

THESIS

1

2003

5748.115

LIBRARIES
MICHIGAN STATE UNIVERSITY
EAST LANSING, MICH 48824-1048

This is to certify that the
thesis entitled

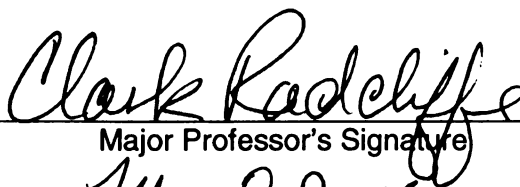
MODELING OF DYNAMIC SYSTEMS USING INTERNET
ENGINEERING DESIGN AGENTS

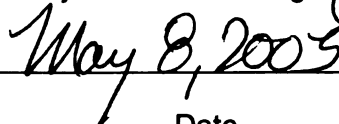
presented by

DREW R. REICHENBACH

has been accepted towards fulfillment
of the requirements for the

Master of Science degree in Mechanical Engineering


Major Professor's Signature


Date

PLACE IN RETURN BOX to remove this checkout from your record.
TO AVOID FINES return on or before date due.
MAY BE RECALLED with earlier due date if requested.

DATE DUE	DATE DUE	DATE DUE

MODELING OF DYNAMIC SYSTEMS USING INTERNET
ENGINEERING DESIGN AGENTS

By

Drew R. Reichenbach

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Department of Mechanical Engineering

2003

process

become

are migr

number

Addition

design an

this requi

global mo

integrator

contained

modeling

revalidation

companies

at any time

method for

information

modeling me

system mode

ABSTRACT

MODELING OF DYNAMIC SYSTEMS USING INTERNET ENGINEERING DESIGN AGENTS

By

Drew R. Reichenbach

As engineering evolves from largely in-house design, validation, and production processes to an interlinked network of global designers, manufacturers, and integrators, it becomes necessary for the engineering processes involved to change as well. Companies are migrating from being product manufacturers to product integrators, combining a large number of engineering subsystems designed and manufactured by their supply base. Additionally, product integrators are coming to rely heavily on those suppliers for the design and model information required to make their integration a success. Currently, this requires the time and effort to reformulate and revalidate the subsystem models into a global model of the product. In addition, suppliers are hesitant to give their models to the integrators, without binding legal agreements to protect the proprietary information contained within their subsystem models. The fixed input/output structure the modular modeling method provides eliminates the need for global model reformulation and revalidation. The distributed, agent-based architecture of the i-EDA system allows companies to focus on their core competencies, and to have access to model information at any time, and from any place. In addition, the synthesis of the two concepts provides a method for communicating dynamic models in a manner that conceals the proprietary information contained within them. This work details the implementation of the modular modeling method within the i-EDA framework to assemble and solve a class of dynamic system models over the internet. Examples are given from structural mechanics.

My th
would not be

My th
my committe

Witho

Thanks to all

ACKNOWLEDGEMENTS

My thanks go to Dr. Radcliffe. Without his guidance and assistance, this work would not be possible.

My thanks also go to Dr. Rosenberg and Dr. Sticklen for serving as members of my committee.

Without the support of my family and friends, I never would have made it this far. Thanks to all of you.

List c

List c

Chap

The

The

Chapt

Mo

Mo

Mo

Dev

Dev

Dev

Dev

Chapte

Simu

Simu

Truss

Mode

Chapter

Appendi

Appendi

TABLE OF CONTENTS

List of Tables	vi
List of Figures	vii
Chapter 1. Introduction	1
The i-EDA System.....	2
The Modular Modeling Method.....	3
Chapter 2. Theoretical Solution	7
Model Formulation	7
Model Connection.....	8
Model Connection Algorithm	10
Development of a One-Dimensional Bar Model	13
Development of a Compound Bar Model.....	14
Development of a Dynamic Truss Model	15
Development of a Dynamic Span Model.....	19
Chapter 3. Dynamic System Simulation.....	21
Simulation of a One-Dimensional Bar.....	24
Simulation of a Compound Bar	26
Truss and Span Simulation	28
Modeling within the i-EDA system	29
Chapter 4. Conclusion.....	32
Appendix A Query/Response Definition	35
Appendix B Matlab Code	36

List of

Cr

Cr

Dy

Ad

add

Join

Pos

Pre

Rota

cell

Solv

Appl

Cell

Cell

Appendix

Appendix

Modular

Combina

Method

List of Refe

List of Included Programs.....	36
CreateTruss.m	37
CreateSpan.m	39
DynamicJoin.m	41
AddDimension.m	44
addpoly.m.....	45
JoinMap.m	47
PostMultCell.m	48
PreMultCell.m.....	50
RotateDynamicBar.m.....	51
cell2str.m.....	53
SolveDynamics.m	54
ApplyBoundaryCond.m	58
CellArrayAdjoint.m	60
CellArrayDet.m.....	64
Appendix C Dynamic Response Client Documentation.....	67
Appendix D Failed Methods for Joining Dynamic System Models	76
Modular Dynamic System Representation	76
Combining Modular Dynamic Systems.....	77
Method Failure.....	81
List of References	83

Table I

LIST OF TABLES

Table 1 Properties for the BA1x2 bar model in the i-EDA system	25
--	----

Figure

sy

Figure

mo

rep

Figure 3

disp

Figure 4

Figure 5 A

modu

Figure 6 A

conne

Figure 7 O

Figure 8 Co

Figure 9 Th

Figure 10 Th

LIST OF FIGURES

Figure 1 An example of measurement perspective causality for a linear mechanical system. Black arrows represent forces; white arrows represent velocities.	4
Figure 2 A Modular Modeling Connector (MMC) element for connecting two modular models of linear mechanical systems. Black arrows represent forces; white arrows represent velocities.....	5
Figure 3 Diagram of a one dimensional discretized bar. Inputs are forces, outputs are displacements.....	13
Figure 4 A compound bar model. Inputs are forces and outputs are displacements	14
Figure 5 A dynamic truss model is composed of three dynamic bar models and three modular modeling connector elements	16
Figure 6 A span model is composed of two trusses, one bar, and three modular modeling connector elements.....	19
Figure 7 One dimensional bar model.....	25
Figure 8 Constrained compound bar model.....	27
Figure 9 The i-EDA system topology.....	29
Figure 10 The i-EDA dynamic response client.....	31

Eng

facilities, a

no longer d

accurate, co

Along with

dynamic mo

design and m

overhead, co

that strong le

protect the pr

these legal ag

to a full year t

model must be

the global refo

time consumin

the individual c

replacement, su

and time-to-mar

models between

with traditional

A comple

models, and the

Chapter 1. Introduction

Engineering is a global effort. Corporations have design teams, manufacturing facilities, and a supply base that spans multiple nations. Manufacturers of finished goods no longer do all of the product design in house but rely on their suppliers for complete, accurate, cost-effective component designs to integrate into their finished products. Along with supplying component designs, suppliers are now expected to provide dynamic models of those systems [Kerr, 2000]. Kerr also states that communicating design and modeling information between companies requires a great amount of overhead, cooperation, and trust between corporations. Model communication requires that strong legal agreements are in place before any information is transmitted in order to protect the proprietary information contained within the model. Creating and maintaining these legal agreements can be time consuming – requiring anywhere from a few months to a full year to establish. Additionally, after model information has been received, the model must be integrated into the existing system model. This process typically requires the global reformulation and revalidation of the entire model, which is one of the most time consuming portions of dynamic model simulation. Additionally, reformulation of the individual component models into an assembly model often makes component model replacement, substitution, or updating extremely difficult. To reduce design cycle length and time-to-market, a process is needed for communicating reusable dynamic system models between companies that protects the proprietary information and interfaces easily with traditional modeling methods.

A complete solution would allow for the communication of dynamic system models, and the protection of the proprietary information contained within them.

Commun
example,
assemblies
the ability
communi
final prod
frequently
protect th
between s
contracts
proprietar
agreement
to drive c

T
designed
model an
The core
describe
oriented
Each inc
virtual p
a networ

Communication of models is important because of the global nature of engineering. For example, an assembly is designed in Detroit for assembly in St. Louis, from sub assemblies designed in Seattle and built in Mexico from Japanese components. Without the ability to model the component, sub-assembly, and assembly designs, and to communicate the model information, the ability to predict the functional behavior of the final product is greatly reduced. The model information that is communicated, however, frequently contains information considered proprietary by the supplier. The ability to protect the proprietary information contained in the models that are communicated between suppliers and their customers will remove the need for lengthy negotiations and contracts before any parts have been purchased. Additionally, the protection of proprietary information will allow manufacturers to test models before any specific agreements were created. This would increase competition in the marketplace and help to drive costs down, while increasing the performance and quality of engineered products.

The i-EDA System

The i-EDA system [Radcliffe, Sticklen, and Gosciak, 2002; Gosciak, 2001] is designed to be a distributed, component-based system capable of providing both external model and descriptive information without divulging internal proprietary information. The core of the i-EDA system is the design agent. In this work, the term "agent" describes one class of agent software; a group of semi-autonomous, stationary, object-oriented programs designed to interact with both users and each other over the internet. Each individual design agent represents a real world object and could be considered a virtual product. Agents representing subsystems and components are linked by means of a network to form larger integrated systems. Agents interact with the user, and with each

of

of

ar

m

ex

de

sys

me

val

me

rev

mo

qui

the

and

are

form

when

powe

other, through a system of strict, predefined queries stored in the global ontology. The ontology, which interacts as an agent itself, contains the set of standardized queries that are complete enough to allow compositional modeling. When used with the modular modeling method, the i-EDA system is capable of assembling and communicating external system models without divulging the proprietary from which the models were developed.

The Modular Modeling Method

The modular modeling method [Byam and Radcliffe, 2000; Byam, 1999] is a systematic, power-based, modeling method that defines strict port causality, eliminating model reformulation, and enhancing the model validation process. The formulation and validation of models are the two most time consuming processes in dynamic system modeling. Traditionally, a model would have to be globally reformulated and then revalidated for each set of possible model input/output configurations. For smaller models, the workload is moderate, however for larger models, the amount of overhead quickly becomes intractable. The number of possible model configurations is related to the input/output selection at each power port. For a power port, the product of the input and output variables must be equal to the power transmitted at that port. Because there are two possible configurations for each power port, the number of possible model formulations (and likewise, required model validations) can be written as

$$N_a = 2^n \quad (1.1)$$

where N_a is the total number of possible model formulations, and n is the number of Power-ports on the model. In a large model, such as the kinematic model used by

Chrysler during the design of their full-size sedans, there could be as many as 5,500 interconnected components. [Computers in Engineering: Chrysler designs paperless cars, 1998] This means that the Chrysler large car model contained a minimum of 2.138×10^{3311} possible model input/output configurations.

With the modular modeling method, the number of possible model formulations is reduced to one. This reduction occurs because of the fixed input/output, or causal, structure of modular models. The causal structure used by the modular modeling method is known as measurement perspective causality. [Byam and Radcliffe, 2000; Byam, 1999] Measurement perspective causality defines the port output as the variable related to the commonly measured quantity (e.g. linear velocity) and the port input as the variable typically assumed to be zero for zero power flow across the port (e.g. linear force).



Figure 1 An example of measurement perspective causality for a linear mechanical system. Black arrows represent forces; white arrows represent velocities.

Measurement perspective causality for a linear mechanical system specifies that the **i**nputs should all be forces, and the outputs should all be velocities, as shown in Figure 1. **L**ike all power-based models, the product of the port variables remains power. Work by **[B]**yam, 1999] and [Byam and Radcliffe, 2000] expands on the port causal structures for **o**ther types of power-based models. Specifying the port causality, however, causes **m**odular models to have incompatible input/output structures. For a connection to be **m**ade, the input port of one model must be connected to the output port of another. With **m**odular modeling, however, this is not possible, because the inputs and outputs of both **m**odels are specified and represented by the same variables.

The Modular Modeling Connector (MMC) element was introduced by Byam to surmount this problem. The MMC is designed to provide complementary causality for the connection of models.

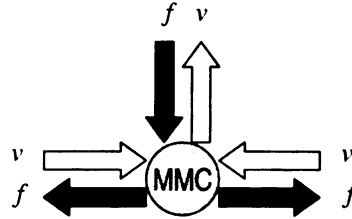


Figure 2 A Modular Modeling Connector (MMC) element for connecting two modular models of linear mechanical systems. Black arrows represent forces; white arrows represent velocities.

A modular modeling connector (Figure 2 e.g.) is composed of $n + 1$ total power ports. Of those ports, the first n ports have an input/output structure that compliments measurement perspective causality. The remaining port has measurement perspective causality and can either be used to connect additional modular models to the port at a later time, or can be condensed out of the model to protect the internal structure of the model.

Mathematically, the MMC applies a set of constraints on the objects being connected. When two objects are physically attached to each other, the output for each model must be equal at the connection. The first constraint applies that connection by equating the port outputs,

$$y_1 = y_2 = \dots = y_n = y_c \quad (1.2)$$

and can be represented mathematically by equation (1.2) where y_1 through y_n represent the output from the models being connected and y_c represents the output of the modular modeling connector. Because the presence of the modular modeling connector itself

should have no effect on the system, the connector is considered an ideal element. As an ideal element, the MMC must conserve power by ensuring that the power flow into the connector from a model or the external port is equal to the power flow out of the connector. The second constraint conserves power at the MMC,

$$\sum_{i=1}^n y_i u_i = y_c u_c \quad (1.3)$$

and can be represented by equation (1.3) where power port pairs $y_i u_i$ for all n connected power ports represent the power flow from the models being connected and $y_c u_c$ represents the power flow into the modular modeling connector.

Chapter 2. Theoretical Solution

Model Formulation

The formulation of a dynamic model for use in the i-EDA system begins with the equations of motion for the system written as ordinary differential equations of any order, N , in the form

$$\sum_{i=0}^N \mathbf{A}_i \left[\frac{d^i}{dt^i} \mathbf{y} \right] = \mathbf{u} \quad (2.1)$$

where \mathbf{y} is the vector of system outputs, \mathbf{u} is the vector of system inputs, and \mathbf{A}_i is a time-invariant matrix. Many mechanical engineering systems are represented by coupled, second order, differential equations written as

$$\mathbf{M}\ddot{\mathbf{y}} + \mathbf{C}\dot{\mathbf{y}} + \mathbf{K}\mathbf{y} = \mathbf{u} \quad (2.2)$$

where \mathbf{y} and \mathbf{u} are defined as in equation (2.1), and \mathbf{M} , \mathbf{C} , and \mathbf{K} are matrices of time-invariant coefficients related to system parameters – typically mass, damping, and stiffness for a mechanical system. Equations like those represented by equation (2.2) are common among lumped-mass systems like Finite Element models.

The model format within the i-EDA system can be found by taking the Laplace transform of equation (2.2) and applying zero initial conditions – similar to the process for creating the transfer function for a system – which yields

$$\left[\mathbf{M}s^2 + \mathbf{C}s + \mathbf{K} \right] \mathbf{Y} = \mathbf{U} \quad (2.3)$$

where s is the Laplace variable, \mathbf{Y} and \mathbf{U} represent the Laplace transform of the output and input vectors respectively, and all other parameters are as defined in equation (2.2).

The representation in equation (2.3) can also be written as

$$\mathbf{R}(s)\mathbf{Y} = \mathbf{U} \quad (2.4)$$

where $\mathbf{R}(s)$ is defined as

$$\mathbf{R}(s) = \left[\mathbf{M}s^2 + \mathbf{C}s + \mathbf{K} \right] \quad (2.5)$$

and each entry is a polynomial in terms of the Laplace variable, s . This model format is very similar to the dynamic stiffness matrix, \mathbf{K}_{dyn} , used by Genta, [Genta, 1999], and in the mechanics community. The key difference lies in the form of the assumed solution. Because of the systems oriented nature of this work, the model form in equations (2.3), (2.4), and (2.5) was selected.

The i-EDA canonical form, as represented by equation (2.4), is preferred to the representations in either equation (2.1) or equation (2.2), in that it requires a single network transmission to communicate the information from one agent to another. Another advantage provided by the i-EDA canonical form is the ability to implement boundary conditions as part of the solution process. Additionally, the similarity between the i-EDA canonical form and the model form used to model static systems in i-EDA [Radcliffe, 2003] provides starting point for developing the algorithms to connect models.

Model Connection

Consider some number, n , of i-EDA models to be connected. Each can individually be represented in i-EDA canonical form, as shown in equation (2.4). When joined using modular modeling connectors, they will form a single system, which can also be written in the i-EDA canonical form. This allows assemblies to be created using a recursive algorithm, rather than a unique algorithm for assembly. This will be done through the constraints supplied by the modular modeling connectors.

The first constraint is the output constraint provided by the modular modeling connector, as shown in equation (1.2). This constraint equates the output from each port involved in the connection to the external output at that point. This represents the physical connection between the ports involved in the connection. In such a connection, the outputs at that point must all be equal.

The second constraint is a statement of the conservation of power that must occur through the rigid, ideal, modular modeling connector, represented by equation (1.3).

When considered from the assembly coordinate frame, instead of the coordinate frame of the components, the conservation of power can be written as the conservation of work.

Consider the physical port location where the connection is going to occur. In each case, the systems are physically connected. In terms of the assembly coordinate frame, the initial position of the connected ports is constrained to be the same. With this initial condition, the conservation of power equation, equation (1.3), is equivalent to the conservation of work which can be written

$$\sum_{j=1}^k W_j = \sum_{j=1}^k (u_j f_j) = u_c f_c = W_c \quad (2.6)$$

where the work at each port involved in the connection, W_j , is summed up and equated to the external work input to the connection, W_c . Combining equations (1.2) and (2.6) gives the constraint on the input forces, which can be written

$$\sum_{j=1}^k f_j = f_c \quad (2.7)$$

where f_j is the force provided by each object involved in the connection and f_c is the external force input into the connector.

Model Connection Algorithm

The computational algorithm for connecting system matrices allows the connected stiffness matrix, to be constructed without first creating the unconstrained system matrix. Performing the connection operations adds component dynamic stiffnesses in much the same way as a finite element code assembles the global stiffness matrix from each of the element stiffnesses.

Creation of the assembly dynamic stiffness matrix is performed by inserting the entries in each object's dynamic stiffness matrix into the assembly stiffness matrix based on the values in a connectivity table. The connectivity tables used in this process can either be connection-based, or object based. Connection-based connectivity tables are preferred when selecting which object ports will be connected in the assembly. Object-based connectivity tables are preferred when using a computerized process for the assembly of dynamic stiffness matrices. Conversion between object-based and connection-based connectivity tables is possible and allows the port connections to be described with the connection-based connectivity table and model assembly to be performed using an object based connectivity table. Consider the connectivity tables for a two-dimensional truss model. In equation (2.8), below

$$\mathbf{CT_J} = \begin{bmatrix} 1 & 1 & 3 & 3 \\ 1 & 2 & 3 & 4 \\ 1 & 3 & 2 & 1 \\ 1 & 4 & 2 & 2 \\ 2 & 3 & 3 & 1 \\ 2 & 4 & 3 & 2 \end{bmatrix} \quad (2.8)$$

the rows correspond to the degrees of freedom in the constrained assembly. Each line lists the object and port pair that are going to be connected at that particular node in the

assembly. While equation (2.8) only shows the connection between two objects at each node in the assembly, an assembly node can contain an infinite number of connections. For unconnected ports, the row in the connection-based connectivity table will only have two entries.

In the object-based connectivity table shown in equation (2.9)

$$\mathbf{CT}_O = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 5 & 6 \\ 5 & 6 & 1 & 2 \end{bmatrix} \quad (2.9)$$

each row corresponds to one of the objects included in the assembly, and the columns are equivalent to the ports on each object. Each entry in the object-based connectivity table corresponds to a node in the assembly, repeated node values indicate the connection between object ports. While equation (2.9) only shows objects with an equal number of degrees of freedom, an object can have an infinite number of degrees of freedom. An object with fewer ports than the object with the maximum number of ports will have one or more empty columns in the object-based connectivity table.

The dynamic stiffness matrix for the assembly can be formed using the connectivity table, and the dynamic stiffness matrices for each of the subcomponents, using the equation

$$\begin{aligned} &\text{For the } k^{\text{th}} \text{ object, } \forall i, j \\ &\mathbf{R}_{A,mn} = \mathbf{R}_{A,mn} + \mathbf{R}_{k,ij} \end{aligned} \quad (2.10)$$

where \mathbf{R}_A is the dynamic stiffness matrix for the assembly, \mathbf{R}_k is the dynamic stiffness matrix for the k^{th} object, and m and n are defined using the object-based connectivity tables as

$$m = \mathbf{CT}_{\mathbf{O},ki} \quad (2.11)$$

$$n = \mathbf{CT}_{\mathbf{O},kj} \quad (2.12)$$

where $\mathbf{CT}_{\mathbf{O}}$ is the object-based connectivity table, and i and j are the indices for the entries in the object dynamic stiffness matrices. For example, consider an entry from the first object stiffness matrix. For $k = 1$, $i = 3$, and $j = 3$ equation (2.10) becomes

$$\begin{aligned} m &= \mathbf{CT}_{\mathbf{O},13} = 3 \\ n &= \mathbf{CT}_{\mathbf{O},13} = 3 \\ \mathbf{R}_{\mathbf{A},33} &= \mathbf{R}_{\mathbf{A},33} + \mathbf{R}_{1,33} = \mathbf{R}_{1,33} \end{aligned} \quad (2.13)$$

Consider next an entry from the second object stiffness matrix. For $k = 2$, $i = 1$, and $j = 1$ equation (2.10) gives

$$\begin{aligned} m &= \mathbf{CT}_{\mathbf{O},21} = 3 \\ n &= \mathbf{CT}_{\mathbf{O},21} = 3 \\ \mathbf{R}_{\mathbf{A},33} &= \mathbf{R}_{\mathbf{A},33} + \mathbf{R}_{2,11} = \mathbf{R}_{1,33} + \mathbf{R}_{2,11} \end{aligned} \quad (2.14)$$

This corresponds to the results for the element in the third row and third column of the assembly dynamic stiffness matrix when the equations are assembled according to equations (1.2) and (1.3). One further example considers an element from the third object dynamic stiffness matrix. For $k = 3$, $i = 1$, and $j = 4$ the result from equation (2.10) becomes

$$\begin{aligned} m &= \mathbf{CT}_{\mathbf{O},31} = 5 \\ n &= \mathbf{CT}_{\mathbf{O},34} = 2 \\ \mathbf{R}_{\mathbf{A},52} &= \mathbf{R}_{\mathbf{A},52} + \mathbf{R}_{3,14} = \mathbf{R}_{3,14} \end{aligned} \quad (2.15)$$

which corresponds to the effects of the fifth assembly output on the second assembly input.

Development of a One-Dimensional Bar Model

Developing a one dimensional bar model begins with the discretization of the bar.

Consider the one-dimensional bar in Figure 3 below,

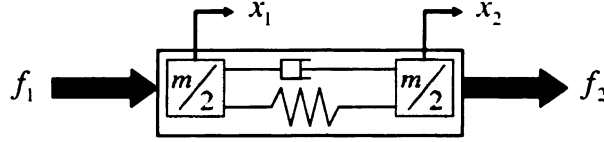


Figure 3 Diagram of a one dimensional discretized bar. Inputs are forces, outputs are displacements.

The inputs to the bar are the applied forces, and are positive in the direction shown. The outputs from the bar are displacements, and are also positive in the direction shown.

Using either the LaGrange or Newton-Euler method, the equations of motion

$$\frac{m}{2}\ddot{x}_1 + c\dot{x}_1 - c\dot{x}_2 + kx_1 - kx_2 = f_1 \quad (2.16)$$

$$\frac{m}{2}\ddot{x}_2 + c\dot{x}_2 - c\dot{x}_1 + kx_2 - kx_1 = f_2 \quad (2.17)$$

which form the model for the system. Equations (2.16) and (2.17) can also be

represented in matrix form as

$$\begin{bmatrix} \frac{m}{2} & 0 \\ 0 & \frac{m}{2} \end{bmatrix} \begin{bmatrix} \ddot{x}_1 \\ \ddot{x}_2 \end{bmatrix} + \begin{bmatrix} c & -c \\ -c & c \end{bmatrix} \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} + \begin{bmatrix} k & -k \\ -k & k \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \quad (2.18)$$

Taking the Laplace transform of equation (2.18) and factoring out the input terms allows **the** model to be written in i-EDA canonical form

$$\mathbf{R}(s)\mathbf{Y} = \mathbf{F} \quad (2.19)$$

where \mathbf{Y} is the Laplace Transform of the vector of system outputs, \mathbf{F} is the Laplace

Transform of the vector of system inputs, and $\mathbf{R}(s)$ is

$$\mathbf{R}(s) = \begin{bmatrix} \frac{m}{2}s^2 + cs + k & -cs - k \\ -cs - k & \frac{m}{2}s^2 + cs + k \end{bmatrix} \quad (2.20)$$

Once the model is in this form, it is ready to be combined with other modular models within the i-EDA system.

Development of a Compound Bar Model

Development of a compound bar model begins with two one-dimensional bar models joined with a modular modeling connector.

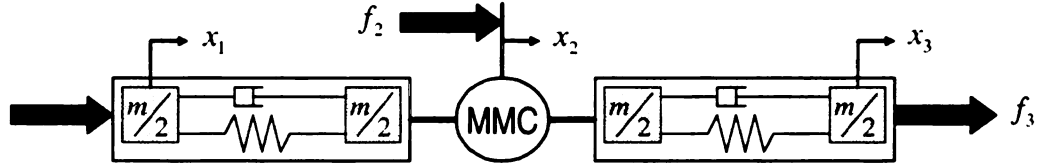


Figure 4 A compound bar model. Inputs are forces and outputs are displacements

In Figure 4, above, the inputs to the model are forces, and the outputs to the model are displacements. The positive direction for each is as shown. The first step in creating the compound bar model is to define the connection-based connectivity table. The connection-based connectivity table for a compound bar is shown in equation (2.21), below.

$$\mathbf{CT_J} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 2 & 2 & 1 \\ 2 & 2 & 0 & 0 \end{bmatrix} \quad (2.21)$$

The object-based connectivity table can be built from the connection-based connectivity table. The object-based connectivity table for a compound bar is shown in equation (2.22).

$$\mathbf{CTO} = \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix} \quad (2.22)$$

This connectivity table can then be used with equation (2.10), and the dynamic stiffness matrix (found in equation (2.20)) for a one-dimensional bar to assemble the dynamic stiffness matrix for a compound bar. Applying equation (2.10) creates the assembly dynamic stiffness matrix, which is written as

$$\begin{bmatrix} R_{1,11} & R_{1,12} & 0 \\ R_{1,21} & R_{1,22} + R_{2,11} & R_{2,12} \\ 0 & R_{2,21} & R_{2,22} \end{bmatrix} \begin{bmatrix} x_{c,1} \\ x_{c,2} \\ x_{c,3} \end{bmatrix} = \begin{bmatrix} f_{c,1} \\ f_{c,2} \\ f_{c,3} \end{bmatrix} \quad (2.23)$$

where $R_{k,ij}$ is the entry in the i^{th} row and j^{th} column of the k^{th} object dynamic stiffness matrix. Substituting in for the appropriate entries gives

$$\begin{bmatrix} \frac{m_1}{2}s^2 + c_1s + k_1 & -(c_1s + k_1) & 0 \\ -(c_1s + k_1) & \frac{m_1 + m_2}{2}s^2 + (c_1 + c_2)s + (k_1 + k_2) & -(c_2s + k_2) \\ 0 & -(c_2s + k_2) & \frac{m_2}{2}s^2 + c_2s + k_2 \end{bmatrix} \begin{bmatrix} x_{c,1} \\ x_{c,2} \\ x_{c,3} \end{bmatrix} = \begin{bmatrix} f_{c,1} \\ f_{c,2} \\ f_{c,3} \end{bmatrix} \quad (2.24)$$

where $x_{c,i}$ and $f_{c,i}$ are the constrained system's outputs and inputs at the i^{th} port respectively. The constrained system maintains the overall symmetric structure that the original unconnected i-EDA dynamic stiffness matrices had, and as such, can either be used in a simulation, or as part of a larger system.

Development of a Dynamic Truss Model

Creating a truss assembly, as shown in Figure 5, from three bar models follows the same methodology used for forming a compound bar. In the truss model, three two-

dimensional bar models are pinned at both ends. The pin joint allows the transmission of axial and transverse loading to the bar models, without transmitting moments.

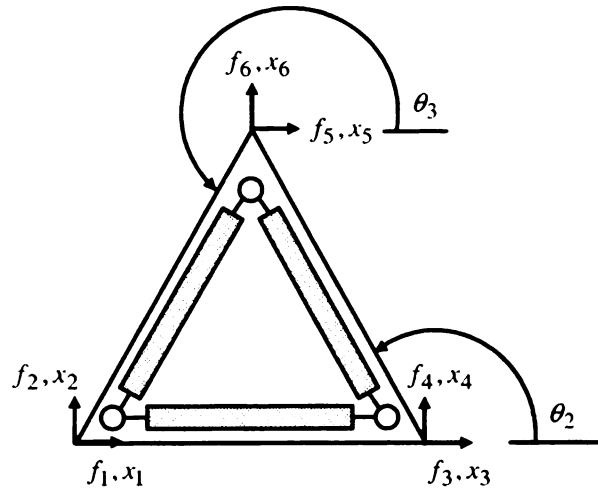


Figure 5 A dynamic truss model is composed of three dynamic bar models and three modular modeling connector elements

Each bar is placed in the model after being rotated from its original coordinate frame into the coordinate frame of the assembly. Rotations are always performed in a counterclockwise direction with the bars' original horizontal position as the reference point.

Several modifications must be made to the original one-dimensional bar model (shown in Figure 3 and equation 2.20) before they can be used to form the truss model. First, the one-dimensional bar model must be expanded to two-dimensions. Adding an additional degree of freedom at each port in the transverse direction expands the model from one, axial, dimension, to the two-dimensions required by the truss model. Some assumptions are made when expanding the model. The first assumption is that the transverse displacements are small, compared to the length of the bar, which allows the model to neglect the any rotation of the bar. The second assumption made is that there

are no stiffness or damping effects in the transverse direction. With these assumptions, the one-dimensional bar from equation (2.20) can be expanded to

$$\begin{bmatrix} \frac{m}{2}s^2 + cs + k & 0 & -cs - k & 0 \\ 0 & \frac{m}{2}s^2 & 0 & 0 \\ -cs - k & 0 & \frac{m}{2}s^2 + cs + k & 0 \\ 0 & 0 & 0 & \frac{m}{2}s^2 \end{bmatrix} \begin{bmatrix} Y_{1x} \\ Y_{1y} \\ Y_{2x} \\ Y_{2y} \end{bmatrix} = \begin{bmatrix} U_{1x} \\ U_{1y} \\ U_{2x} \\ U_{2y} \end{bmatrix} \quad (2.25)$$

where Y_{ix} and U_{ix} are the axial output and input for the i^{th} port, and Y_{iy} and U_{iy} are the transverse output and input for the i^{th} port respectively.

The second additional step when forming a truss model is that the bar models need to be transformed from their original coordinate frame into the coordinate frame of the assembly. This is done through a rotational transformation matrix [Greenwood, 1988], \mathbf{T} , which is defined as

$$\mathbf{T} = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & \cos(\theta) & \sin(\theta) \\ 0 & 0 & -\sin(\theta) & \cos(\theta) \end{bmatrix} \quad (2.26)$$

where the rotation matrix, \mathbf{T} , is constant for each bar, and the bar's rotation angle, θ , is defined as the angular distance measured counterclockwise from the bars' original horizontal position..

The rotation matrix for the second bar with $\theta_2 = 120^\circ$, for example, is

$$\mathbf{T} = \begin{bmatrix} \cos(120) & \sin(120) & 0 & 0 \\ -\sin(120) & \cos(120) & 0 & 0 \\ 0 & 0 & \cos(120) & \sin(120) \\ 0 & 0 & -\sin(120) & \cos(120) \end{bmatrix} = \begin{bmatrix} \frac{-1}{2} & \frac{\sqrt{3}}{2} & 0 & 0 \\ \frac{-\sqrt{3}}{2} & \frac{-1}{2} & 0 & 0 \\ 0 & 0 & \frac{-1}{2} & \frac{\sqrt{3}}{2} \\ 0 & 0 & \frac{\sqrt{3}}{2} & \frac{-1}{2} \end{bmatrix} \quad (2.27)$$

The transformation matrix is then applied to the dynamic stiffness matrix for each bar using

$$\mathbf{R}_{\text{rot}} = \mathbf{T}^T \mathbf{R} \mathbf{T} \quad (2.28)$$

where \mathbf{R}_{rot} is the dynamic stiffness matrix for each bar, rotated into the truss coordinate frame. After the rotation is complete, the connectivity tables for the truss can be developed. The connection-based and object-based connectivity tables for a truss are defined in equations (2.8) and (2.9) respectively. These tables can then be used with equation (2.10) to build the dynamic stiffness matrix for the assembly. Evaluating equation (2.10) for each of the three components gives

$$\mathbf{R} = \begin{bmatrix} R_{1,11} + R_{3,33} & R_{1,12} + R_{3,34} & R_{1,13} & R_{1,14} & R_{3,31} & R_{3,32} \\ R_{1,21} + R_{3,43} & R_{1,22} + R_{3,44} & R_{1,23} & R_{1,24} & R_{3,41} & R_{3,42} \\ R_{1,31} & R_{1,32} & R_{1,33} + R_{2,11} & R_{1,34} + R_{2,12} & R_{2,13} & R_{2,14} \\ R_{1,41} & R_{1,42} & R_{1,43} + R_{2,21} & R_{1,44} + R_{2,22} & R_{2,23} & R_{2,24} \\ R_{3,13} & R_{3,14} & R_{2,31} & R_{2,32} & R_{2,33} + R_{3,11} & R_{2,34} + R_{3,12} \\ R_{3,23} & R_{3,24} & R_{2,41} & R_{2,42} & R_{2,43} + R_{3,21} & R_{2,44} + R_{3,22} \end{bmatrix} \quad (2.29)$$

for the dynamic stiffness matrix of the truss. Note that the dynamic stiffness matrix in equation (2.29) maintains the same symmetry as the original object dynamic stiffness matrices and can now either be used in a simulation, or in the construction of another assembly.

Development of a Dynamic Span Model

Creating a span assembly follows the same processes used to create both trusses and compound bars. A span is a two-dimensional object created by using pins to join two trusses and a bar as shown in Figure 6. Because a span is two-dimensional, the bar model must be expanded from one-dimension to two-dimensions before it can be used. A span model has the same joint characteristics as a truss model, allowing the expanded bar model to be used instead of a beam model.

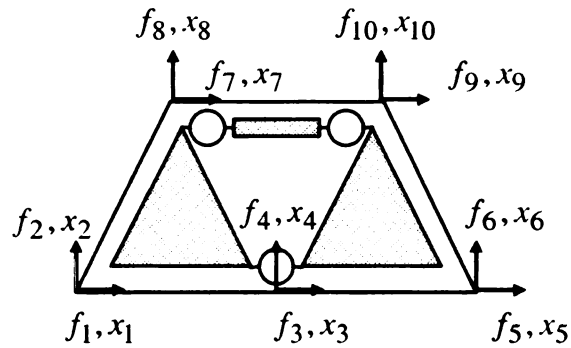


Figure 6 A span model is composed of two trusses, one bar, and three modular modeling connector elements.

In addition, the bar must go through the same rotation process as the bars required for the truss model. After the bar model has been prepared for use in the span model, the connectivity tables can be created. Using the connection definitions from Figure 6, the connection-based connectivity table can be defined as

$$\mathbf{CT_J} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 2 & 0 & 0 \\ 1 & 3 & 2 & 1 \\ 1 & 4 & 2 & 2 \\ 2 & 3 & 0 & 0 \\ 2 & 4 & 0 & 0 \\ 1 & 5 & 3 & 1 \\ 1 & 6 & 3 & 2 \\ 2 & 5 & 3 & 3 \\ 2 & 6 & 3 & 4 \end{bmatrix} \quad (2.30)$$

The object-based connectivity table can be created from the connection-based connectivity table shown in equation (2.30). The object-based connectivity table is shown in equation (2.31), below.

$$\mathbf{CT_O} = \begin{bmatrix} 1 & 2 & 3 & 4 & 7 & 8 \\ 2 & 3 & 4 & 5 & 9 & 10 \\ 7 & 8 & 9 & 10 \end{bmatrix} \quad (2.31)$$

With the object-based connectivity table and equation (2.10), the span model can be assembled in the same manner as the compound bar and truss models.

Chapter 3. Dynamic System Simulation

Solving for dynamic system response is an important part of the dynamic modeling process. A wide variety of tools exist for performing simulations and analyzing the results. These tools can be put to use on models generated by the i-EDA system, if the models can be transformed from i-EDA canonical form to a model format, like the transfer function, that can be solved directly. Transfer functions provide the ratio of output signal to input signal for a model. The most common form for a transfer function is

$$\frac{Y(s)}{U(s)} = G(s) \quad (3.1)$$

where, for a single input, single output (SISO) system, $Y(s)$ is the system output, $U(s)$ is the system input, and $G(s)$ is defined as

$$G(s) = \frac{a_n s^n + a_{n-1} s^{n-1} + \dots + a_1 s^1 + a_0}{b_m s^m + b_{m-1} s^{m-1} + \dots + b_1 s^1 + b_0} \quad (3.2)$$

with a_i and b_i as linear, constant coefficients that represent the system behavior. For a multi-input, multi-output (MIMO) system, the system is written as a matrix, $\mathbf{G}(s)$, where each entry represents the transfer function from the i -th input to the j -th output.

The i-EDA canonical form

$$\mathbf{R}(s)\mathbf{Y}(s) = \mathbf{U}(s) \quad (3.3)$$

bears an inverse relationship to the matrix form of equation (3.1), which is written as

$$\mathbf{Y}(s) = \mathbf{G}(s)\mathbf{U}(s) \quad (3.4)$$

where $\mathbf{R}(s)$ is the i-EDA dynamic stiffness matrix, $\mathbf{G}(s)$ is a matrix of transfer functions, and $\mathbf{U}(s)$ and $\mathbf{Y}(s)$ are the vectors of system inputs and outputs respectively. This

property allows all of the solution and analysis techniques that exist for transfer functions, as well as many other types of linear, constant-coefficient, input-output systems to be used on models generated by the i-EDA system.

Boundary conditions must be applied before the system can be inverted.

Inverting the system without applying boundary conditions will result in at least one eigenvalue of zero, and the simulation will predominantly display the rigid body mode of the object. The ability to apply boundary conditions as part of the solution process, and not part of the model formulation process is one advantage that the i-EDA modeling system has over conventional methods. With an i-EDA model, the boundary conditions applied can be unique to each situation, and do not require reformulation of the model.

For example, consider the one-dimensional bar model from Figure 3. As it is shown, the application of a force to either input will result in the translation and vibration of the bar. To study the vibration response alone, one end of the bar must have a zero displacement boundary condition applied. For a system with n degrees of freedom, the unconstrained dynamic stiffness will take the form

$$\mathbf{R}(s) = \begin{bmatrix} R_{11} & R_{12} & \cdots & R_{1n} \\ R_{21} & R_{22} & \cdots & R_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ R_{n1} & R_{n2} & \cdots & R_{nn} \end{bmatrix} \quad (3.5)$$

which, in general, will be full. To constrain this matrix, set the i^{th} output to zero. When that is done, the constrained matrix will take the form

$$\mathbf{R}_{\text{Constrained}} = \begin{bmatrix} R_{1,1} & \cdots & R_{1,i-1} & 0 & R_{1,i+1} & \cdots & R_{1,n} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ R_{i-1,1} & \cdots & R_{i-1,i-1} & 0 & R_{i-1,i+1} & \cdots & R_{i-1,n} \\ R_{i,1} & \cdots & R_{i,i-1} & 0 & R_{i,i+1} & \cdots & R_{i,n} \\ R_{i+1,1} & \cdots & R_{i+1,i-1} & 0 & R_{i+1,i+1} & \cdots & R_{i+1,n} \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ R_{n,1} & \cdots & R_{n,i-1} & 0 & R_{n,i+1} & \cdots & R_{n,n} \end{bmatrix} \quad (3.6)$$

Removing the column of zeros from the constrained dynamic stiffness matrix in equation (3.6) leaves the system of equations ill-formed, with n equations and $n - 1$ unknowns.

Additionally, the i^{th} row is now only a function of the remaining variables. The i^{th} row can then be removed from the system, leaving it as

$$\begin{bmatrix} R_{1,1} & \cdots & R_{1,i-1} & R_{1,i+1} & \cdots & R_{1,n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ R_{i-1,1} & \cdots & R_{i-1,i-1} & R_{i-1,i+1} & \cdots & R_{i-1,n} \\ R_{i+1,1} & \cdots & R_{i+1,i-1} & R_{i+1,i+1} & \cdots & R_{i+1,n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ R_{n,1} & \cdots & R_{n,i-1} & R_{n,i+1} & \cdots & R_{n,n} \end{bmatrix} \begin{bmatrix} Y_1 \\ \vdots \\ Y_{i-1} \\ Y_{i+1} \\ \vdots \\ Y_n \end{bmatrix} = \begin{bmatrix} U_1 \\ \vdots \\ U_{i-1} \\ U_{i+1} \\ \vdots \\ U_n \end{bmatrix} \quad (3.7)$$

If the reaction force at the constraint is desired, the i^{th} equation can be solved to provide that information after the rest of the system has been solved.

Applying a non-fixed boundary condition is similar to applying a fixed boundary condition. The primary difference is that instead of removing the i^{th} column, it needs to be evaluated and subtracted from the input force vector. After performing this operation, the system of equations will again be ill-formed and require the removal of the i^{th} row in order to solve the system. A system with a non-fixed boundary condition applied will take the form

$$\begin{bmatrix} R_{1,1} & \cdots & R_{1,i-1} & R_{1,i+1} & \cdots & R_{1,n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ R_{i-1,1} & \cdots & R_{i-1,i-1} & R_{i-1,i+1} & \cdots & R_{i-1,n} \\ R_{i+1,1} & \cdots & R_{i+1,i-1} & R_{i+1,i+1} & \cdots & R_{i+1,n} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ R_{n,1} & \cdots & R_{n,i-1} & R_{n,i+1} & \cdots & R_{n,n} \end{bmatrix} \begin{bmatrix} Y_1 \\ \vdots \\ Y_{i-1} \\ Y_{i+1} \\ \vdots \\ Y_n \end{bmatrix} = \begin{bmatrix} U_1 - R_{1,i}Y_i \\ \vdots \\ U_{i-1} - R_{i-1,i}Y_i \\ U_{i+1} - R_{i+1,i}Y_i \\ \vdots \\ U_n - R_{n,i}Y_i \end{bmatrix} \quad (3.8)$$

Like the fixed boundary condition case, if the reaction force at the i^{th} port is required, the i^{th} equation can be evaluated after the rest of the system has been solved.

After the boundary conditions have been applied, the system can be inverted. In general, this requires taking the matrix inverse of a matrix of polynomials. One definition of the matrix inverse states that

$$\mathbf{A}^{-1} = \frac{\mathbf{A}^*}{|\mathbf{A}|} \quad (3.9)$$

where \mathbf{A}^* is the adjoint of \mathbf{A} , and $|\mathbf{A}|$ is the determinant of \mathbf{A} [Phillips and Harbor, 2000]. Equation (3.9) proves to be the most useful method for finding the matrix inverse in this case because it does not rely on inspection, or equating polynomials. After taking the inverse of the i-EDA dynamic system matrix to find the transfer function for the system, all of the traditional simulation (Matlab, e.g.) and analysis (eigenvalue analysis, e.g.) tools can be used with the model.

Simulation of a One-Dimensional Bar

Consider the one-dimensional bar model shown in Figure 7 below.

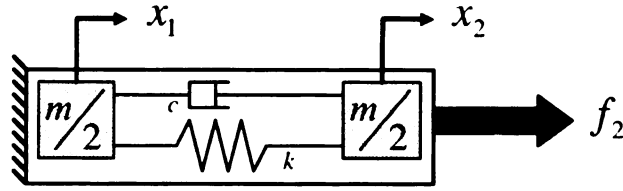


Figure 7 One dimensional bar model

Given the mass, stiffness, and damping properties in Table 1, below,

Table 1 Properties for the BA1x2 bar model in the i-EDA system

Property	Value	Units
Mass	1.6677E+02	kg
Stiffness	1.4700E+08	N/m
Damping	2.2143E+04	kg/s

the i-EDA dynamic stiffness matrix for a one dimensional bar from equation (2.20) can be evaluated to

$$\mathbf{R}(s) = \begin{bmatrix} 83.385s^2 + 2.2143e4s + 1.47e8 & -2.2143e4s - 1.47e8 \\ -2.2143e4s - 1.47e8 & 83.385s^2 + 2.2143e4s + 1.47e8 \end{bmatrix} \frac{\text{kg}}{\text{s}^2} \quad (3.10)$$

The boundary condition for this model shows that the left hand end of the bar (considered port 1), should be fixed, that is, have zero displacement as an output. Applying this boundary condition means that the first row and the first column (the entries in $\mathbf{R}(s)$ associated with port 1) are removed from the matrix, resulting in

$$\mathbf{R}(s) = \left[83.385s^2 + 2.214e4s + 1.47e8 \right] \frac{\text{kg}}{\text{s}^2} \quad (3.11)$$

for the system. Now that the boundary conditions have been applied, it is possible to invert the i-EDA dynamic stiffness matrix to form the transfer function for the system. In this case, $\mathbf{R}(s)$ is a scalar so a matrix inverse is not required. The transfer function for the system is

$$G(s) = \frac{1}{83.385s^2 + 2.214e4s + 1.47e8} \quad (3.12)$$

and can now be solved with existing simulation and analysis methods. Finding the eigenvalues for equation (3.12) gives

$$\lambda_{1,2} = -132.78 \pm 1321.1i \quad (3.13)$$

for the constrained system transfer function eigenvalues. Independently, a model can be developed for the one-dimensional bar model shown in Figure 7 using a bond graph to develop the state equations. For the bar model shown, the state equations are

$$\begin{aligned} \begin{bmatrix} \dot{x}_2 \\ \dot{v}_2 \end{bmatrix} &= \begin{bmatrix} 0 & 1 \\ -\frac{2k}{m} & -\frac{2c}{m} \end{bmatrix} \begin{bmatrix} x_2 \\ v_2 \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{2}{m} \end{bmatrix} [F_2] \\ [x_2] &= [1 \quad 0] \begin{bmatrix} x_2 \\ v_2 \end{bmatrix} + [0] [F_2] \end{aligned} \quad (3.14)$$

which can be evaluated using the properties in Table 1 as

$$\begin{aligned} \begin{bmatrix} \dot{x}_2 \\ \dot{v}_2 \end{bmatrix} &= \begin{bmatrix} 0 & 1 \\ -1.7629e6 & -2.6555e2 \end{bmatrix} \begin{bmatrix} x_2 \\ v_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1.1993e-2 \end{bmatrix} [f_2] \\ [x_2] &= [1 \quad 0] \begin{bmatrix} x_2 \\ v_2 \end{bmatrix} + [0] [f_2] \end{aligned} \quad (3.15)$$

Performing an eigenvalue analysis on equation (3.15) results in the eigenvalues

$$\lambda_{1,2} = -132.78 \pm 1321.1i$$

for the system. Comparing the eigenvalues given by equations (3.11) and (3.15) shows that there is no error in the formulation and solution process for a one-dimensional bar.

Simulation of a Compound Bar

Consider the compound bar model shown in Figure 8 below.

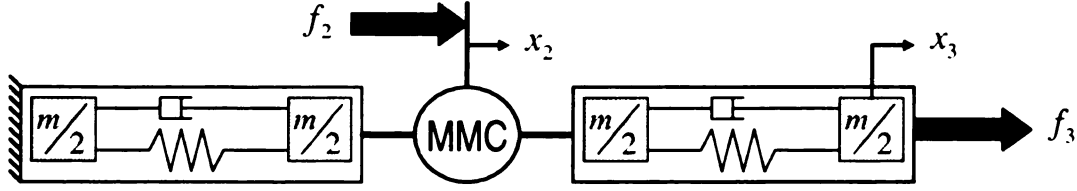


Figure 8 Constrained compound bar model

The compound bar model is going to be created from two bars with the mass, stiffness, and damping values from Table 1. Using these properties, the compound bar model in equation (2.24) can be evaluated as

$$\begin{bmatrix} 83.385s^2 + 22143s + 1.47e8 & -22143s - 1.47e8 & 0 \\ -22143s - 1.47e8 & 166.77s^2 + 44286s + 2.94e8 & -22143s - 1.47e8 \\ 0 & -22143s - 1.47e8 & 83.385s^2 + 22143s + 1.47e8 \end{bmatrix} \quad (3.16)$$

The boundary condition shown in Figure 8, indicates that the left hand end of the compound bar model (considered port 1) should be fixed. Applying the fixed condition to equation (3.16) yields

$$\begin{bmatrix} 166.77s^2 + 44286s + 2.94e8 & -22143s - 1.47e8 \\ -22143s - 1.47e8 & 83.385s^2 + 22143s + 1.47e8 \end{bmatrix} \quad (3.17)$$

as the constrained system matrix. Now that the boundary conditions have been applied, it is possible to invert the dynamic stiffness matrix to form the transfer function for the system. As shown in equation (3.9), the matrix inverse is formed by taking the adjoint and dividing by the determinant. The adjoint of the constrained system matrix (equation 3.17) is

$$\begin{bmatrix} 83.385s^2 + 22143s + 1.47e8 & 22143s + 1.47e8 \\ 22143s + 1.47e8 & 166.77s^2 + 4.4286s + 2.94e8 \end{bmatrix} \quad (3.18)$$

The determinant of the constrained system matrix is

$$13906s^4 + 7.3856e6s^3 + 4.9521e10s^2 + 6.51e12s + 2.1609e16 \quad (3.19)$$

combining equations (3.18) and (3.19) with equation (3.9) gives us the transfer function for the system. Performing a transfer function analysis on the transfer function yields the eigenvalues

$$\begin{aligned} \lambda_{1,2} &= -226.66 \pm 1719.9i \\ \lambda_{3,4} &= -38.889 \pm 717.52i \end{aligned} \quad (3.20)$$

as the unique eigenvalues for the system. Independently, a model can be developed for the compound bar model shown in Figure 8, using a bond graph to develop the state equations. Performing an eigenvalue analysis on the state equations yields

$$\begin{aligned} \lambda_{1,2} &= -226.66 \pm 1719.9i \\ \lambda_{3,4} &= -38.889 \pm 717.52i \end{aligned} \quad (3.21)$$

as the unique eigenvalues for the state space model. Comparing the eigenvalues in equations (3.20) and (3.21) shows that there is no error in the model formulation.

Truss and Span Simulation

The solution process for dynamic truss and span models, in general, is similar to the solution process for the compound bar model detailed above. One of the primary differences in the solution method is the number of boundary conditions that must be applied to remove the rigid body modes from the system model. Because truss and span models are two-dimensional, three boundary conditions must be applied to fully constrain the system and remove the horizontal, vertical, and rotational rigid body modes from the system. Removing multiple degrees of freedom from a model simply requires applying the process detailed in equations (3.5) through (3.7) for each degree of freedom that is removed. After the boundary conditions are applied, the adjoint and the determinant can

be used to invert the system and find the transfer function, and existing simulation and analysis methods can be used.

Modeling within the i-EDA system

The i-EDA system, uses a collection of internet agents to create dynamic system models. The i-EDA system topology is shown in Figure 9.

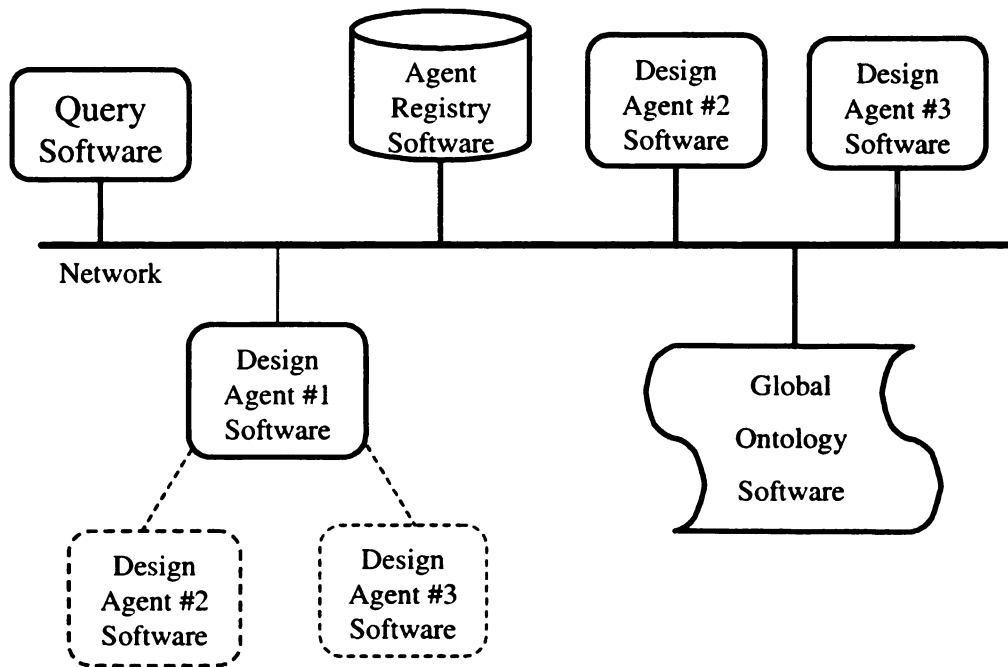


Figure 9 The i-EDA system topology

One of the most important distinctions to make within the i-EDA system is the capabilities of the various objects. The agent registry and ontology software interacts with all objects within the i-EDA system, identifying the agent addresses, and the list of available queries. The individual design agents can be broken into two categories. Component-level agents are agents which represent objects that are not composed of any other objects. One example of a component level agent is the one-dimensional bar.

Assembly-level agents are agents that create assemblies from information gathered from other agents. To an assembly-level agent, all other agents appear to be component-level agents, regardless of their actual status. Within each assembly-level agent, a dynamic stiffness model is assembled using the individual dynamic stiffness matrices from the components that make up the object, a specified connectivity table, and the algorithm described by equation (2.10). For example, a truss agent receives a query for the “Dynamics” of the truss-object that it represents. The truss agent then immediately sends out a set of sub-queries, including the original “Dynamics” query, to each of the agents that represent the bars that compose that truss. Upon receiving the response, the information is used, along with the connectivity table which is built into the truss agent, to assemble the component dynamic stiffness matrices into the dynamic stiffness matrix for the truss assembly. A second example of an assembly-level agent is a span. When a span agent receives the “Dynamics” query, it retrieves information from the component-level agents for the bar model and from the assembly-level agents for each of the two truss models. The span model can then be assembled using the dynamic stiffness matrices retrieved from each of the component objects, the connectivity table built into the span agent, and the algorithm defined in equation (2.10). One unique feature of the i-EDA system is the fact that every assembly considers its components as if they were component-level agents. This allows the system to create models of any shape and size without a change in the algorithms used to create the models and increases system’s ability to protect the proprietary information from which the models are built..

Client software provides the ability for users to interact with the i-EDA system. A client, using information from the global registry and ontology, send queries to agents

and either display the response to the user, or perform an analysis on the response. Client software is designed to perform specific functions, either through the actual client interface, or across the internet through a web-based interface. One type of client software queries the different agents for cost, length, weight, and size information, then displays the agents' responses to the user. A second type of client software can retrieve static or dynamic model information, apply boundary conditions, These clients apply boundary conditions and load cases and then run a simulation and display the results. The i-EDA Dynamic Response Client, shown in Figure 10, performs the solution procedure described in by equations (3.5) through (3.9).

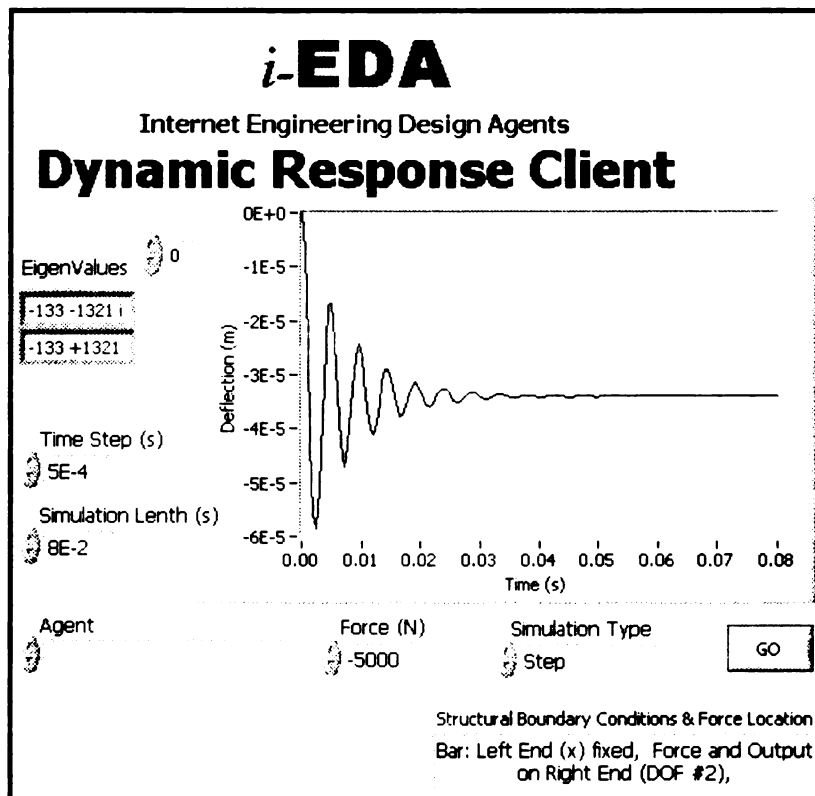


Figure 10 The i-EDA dynamic response client

Chapter 4. Conclusion

A method for communicating reusable dynamic system models that protects the proprietary information and interfaces easily with traditional modeling methods was successfully developed. Combining modular modeling algorithms with the i-EDA system gives engineers around the world access to dynamic model data around the clock. Proprietary information is protected by the input/output form of the model, and by the ability of the i-EDA system to treat every virtual product, whether an assembly or a component, as unique component. This represents a significant advancement in modeling technology, because it overcomes the need for a company to create and maintain a library of models that contains each possible input/output configuration. Additionally, it eliminates the need for legal agreements that protect proprietary information to be put into place before model information is transmitted.

The methods developed in this work, while designed for dynamic system models, can also be used for assembling linear finite element models. When creating the i-EDA canonical form, the common matrices used are the mass, damping, and stiffness matrices for the dynamic system. When considering the method with a finite element model, only the stiffness matrix would be present. This would allow the system to handle the assembly of static, and dynamic finite element models, as well as SEA, and many other types of linear finite element models.

In addition to providing a method for modeling dynamic systems over the internet, the solution presented introduces many areas for future work. One such area is the process of model condensation. Model condensation is the removal of input/output ports that are considered interior (and possibly proprietary) to the model. An example of this is

the third and fourth degrees of freedom in a span model (Figure 6). Removing those degrees of freedom would reduce the overall size of the span model, and would further protect the proprietary information of the model. In addition to limiting the size of the model, condensation of models significantly increases the difficulty associated with reverse engineering a model.

A second area where there are possibilities for future work is in the creation of algorithms for limiting system bandwidth to a particular range. One side effect of maintaining all of the system dynamics for each component involved in an assembly is the possibility that the frequency bandwidth the system occupies will become quite large. In many cases, only a very specific bandwidth is of interest to the modeler. For example, in a large electro-mechanical (or Mechatronic) system, there will be electrical components which react very quickly and have a time constant measured in milliseconds or microseconds. There will also be mechanical components, with comparatively large time constants, measured in seconds, or even minutes. When designing control algorithms for the assembly, the only time constants under consideration will be related to the mechanical portions of the system. When compared to the mechanical system, the electrical system seems to react instantaneously. Because of this, the ability to limit the bandwidth under consideration will improve the algorithms set forth in this work.

A third area where there are possibilities for future study is in the formulation of models for use within the i-EDA system. Because the i-EDA dynamic stiffness matrix can be easily converted into the transfer function of the system, there exists a strong possibility that the reverse is also true. This would allow companies to formulate models using existing, proven, techniques. (Note that proper port causality must still be

maintained). For example, an electrical network model could be formulated using bond graphs to generate the state equations for the system. The state-equations could then be transformed into the transfer function for the system and from there into i-EDA canonical form. If this was the case, companies would be able to use models that currently exist within their model libraries, instead of creating separate models from scratch. Additionally, companies would not have to invest in re-educating their model design staff with new modeling techniques.

Appendix A Dynamics Query and Response Syntax

Query Syntax: Dynamics+<PN>

Where <PN> is a string representing the component part number.

Response Syntax: { <Dynamics> } <units>

Where <Dynamics> is a string representing the contents of the i-EDA canonical form and <units> is a string representing the units for the response.

An example is shown below with the control characters (as defined by LabView) shown.

Query:

BB3x6+Dynamics

Response:

```
\s{\s[\s1777.572000\s157978.197483\s3.510000E+8\s]\s\s[\s-  
157978.197483\s-3.510000E+8\s]\s\s;\s\n\s\s[\s-157978.197483\s-  
3.510000E+8\s]\s\s[\s1777.572000\s157978.197483\s3.510000E+8\s]\s\s\s}\sk  
g\\(s^2)\s
```

The same example is shown again without the control characters shown.

Query:

BB3x6+Dynamics

Response

```
{ [ 1777.572000 157978.197483 3.510000E+8 ] [ -157978.197483 -  
3.510000E+8 ] ;  
[ -157978.197483 -3.510000E+8 ] [ 1777.572000 157978.197483  
3.510000E+8 ] } kg\\(s^2)
```

Please note that in this example the response has been wrapped to fit the page width. The true response has two lines. The first line ends at the line break following the semicolon. The second line ends following the units.

Appendix B Matlab Code

List of Included Programs

CreateTruss.m	Creates Trusses. Primary LabView Truss interface.
CreateSpan.m	Creates Spans. Primary LabView Span interface.
DynamicJoin.m	Actually performs the join operation of two or more dynamic systems.
AddDimension.m	Transforms a bar from a 1-D object to a 2-D object.
addpoly.m	Adds two polynomials of unequal length. Pads with zeros.
JoinMap.m	Transforms the Connectivity Table from join-based to object-based.
PostMultCell.m	Performs the matrix post-multiply operation with a cell array and a matrix.
PreMultCell.m	Performs the matrix pre-multiply operation with a matrix and a cell array.
RotateDynamicBar.m	Transforms the bar dynamics into the coordinate frame of the assembly.
cell2str.m	Converts a matlab cell array into a character string
SolveDynamics.m	Applies boundary conditions, inverts, and solves
ApplyBoundaryCond.m	Applies the specified boundary conditions
CellArrayAdjoint.m	Takes the adjoint of a cell array
CellArrayDet.m	Takes the determinant of a cell array

CreateTruss.m

```

function [TrussAscii]=CreateTruss(B1Ascii, B2Ascii, B3Ascii, Sines, Cosines)
%*****
%*
%* Author: Drew R. Reichenbach
%* Date: 04-04-2003
%* Group: i-EDA System Dynamics
%*
%* Function: CreateTruss.m v1.1
%* Inputs: B1Ascii - A vector of ASCII character codes from LabView
%*           which, when converted, form the i-EDA string for
%*           dynamic bar #1.
%*           B2Ascii - A vector of ASCII character codes from LabView
%*           which, when converted, form the i-EDA string for
%*           dynamic bar #2.
%*           B3Ascii - A vector of ASCII character codes from LabView
%*           which, when converted, form the i-EDA string for
%*           dynamic bar #3.
%*           Sines - A vector of double precision values equal to the
%*           directional sines for each bar.
%*           Cosines - A vector of double precision values equal to the
%*           directional cosines for each bar.
%* Outputs: TrussAscii - A vector of ASCII character codes to pass to
%*           LabView which, when converted, form the i-EDA
%*           string for the unconstrained dynamic truss.
%*
%* Purpose: CreateTruss(args) is the main program for connecting i-EDA
%*           Dynamic Bar models into an i-EDA dynamic Truss model.
%*
%* Required Matlab Functions:
%*   AddDimension.m - Turns 1-D bars into 2-D bars.
%*   RotatedDynamicBar.m - Rotates 2-D Dynamic Bars.
%*   DynamicJoin.m - Performs the i-EDA join process on 2 or more
%*                   dynamic objects.
%*   cell2str.m - Converts cell array objects to character strings.
%*
%* NOTES:
%*
%*****

%=====
% Declare and Initialize Variables
%=====
B1String=[]; %The string for Bar 1
B2String=[]; %The string for Bar 2
B3String=[]; %The string for Bar 3
Bar1ld={}; %The cell array for Bar 1
Bar2ld={}; %The cell array for Bar 2
Bar3ld={}; %The cell array for Bar 3
Bar1l={}; %The cell array for 2D bar 1
Bar2l={}; %The cell array for 2D bar 2
Bar3l={}; %The cell array for 2D bar 3
Bar1r={}; %The cell array for rotated 2D bar 1
Bar2r={}; %The cell array for rotated 2D bar 2

```

```

Bar3r={}; %The cell array for rotated 2D bar 3
JoinCT=[] %The join based nodal connectivity table
TrussArray={} %The Cell Array of Truss Polynomials

%=====
% Main Program
%=====

%-----
% Convert vectors of ASCII codes to strings
%-----
B1String=char(B1Ascii);
B2String=char(B2Ascii);
B3String=char(B3Ascii);

%-----
% Convert strings to Cell Arrays
%-----
Bar1ld=eval(B1String);
Bar2ld=eval(B2String);
Bar3ld=eval(B3String);

%-----
% Turn 1-D bars into 2-D bars
%-----
Bar1=AddDimension(Bar1ld);
Bar2=AddDimension(Bar2ld);
Bar3=AddDimension(Bar3ld);

%-----
% Rotate the 2-D Bars
%-----
Bar1r=RotateDynamicBar(Bar1, Sines(1), Cosines(1));
Bar2r=RotateDynamicBar(Bar2, Sines(2), Cosines(2));
Bar3r=RotateDynamicBar(Bar3, Sines(3), Cosines(3));

%-----
% Create the Join-Based Connectivity Table
%-----
JoinCT = [ 1 1 3 3; %Bar 1, DOF 1 to Bar 3, DOF 3
           1 2 3 4; %Bar 1, DOF 2 to Bar 3, DOF 4
           1 3 2 1; %Bar 1, DOF 3 to Bar 2, DOF 1
           1 4 2 2; %Bar 1, DOF 4 to Bar 2, DOF 2
           2 3 3 1; %Bar 2, DOF 3 to Bar 3, DOF 1
           1 4 3 2]; %Bar 2, DOF 4 to Bar 3, DOF 2

%-----
% Assemble The Truss
%-----
TrussArray=DynamicJoin(JoinCT, Bar1r, Bar2r, Bar3r);

%-----
% Convert the Cell Array to a String
%-----
TrussString=cell2str(TrussArray);

%-----

```

```

% Convert the String to ASCII Character codes
%-----
TrussAscii=double(TrussString)

%*****
%* End Of Program                                     *
%*                                                     *
%* Last Update: 04-09-2003                             *
%*                                                     *
%* v1.1:  Corrected the connectivity table to reflect proper node *
%*         numbering.                                           *
%* v1.0:  creation version                                       *
%*                                                     *
%*****

```

CreateSpan.m

```

function [SpanAscii]=CreateSpan(T1Ascii, T2Ascii, B3Ascii)
%*****
%*                                                     *
%* Author: Drew R. Reichenbach                             *
%*   Date: 04-07-2003                                       *
%*   Group: i-EDA System Dynamics                           *
%*                                                     *
%* Function: CreateSpan.m v1.1                               *
%*   Inputs: T1Ascii - A vector of ASCII character codes from LabView *
%*               which, when converted, form the i-EDA string for *
%*               dynamic truss #1.                             *
%*           T2Ascii - A vector of ASCII character codes from LabView *
%*               which, when converted, form the i-EDA string for *
%*               dynamic truss #2.                             *
%*           B3Ascii - A vector of ASCII character codes from LabView *
%*               which, when converted, form the i-EDA string for *
%*               the dynamic bar.                               *
%*   Outputs: SpanAscii - A vector of ASCII character codes to pass to *
%*               LabView which, when converted, form the i-EDA *
%*               string for the unconstrained dynamic span.     *
%*                                                     *
%* Purpose: CreateSpan(args) is the main program for connecting i-EDA *
%*           Dynamic Truss and Bar models into an i-EDA dynamic Span *
%*           model.                                             *
%*                                                     *
%* Required Matlab Functions:                                 *
%*   AddDimension.m - Turns 1-D bars into 2-D bars.           *
%*   RotatedDynamicBar.m - Rotates 2-D Dynamic Bars.          *
%*   DynamicJoin.m - Performs the i-EDA join process on 2 or more *
%*                   dynamic objects.                           *
%*   cell2str.m - Converts cell array objects to character strings. *
%*                                                     *
%* NOTES:                                                     *
%*                                                     *
%*****

%=====
% Declare and Initialize Variables
%=====

```

```

T1String=[]; %The string for Truss 1
T2String=[]; %The string for Truss 2
B3String=[]; %The string for the Bar
Bar3ld={}; %The cell array for the Bar
Truss1={}; %The cell array for 2D Truss 1
Truss2={}; %The cell array for 2D Truss 2
Bar3={}; %The cell array for the 2D Bar
JoinCT=[] %The join based nodal connectivity table
SpanArray={} %The Cell Array of Truss Polynomials

%=====
% Main Program
%=====

%-----
% Convert vectors of ASCII codes to strings
%-----
T1String=char(T1Ascii);
T2String=char(T2Ascii);
B3String=char(B3Ascii);

%-----
% Convert strings to Cell Arrays
%-----
Bar3ld=eval(B3String);
Truss1=eval(T1String);
Truss2=eval(T2String);

%-----
% Turn the 1-D bar into a 2-D bars
%-----
Bar3=AddDimension(Bar3ld);

%-----
% Rotate the 2-D Bars
%-----

%No rotation required to build a span.

%-----
% Create the Join-Based Connectivity Table
%-----
JoinCT = [ 1 1 0 0 ; %Truss 1, DOF 1 is unconnected
          1 2 0 0 ; %Truss 1, DOF 2 is unconnected
          1 3 2 1 ; %Truss 1, DOF 3 to Truss 2, DOF 1
          1 4 2 2 ; %Truss 1, DOF 4 to Truss 2, DOF 1
          2 3 0 0 ; %Truss 2, DOF 3 is unconnected
          2 4 0 0 ; %Truss 2, DOF 4 is unconnected
          1 5 3 1 ; %Truss 1, DOF 5 to Bar 3, DOF 1
          1 6 3 2 ; %Truss 1, DOF 6 to Bar 3, DOF 2
          2 5 3 3 ; %Truss 2, DOF 5 to Bar 3, DOF 3
          2 6 3 4 ]; %Truss 2, DOF 6 to Bar 3, DOF 4

%-----
% Assemble The Truss
%-----
SpanArray=DynamicJoin(JoinCT, Truss1, Truss2, Bar3);

```

```

%-----
% Convert the Cell Array to a String
%-----
SpanString=cell2str(SpanArray);

%-----
% Convert the String to ASCII Character codes
%-----
SpanAscii=double(SpanString)

%*****
%* End Of Program
%*
%* Last Update: 04-09-2003
%*
%* v1.1: Fixed a variable reference error.
%* v1.0: creation version
%*
%*****

```

DynamicJoin.m

```

function [CellOut] = DynamicJoin(JoinCT, Cell1, Cell2, varargin)
%*****
%*
%* Author: Drew R. Reichenbach
%* Date: 2-21-2003
%* Group: i-EDA System Dynamics
%*
%* Function: DynamicJoin.m v1.2
%* Inputs: JoinCT - A matrix containing the Join-based nodal
%* connectivity table.
%* Cell1 - A cell array of Laplace Domain Parameters for the
%* first Dynamic System
%* Cell2 - A cell array of Laplace Domain Parameters for the
%* second Dynamic System.
%* varargin - A cell array of any additional cell arrays for
%* extending the capability of the Dynamic Join
%* function beyond a single join operation. The
%* number of arguments will be checked vs. the
%* number of matrices handed to the Dynamic Join
%* function; a mismatch will trigger an error and
%* will end the join process.
%* Outputs: CellOut - A cell array of Laplace Domain Parameters for
%* the output composite Dynamic System.
%*
%* Purpose: DynamicJoin(JoinCT, Cell1, Cell2, varargin) performs the
%* join procedure on at least two dynamic systems as
%* specified in the JoinCT nodal connectivity table.
%* DynamicJoin will generate an error message if the number
%* of input cell arrays does not match the number of systems
%* indicated in the connectivity table.
%*
%* Required Matlab Functions:
%* JoinMap(JoinConnect) - converts modular modeling connector
%*

```

```

%*          data from a join-based system to an object based system.      *
%*          addpoly(Poly1,Poly2) - Adds two vectors representing           *
%*          polynomials in canonical form, independent of polynomial       *
%*          order.                                                          *
%*                                                                           *
%* NOTES: Join Based Nodal Connectivity Tables:                            *
%*          A connection based nodal connectivity table has each row       *
%*          defining the connections. For example, the JoinConnect         *
%*          matrix                                                         *
%*          JoinConnect = [ 1 1 2 2 ;                                     *
%*                          1 3 2 3 ;                                     *
%*                          1 2 2 1 ]                                     *
%*          defines a physical system where                                *
%*          join 1 (row 1):(component#1, port#1) & (component#2, port#2) *
%*          join 2 (row 2):(component#1, port#3) & (component#2, port#3) *
%*          join 3 (row 3):(component#1, port#2) & (component#2, port#1) *
%*          note that unconnected ports still need to have a row in the   *
%*          connectivity table. For example, the physical system where     *
%*          join 1 (row 1):(component#1, port#1) is unconnected           *
%*          join 2 (row 2):(component#1, port#2) & (component#2, port#1) *
%*          join 3 (row 3):(component#2, port#2) is unconnected           *
%*          would be defined by the JoinConnect matrix                    *
%*          JoinConnect = [ 1 1 0 0 ;                                     *
%*                          1 2 2 1 ;                                     *
%*                          2 2 0 0 ]                                     *
%*          and could be the coupling of two 1-D bars (see BarJoin1D.m)   *
%*                                                                           *
%*****

%The Process:
% Error Check Input (done)
% Repackage Input (done)
% Convert Connectivity Table (done)
% Process Join Process based on ObjectCT.
% Update process for "cell array of cell arrays"
% End.

%=====
% Declare and Initialize Variables
%=====
ObjectCT=[]; %The Object Based Nodal Connectivity Table
i=[]; %Iteration variable
j=[]; %Iteration variable
temp=[]; %Temporary Variable
InptArgs=[]; %The total number of input arguments containing cell arrays
NumObj=[]; %The number of object present in the Connectivity Table
ErrMsg=[]; %An Error Message variable
DynArray=[]; %A Cell Array of the Dynamic Cell Arrays

%=====
% Error Check Input
%=====
InptArgs=nargin-1; %find the total number of input cell arrays
NumObj=max(max(JoinCT(:,1:2:size(JoinCT,2))))); %find largest obj # in inpt
if InptArgs ~= NumObj %if # of Input Cell arrays is not = to Obj # from CT
    ErrMsg=['Number of Input Cell arrays does not match number of objects'];

```

```

    ErrMsg=[ErrMsg, ' in connectivity table.'];
    error(ErrMsg); %Display an error message
end

%=====
% Repackage Input
%=====
DynArray=cell(1,InptArgs); %Create an empty array of proper size
for i=1:InptArgs %Loop through all input cell arrays
    switch i
    case 1
        DynArray{i}=Cell1; %Put Cell1 in first cell
    case 2
        DynArray{i}=Cell2; %Put Cell2 in second cell
    otherwise
        DynArray{i}=varargin{(i-2)}; %Put cells from varargin in cells 3 to n
    end
end

%=====
% Define Nodal Connectivity
%=====
%-----
% Convert Join Based Connectivity Table to Object-based Connectivity Table
%-----
ObjectCT=JoinMap(JoinCT)

%=====
% Main Program
%=====

%-----
% The size of the output matrix is equal to the number of ports in the
% assembly. This is equal to the number of rows in the join-based
% nodal connectivity table.
%-----
CellOut=cell(size(JoinCT,1)) %Define the empty output cell array

%-----
% Initialize Output Cell Array
%-----
for i=1:size(CellOut,1) %Loop over the rows
    for j=1:size(CellOut,2) %Loop over the columns
        CellOut{i,j}=[0]; %insert zero polynomials into every cell
    end
end

%-----
% Generate the output cell array, one component at a time
%-----
for i=1:(size(ObjectCT,1)) %Loop over each object (row in ObjectCT)
    NumPorts=length(find(ObjectCT(i,:))); %The Number of Input Object Ports
                                         %is equal to the number of
                                         %Non-Zero columns in ObjectCT
    for j=1:NumPorts %Loop over each row in the result
        for k=1:NumPorts %Loop over each column in the result
            CellOut{ObjectCT(i,j),ObjectCT(i,k)}=...

```

```

        addpoly(CellOut{ObjectCT(i,j),ObjectCT(i,k)},DynArray{i}{j,k});
    %Add the jth row, kth column from the ith object to the output
    end
end
end

```

```

%*****
%* End Of Program
%*
%* Last Update: 04-09-2003
%*
%* v1.2: corrected array indexing problem for systems with components
%*       that have less than the maximum number of ports.
%* v1.1: corrected issue with output array initialization
%* v1.0: creation version (iteration on BarJoin1D.m)
%*
%*****

```

AddDimension.m

```

function [Dynamic2D]=AddDimension(Dynamic1D)
%*****
%*
%* Author: Drew R. Reichenbach
%* Date: 04-04-2003
%* Group: i-EDA System Dynamics
%*
%* Function: AddDimension.m v1.0
%* Inputs: Dynamic1D - An i-EDA cell array containing the dynamics of a
%*                  one dimensional bar model.
%* Outputs: Dynamic2D - An i-EDA cell array containing the dynamics of a
%*                  two dimensional bar model.
%*
%* Purpose: AddDimension(Dynamic1D) converts a standard one
%*          dimensional bar model into a two dimensional bar model.
%*          This is done by adding rows and columns on to the original
%*          cell array. The vector of inputs goes from [x1 x2] to
%*          [x1 y1 x2 y2]. The y-direction ports have only inertial
%*          dynamics at this time.
%*
%* Required Matlab Functions:
%*
%* NOTES: Output array should take the form:
%*
%*          [ [ (m/2)s^2+k    0    -k    0    ]
%*            [ 0    (m/2)s^2    0    0    ]
%*            [ -k    0    (m/2)s^2+k    0    ]
%*            [ 0    0    0    0    (m/2)s^2 ] ]
%*
%*****

%=====
% Declare and Initialize Variables
%=====
BarMass=[]; %The mass of the dynamic bar
Dynamic2D={}; %The output cell array

```

```

MassVector=[]; %The vector representing the x'' term for y-dir dynamics

%=====
% Main Program
%=====

%-----
% Retrieve the mass from the dynamic bar
%-----
BarMass=Dynamic1D{1,1}(1); %The mass is the first element stored in the
                           % first cell.
MassVector=[BarMass, 0, 0]; %Create the polynomial for inertia-only effects

%-----
% Create the output array
%-----
Dynamic2D=cell(4,4); %Create an empty output array of the proper size

for i=1:4 %loop over the rows
    for j=1:4 %loop over the columns
        if (i==j) %selects diagonal elements
            if mod(i,2)==0 %selects even diagonals
                Dynamic2D{i,j}=MassVector; %Inertia-only vector on even diag.
            else
                Dynamic2D{i,j}=Dynamic1D{1,1}; %Inertia + stiffness on odd diag
            end
        elseif (i==1 & j==3) | (i==3 & j==1) %select -k elements
            Dynamic2D{i,j}=Dynamic1D{1,2};
        else
            Dynamic2D{i,j}=[0]; %Put zeros everywhere else
        end
    end
end

%*****
%* End Of Program
%*
%* Last Update: 04-04-2003
%*
%* v1.0: creation version
%*
%*****

```

addpoly.m

```

function [PolyOut]=addpoly(Poly1,Poly2)
%*****
%*
%* Author: Drew R. Reichenbach
%* Date: 02-19-2003
%* Group: i-EDA System Dynamics
%*
%* Function: addpoly.m v1.0
%* Inputs: Poly1 - The first of two polynomials in vector form. The
%*             polynomial should be in decreasing powers of x,
%*             with the largest power of x first.(canonical form)
%*
%*****

```

```

%*          Poly2 - The second of two polynomials in vector form. The      *
%*                  polynomial should be in decreasing powers of x,      *
%*                  with the largest power of x first.(cannonical form)  *
%* Outputs: PolyOut - The polynomial obtained when the two input        *
%*                  polynomials are added togeather. The polynomial      *
%*                  is in decreasing powers of x, with the largest      *
%*                  power of x first. (cannonical form)                  *
%*                                                                    *
%* Purpose: addpoly(Poly1,Poly2) adds two single variable polynomials of *
%*                  unknown length. The polynomials should be stored as  *
%*                  vectors of coefficients of decreasing powers of the  *
%*                  independant variable. 2x+3 would be [2 3].          *
%*                                                                    *
%* Required Matlab Functions:                                           *
%*                                                                    *
%* NOTES:                                                                *
%*                                                                    *
%*****

%=====
% Declare and Initialize Variables
%=====
Order1=[]; %The order of the first polynomial
Order2=[]; %The order of the second polynomial
Pad=[]; %The pad (of zeros) to adjust the length of the smaller polynomial
PolyOut=[]; %The output polynomial

%=====
% Main Program
%=====

%-----
% Find the order of the two polynomials
%-----
Order1=size(Poly1,2);
Order2=size(Poly2,2);

%-----
% Pad the smaller polynomial then add
%-----
if Order1 > Order2 %The first polynomial is larger
    Pad=zeros(1,(Order1-Order2)); %Create the Pad
    Poly2=[Pad, Poly2]; %Pad the smaller polynomial
    PolyOut=Poly1+Poly2; %Add the polynomials
elseif Order2 > Order1 %The second polynomial is larger
    Pad=zeros(1, (Order2-Order1)); %Create the Pad
    Poly1=[Pad, Poly1]; %Pad the smaller polynomial
    PolyOut=Poly1+Poly2; %Add the polynomials
else %The polynomials are equal in size
    PolyOut=Poly1+Poly2; %Add the polynomials
end

%*****
%* End Of Program *
%* *
%* Last Update: 02-19-2003 *
%* *

```

```

%* v1.0:  creation version
%*
%*****

```

JoinMap.m

```

function ObjectConnect=JoinMap(JoinConnect)
%*****
%*
%* Author: Clark J. Radcliffe
%*   Date: 05-14-2002
%*  Group: i-EDA System Dynamics
%*
%* Function: JoinMap.m v 1.1
%*   Inputs: JoinConnect - A Join-Centric Nodal Connectivity Table (see
%*                                   notes for details).
%*   Outputs: ObjectConnect - An Object-Centric Nodal Connectivity Table
%*                                   (see nodes for details).
%*
%* Purpose: JoinMap(JoinConnect) converts from the join-based nodal
%*           nodal connectivity table that is easily understood by users
%*           to the object based nodal connectivity table which is more
%*           practical for programming.
%*
%* Required Matlab Functions:
%*
%* NOTES: Join Based Nodal Connectivity Tables:
%*       A connection based nodal connectivity table has each row
%*       defining the connections.  For example, the JoinConnect
%*       matrix
%*       JoinConnect = [ 1 1 2 2 ;
%*                       1 3 2 3 ;
%*                       1 2 2 1 ]
%*       defines a physical system where
%*       join 1 (row 1):(component#1, port#1) & (component#2, port#2)
%*       join 2 (row 2):(component#1, port#3) & (component#2, port#3)
%*       join 3 (row 3):(component#1, port#2) & (component#2, port#1)
%*
%* Object Based Nodal Connectivity Tables:
%*       An object based nodal connectivity table has each row
%*       corresponding to a component, and each column corresponds to an
%*       object's port.  For example, the JoinConnect Matrix above
%*       would convert to the ObjectConnect Matrix
%*       ObjectConnect = [ 1 3 2 ;
%*                         3 1 2 ]
%*       which is the equivalent of saying that
%*       For component #1: port1=>join1, port2=>join3, port3=>join2
%*       For component #2: port1=>join3, port2=>join1, port3=>join2
%*
%*****

%=====
% Declare and Initialize Variables
%=====
MaxComp=[]; %The largest component number
MaxDOF=[]; %The largest component port #

```

```

ObjectConnect=[]; %The object based nodal connectivity table

%=====
% Main Program
%=====

%-----
% Find the number of components and DOF involved
%-----
MaxComp=max(max(JoinConnect(:,1:2:size(JoinConnect,2)))));
MaxDOF=max(max(JoinConnect(:,2:2:size(JoinConnect,2)))));

%-----
% Create the empty resultant matrix
%-----
ObjectConnect=zeros(MaxComp,MaxDOF);

%-----
% Compute Object based connectivity from Join based connectivity
%-----
for i=1:size(JoinConnect,1) %Loop over list of connections
    for j=1:(size(JoinConnect,2)/2) %Loop over ports in connection
        a=(2*j)-1;
        CJ=JoinConnect(i,a:a+1);
        if CJ(1)~=0
            ObjectConnect(CJ(1),CJ(2))=i;
        end
    end
end

%*****
%* End Of Program *
%* *
%* Last Update: 02-19-2003 *
%* *
%* v1.1: Reformatted Header and Comment Information (D.R.R) *
%* v1.0: Creation Version (C.J.R.) *
%* *
%*****

```

PostMultCell.m

```

function [ArrayOut]=PostMultCell(CellArray, Matrix)
%*****
%*
%* Author: Drew R. Reichenbach *
%* Date: 04-04-2003 *
%* Group: i-EDA System Dynamics *
%* *
%* Function: PostMultCell.m v1.0 *
%* Inputs: CellArray - A NxM Cell Array of numeric values. Cell *
%*           contents are 1-dimensional numeric arrays. *
%*           Matrix - A MxM square matrix of scalar values. *
%* Outputs: ArrayOut - A NxM Cell array that has been through the matrix *
%*           multiply operation. Post-multiplication ONLY. *
%*
%*****

```

```

%* Purpose: PostMultCell(Matrix, CellArray) performs the matrix      *
%*           post-multiplication operation on a 2-dimensional scalar  *
%*           matrix and a 2-dimensional cell array of 1-dimensional  *
%*           numeric arrays which represent polynomial coefficients.  *
%*                                                                    *
%* Required Matlab Functions:                                         *
%*           addpoly.m - Adds polynomials of non-similar order.      *
%*                                                                    *
%* NOTES:                                                             *
%*                                                                    *
%*****

%=====
% Declare and Initialize Variables
%=====
N=[]; %The number of rows in the input cell array
M=[]; %The number of columns in the input cell array
temp1=[]; %A temporary variable
temp2=[]; %A second temporary variable
CellOut={}; %The output cell array
tempvect=[]; %A temporary vector (one row of Matrix)

%=====
% Main Program
%=====

%-----
% Identify the size of the cell array
%-----
[N, M] = size(CellArray);

%-----
% Check matrix size, create output array
%-----
[temp1, temp2]=size(Matrix);
if temp1~=M | temp2~=M %Error Check Input
    error('Matrix and Cell Array Inner Dimensions do not match');
end

ArrayOut=cell(size(CellArray)); %Create output array

%-----
% Perform Matrix Post-multiplication
%-----
for i=1:N %Loop over output array rows
    for j=1:M %Loop over output array columns
        tempvect=[0];
        for k=1:N %Loop over the elements to be added
            tempvect=addpoly(tempvect, (CellArray{i,k}*Matrix(k,j)));
        end
        ArrayOut{i,j}=tempvect;
    end
end

%*****
%* End Of Program
%*

```

```

%* Last Update: 04-04-2003
%*
%* v1.0: creation version
%*
%*****

```

PreMultCell.m

```

function [ArrayOut]=PreMultCell(Matrix, CellArray)
%*****
%*
%* Author: Drew R. Reichenbach
%* Date: 04-04-2003
%* Group: i-EDA System Dynamics
%*
%* Function: PreMultCell.m v1.0
%* Inputs: Matrix - A NxN square matrix of scalar values.
%*          CellArray - A NxM Cell Array of numeric values. Cell
%*                   contents are 1-dimensional numeric arrays.
%* Outputs: ArrayOut - A NxM Cell array that has been through the matrix
%*                   multiply operation. Pre-multiplication ONLY.
%*
%* Purpose: PreMultCell(Matrix, CellArray) performs the matrix
%*          pre-multiplication operation on a 2-dimensional scalar
%*          matrix and a 2-dimensional cell array of 1-dimensional
%*          numeric arrays which represent polynomial coefficients.
%*
%* Required Matlab Functions:
%*          addpoly.m - Adds polynomials of non-similar order.
%*
%* NOTES:
%*
%*****

%=====
% Declare and Initialize Variables
%=====
N=[]; %The number of rows in the input cell array
M=[]; %The number of columns in the input cell array
temp1=[]; %A temporary variable
temp2=[]; %A second temporary variable
CellOut={}; %The output cell array
tempvect=[]; %A temporary vector (one row of Matrix)

%=====
% Main Program
%=====

%-----
% Identify the size of the cell array
%-----
[N, M] = size(CellArray);

%-----
% Check matrix size, create output array
%-----

```

```

[templ, temp2]=size(Matrix);
if templ~=N | temp2~=N %Error Check Input
    error('Matrix and Cell Array Inner Dimensions do not match');
end

ArrayOut=cell(size(CellArray)); %Create output array

%-----
% Perform Matrix Pre-multiplication
%-----
for i=1:N %Loop over output array rows
    for j=1:M %Loop over output array columns
        tempvect=[0];
        for k=1:N %Loop over the elements to be added
            tempvect=addpoly(tempvect, (Matrix(i,k)*CellArray{k,j}));
        end
        ArrayOut{i,j}=tempvect;
    end
end
end

%*****
%* End Of Program
%*
%* Last Update: 04-04-2003
%*
%* v1.0: creation version
%*
%*****

```

RotateDynamicBar.m

```

function [RotatedBar]=RotateDynamicBar(Bar, Sine, Cosine)
%*****
%*
%* Author: Drew R. Reichenbach
%* Date: 04-04-2003
%* Group: i-EDA System Dynamics
%*
%* Function: RotateDynamicBar.m v1.1
%* Inputs: Bar - A Cell array containing the vectors of polynomial
%*           coefficients that make up a two dimensional i-EDA
%*           dynamic bar model.
%*           Sine - A floating point value that represents the sine of the
%*           angle to which the bar must be rotated.
%*           Cosine - A floating point value that represents the cosine of
%*           angle to which the bar must be rotated.
%* Outputs: RotatedBar - A Cell array containing the vectors of
%*           polynomial coefficients that make up a two
%*           dimensional i-EDA dynamic bar model after it
%*           has been rotated.
%*
%* Purpose: RotateDynamicBar(args) uses the directional sine and cosine
%*           to rotate the model of a dynamic bar so that it can be used
%*           in either a dynamic truss or a dynamic span.
%*
%* Required Matlab Functions:
%*
%*****

```

```

%*          PreMultCell.m - Performs the matrix pre-multiply operation      *
%*          with a NxN numeric matrix and a NxM cell array of numeric      *
%*          vectors.                                                         *
%*          PostMultCell.m - Performs the matrix post-multiply operation    *
%*          with a NxM cell array of numeric vectors and a MxM numeric      *
%*          matrix.                                                           *
%*                                                                           *
%* NOTES:                                                                    *
%*                                                                           *
%*****

%=====
% Declare and Initialize Variables
%=====
T=[]; %The rotational matrix, T
HalfRot={}; %The half rotated cell array (pre multiplied by T')
M=[]; %The Number of rows in the input cell array
N=[]; %The Number of columns in the input cell array
DirtyVect=[]; %The vector which may have leading zeros
CleanVect=[]; %The vector which does not have leading zeros

%=====
% Main Program
%=====

%-----
% Define the Two Dimensional rotation matrix, T
%-----

T=[ Cosine   Sine    0    0 ;
   -1*Sine Cosine    0    0 ;
     0        0   Cosine  Sine ;
     0        0  -1*Sine Cosine ];

%-----
% To rotate the dynamics, use T'*R*T
%   Done in two steps: Pre-multiply by T'
%                       then Post-multiply by T
%-----

%perform the first part of the rotation
HalfRot=PreMultCell(T',Bar);

%perform the second part of the rotation
RotatedBar=PostMultCell(HalfRot,T);

%-----
% Remove any incidental leading zeros
%-----
[M,N]=size(RotatedBar); %determine the size of the array

for i=1:M %Loop over the rows
    for j=1:N %Loop over the columns
        DirtyVect=RotatedBar{i,j};
        if isempty(find(DirtyVect)) %If the vector has no Non-Zero entries
            CleanVect=[0]; %Store a single zero
        end
    end
end

```

```

        else %Otherwise
            CleanVect=DirtyVect(min(find(DirtyVect)):length(DirtyVect));
            %Store the input vector from the first non-zero entry to
            % the last element in the vector
        end
        RotatedBar{i,j}=CleanVect; %Put the cleaned vector back in the cell
    end
end

%*****
%* End Of Program *
%* *
%* Last Update: 04-09-2003 *
%* *
%* v1.1: Added code to remove leading zeros from entries that only *
%* stiffnesses, but were padded due to the matrix multiply *
%* operation. *
%* v1.0: creation version *
%* *
%*****

```

cell2str.m

```

function [StrOut] = cell2str(CellIn)
%*****
%*
%* Author: Drew R. Reichenbach *
%* Date: 11-24-2002 *
%* Group: i-EDA System Dynamics *
%* *
%* Function: cell2str.m v1.1 *
%* Inputs: CellIn - A cell array of unknown size that is to be converted *
%* Outputs: StrOut - A single string which can be reconverted to a cell *
%* array with the command str2num *
%* *
%* Purpose: cell2str(CellIn) takes a cell array of unknown size (2D only) *
%* and converts it into a string. The string includes all *
%* of the required brackets to be easily reconverted into a *
%* cell array using the str2num() function. *
%* *
%* Required Matlab Functions: *
%* size() - finds the n-dimensional size of the input args. *
%* sprintf(format,arg) - converts arg into a string array using *
%* the format string. *
%* *
%* NOTES: cell2str was designed to produce an output string properly *
%* formatted for use within the i-EDA Dynamic System *
%* models. The formatting can be adjusted by changing *
%* format string, according to the information found in *
%* the Matlab help file for sprintf.m *
%* *
%*****

%=====
% Declare and Initialize Variables
%=====

```

```

i = []; %Row Dimension Counter
j = []; %Column Dimension Counter
format = ['%E ']; %Format the output string to have Exponential Notation
StrOut = []; %The Output String (begins empty)
CellRow = []; %The Number of rows in the Cell Array
CellCol = []; %The Number of columns in the Cell Array

%=====
% Main Program
%=====

%-----
% Determine the size of the input cell array
%-----
[CellRow,CellCol] = size(CellIn);

%-----
% Create the output string by looping through each cell
%-----
StrOut = ['{']; %Begin the string with the cell array curly brace
for i=1:CellRow %Loop Through the Array Rows
    for j=1:CellCol %Loop Through the Array Columns
        StrOut=[StrOut,' [ ',sprintf(format,CellIn{i,j}),' ] '];
        %Append the contents of the (i,j)th cell to the string
    end
    StrOut=[StrOut,' ; ']; %End the ith row with a semi-colon
end
StrOut=[StrOut,' }']; %Close the string with the cell array curly brace

%*****
%* End Of Program *
%* *
%* Last Update: 12-05-2002 *
%* *
%* v1.1: added a space to the format string to correct format error *
%* v1.0: creation version *
%* *
%*****

```

SolveDynamics.m

```

function [Solution,TimeVector,lamda]=SolveDynamics(SYSAscii, InputDOF,
OutputDOF, FixedDOF, ITypeAscii, IValue,dt,Tsim)
%*****
%* *
%* Author: Drew R. Reichenbach *
%* Date: 04/11/2003 *
%* Group: i-EDA System Dynamics *
%* *
%* Function: Solvedynamics.m v1.0 *
%* Inputs: SYSAscii - A Vector of ASCII character codes that contain *
%* the dynamics for the system being solved. *
%* InputDOF - A double precision value that specifies which DOF *
%* will have the input applied to it *
%* OutputDOF - A double precision value that specifies which DOF *
%* the output will be recorded for. *
%* *
%*****

```

```

%*          FixedDOF - A vector of integer vauses that specify which DOF *
%*                      are going to be fixed (Boundary Conditions) *
%*          ITypeAscii - An Optional input equal to: A vector of ASCII *
%*                      character codes that convert to a string to *
%*                      specify what type of input is being applied. *
%*                      Current Options are: Impulse and Step. Default *
%*                      value is step. *
%*          IValue - An optional input equal to a vector of applied input *
%*                      magnitudes. Default value is unity. *
%*          dt - The time step for running the simulation. *
%*          Tsim - The final simulation time. *
%* Outputs: Solution - A Nx1 vector of double precicion values that are *
%*                      the result of system simulation. *
%*          TimeVector - A Nx1 vector of double precision values that are *
%*                      the time at which each Solution entry occurs. *
%*          EigV - A complex vector containing the eigenvalues for the *
%*                      constrained system. *
%* *
%* Purpose: SolveDynamics(args) solves the i-EDA dynamics problem. *
%*          Inputs are applied at a specified location and outputs *
%*          are measured at the specified location. Boundary *
%*          conditions are applied before inverting the system. *
%* *
%* Required Matlab Functions: *
%*          CellArrayDet.m - Determinant of a cell array. Recursive. *
%*          CellArrayAdjoint.m - Adjoint of a cell array. *
%*          ApplyBoundaryCond.m - Applies fixed boundary conditions *
%* *
%* NOTES: Simulation times are user specified, because eigenvalue *
%*          analysis was not working properly. *
%* *
%*****

%=====
% Declare and Initialize Variables
%=====
SYSunConst={}; %The input system in cell array form
SYS={}; %The constrained system
IType=[]; %A String that represents the type of input to apply to SYS
idof=[]; %The vector of inputs adjusted to fit the constrained system
odof=[]; %The vector of outputs adjusted to fit the constrained system
NUM={}; %A Cell array of Transfer Function Numerator Polynomials
CharPoly=[]; %The system characteristic polynomial
DEN={}; %A Cell array of Transfer Function Denominator Polynomials
TFSYS=[]; %A TF object (LTI OBJECT) for the constrained system
lamda=[]; %The constrained system eigenvalues
sigma=[]; %The real part of the eigenvalues
omega=[]; %The imaginary part of the eigenvalues
TimeVector=[]; %The simulation time vector
U=[]; %A matrix of forcing functions for LSIM
Y=[]; %A full fledged output matrix from LSIM
Solution=[]; %The parsed output vector.

%=====
% Main Program
%=====

```

```

%-----
% Apply default value if necessary
%-----
if nargin < 5
    ITypeAscii=double(['Step']); %Default Value for IType is Step response
end
if nargin < 6
    for i=1:length(InputDOF)
        IValue(i)=1; %Default value for IValue is unity
    end
end

%-----
% Convert the ASCII vectors to character strings
%-----
SYSunConst=eval(char(SYSAscii)); %convert the input system to a cell array
IType=char(ITypeAscii); %Convert the input type to a string

%-----
% Error Check the input
%-----
if nargin < 4
    error('Not Enough Input Arguments!');
elseif size(SYSunConst,1)~=size(SYSunConst,2)
    error('Input System Is Not Square');
elseif size(SYSunConst,1) <= length(FixedDOF)
    error('The system has no unconstrained DOF');
end

%-----
% Constrain the system and adjust the i/o vectors
%-----
[SYS,idof,odof]=ApplyBoundaryCond(SYSunConst,InputDOF,OutputDOF,FixedDOF);

%-----
% Invert the system into Y=G(s)*U form
%-----
switch size(SYS,1)
case 1
    %After constraint, the system is SISO
    NUM=[1];
    DEN=SYS{1,1};
otherwise
    %After constraint, the system is MIMO
    NUM=CellArrayAdjoint(SYS);
    CharPoly=CellArrayDet(SYS);
    %Fill the DEN cell array with the Determinant
    DEN=cell(size(NUM));
    for i=1:size(NUM,1)
        for j=1:size(NUM,2)
            DEN{i,j}=CharPoly;
        end
    end
end
end

%-----

```

```

% Create the Transfer Function for the system
%-----
TFSYS=tf(NUM,DEN);

%-----
% Create the Time Vector for use with LSIM
%-----
clear i          %clear i for use as the imaginary operator
lamda=eig(TFSYS)'; %find the eigenvalues
dt
Tsim

%Calculate Simulation Time
TimeVector=0:dt:Tsim;

%-----
% Prepare U(t), the LSIM input vector
%-----
U=zeros(length(TimeVector),size(SYS,1)); %initialize the forcing function

%insert the forcing values (from IValue)
for i=1:length(idof) %loop over the input degrees of freedom
    switch IType
        case 'Step'
            U(:,idof(i))=IValue(i); %create a step input on the idof(i)th input

        case 'Impulse'
            U(1,idof(i))=IValue(i); %create an impulse input on the idof(i)th
                                   %input

        otherwise
            error(['Input Type ',IType,' is not defined'])
        end
    end
end

%-----
% Run the Simulation
%-----
Y=lsim(TFSYS,U,TimeVector);

%-----
% Parse the output vector to only include the desired columns
%-----
Solution=zeros(length(TimeVector),length(odof));
%initialize the output vector to be TxODOF

for i=1:length(odof)
    Solution(:,i)=Y(:,odof(i));
end
Solution=Solution';

%*****
%* End Of Program                                     *
%*                                                    *
%* Last Update: 04-11-2003                             *
%*                                                    *
%* v1.0:  creation version                               *
%*                                                    *

```

...

A

2

1

1

120

3

;

...

?

9

•

14

endo

3

1

...

...

3

2

...

3

```
%*****
```

ApplyBoundaryCond.m

```
function
[SYSOut, InputDOFOut, OutputDOFOut]=ApplyBoundaryCond(SYS, InputDOF, OutputDOF, FixedDOF)
```

```
%*****
```

```
%*
```

```
%* Author: Drew R. Reichenbach %*
```

```
%* Date: 04-12-2003 %*
```

```
%* Group: i-EDA System Dynamics %*
```

```
%*
```

```
%* Function: ApplyBoundaryCond.m v1.0 %*
```

```
%* Inputs: SYS - A NxN Cell Array of vectors. Each Vector contains the %*
```

```
%* coefficients to a polynomial in terms of s that %*
```

```
%* represents the system dynamics from the i-th output to %*
```

```
%* the j-th input. R(s), inverse of TF, G(s). %*
```

```
%* InputDOF - A vector of integers that list the DOF to which an %*
```

```
%* input is going to be applied. %*
```

```
%* OutputDOF - A vector of integers that lists the DOF at which %*
```

```
%* the output is going to be measured. %*
```

```
%* FixedDOF - A vector of integers that lists the DOF at which a %*
```

```
%* fixed boundary condition (output==0) is going to %*
```

```
%* be applied. %*
```

```
%* Outputs: SYSOut - A MxM Cell Array of vectors obtained by removing the %*
```

```
%* rows and columns for the DOF that are fixed. %*
```

```
%* InputDOFOut - A vector of integers that list the DOF of the %*
```

```
%* constrained system (SYSOut) to which an input %*
```

```
%* is going to be applied. %*
```

```
%* OutputDOFOut - A vector of integers that lists the DOF of the %*
```

```
%* constrained system (SYSOut) at which the %*
```

```
%* output is going to be measured. %*
```

```
%*
```

```
%* Purpose: ApplyBoundaryCond(args) performs several tasks related to the %*
```

```
%* related to the application of the fixed boundary condition %*
```

```
%* to a model written in the i-EDA canonical form. First, the %*
```

```
%* function removes the rows and columns of the R(s) array that %*
```

```
%* correspond to the DOF which are being fixed. Then, the %*
```

```
%* vectors that correspond to the input and output DOF are %*
```

```
%* adjusted to be in terms of the DOF of the output constrained %*
```

```
%* system. %*
```

```
%*
```

```
%* Required Matlab Functions: %*
```

```
%*
```

```
%* NOTES: %*
```

```
%*
```

```
%*****
```

```
%=====
```

```
% Declare and Initialize Variables
```

```
%=====
```

```
N=[]; %The size of the unconstrained system
```

```
RemainDOF=[]; %The DOF remaining after boundary conditions are applied
```

```
trash=[]; %A variable for storing data that does not need to be used
```

```

%=====
% Main Program
%=====

%-----
% Error check the input
%-----
if size(SYS,1)~=size(SYS,2)
    %if the number of rows and columns are not the same
    %the input system is not square, and cannot be solved.
    error('Input System Array is not square')
elseif size(SYS,1) < length(FixedDOF)+length(InputDOF)
    %if the number of DOF in the input system is less
    %than the number of DOF to be fixed and the number of inputs applied
    %then the system cannot be solved (no inputs).
    error('No Unconstrained DOF Remain.  Decrease the number of Fixed DOF');
end

%-----
% Form the Transformation Vector
%-----
N=size(SYS,1); %find the size of the unconstrained system
RemainDOF=(1:N); %Before BC application, all DOF are present

%-----
% Replace the Fixed DOF with zeros in the vector of DOF
%-----
for i=1:length(FixedDOF) %loop over the elements of the FixedDOF vector
    RemainDOF(FixedDOF(i))=0; %put a zero in place of the fixed DOF
end

%-----
% Find the vector of DOF that will remain
%-----
[trash,RemainDOF]=find(RemainDOF);

%-----
% Remove the FixedDOF rows and columns from SYS
%-----
M=length(RemainDOF); %find the size of the constrained array
SYSOut=cell(M,M); %initialize the constrained array

for i=1:M %loop over output array rows
    I=RemainDOF(i); %relate the output and input array rows
    for j=1:M %loop over output array columns
        J=RemainDOF(j); %relate the output and input array columns
        %fill the constrained array cell by cell.
        SYSOut{i,j}=SYS{I,J}; %build the output array
    end
end

%-----
% Adjust the Input DOF matrix to account for the missing DOF
% Adjust the Output DOF matrix to account for the missing DOF
%-----
InputDOFOut=InputDOF; %Initialize InputDOFOut
OutputDOFOut=OutputDOF; %Initialize OutputDOFOut

```

```

for i=1:length(FixedDOF)
    %Adjust Input DOF
    for j=1:length(InputDOF)
        if FixedDOF(i) < InputDOF(j)
            InputDOFOut(j)=InputDOFOut(j)-1;
        end
    end
end

%Adjust Output DOF
for k=1:length(OutputDOF)
    if FixedDOF(i) < OutputDOF(k)
        OutputDOFOut(k)=OutputDOFOut(k)-1;
    end
end
end

%*****
%* End Of Program
%*
%* Last Update: 04-12-2003
%*
%* v1.0: creation version
%*
%*****

```

CellArrayAdjoint.m

```

function [CellAdjOut]=CellArrayAdjoint(InputArray)
%*****
%*
%* Author: Drew R. Reichenbach
%* Date: 04/11/2003
%* Group: i-EDA System Dynamics
%*
%* Function: CellArrayAdjoint.m v1.0
%* Inputs: InputArray - A Cell Array of numeric vectors. Each vector
%* contains the coefficients of polynomials in
%* decreasing order.
%* Outputs: CellDet - A Cell Array containing the Adjoint of the Input
%* Cell Array. Each entry is a vector containing
%* the coefficients of polynomials in decreasing
%* order.
%*
%* Purpose: CellArrayAdjoint(args) takes the Adjoint of an input cell
%* array. The Adjoint definition used is the transpose of the
%* matrix of cofactors. (see notes for ref).
%*
%* Required Matlab Functions:
%* CellArrayDet.m - Determinant of a cell array. Recursive.
%*
%* NOTES: Adjoint definition taken from:
%* Phillips, Charles L. and Harbor, Royce D., 2000, "Feedback Control
%* Systems", Fourth Edition, Prentice Hall, Upper Saddle River, NJ.
%* Pg. 630.
%*
%*

```

```

%*****

%=====
% Declare and Initialize Variables
%=====
M=[]; %Number of rows in the input array
N=[]; %Number of columns in the input array
Minor={}; %The minor array
m=[]; %The size of the minor array

%=====
% Main Program
%=====

%-----
% Find the size of the input array
%-----
[M,N]=size(InputArray);

%-----
% Error Check the Input
%-----
if M~=N
    error('Input Array Must Be Square');
end

%-----
% Take The Adjoint
%-----
CellAdjOut=cell(M,M); %Initialize the output array

for I=1:M %Loop over the input array rows
for J=1:M %Loop over the input array columns

    m=M-1; %Define the size of the minor
    Minor=cell(m,m); %Initialize the minor

    %-----
    % Each entry must be put into the minor
    %   one by one, because they are cells
    %-----
    for i=1:m %Loop over the minor rows
    for j=1:m %Loop over the minor columns

        %-----
        % The row and column being removed from the
        %   input matrix to form the minor determines
        %   how the indexing to form the minor must go
        %-----
        if I == 1 & J == 1
            %The First Row and the First Column is removed
            Minor{i,j}=InputArray{i+1,j+1};

        elseif I == 1 & J == M
            %The First Row and the Last Column is removed
            Minor{i,j}=InputArray{i+1,j};

```

```

elseif I == 1 & J > 1 & J < M
    %The First Row and the Middle Columns are removed
    if j < J
        %The Columns to the left of the removed column
        Minor{i,j}=InputArray{i+1,j};
    elseif j >= J
        %The Columns to the right of the removed column
        Minor{i,j}=InputArray{i+1,j+1};
    else
        error('I==1, J~=(1|M) Minor Failure');
    end

elseif I == M & J == 1
    %The Last Row and The First Column is removed
    Minor{i,j}=InputArray{i,j+1};

elseif I == M & J == M
    %The Last Row and The Last Column is removed
    Minor{i,j}=InputArray{i,j};

elseif I == M & J > 1 & J < M
    %The Last Row and the Middle Columns are removed
    if j < J
        %The Columns to the left of the removed column
        Minor{i,j}=InputArray{i,j};
    elseif j >= J
        %The columns to the right of the removed column
        Minor{i,j}=InputArray{i,j+1};
    else
        error('I==M, J~=(1|M) Minor Failure');
    end

elseif I > 1 & I < M & J == 1
    %The Middle Rows and the First Column are removed
    if i < I
        %The rows above the removed row
        Minor{i,j}=InputArray{i,j+1};
    elseif i >= I
        %The rows below the removed row
        Minor{i,j}=InputArray{i+1,j+1};
    else
        error('I~=(1|M), J==1 Minor Failure');
    end

elseif I > 1 & I < M & J == M
    %The Middle Rows and the Last Column are removed
    if i < I
        %The Rows above the removed row
        Minor{i,j}=InputArray{i,j};
    elseif i >= I
        %The rows below the removed row
        Minor{i,j}=InputArray{i+1,j}
    else
        error('I~=(1|M), J==M Minor Failure');
    end

elseif I > 1 & I < M & J > 1 & J < M

```

```

%The Middle Rows and the Middle Columns are removed

%-----
% This entry goes quadrant by quadrant.  The
% quadrants are defined as:
%
%          -----
%          | Q1 | Q2 |
%          -----R-----
%          | Q3 | Q4 |
%          -----
% where R is the removed entry
%-----

if i < I & j < J
    %The entries in Quadrant 1
    Minor{i,j}=InputArray{i,j};
elseif i < I & j >= J
    %The entries in Quadrant 2
    Minor{i,j}=InputArray{i,j+1};
elseif i >= I & j < J
    %The entries in Quadrant 3
    Minor{i,j}=InputArray{i+1,j};
elseif i >= I & j >= J
    %The entries in Quadrant 4
    Minor{i,j}=InputArray{i+1,j+1};
else
    error('I~=(1|M), J~=(1|M) Minor Failure');
end

else
    error('Minor Indexing Failure. Case does not exist.');
```

end

%This is the End of the indexing into the minor array

end

```

%-----
% Find the signed determinant of the minor array
%-----
Cofactor=((( -1)^(I+J))*CellArrayDet(Minor));

%-----
% The cofactor is the signed determinant
%-----

%REM this used to state that the cofactor required convolution
% that statement was false. (see Phillips & Harbor)

%-----
% The Adjoint is the transpose of the cofactors, therefore
% the I,Jth cofactor goes in the J,Ith position in the
% adjoint output array
%-----
CellAdjOut{J,I}=Cofactor;
end
```

end

```
%*****
%* End Of Program
%*
%* Last Update: 04-11-2003
%*
%* v1.0: creation version
%*
%*****
```

CellArrayDet.m

```
function [CellDetOut]=CellArrayDet(InputArray)
%*****
%*
%* Author: Drew R. Reichenbach
%* Date: 04/11/2003
%* Group: i-EDA System Dynamics
%*
%* Function: CellArrayDet.m v2.1
%* Inputs: InputArray - A Cell Array of numeric vectors. Each vector
%* contains the coefficients of polynomials in
%* decreasing order.
%* Outputs: CellDet - A vector containing the coefficients of the
%* characteristic polynomial for the input array.
%*
%* Purpose: CellArrayDet(args) takes the determinant of a cell array of
%* unspecified size. This function is recursive, so large
%* input arrays could take a while to solve.
%*
%* Required Matlab Functions:
%* addpoly.m - adds vectors (polynomials) of dissimilar length
%*
%* NOTES: This function is recursive in nature. This means that large
%* input arrays could take some time to solve.
%* A 7x7 determinant solves in less than a second. 2521 calls
%* A 10x10 determinant requires over 2 minutes. 1814401 calls
%*
%*****

%=====
% Declare and Initialize Variables
%=====
M=[]; %Number of rows in the input array
N=[]; %Number of columns in the input array
CellDet=[]; %Output Vector (perhaps with leading zeros)
Minor={}; %The minor array
m=[]; %The size of the minor array
CellDetOut=[]; %Output Vector (no leading zeros)

%=====
% Main Program
%=====
```

```

%-----
% Find the size of the input array
%-----
[M,N]=size(InputArray);

%-----
% Error Check the Input
%-----
if M~=N
    error('Input Array Must Be Square');
end

%-----
% Take The Determinant
%-----
switch M %Chose the determinant method based on the input array size
case 1 %For a 1x1 input array
    CellDet=InputArray{1,1}; %the determinant is the cell contents
case 2 %For a 2x2 input array
    CellDet=addpoly(conv(InputArray{1,1},InputArray{2,2}),...
        (-1)*conv(InputArray{2,1},InputArray{1,2}));
    %Take the determinant by hand
    %Note: The 2x2 case is done by hand to improve computational speed
    % And to reduce the number of recursive function calls.
otherwise %For 3x3 to MxM arrays

    CellDet=[0]; %Initialize the Output

    for J=1:M %Loop along the first row of the input array
        m=M-1; %The size of the minor array is M-1xM-1
        Minor=cell(m,m); %Create the Minor Array

        %-----
        % Identify the entries in the Minor Array
        %-----
        for i=1:m %Loop along the rows of the minor array
            for j=1:m %Loop along the columns of the minor array
                switch J %Minor entry selection based on column removed
                case 1 %First column removed
                    Minor{i,j}=InputArray{i+1,j+1};
                case M %Last column removed
                    Minor{i,j}=InputArray{i+1,j};
                otherwise %Remaining columns
                    if j <= J-1 %for the first half
                        Minor{i,j}=InputArray{i+1,j};
                    elseif j >= J %for the second half
                        Minor{i,j}=InputArray{i+1,j+1};
                    else
                        error('The Minor Filling Function Failed');
                    end
                end
            end
        end
    end

    %-----
    % Take the Determinant as the sum of the product of the entry
    % and its cofactor (signed minor)

```

```

        %-----
        CofactorProd=conv(InputArray{1,J},((-1)^(1+J))*CellArrayDet(Minor));
        CellDet=addpoly(CellDet, CofactorProd);
    end
end

%-----
% Strip Unimportant Leading Zeros
%-----
if isempty(find(CellDet)) %No Non-Zero Entries
    CellDetOut=[0]; %Output a single zero
else
    CellDetOut=CellDet(min(find(CellDet)):length(CellDet));
    %Output the input vector from the first non-zero entry to
    % The last element in the vector
end

%*****
%* End Of Program *
%* *
%* Last Update: 04-11-2003 *
%* *
%* v2.1: Added code to strip leading zeros from output vector *
%* v2.0: updated to handle cell arrays of vectors. *
%* v1.0: creation version (replication of matrix determinant) *
%* *
%*****

```

Appendix C Dynamic Response Client Documentation

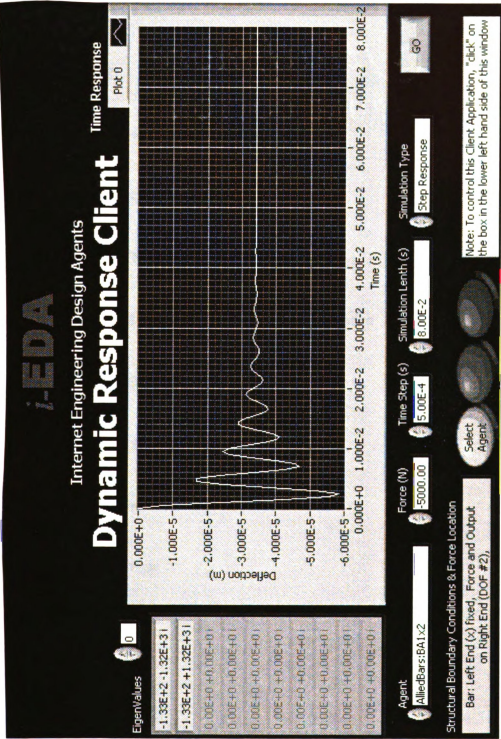
IEDADynamicsWebClient.vi

IEDADynamicsWebClient.vi is an interface for the i-EDA system. The program sends the dynamics query to the specified agent, and retrieves the dynamic stiffness matrix as a response. The client then applies boundary conditions, inverts to find the transfer function, and solves for the time response. Most of the computation is performed using Matlab.

Connector Pane



Front Panel



Controls and Indicators

U16 Agent



Go Button



Force (N)



Time Step (s)



Simulation Length (s)



Simulation Type



Structural Boundary Conditions & Force Location



Retrieving Stiffness



Computing Deflection



Select Agent



Time Response



EigenValues



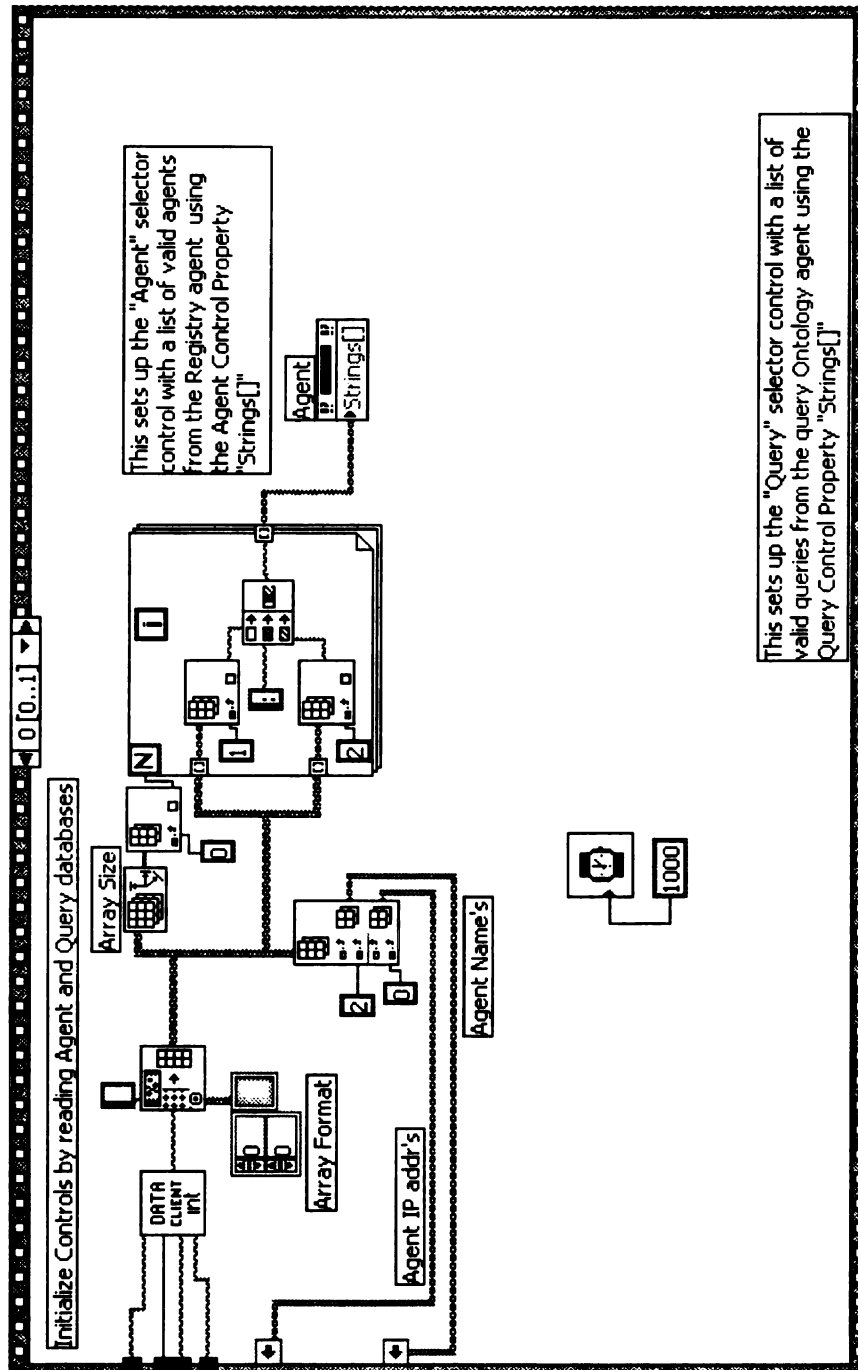
Numeric

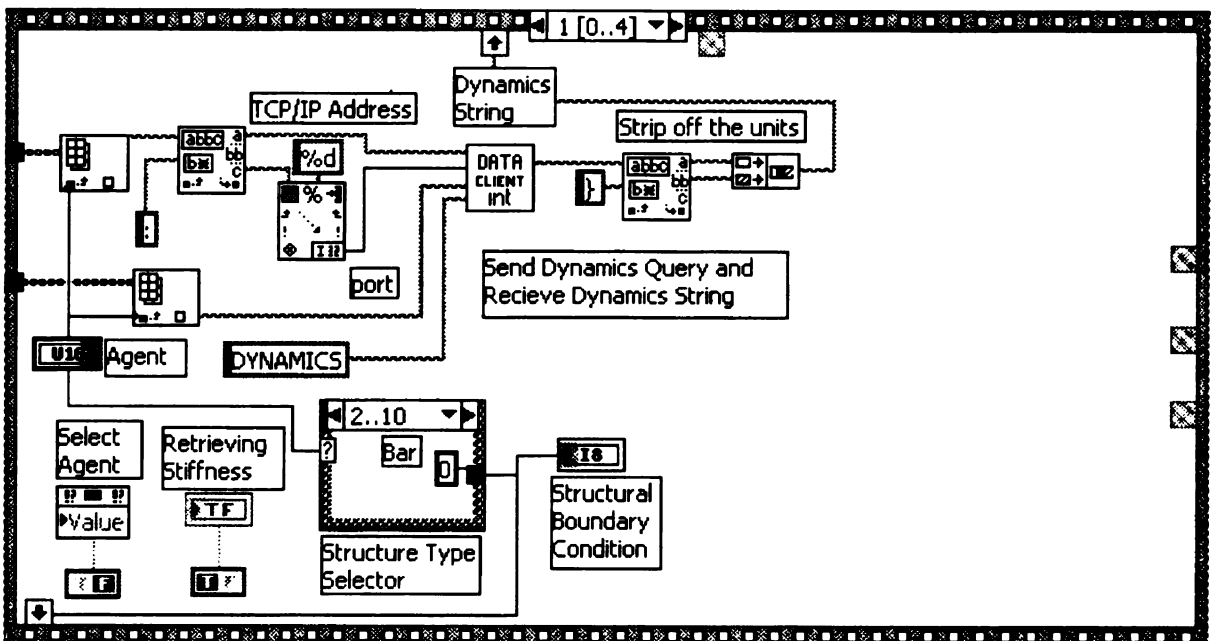
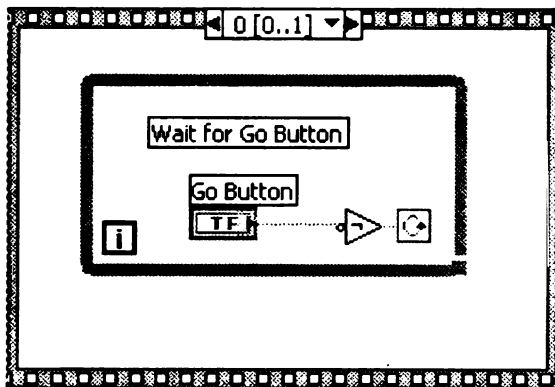
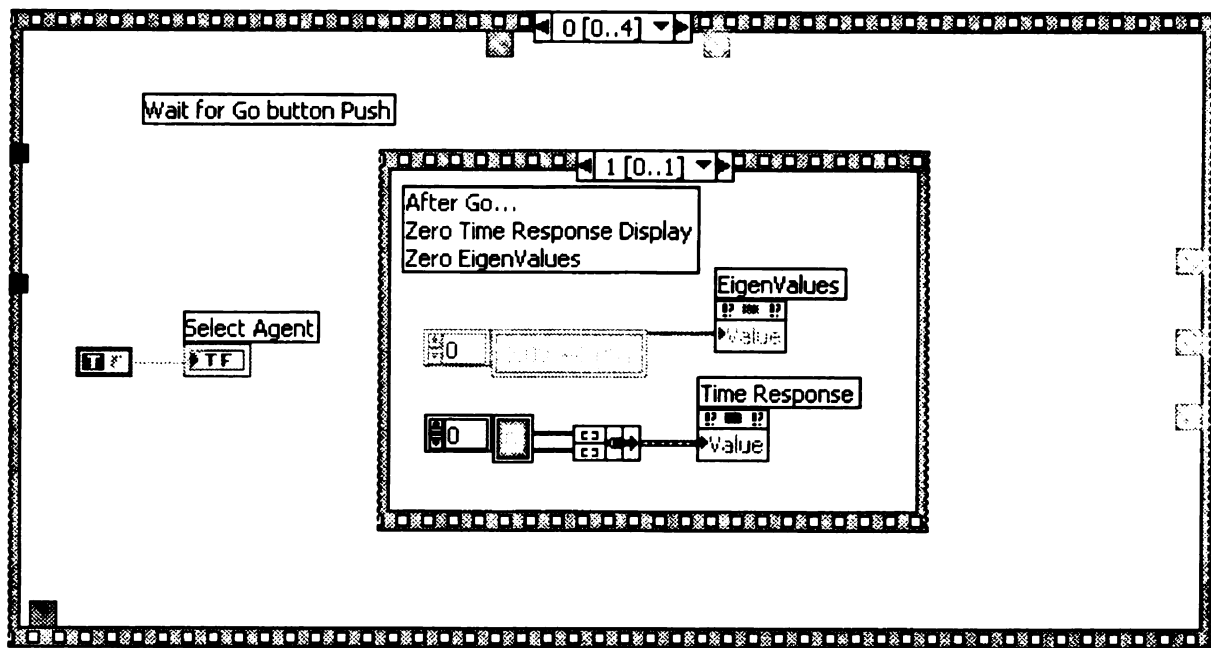
The diagram illustrates the internal architecture of the EGR system. It features several interconnected modules:

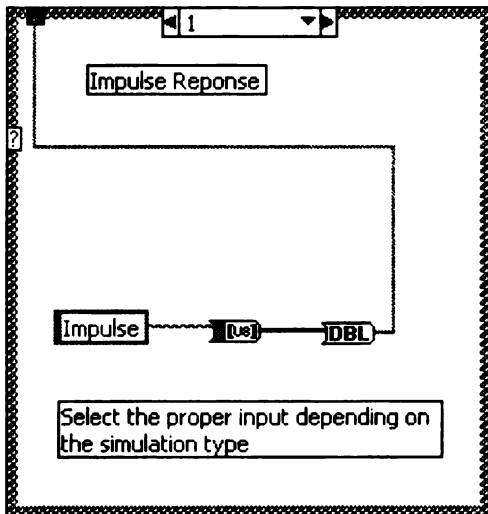
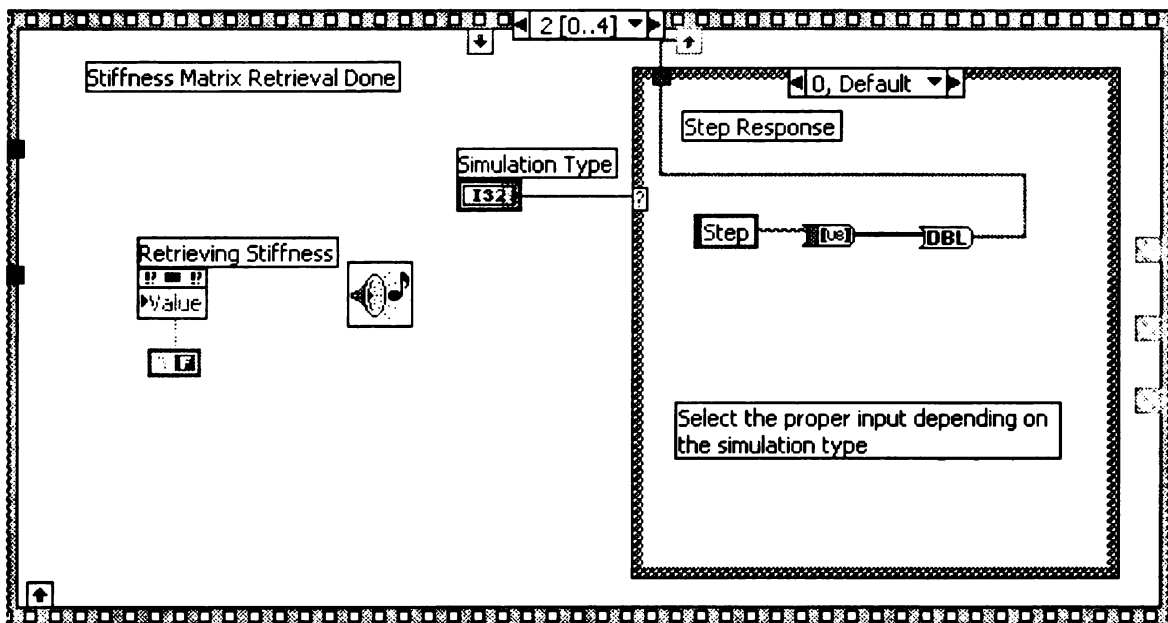
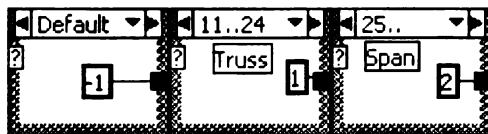
- Solution Vector**: A module at the top left.
- Time Response**: A central module containing a **Time Vector** and **Eigenvalues**.
- Eigenvalues**: Two separate modules below the Time Response.
- Computing Deflection**: A module on the right side.
- Process Complete**: A module on the far right.
- Ports and Connections**: The system includes multiple ports such as **Registry IP Address**, **Registry Port**, **Info**, and **Registry**. These are connected via a network of lines and switches.
- Data Flow**: Arrows indicate the direction of data flow between these components.

Feedback controls by reading Agent and Query databases

0 0 0 0 1





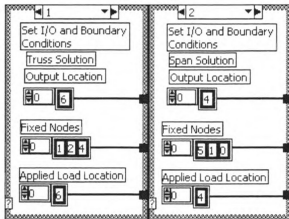
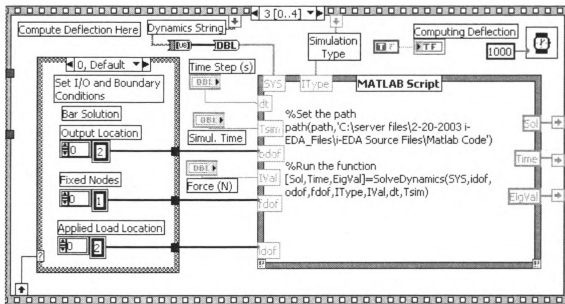




List of



DATA
CLIENT
END



List of SubVIs



Beep.vi

C:\Program Files\National Instruments\LabVIEW 6.1\vi.lib\Platform\system.lib\Beep.vi



InternalClient.vi

C:\server files\2-20-2003 i-EDA_Files\i-EDA Source Files\IEDAUtilities.lib\InternalClient.vi

History

*IEDADy

Current

Position

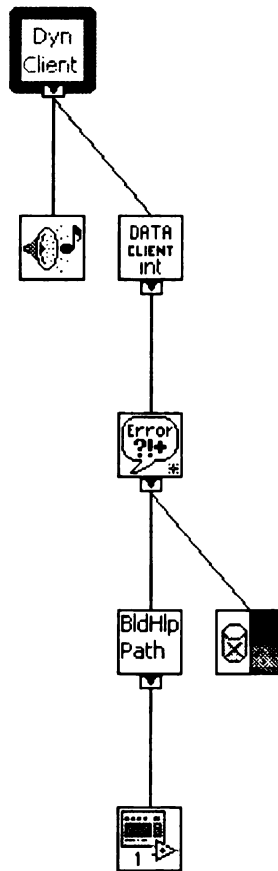


History

"IEDADynamicsWebClient.vi History"

Current Revision: 156

Position in Hierarchy



identical

model

model

modu

equati

transf

and th

when

and

Assu

dete

leve

Appendix D Failed Methods for Joining Dynamic System Models

Modular Dynamic System Representation

One of the primary goals of a modular model is for each model to have an identical input-output topology; both in the model and in the equations that constitute the model. This allows models (and equations) to be combined with ease; requiring no model reformulation before it can be combined into a larger model. The equation modularity also provides system anonymity, as composite systems will have the same equation representation as a smaller, component level system. Beginning with the transfer function representation of the state space system

$$\frac{\mathbf{Y}(s)}{\mathbf{U}(s)} = \frac{\mathbf{C}^T \text{adj}[s\mathbf{I} - \mathbf{A}]\mathbf{B}}{|s\mathbf{I} - \mathbf{A}|} + \mathbf{D} \quad (1)$$

and then rearranging and substituting to get the form

$$\mathbf{G}(s)\mathbf{Y}(s) = \mathbf{H}(s)\mathbf{U}(s) \quad (2)$$

where

$$\mathbf{G}(s) = \{\text{det}[s\mathbf{I} - \mathbf{A}]\}\mathbf{I} \quad (3)$$

and

$$\mathbf{H}(s) = \{\mathbf{C}^T \text{adj}[s\mathbf{I} - \mathbf{A}]\mathbf{B} + (\text{det}[s\mathbf{I} - \mathbf{A}])\mathbf{D}\} \quad (4)$$

Assuming an elegant solution that uses this form can be found, finding the adjoint and the determinant of the system matrices need only happen at the most simplistic (component) level of the model.

conne

the co

manip

where

and

This a

are go

bound

pairs t

generi

have a

will ta

Combining Modular Dynamic Systems

Combining modular dynamic systems stored in the form of (2), a modular connector, or “Join” is required. These Join elements constrain the inputs and outputs at the connected ports with zero power flow. Before the modular joint can be applied, some manipulation of the system must take place.

Beginning with the form:

$$\mathbf{G}(s)\mathbf{Y}(s) = \mathbf{H}(s)\mathbf{U}(s) \quad (5)$$

where

$$\mathbf{G}(s) = \mathbf{D}(s)\mathbf{I}_{n \times n} \quad (6)$$

$$\mathbf{Y}(s) = \mathbf{Y}(s)_{n \times 1} \quad (7)$$

$$\mathbf{H}(s) = [\mathbf{C}^T \text{adj}(s\mathbf{I} - \mathbf{A})\mathbf{B} + \det(s\mathbf{I} - \mathbf{A})\mathbf{D}]_{n \times n} \quad (8)$$

and

$$\mathbf{U}(s) = \mathbf{U}(s)_{n \times 1} \quad (9)$$

This arranges the transfer functions into a system of n equations. In this system, there are going to be external input-output pairs that are only affected by the system, and by boundary conditions. The system will also contain a number of internal input-output pairs that are affected by the original system and by the system being joined to it. A generic system, having n input-output pairs, with m internal input-output pairs, will have a total of $p = (n - m)$ external input-output pairs. The i^{th} equation for that system will take the form

$$\sum_{j=1}^p g_{ij} Y_j + \sum_{k=p+1}^n g_{ik} W_k = \sum_{j=1}^p h_{ij} U_j + \sum_{k=p+1}^n h_{ik} V_k \quad (10)$$

where

and V

m in

If the

equa

now

2000

the f

cons

cons

syste

syste

The

a ne

for

where Y_j is the external outputs, W_k are the internal outputs, U_j are the external inputs, and V_k are the internal inputs. The i^{th} equation for a second system of l equations, with m internal input-output pairs, and $q = (l - m)$ external input-output pairs would be

$$\sum_{j=1}^q g_{ij} Y_j + \sum_{k=q+1}^l g_{ik} W_k = \sum_{j=1}^q h_{ij} U_j + \sum_{k=q+1}^l h_{ik} V_k \quad (11)$$

If the systems being joined are both considered in this fashion, there will be $n + l$ total equations in the composite system, before the constraints are applied. The systems can now be joined in the modular modeling fashion. [Byam, 1999; Byam and Radcliffe, 2000]. According to the modular modeling philosophy, the constraints on the system (in the form of the modular modeling connector, or join) are now applied. Joins apply two constraints that connect modular subsystems into modular composite systems while conserving power. Consider the effects of a single join operation on the composite system. The join will connect the a^{th} port on the first system to the b^{th} port on the second system. The i^{th} equation of each system can now be written as

$$\sum_{j=1}^{a-1} g_{1,ij} Y_{1,j} + g_{1,ia} W_{1,a} + \sum_{k=a+1}^n g_{1,ik} Y_{1,k} = \sum_{j=1}^{a-1} h_{1,ij} U_{1,j} + h_{1,ia} V_{1,a} + \sum_{k=a+1}^n h_{1,ik} U_{1,k} \quad (12a)$$

$$\sum_{j=1}^{b-1} g_{2,ij} Y_{2,j} + g_{2,ib} W_{2,b} + \sum_{k=b+1}^l g_{2,ik} Y_{2,k} = \sum_{j=1}^{b-1} h_{2,ij} U_{2,j} + h_{2,ib} V_{2,b} + \sum_{k=b+1}^l h_{2,ik} U_{2,k} \quad (12b)$$

The first constraint forces the subsystem internal outputs to have equal values and creates a new external output. This constraint is written as

$$W_{1,i} = W_{2,j} = Y_{3,k} \quad (13)$$

for the composite system. This allows us to write the composite system as

$$\sum_{j=1}^{a-1} g_{1,ij} Y_{1,j} + g_{1,ia} Y_{3,c} + \sum_{k=a+1}^n g_{1,ik} Y_{1,k} = \sum_{j=1}^{a-1} h_{1,ij} U_{1,j} + h_{1,ia} V_{1,a} + \sum_{k=a+1}^n h_{1,ik} V_{1,k} \quad (14a)$$

which

cons

new

as

for t

are n

con

whi

con

con

pos

ren

eq

eq

sy

fro

h_2

$$\sum_{j=1}^{b-1} g_{2,ij} Y_{2,j} + g_{2,ib} Y_{3,c} + \sum_{k=b+1}^l g_{2,ik} Y_{2,k} = \sum_{j=1}^{b-1} h_{2,ij} Y_{2,j} + h_{2,ib} V_{2,b} + \sum_{k=b+1}^l h_{2,ik} V_{2,k} \quad (14b)$$

which is in terms of only external outputs. The second join constraint is a net power constraint. This constraint conserves the power output from each subsystem and from the newly created external input-output pair (provided by the join). This constraint is written as

$$V_{1,i} W_{1,i} + V_{2,j} W_{2,j} = U_{3,k} Y_{3,k} \quad (15)$$

for the composite system. From (13) it is known that the internal outputs $W_{1,i}$ and $W_{2,j}$ are forced to equal the newly created external output. Combining this with the power conservation of (15), and canceling the output terms gives

$$V_{1,i} + V_{2,j} = U_{3,k} \quad (16)$$

which is the second constraint equation. This constraint equation requires the linear combination of the equations representing each of the original systems. For the combination of two systems, of n and l equations respectively, there are a total of $n(l)$ possible combinations. However, of those combinations, only $(n + l) - 1$ are unique. The remaining equations can all be found through a linear combination of the unique equations. Restrictions on the selection of equations do exist, and mandate that each equation in each set must be used at least once. Furthermore, the equations from the two systems, once selected, must be multiplied through by the term scaling the internal input from the opposite system as shown in (17a)-(17b).

$$h_{2,b} \left(\sum_{j=1}^{a-1} g_{1,ij} Y_{1,j} + g_{1,ia} Y_{3,c} + \sum_{k=a+1}^n g_{1,ik} Y_{1,k} \right) = h_{2,b} \left(\sum_{j=1}^{a-1} h_{1,ij} U_{1,j} + h_{1,ia} V_{1,a} + \sum_{k=a+1}^n h_{1,ik} U_{1,k} \right) \quad (17a)$$

$$h_{1,2}$$

After

$$\sum_{j=1}^{a-1}$$

When

resp

wh

Re

sys

$$h_{1,a} \left(\sum_{j=1}^{b-1} g_{2,ij} Y_{2,j} + g_{2,ib} Y_{3,c} + \sum_{k=b+1}^l g_{2,ik} Y_{2,k} \right) = h_{1,a} \left(\sum_{j=1}^{b-1} h_{2,ij} U_{2,j} + h_{2,ib} V_{2,b} + \sum_{k=b+1}^l h_{2,ik} U_{2,k} \right) \quad (17b)$$

After this, the system equations can then be combined to form

$$\begin{aligned} \sum_{j=1}^{a-1} h_{2,b} g_{1,ij} Y_{1,j} + \sum_{j=1}^{b-1} h_{1,a} g_{2,ij} Y_{2,j} + (h_{2,b} g_{1,ia} + h_{1,a} g_{2,ib}) Y_{3,c} + \\ \sum_{k=a+1}^n h_{2,b} g_{1,ik} Y_{1,k} + \sum_{k=b+1}^l h_{1,a} g_{2,ik} Y_{2,k} \end{aligned} \quad (18a)$$

$$\begin{aligned} \sum_{j=1}^{a-1} h_{2,b} h_{1,ij} U_{1,j} + \sum_{j=1}^{b-1} h_{1,a} h_{2,ij} U_{2,j} + h_{2,b} h_{1,a} V_{1,a} + \\ h_{1,a} h_{2,b} V_{2,b} + \sum_{k=a+1}^n h_{2,b} h_{1,ik} U_{1,k} + \sum_{k=b+1}^l h_{1,a} h_{2,ik} U_{2,k} \end{aligned} \quad (18b)$$

Where (18a) and (18b) are the left and right-hand sides of the composite system equation respectively. Then, applying (16) to (18a-b) gives

$$\begin{aligned} \sum_{j=1}^{a-1} h_{2,b} g_{1,ij} Y_{1,j} + \sum_{j=1}^{b-1} h_{1,a} g_{2,ij} Y_{2,j} + (h_{2,b} g_{1,ia} + h_{1,a} g_{2,ib}) Y_{3,c} + \\ \sum_{k=a+1}^n h_{2,b} g_{1,ik} Y_{1,k} + \sum_{k=b+1}^l h_{1,a} g_{2,ik} Y_{2,k} \end{aligned} \quad (19a)$$

$$\begin{aligned} \sum_{j=1}^{a-1} h_{2,b} h_{1,ij} U_{1,j} + \sum_{j=1}^{b-1} h_{1,a} h_{2,ij} U_{2,j} + h_{1,a} h_{2,b} U_{3,c} + \\ \sum_{k=a+1}^n h_{2,b} h_{1,ik} U_{1,k} + \sum_{k=b+1}^l h_{1,a} h_{2,ik} U_{2,k} \end{aligned} \quad (19b)$$

where (19a-b) once again represent the right and left hand sides respectively.

Renumbering the system equations to match with the external ports on the composite system gives it the form

$$\sum_{j=1}^{n+l-1} g_{3,ij} Y_{3,j} = \sum_{j=1}^{n+l-1} h_{3,ij} \quad (20)$$

Wh
per

equ
for
gen
the

To
(li
(or
Pe

In
eq
ec

w

Which is exactly the form shown in (10) and (11), if there are no more joins to be performed.

Method Failure

The primary weakness in this method is its dependence on the combination of equations to combine joined internal inputs into the external input at the join. The formulation described above depends on the presence of zeros in the $H(s)$ matrix. In general, $H(s)$ is not sparse, and no combination of rows can occur. For example, consider the system:

$$[G(s)][y] = \begin{bmatrix} h_{1,11} & h_{1,12} & h_{1,13} & h_{1,14} \\ h_{1,21} & h_{1,22} & h_{1,23} & h_{1,24} \\ h_{2,11} & h_{2,12} & h_{2,13} & h_{2,14} \\ h_{2,21} & h_{2,22} & h_{2,23} & h_{2,24} \end{bmatrix} [u] \quad (21)$$

To join the third port from the first system to the second port from the second system (like putting the final join into the truss from three bar models), it requires that the first (or second) equation be equal to the third (or fourth) equation on the input to be joined. Performing the proper cross-multiplication (like described above) gives:

$$\begin{aligned} h_{2,12}h_{1,11}u_1 + h_{2,12}h_{1,12}u_2 + h_{2,12}h_{1,13}u_3 + h_{2,12}h_{1,14}u_4 \\ h_{1,13}h_{2,11}u_1 + h_{1,13}h_{2,12}u_2 + h_{1,13}h_{2,13}u_3 + h_{1,13}h_{2,14}u_4 \end{aligned} \quad (22)$$

In order to properly apply the input constraints, the multipliers on u_2 and u_3 must be equal. Given these two equations, and the constraint equations, there is no way for these equations to be joined in the method required by modular modeling.

It should also be noted that the same logic applies to the failure of the method when a Transfer Function is used instead of the split Transfer function.

do

ent

Additionally, direct application of Byam's methodology using state space systems does not allow proprietary information to be protected, because the join-history of the entire system must be included in the constraint equation.

LIST OF REFERENCES

- Byam, Brooks P., 1999, *Modular Modeling of Engineering Systems Using Fixed Input-Output Structure*, Ph. D. Dissertation, Michigan State University, East Lansing, MI.
- Byam, Brooks P. and Radcliffe, Clark J., 2000, “*Direct-Insertion Realization of Linear Modular Models of Engineering Systems Using Fixed Input-Output Structure*”, ASME, Proceedings of DETC2000: 26th Design Automation Conference, Baltimore, MD.
- Charlton, T.M., 1954, *Model Analysis of Structures*, John Wiley & Sons, Inc., New York, NY.
- Computers in Engineering: Chrysler designs paperless cars, 1998, *Automotive Engineering International*, Volume 106, Number 6 (June), Page 48.
- Friedland, Bernard, 1986, *Control System Design: An Introduction to State Space Methods*, McGraw-Hill, Inc., New York, NY.
- G.A. Hensley Company, 2000, Design data: scrambled or over easy? *Design News*, Volume 55, Number 3 (February), Page 24.
- Genta, Giancarlo, 1999, *Vibration of Structures and Machines: Practical Aspects*, Springer, New York, NY.
- Gosciak, Gary Joseph, 2001, *Internet Engineering Design Agents*, M.S. Thesis, Michigan State University, East Lansing, MI.
- Greenwood, Donald T, 1988, *Principles of Dynamics*, Second Edition, Prentice Hall, Englewood Cliffs, NJ pg. 323.
- Jost, Kevin, 1998, Chrysler redesigns its large cars for 1998, *Automotive Engineering International*, Volume 106, Number 1 (January), Page 10-15.
- Karnopp, Dean C., Margolis, Donald L., and Rosenberg, Ronald C., 2000, *System Dynamics: Modeling and Simulation of Mechatronic Systems*, John Wiley & Sons, Inc., New York, NY.
- Kerr, Brad, 2000, Redesigning work processes and computing environments, *Automotive Engineering International*, Volume 108, Number 7 (July), Page 147-149.
- Meirovitch, L., 1967, *Analytical Methods in Vibrations*, The Macmillan Company, New York, NY.

Phillips, Charles L. and Harbor, Royce D., 2000, *Feedback Control Systems*, Prentice Hall, Upper Saddle River, NJ.

Potter, Caren, 2000, Behind GM's global design success, *Automotive Engineering International*, Volume 108, Number 12 (December), Page 74-75.

Radcliffe, Clark J. and Sticklen, Jon H., "Method and System for Creating Designs Using Internet-Based Agents", United States Patent No.: 6,295,535 B1, September 25, 2001.

Radcliffe, Clark J. and Sticklen, Jon, 2003, "*Modular Distributed Models of Engineering Structures*", ASME IMECE, Proceedings of IMECE 2003: International Mechanical Engineering Congress and Exhibition, Washington DC.

Radcliffe, Clark J. and Sticklen, Jon, 2003, "*Modular Distributed Models of Engineering Structures*", ASME IMECE, Proceedings of IMECE 2003: International Mechanical Engineering Congress and Exhibition, Washington DC.

Radcliffe, Clark J., Sticklen, Jon, and Gosciak, Gary, 2002, "*The Internet Engineering Design Agent System: iEDA*", ASME, Proceedings of 2002 ASME IMECE: International Mechanical Engineering Conference and Exhibition, New Orleans, LA.

Reddy, J. N., 1993, *An Introduction to the Finite Element Method*, McGraw-Hill, Burr Ridge, IL.

Subcommittee on International Economic Policy and Trade of the Committee on International Relations, *Corporate and Industrial Espionage and Their Effects on American Competitiveness*, 106th Congress, Second Session, 2000, Serial Number: 106-180, pg 1.

MICHIGAN STATE UNIVERSITY LIBRARIES



3 1293 02504 6305