraic equations:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x},\mathbf{y},\mathbf{u}) \tag{5-1}$$

$$\mathbf{g}(\mathbf{x},\mathbf{y},\mathbf{u}) = \mathbf{0} \tag{5-2}$$

where **x** are the differential state variables, **y** are the algebraic variables, and **u** are the input variables of the subsystem. In addition, a set of outputs variables **z** that serve as inputs to other subsystem models are defined as nonlinear functions of **x**, **y**, and **u**.

$$\mathbf{z} = \mathbf{h}(\mathbf{x}, \mathbf{y}, \mathbf{u}) \tag{5-3}$$

Mathematically, an instance of *Detailed_Model* is a well-defined system of equations that, given constant or time-dependent values of its input variables and initial values of its state variables, may be used to integrate the time-dependent behavior of its state and output variables.

In order to create a *Detailed_Model*, the user first specifies the number of differential state, algebraic, input, and output variables for the model. The *Detailed_Model* class automatically allocates sufficient computer memory for storing the values of these variables. The differential and algebraic equations are represented by the user-defined functions *calculate_xdot(...)* and *calculate_gy(...)*, respectively. The equations for calculating outputs are represented by the user-defined function *calculate_outputs(...)*. The user also provides a fourth function *initialize_states(...)*, which initializes the differential state variables $\mathbf{x_0}$ to their desired values at the beginning of the simulation.

Most importantly, the *Detailed_Model* class also encapsulates the methodology of the model order reduction algorithm. As a result, at any instance in simulation time, the *generate_simple_model(...)* function of the *Detailed_Model* class can be called to automatically create a simplified, reduced-order model, represented as a *simple_model_key* that may be used to create any number of duplicate instances of a *Simple_Model*.

Table 5-1. Description of *Detailed_Model* Class

Class:	*Detailed_Model*
Inputs specified by user:	*calculate_xdot(...)* function for computing time-derivatives of state variables given current values of state, algebraic, and input variables.
	calculate_gy(...) function for computing residuals of algebraic equations given current values of state, algebraic, and input variables
	calculate_outputs(...) function for computing values of output variables given current values of state, algebraic, and input variables.
	initialize_states(...) function for initializing values of state variables at beginning of simulation.
	number_of_states, number_of_inputs, number_of_outputs integers that define the dimensions of the model.
Key class methods:	*generate_simple_model(...)* generates the data necessary for constructing a reduced-order model of the *Detailed_Model*.
	report(...) function reports current values of state, algebraic, input, and output variables.
Key class variables:	*x[ ], xdot[ ], y[ ], gy[ ], u[ ], and z[ ]* arrays for representing current values of states, rates-of-change, algebraic variables, residuals, inputs, and outputs, respectively

The *Simple_Model* class is summarized in Table 5-2. This class represents a linearized reduced-order model of a particular *Detailed_Model*. Mathematically, it encapsulates Equation (2-13), which calculates the rates of change of the transformed state variables, and Equation (2-14), which calculates its outputs, where both equations depend on the current value of its transformed state and input variables. Given the same time-dependent input variables as to the *Detailed_Model*, the *calculate_xhat_dot(...)* and *calculate_outputs(...)* functions of the *Simple_Model* can be used during integration to accurately predict the outputs of the *Detailed_Model* over short periods of simulation time. Updated periodically, the sequence of *Simple_Models* thus serve as surrogates for calculating the outputs of the *Detailed_Model* over the entire simulation.

Table 5-2. Description of *Simple_Model* Class

Class:	*Simple_Model*
Inputs specified by *Detailed_Model*:	*simple_model_key* data structure containing information required for instantiating *Simple_Model* class
	number_of_transformed states, *number_of_inputs*, *number_of_outputs* which are integers that define the dimensions of the simple model.
	initial_time which specifies the point in simulation time at which the simplified model data was generated.
Key class methods:	*calculate_xhat_dot (...)* function for computing time-derivatives of transformed state variables given current values of transformed state and input variables.
	calculate_outputs(...) function computes values of output variables given current values of transformed state and input variables.
	report(...) function reports current values of transformed state, input, and output variables.
Key class variables:	*x_hat[ ], u[], and z[]* arrays representing current values of the transformed state, input, and output variables..

In our design, we have focused on keeping the *Detailed_Model* and *Simple_Model* streamlined so that engineers developing subsystem models are not burdened with the intricacies of distributed simulation of complex systems. Consequently, in our design we have encapsulated this functionality in two separate classes, *Simulation_Process* and *Simulation_Manager*.

The *Simulation_Process* class is summarized in Table 5-3. This class represents an integrated multi-system model with N-component subsystems. An instance of a *Simulation_Process* is created on each computer where the *Detailed_Model* of a particular subsystem is defined. The overall model is simulated by the *Simulation_Process* by integrating the *Detailed_Model* simultaneously with the N-1 *Simple_Models*. Each *Simulation_Process* also integrates the current simplified version of its *Detailed_Model* in order to evaluate its validity after each time step. If an updated *Simple_Model* is required, it is generated by the *Detailed_Model* and sent by the *Simulation_Process* to the *Simulation_Manager*. Over the course of the integration, updated sets of *Simple_Models* are communicated among all *Simulation_Processes* by an instance of the *Simulation_Manager* using a CORBA-based protocol.

Table 5-3. Description of *Simulation_Process* Class

Class:	*Simulation_Process*
Inputs specified by user:	*Naming_service_IP* address specifying location of *Naming Service*
	Detailed_Model which is the locally defined detailed model of a particular subsystem.
Inputs specified by *Simulation_Manager*:	*Simple_Models* which are the current set of N-1 simplified subsystem models received from the *Simulation_Manager*.
	input_output_maps specify mappings that link each input variables of all detailed and simple models with the appropriate output variable of another detailed or simple models.
Key class methods:	*integrate(...)* function simultaneously integrates the *Detailed_Model* along with the N-1 *Simple_Models* over a specified time interval.
	update_local_simple_model(...) function called by *Simulation_Manager* to get updated *Simple_Model* version of *Detailed_Model*.
	update_simple_models(...) function used by *Simulation_Manager* to transmit new set of *Simple_Model* s of remote processes
	report(...) function gets report of current variable values for *Detailed_Model* and *Simple_Models*.
Key class variables:	*local_simple_model* represents the current simplified version of the local *Detailed_Model*, used to determine when an update is required.

The *Simulation_Manager* class is summarized in Table 5-4. The *Simulation_Manager* coordinates communication between the instances of *Simulation_Process*es running on separate computers. A single instance of *Simulation_Manager* is created for an entire distributed multi-system simulation. This "global" *Simulation_Manager* may reside on the same computer as a *Simulation_Process*, or it may reside independently on a dedicated computer. The *Simulation_Manager* serves as a central repository from which simulations are initiated and where histories of *Simple_Models* for each *Detailed_Model* are maintained. It also serves as a communication middleman between the individual *Simulation_Process*es. In this manner, each *Simulation_Process* does not need to be aware of and communicate with all other *Simulation_Process*es, but rather communicates exclusively with the global *Simulation_Manager*.

In the future, for remotely distributed simulations this architecture may be revised to allow multiple *Simulation_Managers*. In this scheme, a "local" *Simulation_Manager* would be implemented at each remote site in order to reduce the required network bandwidth for transmitting a set of *Simple_Models*. The global *Simulation_Manager* would only transmit the set of *Simple_Models* to the local *Simulation_Manager* at each remote site (rather than to every *Simulation_Process* at each remote site). The local *Simulation_Manager* would then forward the set of *Simple_Models* to each *Simulation_Process* at its site through an assumedly high-speed connection.

Table 5-4. Description of *Simulation_Manager* Class

Class:	*Simulation_Manager*
Inputs specified by user:	*Naming_service_IP* address specifying location of *Naming Service*
	Process_names identify textual names of each simulation process, used for requesting addressable object references from *Naming Service.*
	Input_output_maps specify mappings that link each input variables of all detailed models with the appropriate output variable of another detailed model.
Key class methods:	*initialize_simulation(...)* function requests a *Simple_Model* from each *Simulation_Process*, transmits these *Simple_Model*s among all other *Simulation_Process*es, and notifies all *Simulation_Process*es to prepare for integration.
	integrate(...) function calls all *Simulation_Process*es to integrate over a specified time interval
	report(...) function gets report of current variable values for all *Simulation_Processes.*
Key class variables:	*simple_model_history* maintains a history of *Simple_Models* for each *Detailed_Model* and the time intervals over which they are valid.

The classes of the distributed simulation software library provide a common framework that encapsulates multi-discipline subsystem models as modular, reusable components. These classes are designed to be minimally invasive during the modeling process, allowing engineers to develop new subsystem models or reuse legacy models with little additional overhead. They also streamline aspects of model development by providing routines for numerical integration, by automatically and dynamically creating data structures for storing variable values during simulation, and by reporting simulation results in a structured format. These classes may also be readily extended and modified through the object-oriented concept of inheritance. For example, during the course of this project, the additional capability for solving sets of purely algebraic equations (as opposed to mixed sets of differential and algebraic equations) using a superelement-based approach was implemented. At that point, the *Simple_Model* class was restructured into an abstract class with two subclasses, *Simple_ODE_Model* and *Simple_AE_Model*. This class hierarchy is illustrated in Figure 5-5.

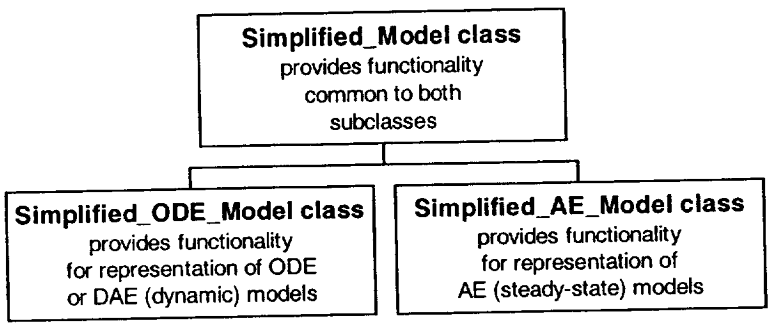
Simplified_Model class
provides functionality common to both subclasses
Simplified_ODE_Model class
provides functionality for representation of ODE or DAE (dynamic) models
Simplified_AE_Model class
provides functionality for representation of AE (steady-state) models

Figure 5-5. *Simple_Model* Class Hierarchy.

The *Simplified_ODE_Model* class provides the necessary functionality for approximating the behavior of models consisting of ODEs or DAEs. The *Simplified_AE_Model* class provides the necessary functionality for using superelements to approximate the behavior of models consisting of AEs. The abstract *Simple_Model* super-class provides the functionality common to both subclasses (e.g., handling of data arrays for model variables) which they access through the property of object-oriented *inheritance*. This eliminates the need for redundant code in the subclasses. In addition, data used exclusively by a particular subclass appears only in that class definition, thus minimizing the amount of data that must be transmitted when instances of a *Simple_Model* subclass are communicated over the network.

5.4 Selection of a CORBA ORB

The software library being developed in this project enables the distributed simulation of an overall model comprised of multiple subsystem models. The implemented scheme requires both the transmission of *Simple_Models* (i.e., data structures) among *Simulation_Process*es as well as the remote invocation of *Simulation_Process* procedures by the *Simulation_Manager*. Several technologies currently exist for implementing such networked communications, the most established of which are Common Object Request Broker Architecture (CORBA), Java Remote Method Invocation (RMI), and Microsoft .NET. Of these, CORBA was selected for this project because, unlike Java RMI, it is language independent and, unlike Microsoft .NET, it is platform independent. In summary, CORBA provides an open, vendor-independent architecture and protocol that enables a program running on almost any computer, operating system, programming language, and network to inter-operate with other CORBA-based programs executing on almost any other computer, operating system, programming language, and network.

Two CORBA Object Request Brokers (ORBs) were evaluated: omniORB (developed at AT&T Laboratories: http://www.uk.research.att.com/omniORB) and MICO (developed as an OpenSource project: http://www.mico.org/). These ORBs were selected because both are freely available, have been branded as CORBA compliant by the OpenGroup standardization organization, and support development on Unix, Linux, and Windows operating systems. While the omniORB version was used in the initial testing last year, the MICO ORB has subsequently proven to be much more robust and compliant. Note, however, that since CORBA is a standardized architecture, any other compliant ORB can also be used.

5.5 Implementation of CORBA Protocol in Software Library

CORBA enables the interoperability of programs across a network. The communication interface between such programs is defined using a declarative, programming language called the interface definition language (IDL). Consequently, the communication protocol between the *Simulation_Manager* and *Simulation_Process*es was defined using an IDL specification. An excerpt from this specification is shown in Table 5-5.

Table 5-5. Excerpt From IDL Specification for Software Library

```
module MOR {

   // ...ByteArray typedef representing ordered list of bytes (i.e., binary data)...
   typedef sequence<octet> ByteArray;

   // ...struct specifying params for generating Simple_Models
   struct ModelParams {
      double time;
      double update_interval;
      double contribution_threshold;
      ...
   };

   ...

   // ...stub class representing interface to Simulation_Process...
   interface Simulation_Process_Stub {

      // ...get name...
      string name_IDL();

      // ...initialize simulation, returning simple model key...
      long initialize_simulation_IDL(in ModelParams stub, out ByteArray model);

      // ...reinitialize state of process so can restart simulation…
      void reinitialize_process_IDL();

      // ...add simple model to list representing external models...
      void add_simple_model_IDL(in ByteArray bytes, in long keylen);

      // ...generate and return updated simple model of local detailed model...
      long generate_local_simple_model_IDL(in ModelParams params, out ByteArray model);
     // ...generate updated simple model and return using oneway callback...
      oneway void update_local_simple_model_oneway_IDL(in ModelParams params);

      // ...update simple models of remote models...
      void update_simple_models_IDL(in ByteArray model, in long size);


      // ...integrate models...
      double integrate_IDL(in double t, in double t2, in double deltaT);
      // ...integrate models using oneway callback...
      oneway void integrate_oneway_IDL(in double t, in double t2, in double deltaT);


      // ...save/restore state...
      void save_state_IDL(in double time);
      void restore_state_IDL(in double time);


      // ...report current values...
      void report_IDL(in double t, in double delta_t);


      // ...retrieve reporting file...
      long close_and_fetch_report_file_IDL(out ByteArray file);

...
   };
};
```

The complete IDL specification explicitly defines all allowable communications between the *Simulation_Manager* and the *Simulation_Processes* (represented in the IDL shown in Table 5-5 for instances of interface class *Simulation_Process_Stub*). The ORB IDL compiler then generates C++ "skeleton" code for compiling with the distributed simulation software library. The changes to the software library necessary to implement inter-process communications were thus limited to these two classes. Further, these changes only involved "interface" implementations, and no changes to the fundamental algorithms or logic were necessary. The *Simulation_Process* was defined as a subclass of the *Simulation_Process_Stub* class. The inherited IDL functions were then mapped to their preexisting counterparts with exception handling added to make networked communication more robust. For the *Simulation_Manager* class, all references to *Simulation_Processes* were replaced with references to *Simulation_Process_Stubs* and the associated function calls redirected to their IDL-derived counterparts.

As a result of this design, separate executable server programs are compiled for each model subsystem encapsulated in a *Simulation_Process*. The various executables can then be run in a distributed manner on any combination of networked computers. A separate executable client program is also compiled for the *Simulation_Manager* which links to the desired model subsystem servers at run-time.

5.6 WORKAROUND FOR CORBA SYNCHRONOUS MESSAGING

It should be noted that CORBA invocations (i.e., function calls) on remote objects are by default synchronous, in which execution is "blocked" until the remote process returns control. In other words, if a *Simulation_Manager* calls the *integrate_IDL* function of a *Simulation_Process*, the *Simulation_Manager* executable is blocked until the *Simulation_Process* executable returns control. Unfortunately, this default situation would prohibit parallel processing as all remote *Simulation_Process*es would integrate in a serial manner. To avoid this situation, a *oneway* invocation scheme was implemented for the *Simulation_Process* functions used for integration and generation of simple models.

When a oneway CORBA invocation is used, control returns immediately to the *Simulation_Manager* after invoking the *integrate_oneway_IDL* function of a *Simulation_Process*. This scheme allows the *Simulation_Manager* to start integration of the remaining *Simulation_Processes* in parallel before the first process finishes its integration. However, since oneway invocations cannot return a value (i.e., the time to which the *Simulation_Process* simulated to successfully), this information must be reported by the *Simulation_Process* using a separate CORBA invocation. In our implementation, the *Simulation_Manager* process spawns a separate *Model_Receiver* process on its local processor for receiving these invocations and data transmissions from the *Simulation_Process*es. The *Model_Receiver* then passes this data to the *Simulation_Manager* using an input pipe channel.

Although the oneway invocation method has been robust in our testing, in the future another proposed method of CORBA message passing may prove to be more desirable. The CORBA standards body (OMG) has specified an *asynchronous method invocation* protocol. Using this method, the *Simulation_Manager* would invoke non-blocking calls on each *Simulation_Process*, then check periodically to see whether a process has concluded. The advantage of this approach is that a separate application thread would not be required for the *Simulation_Manager* client process. As this is a relatively recently proposed protocol, few CORBA vendors have yet implemented it. Consequently, until it is more commonly available, we will continue to use the oneway invocation protocol.

6 CURRENT IMPLEMENTATION OF SOFTWARE LIBRARY

This section describes the current status of the distributed simulation library with respect to software implementation. In summary, the overall architecture and class structure, as well as the remote distributed simulation capabilities using CORBA, have been completed. Model order reduction algorithms have also been implemented for systems comprised of ODEs (based on the relative importance of states for calculating outputs), DAEs (based on eliminating algebraic equations and subsequently using techniques for ODEs), and AEs (based on superelements).

Since the underlying framework of the software library has been completed, future work will focus on enhancements to class functionality and additional algorithm development. This development will be focused in four areas:

1. ***Advanced and supplemental model-order reduction techniques.*** This development area will focus on extending the library of available MOR algorithms, especially with regard to MOR based on the time-dependence of each state, as described in Section 4 of this report.

2. ***Improved integration and error-control routines.*** This development area includes improved model integration routines and error control routines. We will consider more sophisticated integration algorithms (e.g., adaptive time-stepping based on error control, and handling of model discontiuities). We will also investigate error control strategies that dictate the updating frequency of *Simple_Models* (e.g., updating *Simple_Models* of highly nonlinear subsystems more frequently than those of relatively linear subsystems, and implementing a more efficient fallback scheme when *Simple_Model*s exceed error tolerances).

3. ***Supplemental analysis tools.*** This development area includes the use of third-party automatic differentiation software (e.g., ADIC) for automatic generation of code for calculating partial derivatives of model equations encoded in C++, and also the added capability for performing uncertainty analysis on the decoupled modeled subsystems.

4. ***Enhanced usability.*** This development area will focus on ease-of-use enhancements geared toward future end users of the distributed simulation library. Our focus will be to minimize the burden imposed on such a modeler through simplified object-oriented interfaces to external models and class wrappers for integrating existing legacy models.

The status of specific tasks for each of these areas are summarized in Table 6-1.

Table 6-1. Implementation Status of Software Library

Item	**Theory**	**Implementation**
Model-order reduction techniques		
MOR based on time-dependence of each state:		
1x1 blocks	C	F
2x2 blocks	C	F
NxN blocks	C	F
MOR based on relative importance of states for calculating outputs		
1x1 blocks	C (Note 1)	C
2x2 blocks	C (Note 1)	F
NxN blocks	C (Note 1)	F
Using balancing transformations	F	TBD
Segregation of pure integrals prior to diagonalization	C	P
Elimination of algebraic variables in DAEs	C	C
Use of super-elements to perform MOR on AEs	C	P (Note 2)
Improved integration		
Adaptive time step adjustment	C	F
"Fall-back" when simple models become invalid	C	F
Automatic initialization of ODEs and DAEs at t=0	C	F
Model discontinuities	F	TBD
Supplemental analysis tools		
Off-line Jacobian matrix generation using automatic differentiation software	N/A	F (Note 3)
Integrated uncertainty analysis	F	F (Note 3)
Enhanced usability		
Simplified interfaces to external models	N/A	F
Class wrappers for integrating existing models	N/A	P

Key:

C–complete
P–partially complete
F–not yet done but planned for future
TBD–no decision yet on whether to implement

Notes:

1. Assumes that inputs can be set to average values encountered during last time interval.
2. Currently implemented for models consisting entirely on AEs. Not yet implemented for DAEs in which some subsystem models consist solely of AEs.
3. Part of FY03 work scope

7 TEST MODEL DEVELOPMENT

To test the distributed simulation software library, it is necessary to have models of physical systems that are typical of those that will be developed by the intended users. Unfortunately, existing models appropriate for use in testing were difficult to find because they were not publicly available, were poorly documented, or in their present form were not easily partitioned into subsystems. As a result, we found it expedient to develop our own models to support testing.

Taken at face value, this experience suggests that existing code may not be readily amenable to implementation in a distributed fashion using the software library. We believe that such a conclusion is premature. Certainly, the major effort required to develop a good model is the formulation of the physical model in the first place. Thus, adapting an existing model to the new format required for distributed simulation should not prove onerous when this is done by personnel intimately familiar with the existing code.

The effort necessary to develop test models was larger than originally anticipated, but this effort has proved invaluable because the interactions between the simplified model algorithms and the physical models could be explored at a level impossible with closed or poorly-understood physical models derived from other sources. For example, the development of a tubular reactor model introduced the need for block diagonalization (since the Jacobian matrix of the linearized model was not diagonalizable due to degenerate eigenvalues). The development of the turboprop transport plane model led to the representation of complex eigenvalues in 2x2 complex conjugate form in order to eliminate the need for complex arithmetic. The development of the F-16 aircraft model motivated the need to remove pure integrals from the model (since the pure integral states which determine position from velocity led to 3x3 block diagonals).

Described below are five physics-based models that have been used to test the algorithms for simplified subsystem model development.

7.1 HEAT CONDUCTION MODEL

The first model to be developed to test the distributed simulation software was a simple heat conduction model. Based upon the one-dimensional unsteady heat conduction equation, an explicit finite-difference model of a conductive bar was created (Figure 7-1). The bar has been discretized into 90 nodes as shown. With 90 temperature nodes, this model has 90 states.

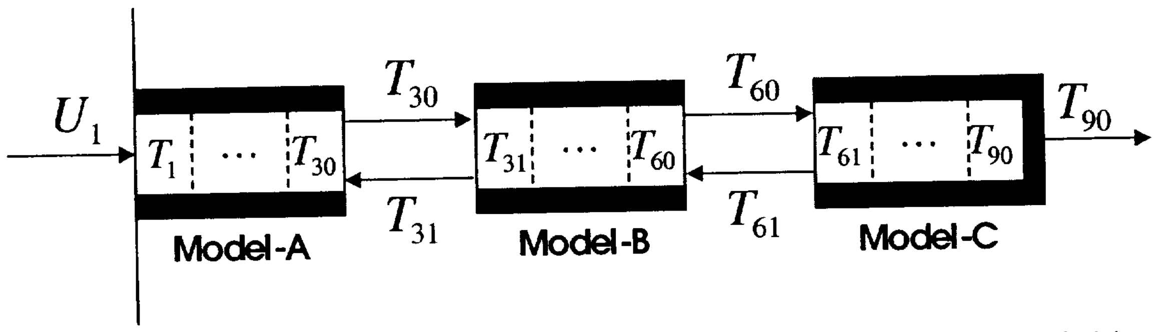

Figure 7-1. One-Dimensional 90-State Discretized Heated Bar Divided Into Three Sub Models.

As for boundary conditions, one end of the bar is adiabatic (zero heat flux) while the other end is subject to an oscillating temperature boundary condition with the following form

$$U_1 = 3 + 0.05\sin(t/10^3) \tag{7-1}$$

For implementation using the software library, the 90 nodes are grouped into three subsystem models with 30 nodes each. Critical to the definition is the definition of the inputs and outputs of each subsystem. For this system, the inputs and outputs for each subsystem are the bounding temperatures as shown in the table below.

Table 7-1. Inputs and Outputs of the Subsystem Models for the 1-D Unsteady Heat Conduction Model.

	Inputs	Outputs
Submodel A	U_1, T_{31}	T_{30}
Submodel B	T_{30}, T_{61}	T_{31},T_{60}
Submodel C	T_{60}	T_{61}, T_{90}

With the unsteady heat conduction problem formulated in this way, the system is governed by 90 coupled linear differential equations. An example of the results generated by the simplified implementation of this model is seen in the figure below.

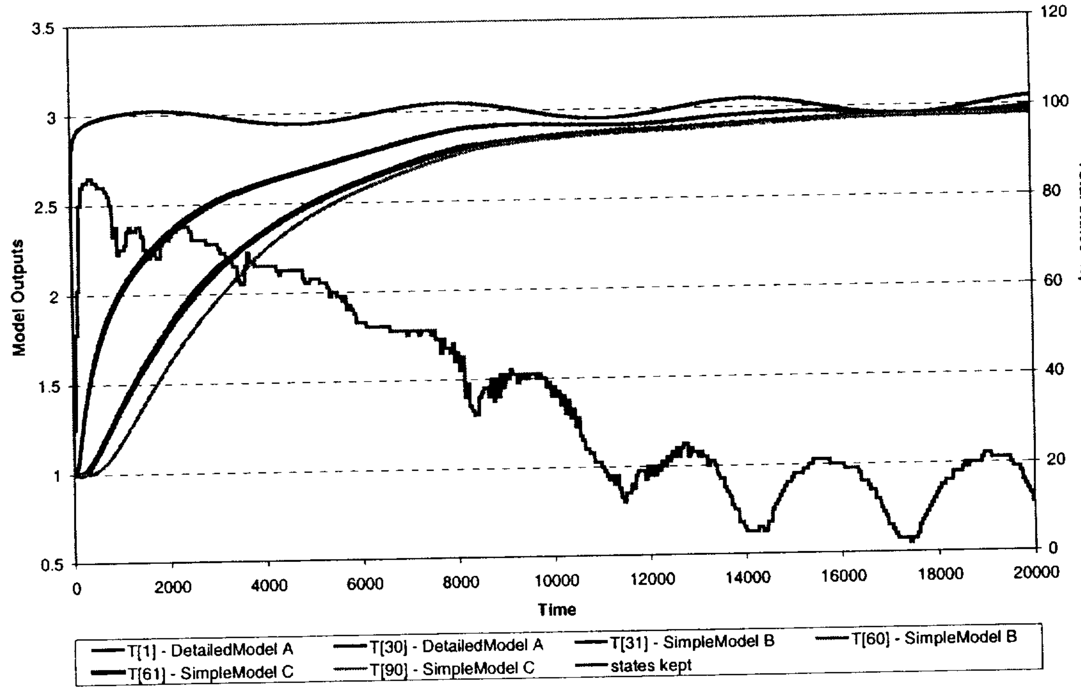

Figure 7-2. Results Generated by the Distributed Implementation of an Unsteady 1-D Thermal Conduction Model. Model uses explicit finite-difference technique with central differencing. The 90 node discretization is divided into three, 30-node subsystem models for computation. The software library algorithms are used to linearize, diagonalize, and then reduce the order of the subsystem models every 50 seconds.

In this test, the thermal diffusivity term is assumed to be equal to 1.0 and all the nodes are initialized to a temperature of 1.0. The equations are then integrated through time with the software library algorithms used to linearize, diagonalize, and reduce the order of the subsystem models after every 50 seconds of simulated time.

As can be seen in the figure, the proper unsteady heat conduction behavior is seen. Of particular interest, though, is the line showing that the number of total states kept by the model order reduction algorithms decrease as the model reaches steady state. This is the expected behavior since fewer states should be necessary to capture the small changes that occur as the system oscillates about steady state.

7.2 Turboprop Aircraft Model

Since linearization is an important component of the development of simplified subsystem models, testing these algorithms on a linear system of equations (see Heat Conduction Model above) is not entirely satisfactory. As a first step to testing the algorithms on nonlinear equations, a simple model for a turboprop transport aircraft was found in the literature (Lewis and Stevens, 1992). This model was extended and implemented here for use in testing the software library.

As shown in Figure 7-3, the overall turboprop aircraft model is composed of three subsystem models: (1) the aircraft body, (2) the engine, and (3) the control system.

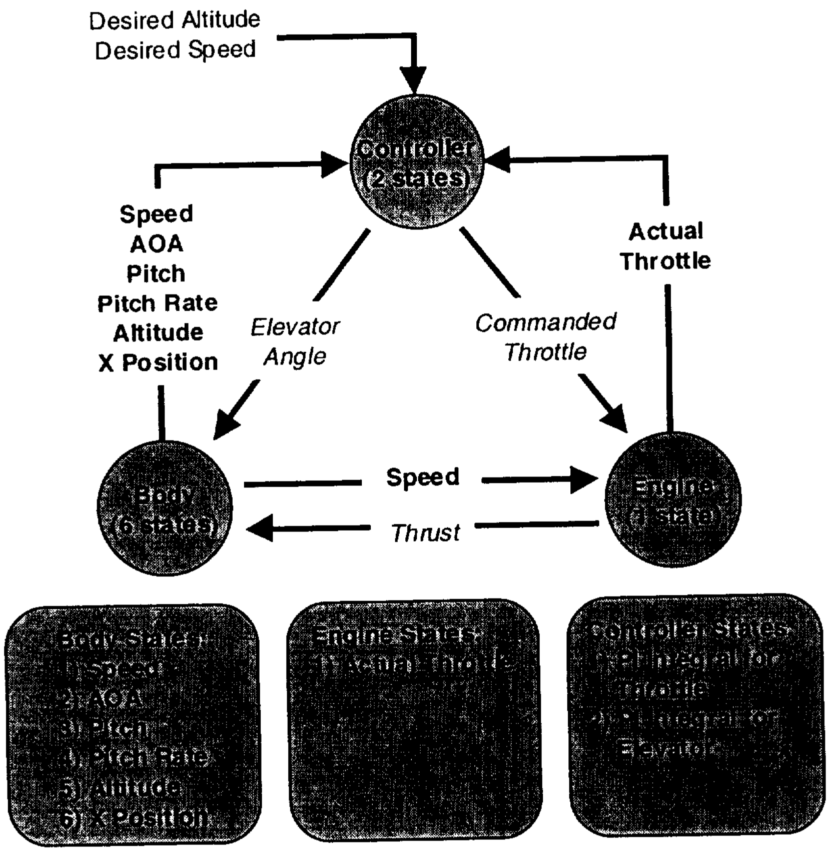

Figure 7-3. Model of Turboprop Transport Aircraft.

The aircraft body subsystem is a three-degree-of-freedom (DOF) model for the forward-backward, up-down, and pitching motions of the airplane. To compute the translational and rotational accelerations, the body model must know the angle of the elevator as commanded by the controller subsystem and it must know the engine thrust as computed by the engine subsystem.

The engine subsystem is a simple, 1 DOF model whose current power level (actual throttle) is a simple lag (time constant = 3 seconds) from the commanded power level as provided by the controller. The thrust produced by the engine is a function of the current power level and the speed of the aircraft. The speed dependence is included to model the behavior of propeller engines where thrust decreases with increasing airplane speed.

The controller subsystem attempts to keep the airplane flying in a stable, trimmed state at the desired speed and altitude. Given a change in the desired speed, the controller uses proportional and integral (PI) feedback on the error in the current speed to change the commanded power level of the engine. Given a change in the desired altitude, the controller uses PI control to change the elevator angle until the desired climb rate is achieved. Additionally, the elevator control law includes a term that is proportional to the pitch rate to maintain stability. To perform these tasks, the controller subsystem is provided with all the states of the airplane body and engine as inputs.

The overall aircraft model, therefore, contains nine states governed by nonlinear ordinary differential equations. The system is nonlinear because of the aerodynamic forces applied to the airplane body, the limits on elevator deflection, the limits on engine power, the limits on the throttle command, and the limits on the allowed climb-rate.

To demonstrate the behavior of the modeled aircraft, the full nonlinear model was subjected to a series of changes in desired speed and desired altitude. The model was initialized to trimmed flight at the desired speed of 400 ft/s (237 knots) and the desired altitude of 20,000 ft. After 10 seconds of simulation time, the speed setpoint of the controller was changed to 500 ft/s. After an additional 70 seconds of simulation time, the altitude setpoint of the controller was changed to 25,000 ft. The figure shows the resulting dynamic behavior of the complete model.

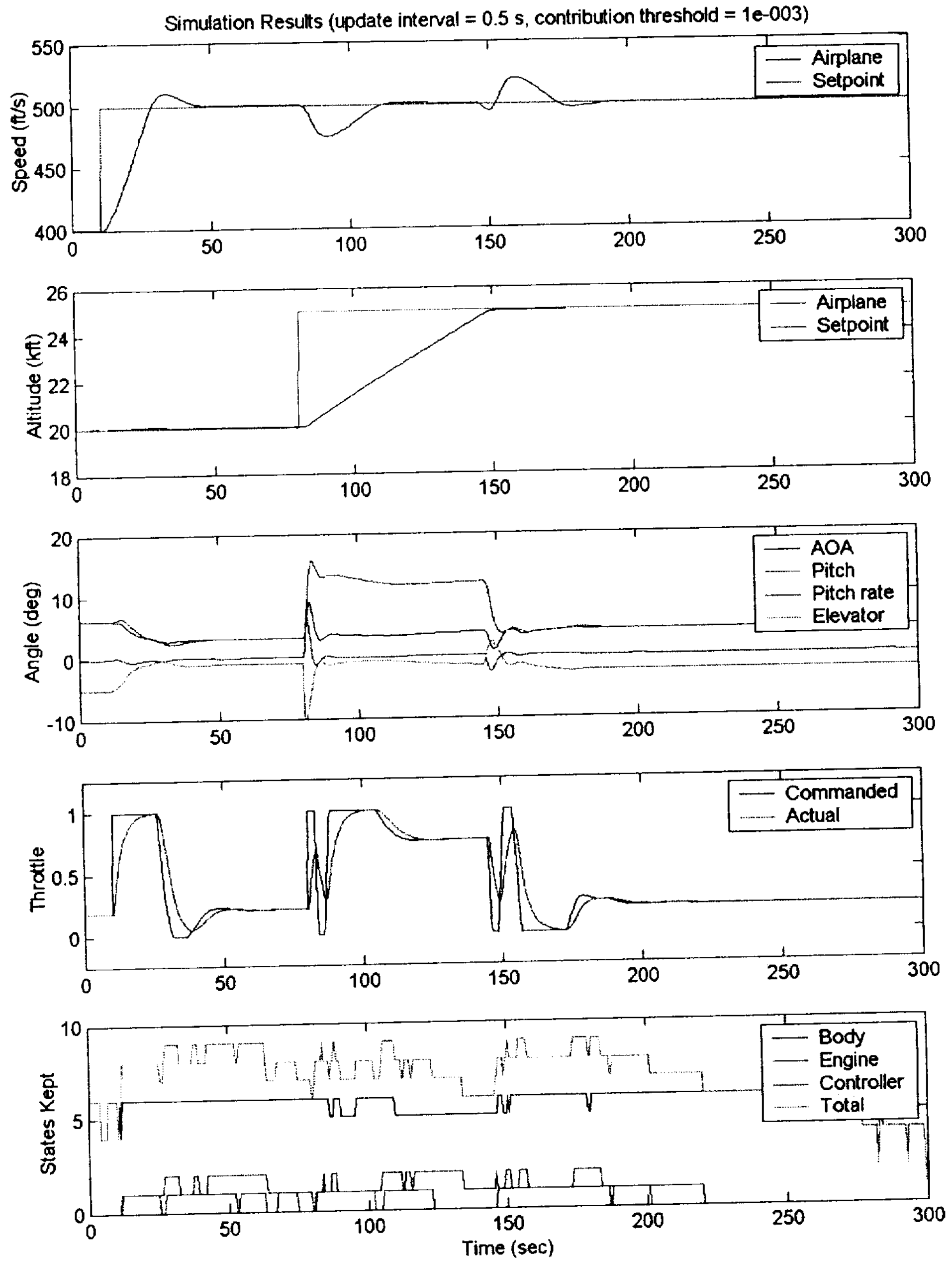

Figure 7-4. Simulation Results for Turboprop Transport Model Performing a Maneuver to Increase Speed and Then Increase Altitude.

To generate the results seen above, the simplified model algorithms implemented in the software library were used to linearize, diagonalize, and reduce the order of the subsystem models. The top plot shows the speed of the aircraft body compared with its specified controller setpoint. The second plot shows the altitude of the aircraft body compared with its specified controller setpoint. The third plot shows the angle of attack (AOA), pitch, pitch rate, and elevator position (specified by the controller) of the aircraft body. The fourth plot shows the throttle position of the engine compared with the position commanded by the controller. The bottom plot shows the total number of states retained for each subsystem. During slow transients, approximately 25–50% of the total states were eliminated.

Comparison of the above results to the results generated without linearization, diagonalization, or model order reduction shows that small errors were introduced by the processing steps. Specifically, error is introduced near model discontinuities such as at about t = 145 s where the commanded throttle is seen to rise slightly when, in fact, it should have jumped down to zero as the climb was finishing. Since any modeling technique that relies upon linearization cannot be expected to perfectly capture discontinuities, the small errors seen in the above results were not considered critical. It should be noted that future versions of the software library are expected to include a simulation manager that detects discontinuities and issues new models to account for them. In any event, the overall ability of the algorithms to generate a sufficiently accurate simplified model is considered excellent for this example.

7.3 F-16 Aircraft Model

To further test the capabilities of the distributed simulation algorithms, a more complex model with more states and more submodels was necessary. In addition to the turboprop transport aircraft model described earlier, Lewis and Stevens (1992) also provided a six-degree-of-freedom (DOF) model of an F-16 military fighter airplane. As before, this model was extended and implemented here for use in testing.

The overall model consists of six submodels (see figure below): a six-degree-of-freedom dynamic model for the airplane body, 1 DOF dynamic models for the engine, elevator, ailerons, and rudder, as well as a multi-input/multi-output model for the controller. Underlying these models are nonlinear, ordinary differential equations.

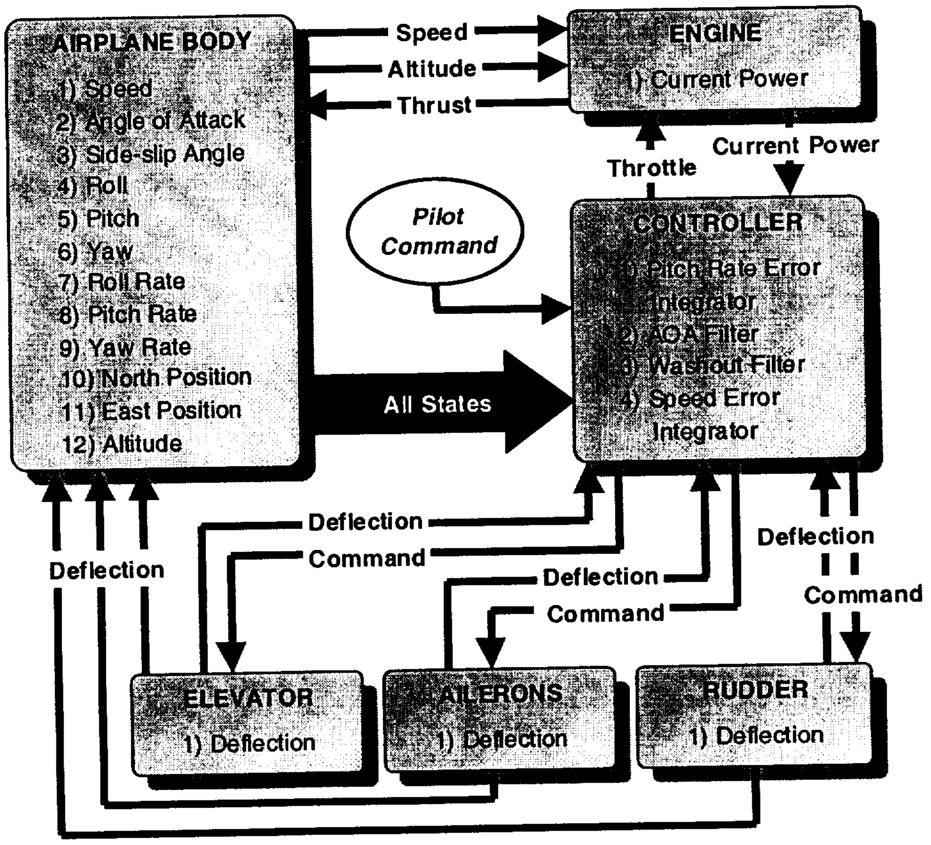

Figure 7-5. States, Inputs, and Outputs of the Components of the F-16 Model.

The body submodel is based on a standard formulation for the equations of motion for translation and rotation in three dimensions. The model includes lookup tables to evaluate all of the aerodynamic forces including lift, drag, and damping as well as to evaluate the forces generated by the control surfaces (elevator, ailerons, and rudder). The tables are based on wind tunnel data for an F-16 (Lewis and Stevens, 1992). Depending on the quantity being evaluated, these tables are angle-of-attack and side-slip angle dependent. The aerodynamic forces are then scaled by the dynamic air pressure, which is dependent on altitude and speed.

The engine submodel is a single-degree-of-freedom (exponential lag) dynamic model of a jet engine. The lag time-constant is power-level dependent and is evaluated by a table lookup. The thrust level generated by the engine is also computed by table lookup and it is based on the current power level, the aircraft's altitude, and the aircraft's speed relative to the local speed of sound. The thrust model spans the engine's power range from minimum idle, through military power, to maximum throttle with afterburner.

The aerodynamic control surfaces (the elevator, ailerons, and rudder) are each modeled as a single-degree-of-freedom, exponential lag between the actual surface deflection and the commanded surface deflection. From Stevens and Lewis, each of the three models has a time constant of 1/20.2 sec. The models for the control surfaces are not given maximum or minimum deflection limits.

Finally, a multi-input/multi-output controller was developed to keep the aircraft stable and to guide the aircraft through a number of maneuvers. The controller must provide commands for the engine and for the elevator, ailerons, and rudder. Throttle control is accomplished either by passing through pilot-commanded throttle level or, when modeling the level-flight autopilot, a proportional-integral feedback controller is used to achieve and maintain the desired speed. On the other hand, the elevator, ailerons, and rudder are feedback controlled to best achieve the angle rates requested by the pilot or by the autopilot.

To achieve the requested pitch rate, a proportional-integral (PI) feedback controller is used to issue the elevator command. Similarly, the desired roll rate is achieved using a proportional feedback controller to generate the aileron command. Finally, the rudder command is determined using the sum of the output from a PI feedback controller on yaw rate and from the output of a aileron-rudder-interconnect controller that attempts to maintain a stability-axis roll regardless of the current angle-of-attack.

For all three control surfaces, the commanded deflection angles are limited to the known deflection limits of the surfaces. When limited, the integral portion of any associated PI controller is bypassed to prevent controller windup.

All together, the system is composed of 20 nonlinear ordinary differential equations that govern the behavior of 20 states. For distributed simulation, the model is implemented as six subsystems with inputs and outputs shown in the previous figure. The software library was employed to linearize, diagonalize, and then reduce the order of the subsystem models.

To test the performance of the simplified subsystem models, a vertical "S" maneuver was simulated (see figure below). This complex maneuver requires a series of control surface and throttle commands that should be much more demanding to capture than the simple altitude and speed changes modeled for the transport plane.

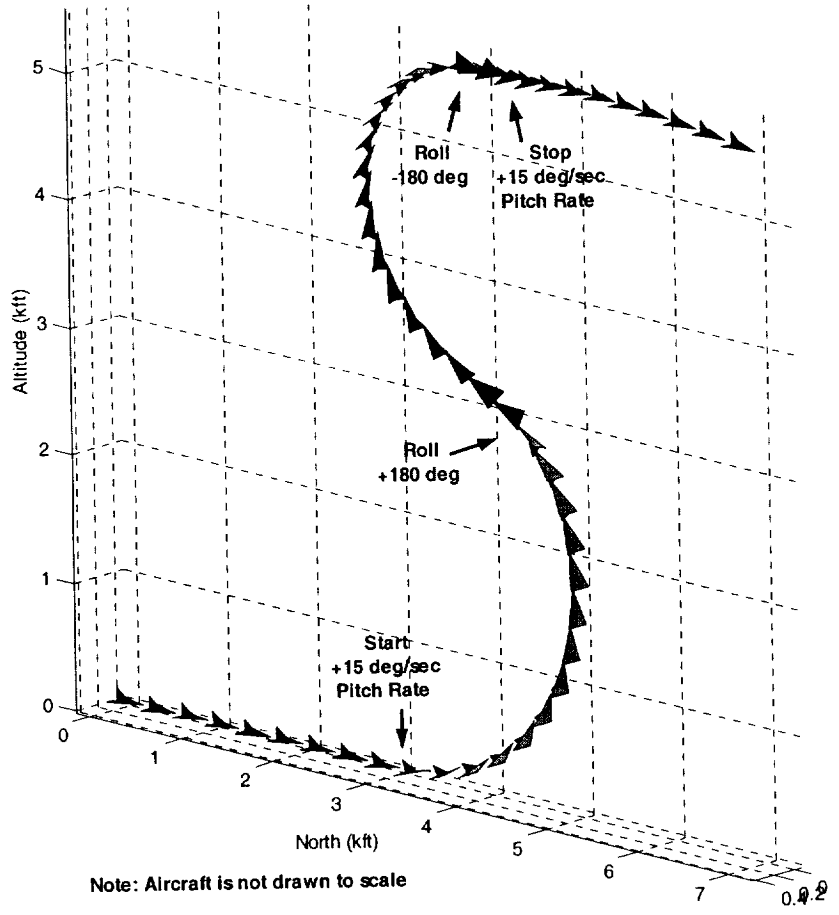

Figure 7-6. Vertical "S" Maneuver Performed by Modeled F-16.

For the full nonlinear simulation, the resulting time-dependent behaviors of the subsystem models during this maneuver are shown in the figure below. Notice that due to physical constraints on the controlled throttle position, sharp model discontinuities occur near 5 seconds and 35 seconds of simulation time.

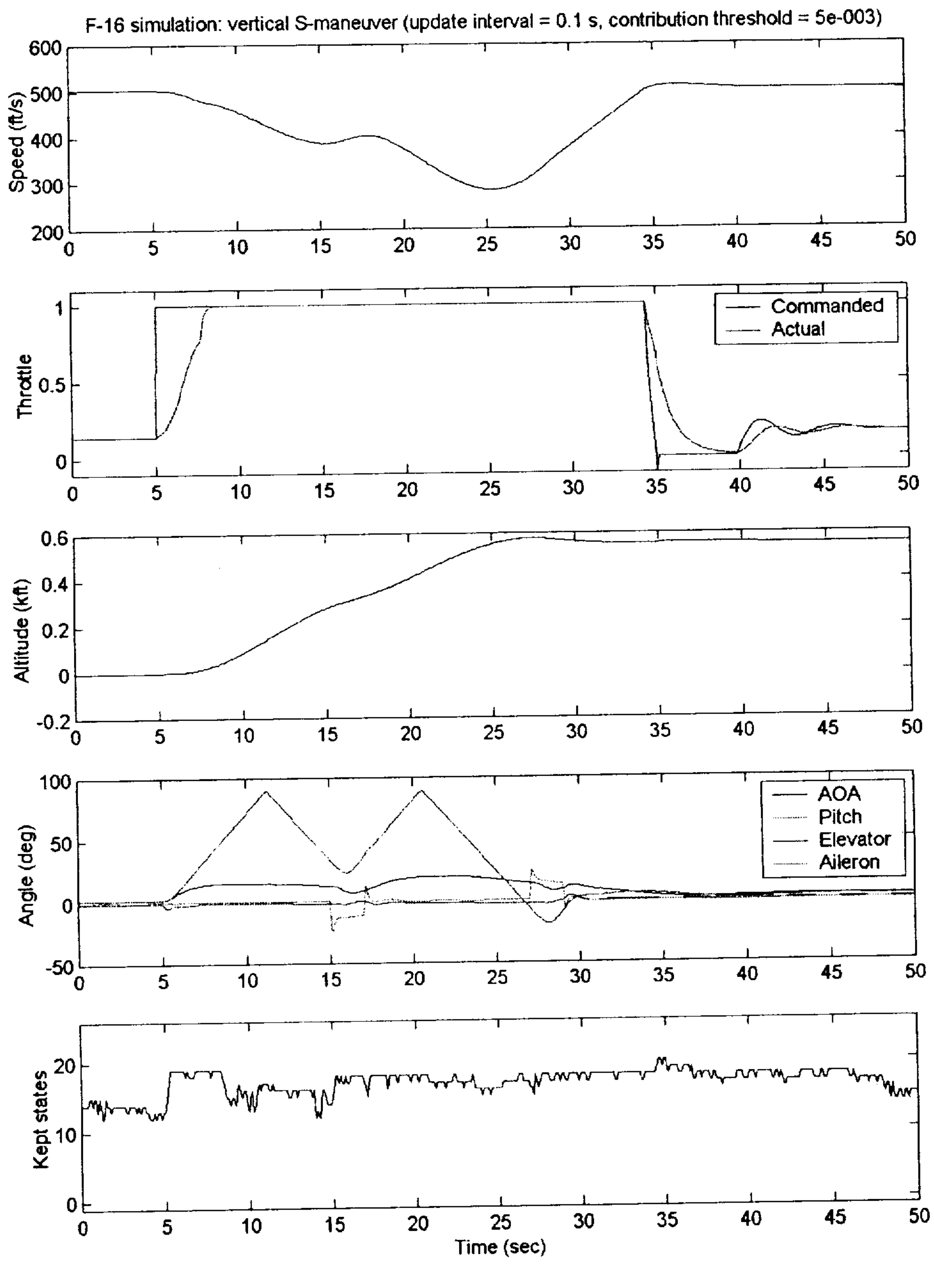

Figure 7-7. Simulation Results for F-16 Model Performing a Vertical "S" Manuever.

As before, the distributed simulation algorithms allow for successful integration of the model through time with only small errors introduced near model discontinuities. Over the entire simulation period, approximately 25–50% of the total states were eliminated. This is encouraging because it shows that a significant fraction of the states in a model can be eliminated even though each subsystem had a relatively small number of states. We expect that a larger fraction of the states can be eliminated as the number of states grows without compromising fidelity.

7.4 Chemical Plant Model

A simple chemical process model was also developed, primarily to test the capabilities of the library for handling coupled differential and algebraic equations (DAEs). This model was primarily chosen for convenience, since one of the engineers on the project was expert in this area. However, the numerical challenges posed by such models are judged to be similar to those that would be encountered in, for example, a spacecraft hydraulic system simulation, so the lessons learned are likely to be applicable to a variety of models.

The process model consists of a mixing tank, a tubular reactor, and a two-phase absorber. The mixing tank is modeled as a uniform volume, while reactor contents and the liquid and vapor phases of the absorber are spatially distributed along one dimension (i.e., their behaviors are modeled by partial differential equations). These partial differential equations are reduced to ODEs using finite control volumes.

The mathematical model for each control volume consists of a set of ODEs representing the conservation of mass for each chemical species and algebraic equations that related the mole fraction of each species to the molar holdup of each species in the control volume.

While this model has not been used extensively, the experience obtained with it was invaluable. In particular, this experience lead to the realization that the block diagonalization algorithm described in Section 2 was necessary. In addition, another version of the model was written that contained only DAEs. The exact agreement of the two versions' results verifies that the procedure used to eliminate algebraic equations described in Section 3 is correct.

7.5 Satellite Formation Flight Model

To provide a more difficult test of the software library, a model for the formation flight behavior of a cluster of satellites was developed. This model provides a very convenient test bed for studying the performance of the distributed simulation software library, since the overall complexity and the number of subsystem models can be adjusted essentially arbitrarily.

The motivation for this model is driven by the recent activities in the field of autonomous formation flight (Bauer et al., 1999; U.S. AFRL, 1998; NASA GSFC), specifically the U.S. Air Force Research Laboratory's effort on TechSat 21 microsatellite cluster program (see figure below) and the NASA Earth Observing One (EO-1) satellite's recent formation flight with the LANDSAT 7 satellite. The purpose of programs such as these is to use multiple satellites operating cooperatively and autonomously to increase mission performance, flexibility, and redundancy, while decreasing overall mission cost.

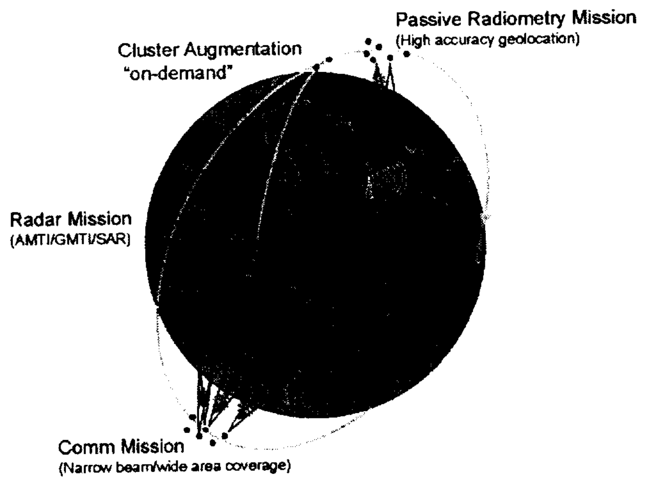
Passive Radiometry Mission
(High accuracy geolocation)
Cluster Augmentation
"on-demand"
Radar Mission
(AMTI/GMTI/SAR)
Comm Mission
(Narrow beam/wide area coverage)

Figure 7-8. Flexibility of Mission Available to Satellite Clusters as Envisioned by the AFRL TechSat 21 Effort (U.S. AFRL, 1998).

For many satellite clusters, such as the EO-1/LANDSAT pair and for space-based radar clusters envisioned as part of the TechSat 21 program, one of the principal technological challenges is to autonomously maintain the physical configuration of the satellite cluster. To make satellite clusters feasible, the individual satellites must be capable of formation flight using their own sensors and control systems. Operator workload and communication blackouts prevent controllers on the ground from performing this task.

If the ability to finely control cluster formation is achieved, a wide array of new missions becomes available by using the satellite cluster as if it were a single sensor with a very large aperture. Such large sensor apertures have the potential to greatly increase the resolution of terrestrial surveillance (for both military and scientific applications) as well as to enable previously impossible astronomical observations such as the search for gravity waves or earth-like planets.

7.5.1 Model Description

For this study, a model has been developed for the motion of earth-orbiting satellites and their ability to maintain formation flight. It is meant to be an example of the type of simulation that might be used by system engineers to gage the effect of various design parameters such as: (1) the type of station-keeping control algorithm being employed, (2) the type and performance of actuators required to maintain formation, and (3) the amount of power and fuel necessary to feed those actuators.

Specifically, the phenomena included and parameter values chosen are intended to model a satellite formation appropriate for radar imaging (see TechSat 21 figure earlier) using satellites of approximately the same size and configuration as the EO-1 earth-observation satellites.

Each satellite model is composed of a number of subsystems including a model for the satellite's rigid body motion, the earth's gravity, the satellite's atmospheric drag, the momentum wheels controlling the satellite's orientation, and the satellite's controller, which determines the proper thruster and orientation commands. The figure below shows how each of the subsytems are interconnected.

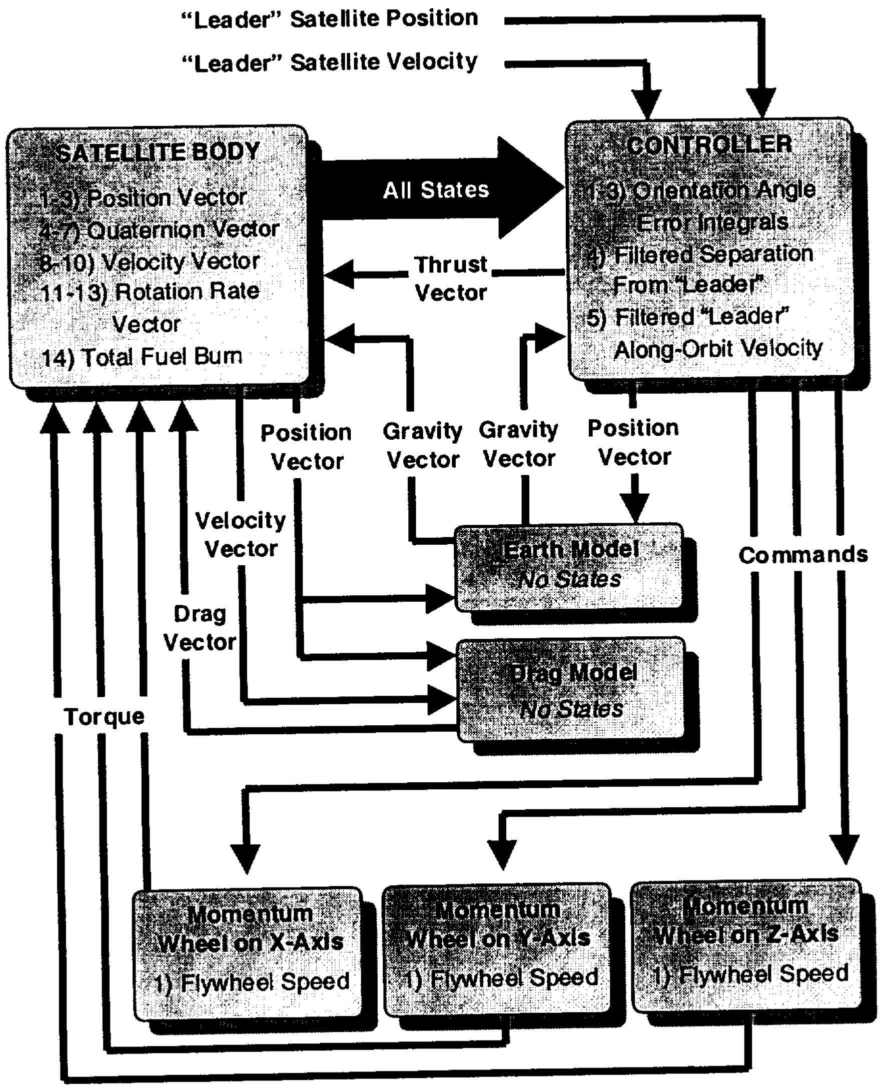

Figure 7-9. Subsystem Models and Their Inter-Connections for Each Satellite Model.

The rigid body motion of the satellite is modeled as a six-degree-of-freedom system based on Newton's laws of motion. Position, velocity, and acceleration are each represented by a three-component vector as measured in a Cartesian earth-centered earth-fixed coordinate system. Satellite orientation is tracked using a four-element quaternion representation. Rotation rates and rotation accelerations are represented using three element vectors as measured about the satellite's principal axes (see figure below). Linear motion accelerations are induced by the sum of forces due to the gravity of the earth, the atmospheric drag imparted by the satellite's velocity, and the thruster firings as commanded by the satellite's controller. Angular accelerations are induced by the sum of torques resulting from atmospheric drag and the acceleration of the momentum wheels as commanded by the satellite's controller.

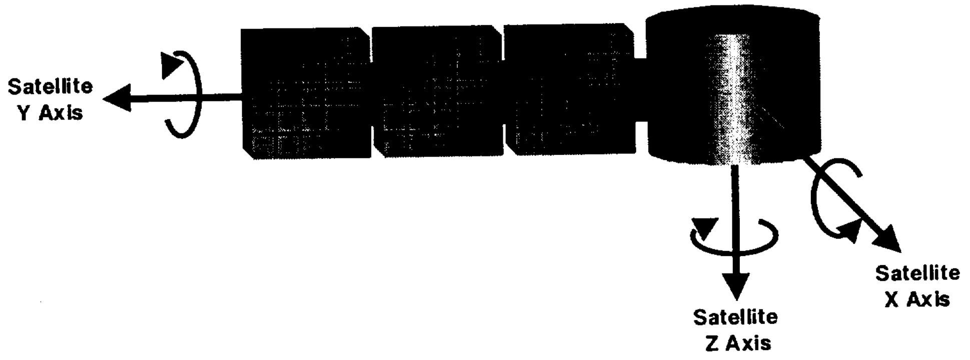

Figure 7-10. Coordinate System Attached to Satellite. Satellite controller attempts to maintain the satellite orientation such that the Z-axis is along the gravity vector and the X-axis is along the velocity vector.

The earth's gravity is modeled simply using Newton's law of gravitation, which can be expressed as

$$\vec{F}_{grav} = -\frac{GM_{earth} m_{satellite}}{|\vec{r}|^2} \frac{\vec{r}}{|\vec{r}|} \qquad (7\text{-}2)$$

where

G is the universal gravitation constant (N-m^2/kg^2),
M_{earth} is the mass of the earth (kg),
$m_{satellite}$ is the mass of the satellite (kg), and
$\vec{r}$ is the current position vector (m) of the satellite.

The atmospheric drag on the satellite's body is modeled using

$$\vec{F}_{drag} = -\tfrac{1}{2}\rho A_{proj} C_d |\vec{V}| \vec{V} \qquad (7\text{-}3)$$

where

ρ is the air density (kg/m^3) at the current altitude as modeled by the 1976 U. S. standard atmosphere,
A_{proj} is the projected area (m^2) of the satellite along the velocity vector,
C_d is the drag coefficient, and
$\vec{V}$ is the current velocity vector (m/s) of the satellite.

The drag force is applied to the satellite body at the centroid of the satellite's projected area. If the centroid of the projected area deviated from the satellite's center of mass (which will occur for asymmetrical satellite shapes), the drag force will induce a torque on the satellite and possibly cause rotational motion. For this model, the size and shape of the satellite are modeled after NASA's EO-1, which is asymmetrical due to its single solar wing (see figure above). Of course, at altitudes typical for EO-1, atmospheric drag should be very small so its effect on both linear and rotational motion should be minimal.

To control the orientation of the modeled satellite, it is assumed that momentum wheels are used (note that this is not the configuration used for EO-1, but it is being used here for this model). Each momentum wheel is composed of a motor attached to a flywheel which are both fit in a housing that is rigidly attached to the spacecraft structure. To generate a torque on the spacecraft, a signal is given to speed or slow the flywheel that is oriented along the proper satellite axis. Because the torque generated to speed or slow the flywheel generates a reaction torque on the satellite, the satellite orientation can be controlled.

The momentum wheel is modeled as a first order system where the satellite controller generates a command for a certain amount of torque. The motor in the momentum wheel is assumed to generate the requested torque instantly. The speed of the flywheel then responds as a first-order exponential lag as expressed by:

$$\frac{d\omega_{wheel}}{dt} = \frac{(\text{Spin Command})(C_{torque})}{J_{wheel}} \tag{7-4}$$

where

ω_{wheel} is the angular velocity (rad/s) of the flywheel in the momentum wheel,
(Spin Command) is the spin command generated by the controller that is a value between −1 and +1 representing the desired fraction of maximum torque that can be generated by the momentum wheel,
C_{torque} is the maximum torque (N-m) that the momentum wheel can generate, and
J_{wheel} is the rotational inertia (kg-m) of the flywheel in the momentum wheel.

The angular velocity of the flywheel is maintained as a system state to make sure that the wheels do not spin up to fast. In a real satellite, an over-spun wheel would require a maneuver to dump angular momentum from the wheel. Typically, a thruster firing is used to perform this task. This satellite model performs no such maneuvers because the duration of the simulations performed do not extend long enough for wheel speeds to get too high.

This model of the satellite uses one momentum wheel for each of its three principal axes. The momentum wheels are assumed to be frictionless.

The final subsystem model of the satellite is the satellite's controller. For this model, the controller has two purposes—generate momentum wheel commands to control the satellite's orientation, and generate thruster commands to control the satellite's position within the satellite formation. An overview of the controller's inputs and outputs are shown in the figure below.

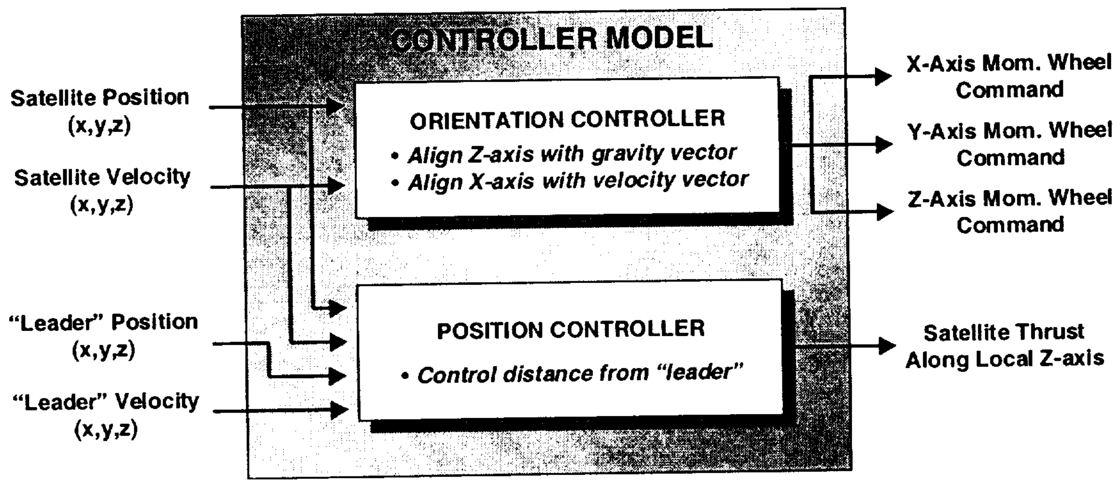
CONTROLLER MODEL
Satellite Position (x,y,z)
Satellite Velocity (x,y,z)
"Leader" Position (x,y,z)
"Leader" Velocity (x,y,z)
ORIENTATION CONTROLLER
• Align Z-axis with gravity vector
• Align X-axis with velocity vector
POSITION CONTROLLER
• Control distance from "leader"
X-Axis Mom. Wheel Command
Y-Axis Mom. Wheel Command
Z-Axis Mom. Wheel Command
Satellite Thrust Along Local Z-axis

Figure 7-11. Model of Satellite Controller.

Momentum wheel commands are currently generated to maintain the orientation of the satellite such that the satellite's Z-axis is always pointing along the gravity vector and such that the satellite's X-axis is always pointing along the portion of the satellite's velocity vector that is perpendicular to the gravity vector. Proportional Integral Derivative (PID) control is used to generate momentum wheel commands along the proper axes to minimize the angular error between the actual location of the principal axes and the desired location of the principal axes. The integral portion of the orientation controller requires one system state for each wheel. The orientation controller, therefore, contributes three states to the overall system model.

Thruster commands are generated to maintain the satellite's position within the formation. The formation control laws are based on a leader-follower structure where a given satellite is trying to position itself relative to its "leader." Of course, the satellite's "leader" might be also be trying to position itself relative to its own "leader," which yields a cascaded leader-follower system.

For this model, the formation is arranged where the satellites are grouped into subgroups (see figure below). Within a subgroup, the satellites are each controlling its along-orbit position relative to the subgroup leader. Each subgroup leader is, in turn, controlling its along-orbit position relative to the subgroup leader in front of it. The overall leader of the whole formation is not controlling its position at all—it is just falling inertially.

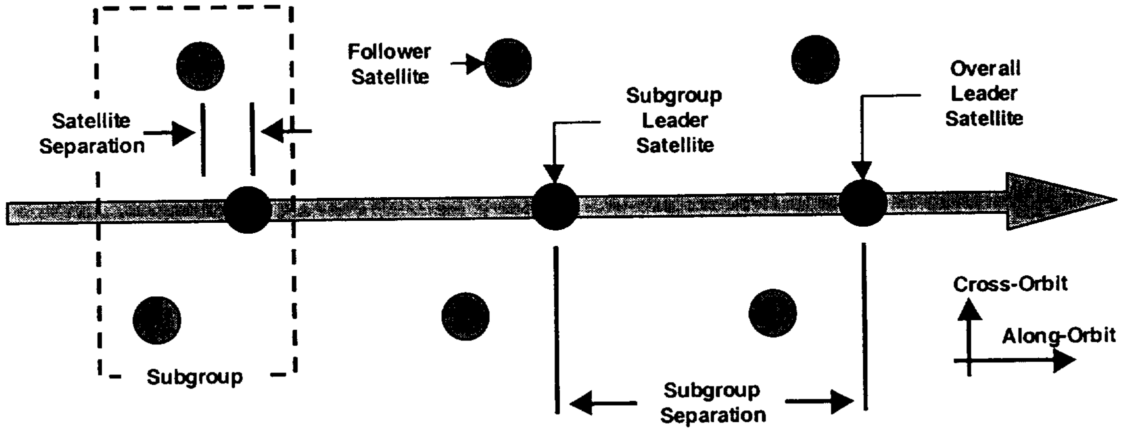

Figure 7-12. Leader-Follower Satellite Formation Where Each Satellite Follows the Leader of its Subgroup. Each subgroup leader follows the subgroup leader in front of it. The overall leader falls inertially.

For all satellites, a thruster is used to control the separation of the satellite relative to its "leader" as to minimize the error between the actual separation and the desired separation. To maximize orbital stability, though, the thruster is *not* fired along the satellite's velocity vector to increase or decrease its speed. Instead, the thruster is fired along the gravity vector as seen by the satellite (usually perpendicular to the velocity vector) to increase or decrease the satellite's altitude. When the satellite's altitude decreases, for example, its velocity relative to its leader should increase. This change in relative speed allows the satellite to catch up to or fall away from its leader. In this way, the separation in controlled.

With this control scheme defined, implementation is fairly direct. For a given difference between the actual satellite separation and the desired separation, a Proportional Derivative (PD) control expression is used to calculate the desired satellite velocity. Based on the orbital trade-off between kinetic and potential energy, this desired velocity corresponds to a new desired altitude. PD control is then used to issue thruster commands to control the satellite's altitude to achieve the desired altitude.

Note that as the error in satellite separation shrinks, the desired closing velocity will also shrink. A smaller closing velocity requires less of a deviation of the desired altitude from the nominal altitude, which causes the controller to fire the thrusters to move the satellite back to its nominal altitude. Finally, as the satellite reaches the desired separation, the desired closing velocity will become zero and the satellite will end up at the same altitude (and, therefore, the same velocity) as the leader satellite. Steady state is achieved.

So, this controller, with its three states for the error integrals for the orientation controller and its two states for filtering (smoothing) in the separation controller, contributes five states to the overall satellite model. With the controller's five states, the three momentum wheels with one state each, the satellite body with 13 states, and with an integral state computing total fuel burn, the satellite model as a whole has 22 states. These states are governed by 22 coupled ordinary differential equations—many of which are nonlinear. For the model of the entire satellite formation, an arbitrarily large number of states and equations can be included simply by increasing the number of satellites in the formation.

7.5.2 Satellite Formation

For a formation to act as a single large radar aperture, the satellites must be distributed both in the along-orbit direction and in the cross-orbit direction. Ideally, they should be distributed and move together to form some type of stable, regular grid pattern—like a school band marching in a parade. Unfortunately, a stable grid formation is not possible for a formation of satellites because it is impossible to maintain a fixed cross-orbit separation without enormous amounts of thrust (and fuel). Looking at the left-hand plot of the figure below, note that the top-most and bottom-most orbits maintain a constant cross-orbit separation relative to the center orbit. Because the top and bottom orbits are not inertially correct (e.g., they do not follow a Great Circle route), an enormous amount of thrust is required to fight gravity throughout the orbit. This is not feasible.

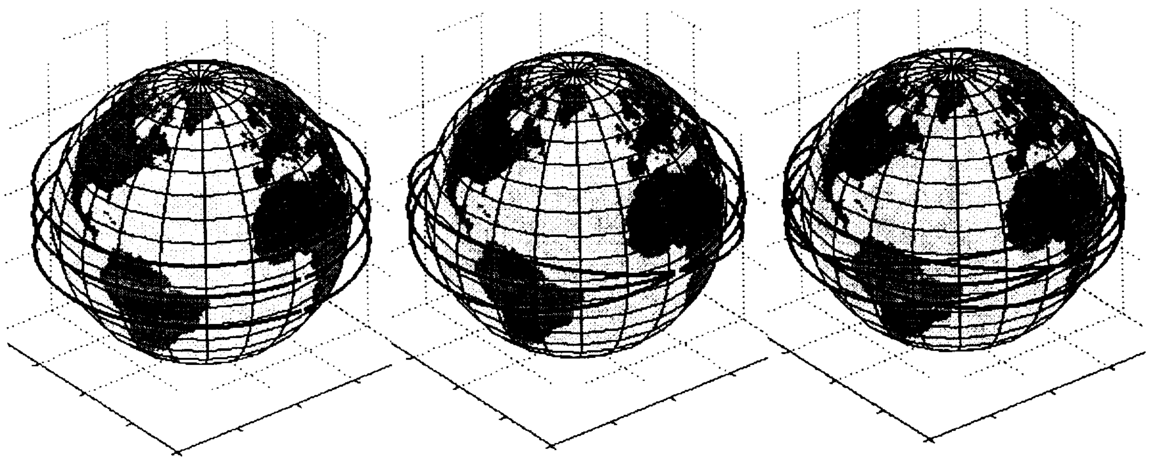

Figure 7-13. Examples of Non-Inertial (left) and Inertial (center and right) Orbits for Achieving Cross-Orbit Satellite Separation. The non-inertial orbits (left) fight gravity thereby requiring an unreasonable amount of thruster power and fuel. Inertial orbits require far less fuel but sacrifice some cross-orbit coverage (center). Consistent cross-orbit coverage can be achieved by adding satellites and by carefully choosing the longitude of each orbit's ascending node.

Instead, a new formation has been chosen that allows each satellite to follow an inertially appropriate orbit (meaning a Great Circle route) while achieving cross-orbit separation (see the center plot of the figure above). The cross-orbit separation is achieved by slightly offsetting one satellite's orbital inclination relative to the others. As the above center plot shows, this offset in inclination allows cross-orbit separation to be achieved throughout much of the orbit. By using more satellites and by properly choosing the longitude of their ascending node (where they cross the equator), a fairly constant amount of cross-orbit coverage can be maintained (right-hand plot).

In setting up and controlling the formation for this simulation, the desired inclination angle for the formation as a whole must be decided. To mimic the EO-1 and other earth-observing satellites, an orbital inclination of 98 degrees is chosen (see figure below). Furthermore, the orbit is assumed to be circular with an altitude of 750 km (again following from the EO-1 example). For the cascaded leader-follower control that has been implemented here for the satellite controller, each subgroup leader (including the overall formation leader) is set to follow this same orbit with each subgroup leader following the previous subgroup leader by the proscribed subgroup separation distance.

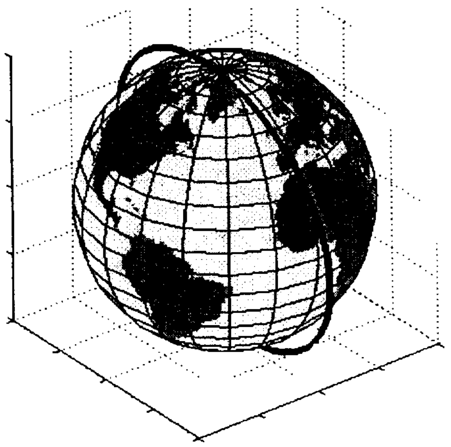

Figure 7-14. Nominal Orbit Used for Simulations (as plotted over a stationary earth) – Circular With an Altitude of 750 km and an Inclination Angle of 98 deg.

Within a subgroup, each follower is given the same orbit except the inclination angle is adjusted to yield the desired maximum cross-orbit separation. For these simulations, the maximum cross-orbit separation is taken to be ±250 meters relative to the subgroup leader's position. This amount of cross-orbit separation yields an exceptionally small 1.01 x 10^{-3} degree deviation from the nominal 98 degree orbital inclination. Within the subgroup, the satellites are also set to be separated from one another in the along-orbit direction by the proscribed satellite separation distance.

The result is that, relative to the overall leader satellite, the satellite formation is distributed both in the along-orbit and cross-orbit directions as shown previously in Figure 7-13. The cross-orbit position of the follower satellites is changing throughout the orbit, but cross-orbit coverage is maintained. Using more satellites allows for an extended and dense virtual aperture to be created by this method.

7.5.3 Simulation Results

With the model and the satellite formation defined, the motion of the formation and the reaction of the formation to changes in commands can be simulated. One typical command that might be given to the formation is to change its configuration in response to mission requirements (see TechSat 21 above). For an imaging satellite formation such as this one, it might be desired to pull the satellites closer (decrease their separation) to widen the formation's radar beamwidth so that it can cover more ground in a single look.

Specifically, a simulation was run with a formation of 30 satellites (grouped into six subgroups). The initial subgroup separation was 200 meters and the initial satellite separation was 25 meters. The new configuration requires a subgroup separation of 100 meters and a satellite separation of 10 meters. The results of just the subgroup leader satellites are presented in the figure below.

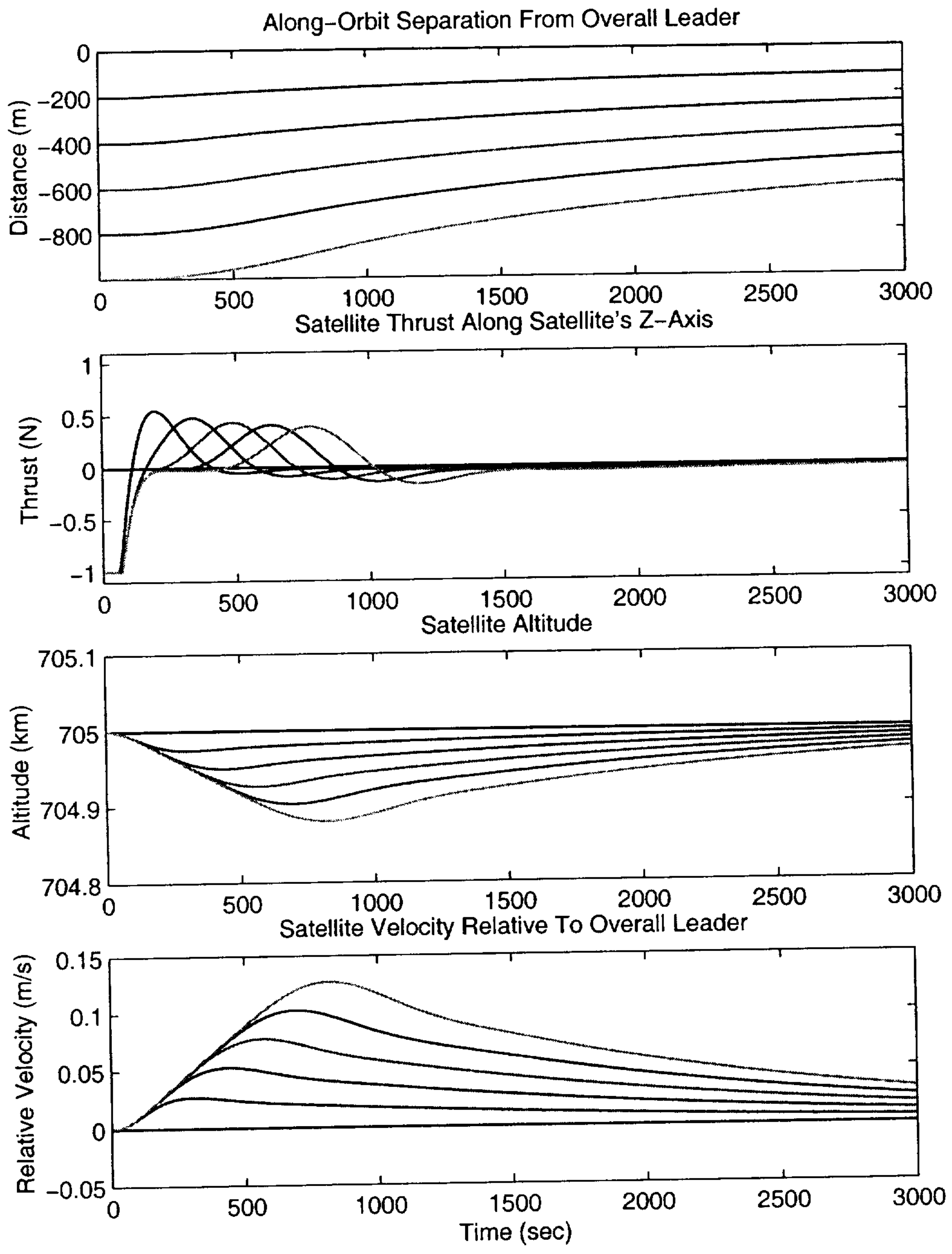

Figure 7-15. Simulation Results of Maneuver to Shrink the Spacing Between Subgroup Leader Satellites From 200 m to 100 m. Only subgroup leaders are plotted.

In this figure, note that the initial separation between subgroup leaders is seen to be the given value of 200 meters. Looking at the plot of thrust, it is seen that the thruster immediately fires up to its maximum thrust to try to reduce this separation to 100 meters. The thruster burn acts to lower the altitude of each of the satellites. The reduction in altitude increases the relative velocity of the satellites. This increased velocity then acts to reduce the separation between the satellites. The simulation results confirm the behavior expected during the design of the controller. Similarly, note that as the desired separation is approached, the thruster firings have begun to bring the satellite's altitude back to its original value and the relative velocity has begun to return to zero. Again, the expected behavior is seen.

An interesting point to note is how the plots differ for each satellite. For example, note that the satellite farthest back (starting at a separation of 1000 m relative to the overall leader) must end up about 500 meters behind the overall leader once the repositioning maneuver is fully complete. It has the most distance to make up. Because of this need, note how the satellite dips to the lowest altitude, stays at a low altitude for the longest time, and achieves the highest relative velocity. To an orbital engineer, this maneuver performed by the last satellite might be too extreme (or not extreme enough). Using this simulation, the engineer could try different formation control laws until the desired performance is achieved.

Another interesting point is seen when looking at the plot below, which shows the data for all 30 satellites in the simulation. In this plot, the four follower satellites in each subgroup are seen closely following their leader. No instabilities arise due to their closeness. In fact, perhaps because of their closeness, note that the followers use much less thrust than the leaders. For an engineer looking to minimize fuel usage, exploring this finding would be critical. Finally, with respect to the performance of the algorithms, note that these results were achieved by keeping only 220–250 of the 660 (30 satellites times 22 states each) total states for this simulation. Combined with the benefits achieved by diagonalization, this represents a substantial reduction in model complexity.

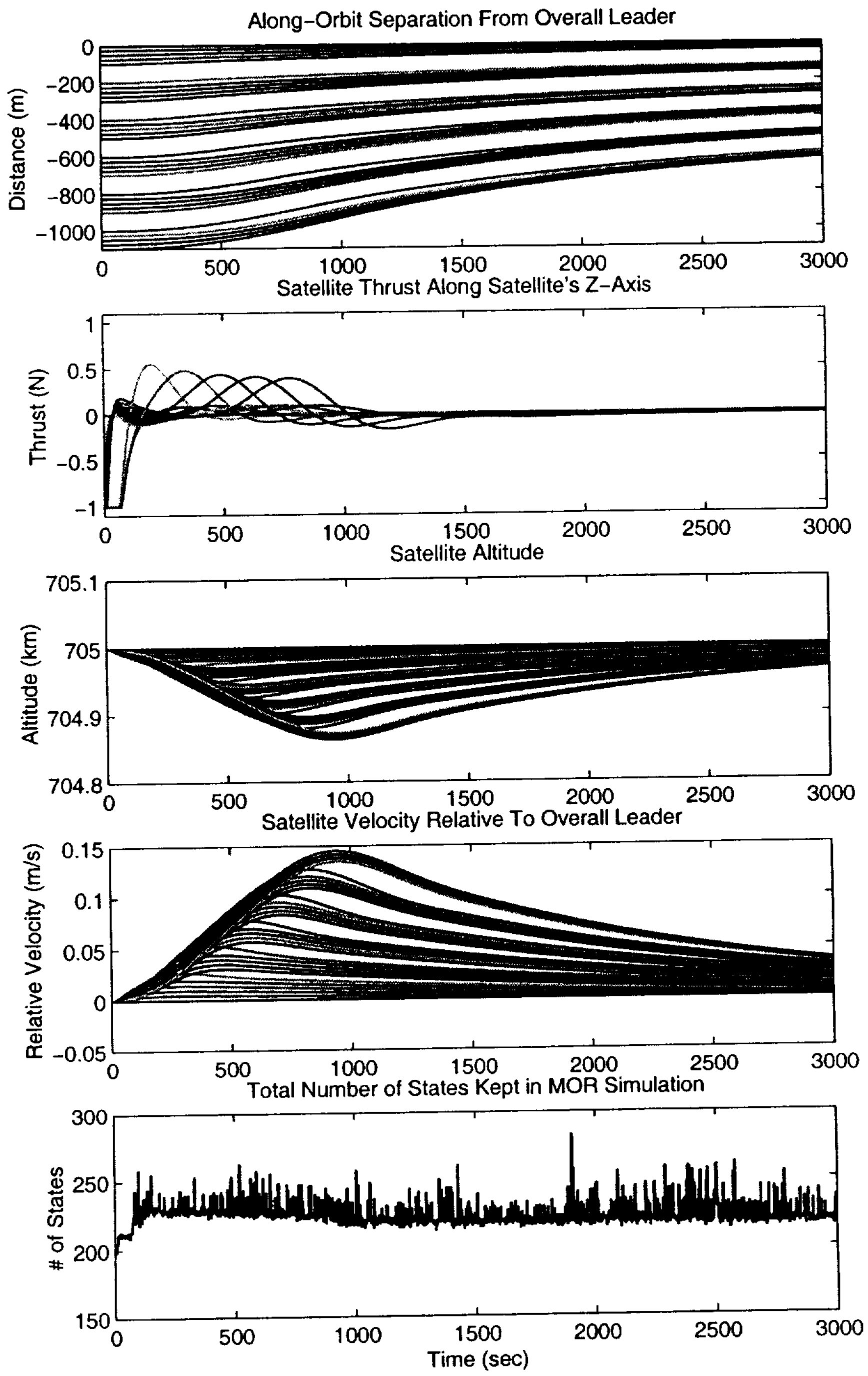

Figure 7-16. Simulation Results of Maneuver to Shrink the Spacing Between Subgroup Leader Satellites From 200 m to 100 m and to Shrink the Spacing Between Satellites Within a Subgroup From 25 m to 10 m.

8 TESTS OF SOFTWARE LIBRARY IMPLEMENTATION

During the past year of this project, the distributed simulation library has been tested extensively using the physics-based models described in the preceding section. Prior progress reports have described the application of the software library to these models, where testing focused on evaluating the performance of the model-order reduction algorithms. In these tests, the effect of varying the number of states retained in a simple model and varying the updating frequency for the simple models were evaluated. Specifically, these results were described in detail in monthly progress report 5 (dated November 28, 2001) for the heat conduction model, monthly progress report 8 (dated February 28, 2002) for the turboprop transport plane model, and monthly progress report 10 (dated April 29, 2002) for the F-16 aircraft model. Summaries of these results are also provided in Section 6 of this report.

During the latter part of the year, we also performed numerous tests of the CORBA-based distributed simulation capabilities of the software library. Various computer configurations were tested, both with multiple subsystem processes running on a single computer and with subsystem processes dispersed across an internal network. In addition, the robustness of the algorithms was evaluated by imposing artificial communication delays during the simulations.

The efforts of this past year culminated with a full-scale test using subsystem processes distributed across the Internet on geographically dispersed computers with real communication delays. These tests successfully demonstrated the utility of the software in running distributed simulations across the Internet, and also showed the robustness of the system with respect to networking communication delays. Details of these tests are described in the remainder of this section.

8.1 DEMONSTRATION OF SOFTWARE LIBRARY FOR EXAMPLE CONFIGURATIONS (F-16 AND SIX SATELLITE FORMATION) ON A SINGLE COMPUTER

The performance of the distributed simulation software library was first tested on two relatively simple example models, the F-16 aircraft model and a six satellite formation model. In these initial demonstrations, we demonstrated our approach with multiple processes, each representing a subsystem, executing on the same computer.

In order to examine the effects of implementing the models in a distributed fashion on multiple computers, simulations were run both with and without artificially simulated communication delays. These were imposed as a random delay of up to one second in the response of the *Simulation_Process* server process whenever a CORBA invocation request was made by the *Simulation_Manager* to the *integrate_oneway_IDL(...)* or the *update_local_simple_model_oneway_IDL(...)* functions. A different sequence of random pauses was generated for each *Simulation_Process* by seeding its pseudo-random number generator using the local clock millisecond time when the first call to it was made.

The configuration used for the F-16 model is illustrated in Figure 8-1. In this test, seven separate executable processes were run on the same computer: six *Simulation_Processes* (one for each model subsystem: body, engine, controller, elevator, ailerons, and rudder) and one *Simulation_Manager*. The CORBA *Naming Service* process was also run on this computer. The single computer tests were repeated on two separate computers, both with and without communication delays. Test results are summarized in Table 8-1. For all cases, the simulation results were not affected by communication delays as they were exactly identical to all bits of machine precision (results are illustrated in Figure 8-2). However, as expected, communication delays as well as processor speed both had a large impact on the required execution time.

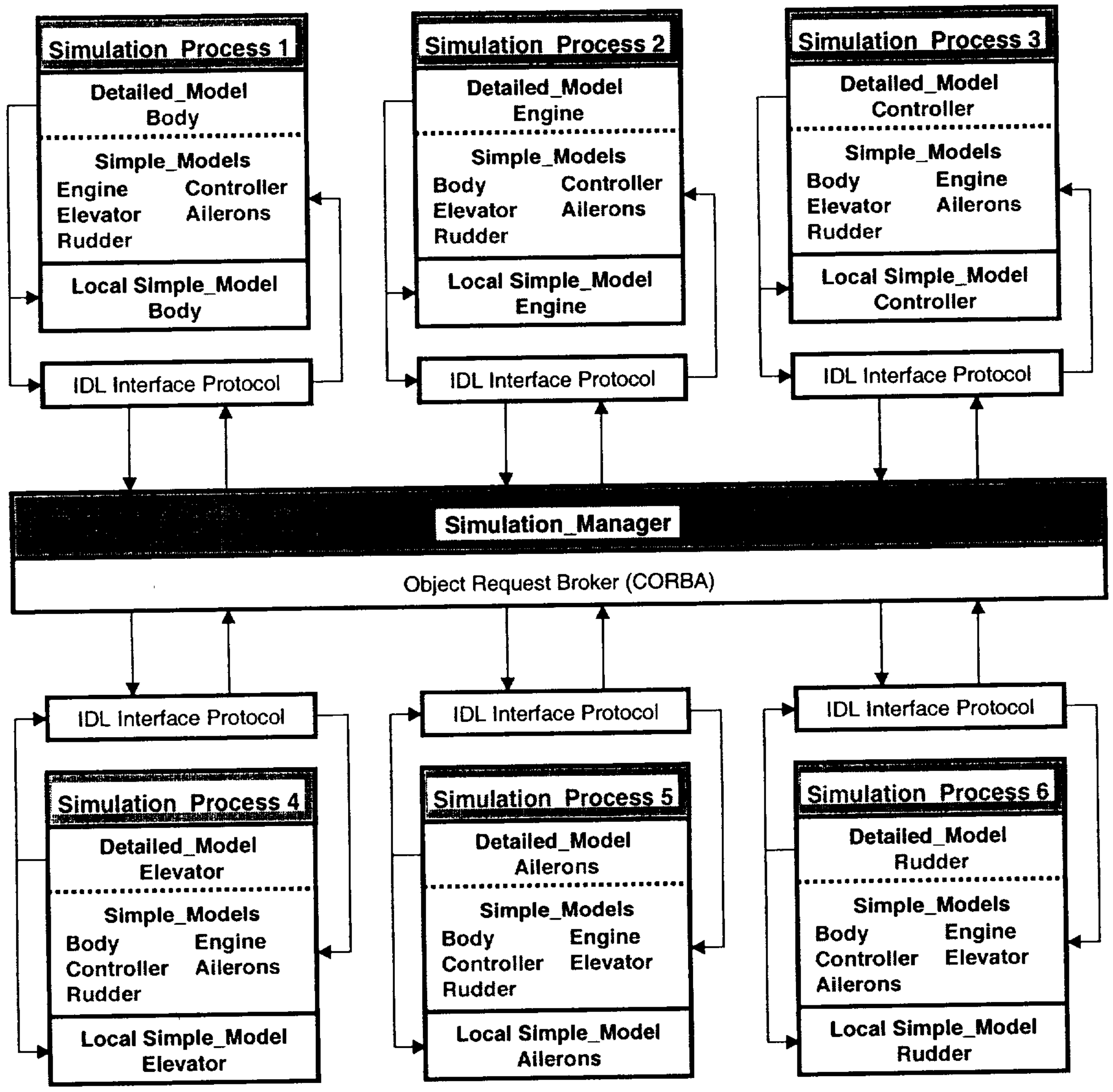
Simulation Process 1
Detailed_Model Body
Simple_Models Engine Controller Elevator Ailerons Rudder
Local Simple_Model Body
IDL Interface Protocol
Simulation Process 2
Detailed_Model Engine
Simple_Models Body Controller Elevator Ailerons Rudder
Local Simple_Model Engine
IDL Interface Protocol
Simulation Process 3
Detailed_Model Controller
Simple_Models Body Engine Elevator Ailerons Rudder
Local Simple_Model Controller
IDL Interface Protocol
Simulation_Manager
Object Request Broker (CORBA)
IDL Interface Protocol
Simulation Process 4
Detailed_Model Elevator
Simple_Models Body Engine Controller Ailerons Rudder
Local Simple_Model Elevator
IDL Interface Protocol
Simulation Process 5
Detailed_Model Ailerons
Simple_Models Body Engine Controller Elevator Rudder
Local Simple_Model Ailerons
IDL Interface Protocol
Simulation Process 6
Detailed_Model Rudder
Simple_Models Body Engine Controller Elevator Ailerons
Local Simple_Model Rudder

Figure 8-1. Configuration for F-16 Multiple Process Simulation.

Table 8-1. Test Results for F-16 With Multiple Processes on Single Computer

Computer	Processor	RAM	Execution time without delays	Execution time with delays
jyb.creare.com	Pentium III 1.1 GHz	512 MB	60 sec	1000 sec
test1.creare.com	Pentium Pro 200 MHz	128 MB	290 sec	1100 sec

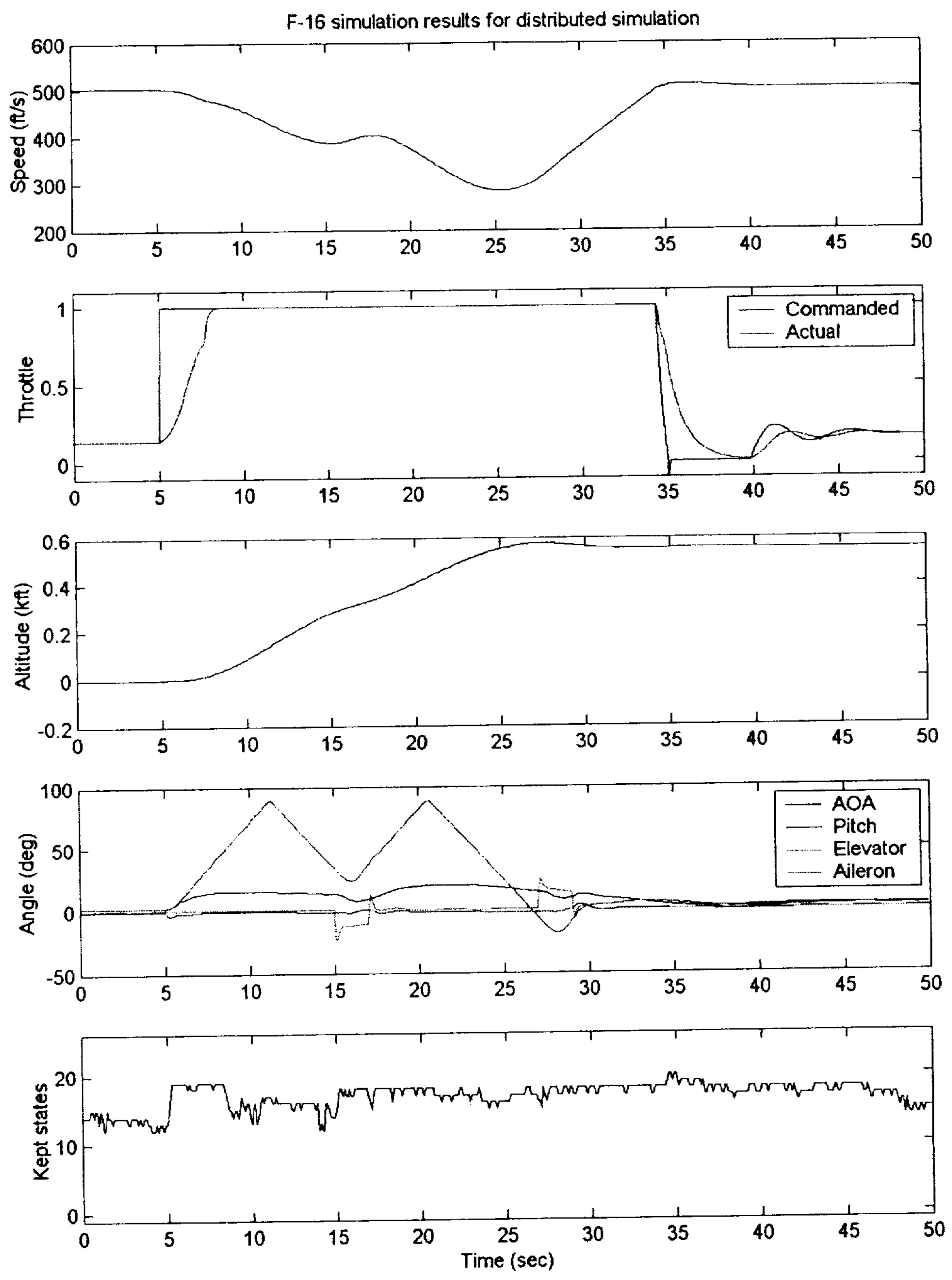

Figure 8-2. Simulation Results for F-16 Model With Multiple Processes on Single Computer.

These tests were repeated for the six satellite formation model using the configuration illustrated in Figure 8-3. In this test, seven separate executable processes were run on the same computer: one *Simulation_Process* for each satellite and one *Simulation_Manager*, along with the CORBA *Naming Service*. The test was again repeated on two separate computers, both with and without communication delays. Test results are summarized in Table 8-2. Simulation results (illustrated in Figure 8-4) were again exactly identical to all bits of machine precision for all cases.

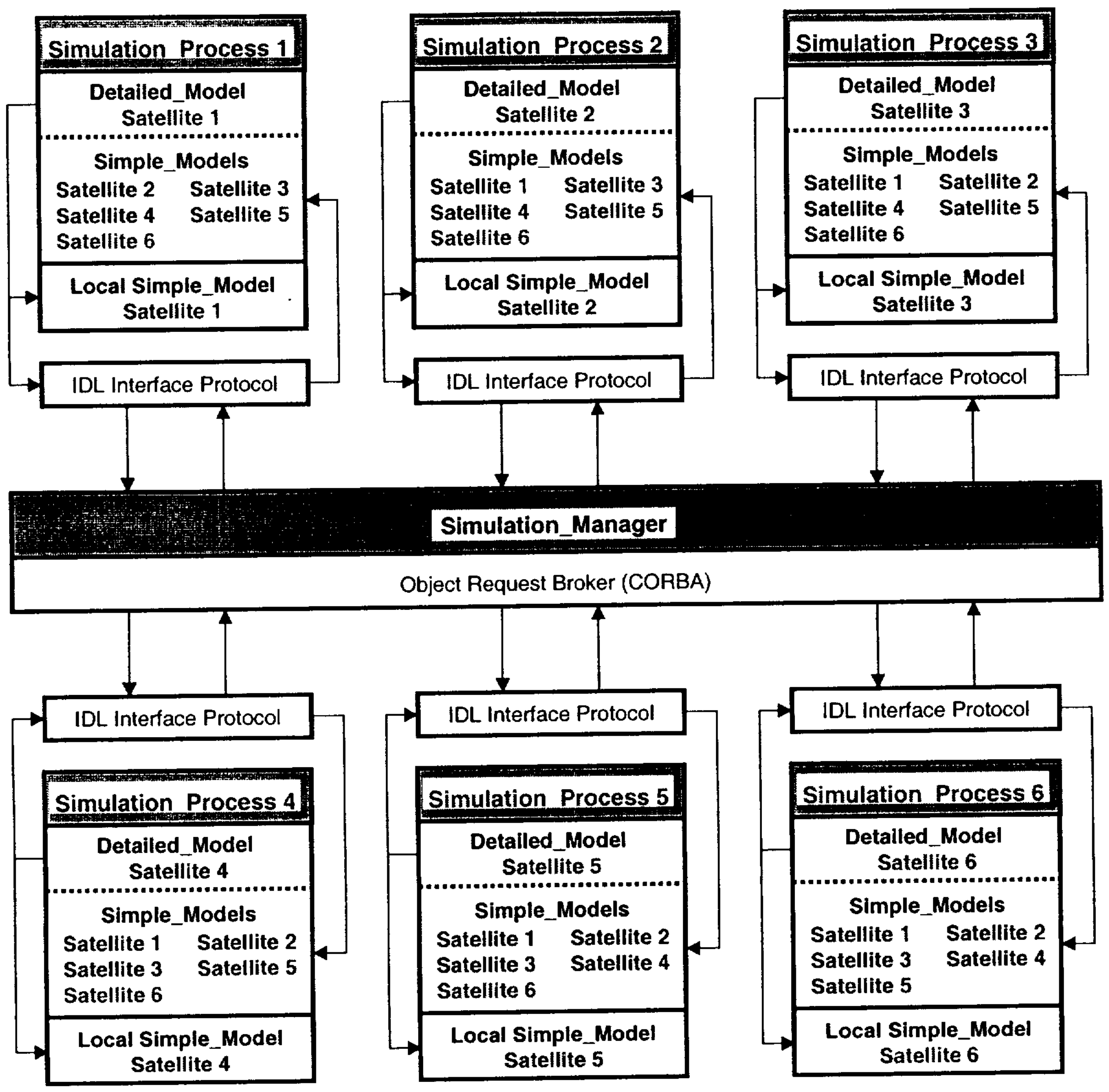
Simulation Process 1
Detailed_Model Satellite 1
Simple_Models Satellite 2 Satellite 3 Satellite 4 Satellite 5 Satellite 6
Local Simple_Model Satellite 1
IDL Interface Protocol
Simulation Process 2
Detailed_Model Satellite 2
Simple_Models Satellite 1 Satellite 3 Satellite 4 Satellite 5 Satellite 6
Local Simple_Model Satellite 2
IDL Interface Protocol
Simulation Process 3
Detailed_Model Satellite 3
Simple_Models Satellite 1 Satellite 2 Satellite 4 Satellite 5 Satellite 6
Local Simple_Model Satellite 3
IDL Interface Protocol
Simulation_Manager
Object Request Broker (CORBA)
IDL Interface Protocol
Simulation Process 4
Detailed_Model Satellite 4
Simple_Models Satellite 1 Satellite 2 Satellite 3 Satellite 5 Satellite 6
Local Simple_Model Satellite 4
IDL Interface Protocol
Simulation Process 5
Detailed_Model Satellite 5
Simple_Models Satellite 1 Satellite 2 Satellite 3 Satellite 4 Satellite 6
Local Simple_Model Satellite 5
IDL Interface Protocol
Simulation Process 6
Detailed_Model Satellite 6
Simple_Models Satellite 1 Satellite 2 Satellite 3 Satellite 4 Satellite 5
Local Simple_Model Satellite 6

Figure 8-3. Configuration for Six-Satellite Formation Multiple Process Simulation.

Table 8-2. Test Results for Six-Satellite Formation With Multiple Processes on Single Computer

Computer	Processor	RAM	Execution time without delays	Execution time with delays
jyb.creare.com	Pentium III 1.1 GHz	512 MB	3300 sec	31000 sec
test1.creare.com	Pentium Pro 200 MHz	128 MB	14000 sec	42000 sec

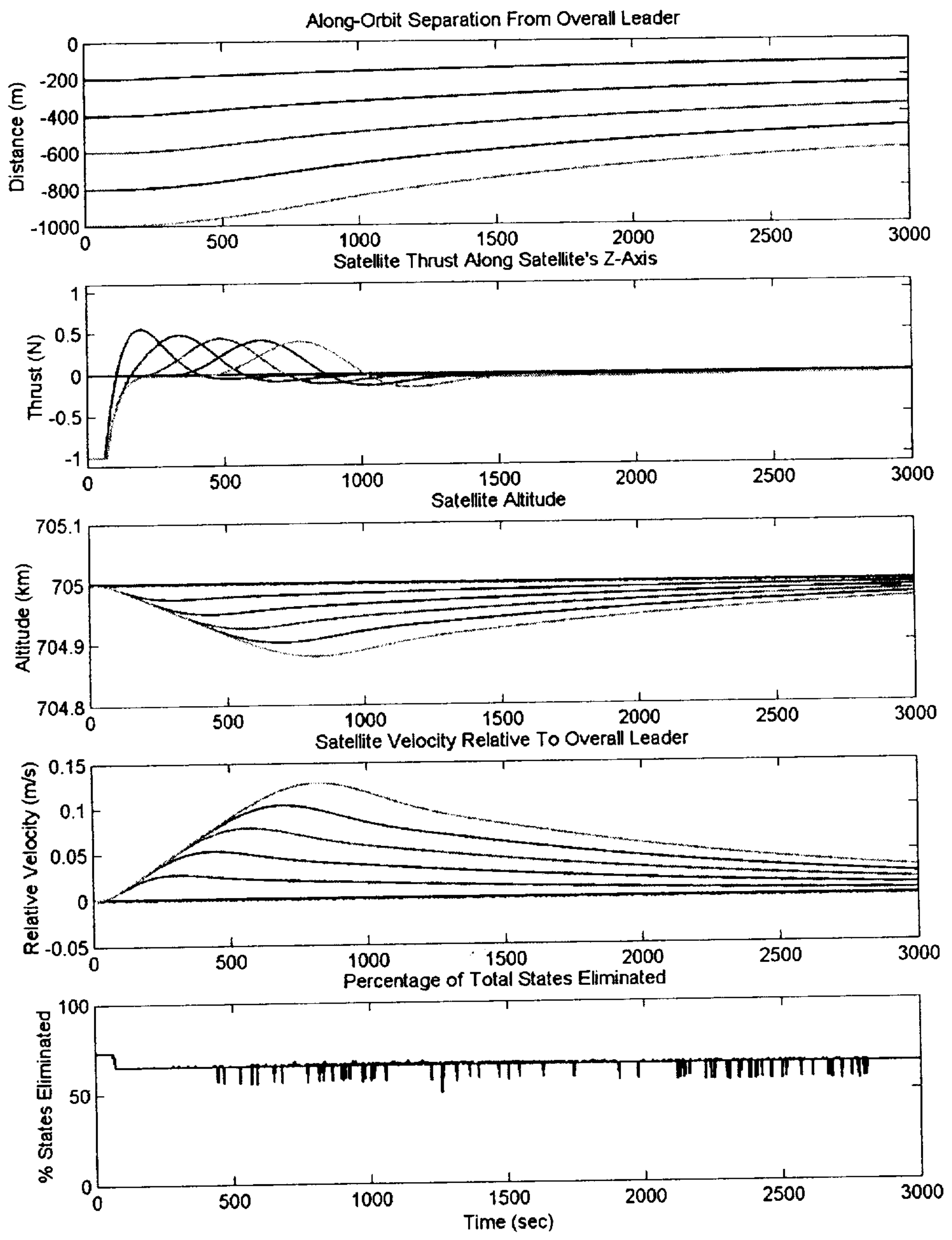

Figure 8-4. Simulation Results for Six-Satellite Formation Model With Multiple Processes on Single Computer.

For both the F-16 and six-satellite formation models running on a single computer, neither communication delays nor processor speed had any effect on the computed simulation results. For the non-delayed cases, execution run-time was approximately inversely proportional to processor clock speed. For the artificially delayed cases, processor speed had a much smaller effect, since the imposed communication delay became a more significant "bottleneck" that dominated execution run-time.

It should be noted that during these tests, a significant "call latency" was observed with the CORBA communication protocol. Even for function calls that pass no parameters, the maximum call rate using a 10 MB Ethernet connection is approximately 200 to 2000 operation invocations per second (as compared to 1,000,000 or more internal C++ function calls per second). Henning and Vinoski (1999) state that "the overall cost of remote calls is dominated by call latency until parameters reach several hundred bytes in size, so an invocation without parameters takes about the same time as an invocation that transmits few parameters." Following this observation, several modifications were made to the software library to minimize the number of CORBA invocations. For example, the set of *Simple_Models* was originally sent by the *Simulation_Manager* to each *Simulation_Process* using an individual function call for each simple model. This was modified so that all *Simple_Model* updates are now passed using a single invocation. Such straightforward changes resulted in a significant reduction in run-time for the multiple process simulations. For example, these changes reduced the time required to run the non-delayed F-16 simulation on test1.creare.com from 405 seconds to 286 seconds.

8.2 Demonstration of Software Library for Example Configurations (F-16 and Six-Satellite Formation) on Multiple Networked Computers

Both the F-16 and six-satellite formation models simulations were then run with each subsystem distributed across multiple networked computers. The configurations used for either model are summarized in Table 8-3. For both models, each subsystem process was run on a separate computer. The *Simulation_Manager* and *Naming Service* processes were also run on one of these computers (test1.creare.com) alongside a subsystem process. The processor speed (~200 MHz) and RAM (~128 MB) were comparable for each computer. The computers were physically linked on the same 10 Mbps Ethernet line which was connected to the Internet through a T1 connection.

No problems were observed with the simulations distributed across these multiple networked computers, and the exact same simulation results were obtained. Compared to the previous simulations run on a single computer, we expected a trade-off between faster local execution of each process (since each process had its own processor) but slower communication among processes. The total execution time for the F-16 and six-satellite formation simulations were 130 seconds and 4200 seconds, respectively, on the six computers, as compared to 290 and 14000 seconds, respectively, on a single 200 MHz computer (test1.creare.com). Thus, for these examples the benefits of distributed computing power outweighed the added cost of network communication.

Table 8-3. Configurations for F-16 and Six-Satellite Formation Models With Processes on Multiple Networked Computers

Computer	Processor	RAM	Connection Type & Speed	F-16 Subsystem Model	Satellite Subsystem Model
test1.creare.com*	Pentium Pro 200 MHz	128 MB	T1 750 kbps	engine	satellite 1
test2.creare.com	Pentium Pro 200 MHz	128 MB	T1 750 kbps	controller	satellite 2
test3.creare.com	Pentium Pro 200 MHz	128 MB	T1 750 kbps	elevator	satellite 3
test4.creare.com	Pentium Pro 200 MHz	128 MB	T1 750 kbps	ailerons	satellite 4
test5.creare.com	Pentium Pro 180 MHz	96 MB	T1 750 kbps	rudder	satellite 5
test6.creare.com	Pentium Pro 266 MHz	128 MB	T1 750 kbps	body	satellite 6

* *Simulation_Manager* and *Naming Service* also run on test1.creare.com

The six-satellite formation simulation was then repeated for a shorter simulation time period (300 seconds) under three different conditions. In the first case, the models were subjected to the linearization, block diagonalization, and model order reduction algorithms. For this first case, the time required to generate the *Simple_Models* would be greatest, but the time required to transmit and integrate them would be the least due to their relatively small size. In the second case, the models were linearized and block diagonalized, but not subjected to model order reduction. For this second case, the time required for *Simple_Model* generation would be somewhat less than the first case while the time required for transmission and integration would be somewhat more. In the third case, the models were only linearized, and subjected to neither block diagonalization nor model order reduction. For this third case, the time required to generate the *Simple_Model*s would be the least, but the time required to transmit and integrate them would be the greatest due to their relatively large size. The results of these tests are shown in Table 8-4. Note that the typical *Simple_Model* size for these 22 state linearized models is reduced by about 25% by block diagonalization alone, while block diagonalization coupled with model order reduction reduces the linearized model size by about 75% for this model.

Unfortunately, the reduction in execution time was not nearly as dramatic. This suggests that much of the run-time cost is dominated by linearization of the *Detailed_Model*s. This is not surprising since the software library currently estimates partial derivatives of the *Detailed_Model*s using finite differences, a very computationally expensive technique. During the next year of this project, we will implement a model preprocessing step, where the partial derivatives of the model will be hard-coded explicitly using third-party automatic differentiation software (probably ADIC). This should not only greatly reduce the execution time required for the distributed simulation software library, but also result in more accurate linearizations of the model (and hence more accurate *Simple_Model*s).

Table 8-4. Results for Various *Simple_Model* Generate Schemes for Six-Satellite Formation Models With Processes on Multiple Networked Computers

Case	Linearization	Block Diagonalization	Model Order Reduction	Typical Simple_Model Size	Execution Time
1	√	√	√	1200 bytes	4200 sec
2	√	√		3600 bytes	4300 sec
3	√			4800 bytes	4700 sec

8.3 DEMONSTRATION OF SOFTWARE LIBRARY FOR COMPLEX CONFIGURATION (30-SATELLITE FORMATION) ON MULTIPLE NETWORKED COMPUTERS

In the next test, a substantially more complex model of a 30 satellite formation (consisting of six subformations with five satellites each) was simulated on the same set of multiple networked computers. The configuration used is summarized in Table 8-5. Each of the 30 satellites was simulated as a separate process. However, due to computer availability limitations, five independent satellite processes were simulated on each of the six computers. The *Simulation_Manager* and *Naming Service* processes were also run on one of these computers (test1.creare.com) alongside the first five satellite processes.

The results for 3000 seconds of simulation time are illustrated in Figure 8-5. No problems were observed with the distributed simulation of this complex configuration on multiple networked computers. However, the processing time for the overall simulation was rather lengthy (approximately 16.4 hours). This is attributed not to network communication delays but rather a processing speed bottleneck since five processes were simulated together on each computer. This problem is again exacerbated by the use of finite differences to estimate partial derivatives of the *Detailed_Models*. Moreover, each *Simulation_Process* subsystem process, as well as the *Simulation_Manager* process, would ideally be run on its own dedicated computer (or processor of a multiple processor computer). We expect that such a configuration would reduce the total execution time for the 30 satellite formation model by approximately 80% (since the processing speed of each process would increase by roughly a factor of five).

Table 8-5. Configurations for 30-Satellite Formation Models With Processes on Multiple Networked Computers				
Computer	**Processor**	**RAM**	**Connection Type & Speed**	**Satellite Subsystem Models**
test1.creare.com*	Pentium Pro 200 MHz	128 MB	T1 750 kbps	satellites 1, 2, 3, 4, 5
test2.creare.com	Pentium Pro 200 MHz	128 MB	T1 750 kbps	satellites 6, 7, 8, 9, 10
test3.creare.com	Pentium Pro 200 MHz	128 MB	T1 750 kbps	satellites 11, 12, 13, 14, 15
test4.creare.com	Pentium Pro 200 MHz	128 MB	T1 750 kbps	satellites 16, 17, 18, 19, 20
test5.creare.com	Pentium Pro 180 MHz	96 MB	T1 750 kbps	satellites 21, 22, 23, 24, 25
test6.creare.com	Pentium Pro 266 MHz	128 MB	T1 750 kbps	satellites 26, 27, 28, 29, 30

* *Simulation_Manager* and *Naming Service* also run on test1.creare.com

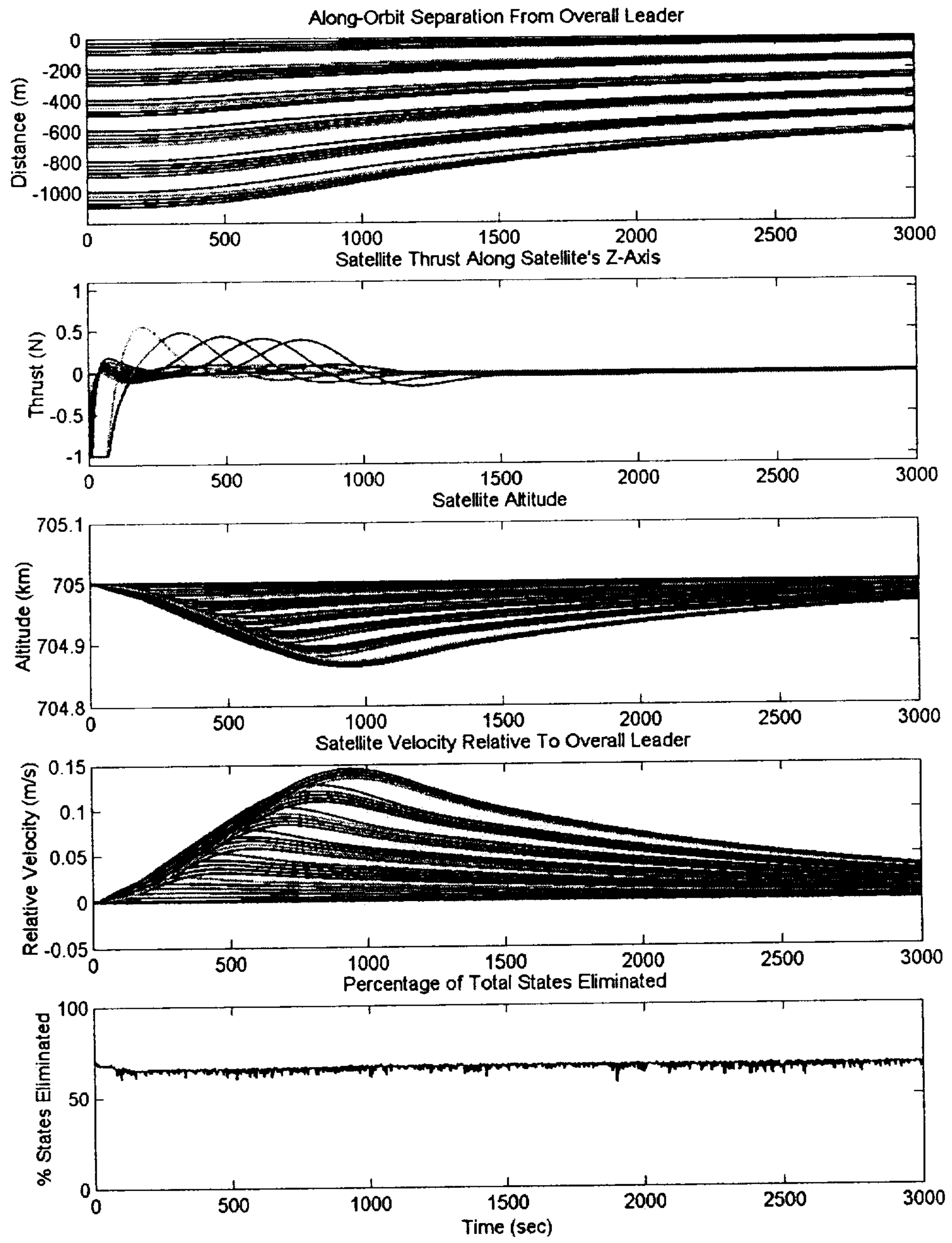

Figure 8-5. Simulation Results for 30-Satellite Formation Model With Multiple Processes on Multiple Computers.

8.4 DEMONSTRATION OF SOFTWARE LIBRARY FOR COMPLEX CONFIGURATION (30-SATELLITE FORMATION) ON INTERNET/GEOGRAPHICALLY DISPERSED COMPUTERS

The testing of the distributed simulation software library culminated with a demonstration on geographically dispersed computers communicating over the Internet. In addition to the six computers at Creare Inc., two engineers volunteered use of their home computers as part of the demonstration. One of these computers was connected through a relatively fast DSL network connection. The other, however, was connected through a significantly slower 56 Kband dial-up connection. While dial-up connections would not be the preferred scheme for production calculations, this configuration allowed us to test the robustness of the integrated system to the relatively unpredictable time delays that are encountered with a dial-up network connection.

For this test, the 30-satellite formation (consisting of six subformations with five satellites each) was again simulated. The configuration of the computers used is summarized in Table 8-6. Note that although only one satellite subsystem was run on each remote computer, this was not a restriction imposed by the software library. Also, due to the slow speed of the dial-up connection, the simulation was run for only 300 seconds of simulation time.

Two minor problems were encountered before the geographically distributed simulation was run successfully. First, networking firewalls had been installed on both remote computers, and these firewalls did not allow the *Simulation_Manager* process on test1.creare.com to contact the *Simulation_Process*es on the remote computers. To work around this, the firewall was temporarily disabled on one computer and explicit permission was granted for the IP address of test1.creare.com to communicate through the firewall of the other computer. The second problem arose when the computer on the dial-up network connection went into "standby mode" (an energy saving feature which puts the computer into a low power consumption state) that caused the network connection to be lost. To work around this, the automatic standby feature of the computer was also temporarily disabled. Once these problems were addressed, the simulation proceeded without any additional difficulties. Note, however, that if a network connection is lost during a simulation in the current version of the software library, then the entire simulation must be restarted since there is no explicit recovery mechanism in place.

The simulation results for the geographically distributed test are illustrated in Figure 8-6. These results are exactly the same as for the first 300 seconds of the results illustrated in Figure 8-5. The total execution time required for the simulation was 9000 seconds, an increase of approximately 50% from the configuration shown in Table 8-5 that did not involve geographically dispersed computers. Again, the increase in execution time is attributable to the slow dial-up connection used for one of the remote computers, even though the processing speed of this computer was substantially greater than the others. This illustrates the fact that the distributed simulation software library is most productive when the simulation is distributed on computers with approximately equal processing speeds and network connection speeds. Otherwise, a single computer can act as a bottleneck which slows down the entire simulation.

Table 8-6. Configurations for 30-Satellite Formation Models With Processes on Geographically Dispersed Computers

Computer	Processor	RAM	Location & Distance from Creare Inc.	Connection Type & Speed	Satellite Subsystem Models
test1.creare.com*	Pentium Pro 200 MHz	128 MB	Hanover, NH --	T1 750 Kbps	satellites 1, 2, 3, 4, 5
test2.creare.com	Pentium Pro 200 MHz	128 MB	Hanover, NH --	T1 750 Kbps	satellites 6, 7, 9, 10
test3.creare.com	Pentium Pro 200 MHz	128 MB	Hanover, NH --	T1 750 Kbps	satellites 11, 12, 13, 14, 15
test4.creare.com	Pentium Pro 200 MHz	128 MB	Hanover, NH --	T1 750 Kbps	satellites 16, 17, 18, 19, 20
test5.creare.com	Pentium Pro 180 MHz	96 MB	Hanover, NH --	T1 750 Kbps	satellites 21, 22, 24, 25
test6.creare.com	Pentium Pro 266 MHz	128 MB	Hanover, NH --	T1 750 Kbps	satellites 26, 27, 28, 29, 30
jks (vermontel.net)	Pentium II 266 MHz	64 MB	Hartland, VT 20 miles	DSL 1000 Kbps	satellite 23
jyb (earthlink.net)	Pentium III 1.2 GHz	512 MB	Grantham, NH 20 miles	phone dial-up 50 Kbps	satellite 8

* *Simulation_Manager* and *Naming Service* also run on test1.creare.com

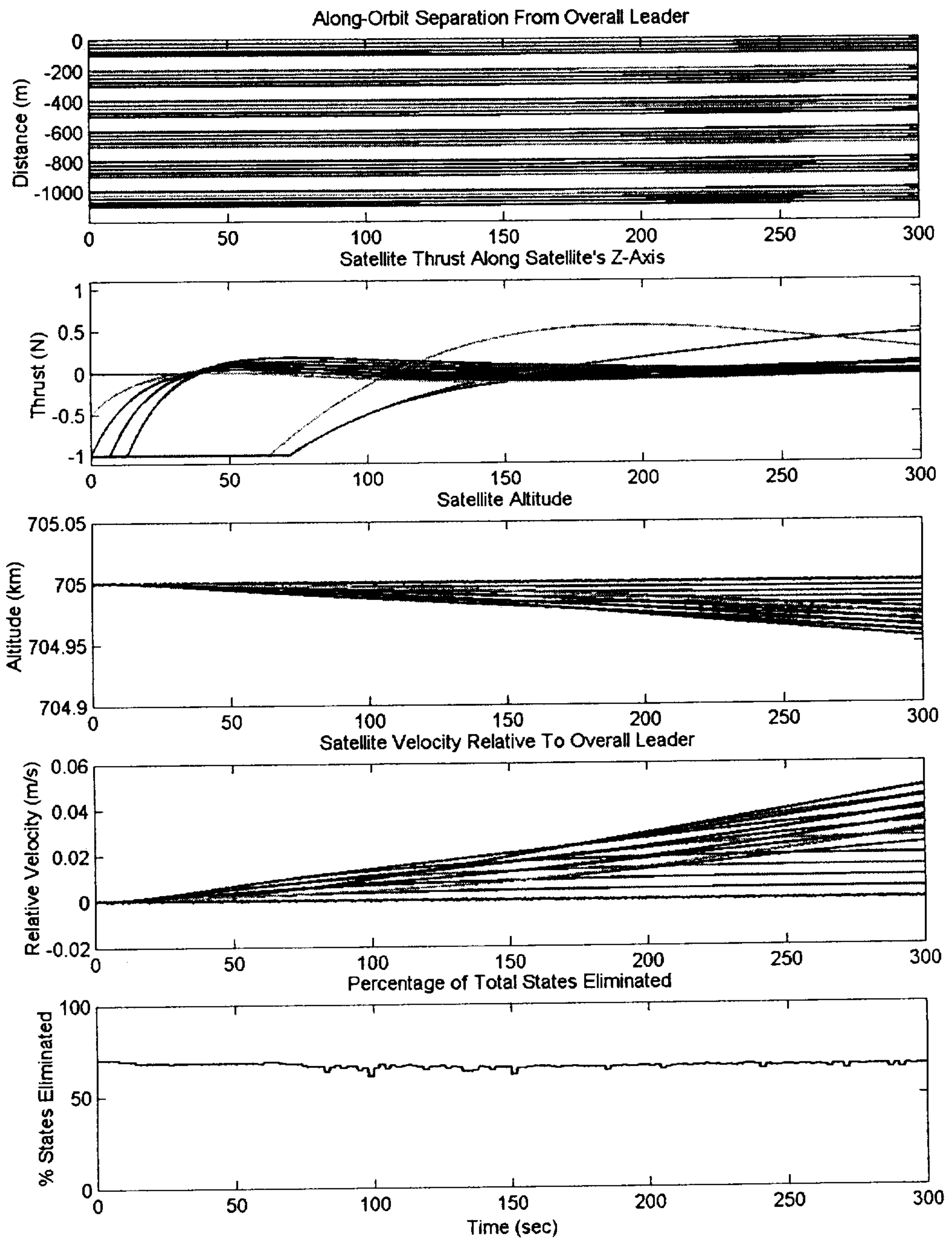

Figure 8-6. Simulation Results for 30-Satellite Formation Model With Multiple Processes on Geographically Dispersed Computers.

8.5 CONCLUSIONS

The capabilities and performance of the distributed simulation software library have been demonstrated for multiple subsystem processes of varying complexity, running on a single computer, distributed across a local intranet, and geographically distributed across the Internet. Use of object-oriented partitioning and surrogate *Simple_Models* greatly reduces the frequency at which the distributed processes must communicate. Furthermore, the use of model order reduction in the generation of *Simple_Models* greatly reduces the amount of data that must be communicated among these processes.

The use of a CORBA-based communication scheme in the software library has proven to be robust, and simulation results are unaffected by both simulated and real communication delays. Although CORBA invocations introduce an overhead due to call latency, special consideration has been taken in the design of the software library to minimize this.

9 CONCLUSIONS AND FUTURE PLANS

In this fiscal year, we largely completed the theoretical underpinnings of our approach for distributed simulation of aerospace models. Many of these features were implemented in the associated software library, and the software was then demonstrated in a series of tests. Over the course of the year, tests were successfully conducted on five different physical models of generally increasing complexity. The tests included simulations where the entire model was executed on a single computer, on several computers connected by a local area network, and on geographically distributed computers connected by the Internet.

While the effort necessary to develop the various physical models for testing the library was larger than originally anticipated, this effort has proved invaluable because of the different lessons that were learned in each case about the behavior of these models. In particular, the need to treat block-diagonalized forms of the linearized subsystem models was learned by testing the library on the more complicated, highly nonlinear models.

In the coming year, most of our effort will be focused in two areas. First, we will develop the capability to automatically generate hard-coded versions of the Jacobian matrix of the subsystem models using the ADIC or ADIFOR codes developed at Argonne National Laboratory (assuming, as we anticipate, that no difficulty is encountered in acquiring access to these programs). Second, we will add an uncertainty analysis capability to the software library using an adjoint-based technique. We will also continue adding to the functionality of the model order reduction algorithms. All modifications to the library will be tested using the suite of test models described above.

10 REFERENCES

Bauer, F. (NASA GSFC) et al., "Enabling Spacecraft Formation Flying through Spaceborne GPS and Enhanced Automation Technologies," 1999 ION-GPS Conference, <http://www.mit.edu/people/jhow/ff/ion99-final.pdf>.

Bavely, C. A. and Stewart, G. W., "An Algorithm for Computing Reducing Subspaces by Block Diagonalization," *SIAM J. Numer. Anal.*, 16, 2, April 1979, pp. 359–367.

Biedron, R. T., Mehrotra, P., Nelson, M. L., Preston, F. S., Rehder, J. J., Rogers, J. L., Rudy, D. H., Sobieski, J. and Storaasli, O.O., "Compute as Fast as the Engineers Can Think!," Langley Research Center, NASA/TM-1999-209715, September 1999.

Brogan, W. L., *Modern Control Theory*, Prentice-Hall, 1991.

Dullerud, G. E. and Paganini, F., *A Course in Robust Control Theory*, Springer-Verlag, 2000.

Golub, G. H. and Van Loan, C. F., *Matrix Computations*, Johns Hopkins University Press, 1983.

Heath, M. T. and Dick, W. A., "Virtual Prototyping of Solid Propellant Rockets, *Computing in Science and Engineering*, March/April 2000.

Hendricks, E., Jannerup, O. and Sorensen, P. H., Forelaesningsnoter til Kursus 50310 Reguleringsteknik 2 (Notes for Course 50310 in Controls Technology II), Institute for Automation, Danish Technical University, August 2000.

Henning, M. and Vinoski, S., *Advanced CORBA Programming with C++*, Addison-Wesley Longman, Reading, MA, 1999.

Hogan, N., "Integrated Modeling of Physical System Dynamics," 1994, unpublished notes.

Lewis, F. L., and Stevens, B. L., *Aircraft Control and Simulation*. John Wiley & Sons, Inc., 1992.

Moler, C. B. and C. F. Van Loan, "Nineteen Dubious Ways to Compute the Exponential of a Matrix," *SIAM Review*, 20, 1979, pp. 801–836.

Moore, B. C., "Principal Component Analysis in Linear Systems: Controllability, Observability, and Model Reduction," *IEEE Trans. Automatic Control*, 26, 1, February 1981, pp. 17–31.

NASA GSFC. "Earth Observing One," <http://eo1.gsfc.nasa.gov/>.

Tomasi, C., "Mathematical Methods for Robotics and Vision," Stanford University, 2000, <http://www.stanford.edu/class/cs205/notes/book/book.html>.

U.S. Air Force Research Laboratory. "TechSat 21 – Space Missions Using Satellite Clusters," 1998, <http://www.vs.afrl.af.mil/Factsheets/techsat21.html>.

Varga, A., "Computational Techniques Based on the Block-Diagonal Form for Solving Large Systems Modeling Problems," *Proc. IEEE Conf. Aerospace Control Systems*, Westlake Village CA, 1993.

Varga, A., "Enhanced modal approach for model reduction," *Mathematical Modelling of Systems*, 1, 1995, pp. 91–105.