Implementation of Mathematica® Interface for DOME
(Distributed Object-based Modeling Environment)

by

Kathleen Jheehye Lee

SUBMITTED TO THE DEPARTMENT OF MECHANICAL ENGINEERING IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE

AT THE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 2001

Signature redacted

Signature of Author ______________________
Department of Mechanical Engineering
June 11, 2001

Signature redacted

Certified by ______________________
Professor David R. Wallace
Thesis Supervisor

Signature redacted

Accepted by ______________________
Professor Ernest G. Cravalho
Chairman, Undergraduate Thesis Committee

Implementation of Mathematica® Interface for DOME (Distributed Object-based Modeling Environment)

by

Kathleen Jheehye Lee

Submitted to the Department of Mechanical Engineering
on May 11, 2001 in Partial Fulfillment of the
Requirements for the Degree of Bachelor of Science in
Mechanical Engineering

ABSTRACT

Product design spans many different disciplines, each of which attempts to meet a unique set of design objectives. Because these design objectives are often in conflict, product design can be described as a collaborative effort to optimize the resolution of competing design goals. In this paradigm, the speed and quality of communication are critical to an efficient and timely design cycle (Wang, 1998). Integrated design simulations are powerful tools that can be used to help satisfy these competing goals. However, creating integrated design models is very difficult as different disciplines use different representations and modeling tools.

The Distributed Object-based Modeling Environment (DOME) system, developed at the Computer Aided Design Laboratory (CADLAB) at the Massachusetts Institute of Technology (MIT) embodies a design service marketplace that can be used to address integration barriers so that integrated simulations may be created easily.

This thesis implements a DOME design service to Wolfram Research's engineering analysis and modeling software Mathematica®. This software module will allow any mathematical simulation to be easily interfaced with many other modeling environments and databases, and a simple example model is used to demonstrate this integration.

Thesis Supervisor: David R. Wallace
Title: Ester and Harold Edgerton Associate Professor of Mechanical Engineering

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to extend thanks to my thesis advisor Professor Wallace for this opportunity to build something of my own, to Ed and Elaine for their generous help. Thanks Soohyun and Paul for a countless many memorable things, and thanks Mom, Dad, Grace and Jenny, for everything.

Now on with the paper.

1 INTRODUCTION

1.1 MOTIVATION

Product design spans many different disciplines, each of which attempts to meet a unique set of design objectives. Because these design objectives are often in conflict, product design can be described as a collaborative effort to optimize the resolution of competing design goals. In this paradigm, the speed and quality of communication are critical to an efficient and timely design cycle (Pahng, Senin and Wallace, 1998).

A Distributed Design Environment (DDE) utilizes a computer network to enhance communication during such product development cycles where design parties are geographically dispersed and of different disciplines. However, even in this networked paradigm, the product development cycle is bottlenecked because designers do not have real-time access to the expertise of those from other disciplines. Designers have no choice but to rely on techniques that provide simplified versions of the specialized analysis or simulation. For example, manufacturing designers generally lack the environmental training to perform a detailed environmental assessment of a design, and vice versa (Borland and Wallace, 2000).

The concept of a design service marketplace addresses this problem by enabling designers to offer their disciplinary simulation and analysis expertise(environmental assessment, geometric design, cost analysis, etc.) as services operable over the Internet (Borland and Wallace, 2000). These decentralized services form a marketplace of services that enable integrated, distributed, collaborative and concurrent product design. One of the key requirements of an integrated design service marketplace is the ability to make interactive services widely accessible for use by others while still allowing each expert to use their modeling tool of choice.

1.2 Distributed Object-based Modeling Environment (DOME)

The Distributed Object-based Modeling Environment (DOME) system (Pahng, Senin and Wallace, 1998), developed at the Computer Aided Design Laboratory (CADLAB) at the Massachusetts Institute of Technology (MIT) is a web-based integrated product modeling and simulation environment. DOME embodies a design service marketplace that can be used to address integration barriers so that integrated simulations may be created easily.

Design services will most often be models that have been previously defined in software applications such as Excel®, Matlab®, SolidWorks® or ProEngineer®. DOME integrates design services built in third party applications by defining software objects known as wrappers that map a DOME object interface to the external application.

1.3 Goal and Deliverable of Thesis

The goal of this thesis is to implement a DOME wrapper to Wolfram Research's engineering analysis and modeling software, Mathematica®. The corresponding deliverable is a software module written in C++ that implements a communication interface to Mathematica's back-end, the Mathematica kernel. This will allow any mathematical analysis or simulation defined in Mathematica to be easily interfaced with many other modeling environments and databases.

1.4 Organization of Thesis

This thesis presents the implementation of a software module as part of the existing DOME framework. Chapter 2 provides background information relevant to the implementation. The first part describes the object-based modeling formalism of DOME; the second part, MathLink, the Mathematica application programming interface (API); and the third part, publisher data. This chapter assumes a basic knowledge of object-oriented programming. Chapter 3 gives a brief overview of implementation in three steps along with a schematic of the implementation.

Chapter 4 presents the results of the implementation by demonstrating a working interface through a simple example model, a cantilever beam under uniform load. This chapter can also be considered the documentation for the wrapper since it steps through the process of creating a Mathematica model in DOME. Chapter 5 outlines opportunities for future development on the wrapper.

2 Pre-Implementation Background

The first part of this chapter provides background on the object-based modeling formalism of DOME focusing on service-objects, DOME's basic building blocks. The second part discusses MathLink, the Mathematica API, and the third part briefly discusses publisher data.

2.1 DOME Modeling Formalism

Chapter 1 gave an abstract description of DOME using terms such as integrated, distributed, and concurrent. This section presents a more detailed description by illustrating how DOME integrates specialized design services by encapsulating them in distributed objects (Abrahamson, Wallace, Senin and Sferro, 2000). These distributed objects are also known as service-objects.

2.2.1 Different Types of Service-objects

Service-objects are the basic building blocks of DOME's object-oriented modeling formalism. In its simplest form, service-objects represent different data types such as real numbers or complex numbers; probability distributions; functions or strings. A more complex service-object is the container service-object which is an object that may encapsulate an embedded model of any type. These containers allow designers to wrap custom models with a DOME object interface. For third party application containers known as plug-ins or wrappers, the embedded model is a back-end interface to the external application. Once this interface is defined, subscribers of this service may create models or simulations in DOME that utilize the third party application without any programming or understanding of how the external application works.

Figure 1 contains code segments from the final C++ wrapper implementation. They are not meant to be read in sequence; individual lines were taken as excerpts to demonstrate the different types of service-objects and how they related to each other within the DOME framework. The numbers in the right-most column map to the line numbers of the source code found in Appendix A. In this discussion, the excerpts will be referred to by the line numbers in the left-most column.

```
1   struct ServiceModule                                               71
2   {                                                                  72
3   PRealModule* service;                                              73
4   string name;                                                       74
5   string domeName;                                                   75
6   }                                                                  76
7
8   ServiceModule* module = new ServiceModule;                        413
9   module->service = new PRealModule(domeName.c_str(), value,        431
    unit.c_str());
10  module->name = string(varName);                                   414
11  module->domeName = string (domeName);                             415
12
13  PStringModule* fileNameService;                                    81
14  fileNameService = new PStringModule(MATHEMATICA_FILENAME);        162
15
16  PBooleanModule* isOnline;                                          87
17  isOnline = new PBooleanModule(MATHEMATICA_IS_ONLINE, pTrue);      169
18
19  PSimpleArray<ServiceModule*> inputs;                               78
20  inputs.add(module);                                               431
21
22  PContainerModule* inputContainer;                                  83
21  inputContainer = new PContainerModule(inputs);                    164
```

Figure 1 Code excerpts from final wrapper implementation to demonstrate different types of DOME service-objects

`PRealModule (line 3)`, `PstringModule (line 13)`, and `PBooleanModule (line 16)` are basic service-objects. `PRealModule` consists of contains a name, value of type double, and a unit of type string as seen in its instantiation in line 9. There is also a container service-object, `inputContainer (line 22)`. Figure 2 is an illustration of these different types of service-objects and how the more complex types encapsulate the simpler types.

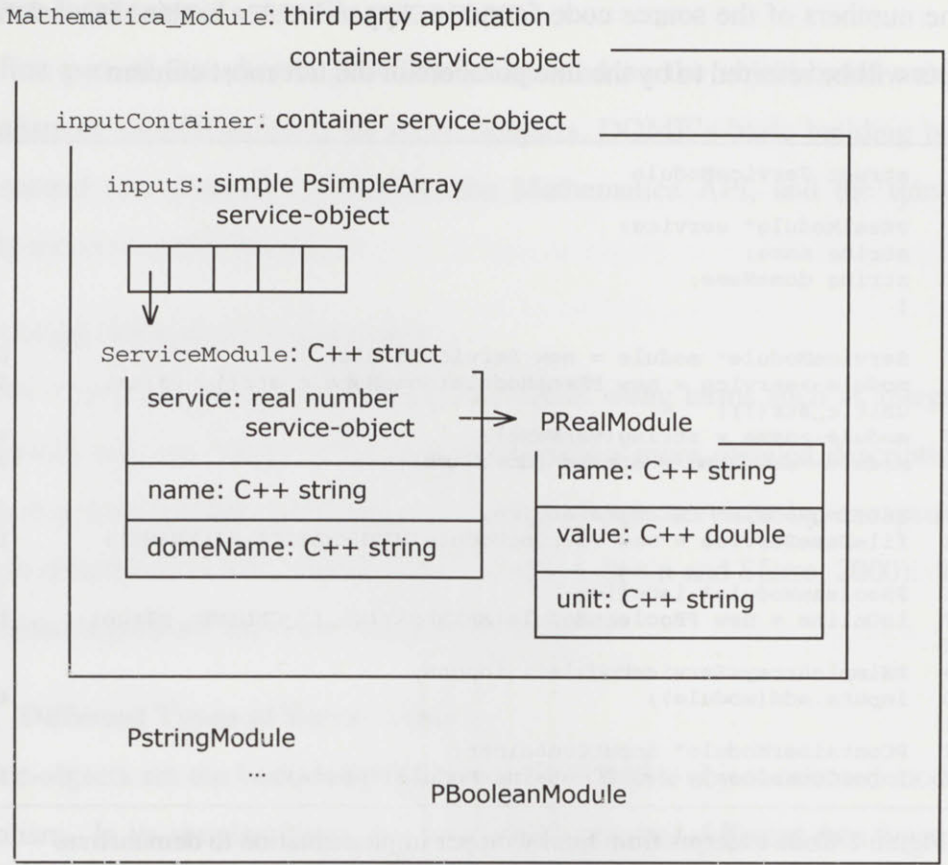

Figure 2 Encapsulation of different types of service-objects within the wrapper module (not all objects are shown)

`Inputs (line 19)` is a simple-typed service-object, and it comprises the container service-object `inputContainer (line 22)`. In turn, `inputContainer` comprises the third party application container service-object, `Mathematica_Module`. `Mathematica_Module` consists of several service-objects of varying complexity that provide a mapping to variables defined in Mathematica.

2.2 MathLink: The Mathematica API

Creating a back-end interface to a proprietary application in the embedded model of a container service-object as described above requires learning the proprietary API. MathLink is a library of functions that implement a protocol for sending and receiving Mathematica expressions. It is used to allow DOME, the front-end, to use the Mathematica kernel, the back-end, as a computational engine. By using

MathLink, Mathematica can essentially be treated like a subroutine embedded inside an external program. Appendix B gives a list of basic MathLink functions in signature form.

Although the MathLink methods are not hard to grasp conceptually, an understanding of the data exchange scheme between the kernel and the external program is required to generate the correct sequence of MathLink methods for a particular task such as reading data from the kernel. Appendix C gives a description of Mathematica's data transfer mechanism.

2.3 Publisher Data

The two previous sections described elements in DOME and Mathematica that allowed them to communicate. DOME embeds service-objects that can map an interface to the external program Mathematica, and MathLink provides methods to interact with the kernel. In DOME, a Mathematica wrapper object is defined and ready to be instantiated, while the Mathematica kernel is ready to process commands. What is yet to be defined, however, is a protocol by which the interaction will be mediated. What kind of variables did the designer define in the Mathematica model? What exactly is the model? The publisher data or publishing meta-data contains this information. It dictates how the Mathematica model is defined within the DOME domain. It allows designers that publish their services to define variables in the model as inputs or outputs and assign initial values and units. In this implementation of the Mathematica wrapper, it is simply an external text file that defines the model's variables. The contents of this file are discussed in more detail in Chapter 4.

3 IMPLEMENTATION OVERVIEW

The Mathematica wrapper for DOME was implemented in C++ under the Windows NT® environment. This chapter gives a brief overview of the steps that were taken in implementing the wrapper object. Implementation started by building a prototype C++ object that utilized the MathLink API to launch, open files, set and get values of predefined variables within the Mathematica kernel. The format of the publisher data file was defined next, and lastly, the methods written in the first step were wrapped into a DOME third party application container object. The last section of the chapter gives a schematic of the implementation.

3.1 C++ PROTOTYPE

The initial C++ prototype class was modeled after the MathLink communication class defined in DensityViewer (Density Viewer), a simple MathLink program. Although DensityViewer is implemented in C++ using Apple Computer's MacApp Object Framework, the essential MathLink communication is implemented as a platform-independent C++ object. Appendix B lists the MathLink methods that were used in the prototype.

3.2 PUBLISHER DATA FILE

The publisher data format was defined as follows. The first line of the file contains the name of the Mathematica model, and the following lines list the variables of the model. The variable is stated as an input or output of the model and is defined with separate names in the DOME and Mathematica domains and with initial values and units. Figure 3 shows the publisher data file format.

Model name				
Input or Output	Mathematica name	DOME name	Initial value	unit

Figure 3 Publisher data file format for DOME Mathematica wrapper

3.3 DOME Container Object: Mathematica_Module

Once the MathLink methods were written and tested, they were wrapped into a DOME third party application container service-object. The wrapping C++ code was modeled after a DOME SolidWorks® wrapper object. The methods were iteratively tested within the new DOME-wrapped environment.

3.4 Schematic of Implementation

Figure 4 provides a schematic of the implementation. It conveys a somewhat topological perspective of how all the sub-components of the implementation fit with respect to each other and the order in which interactions are initiated. First, the C++ DOME wrapper reads in the pre-defined external publisher data file. The data that are read are used to create a DOME container service-object within the DOME domain, and finally, the instantiated wrapper object communicates directly with the Mathematica kernel via its MathLink methods.

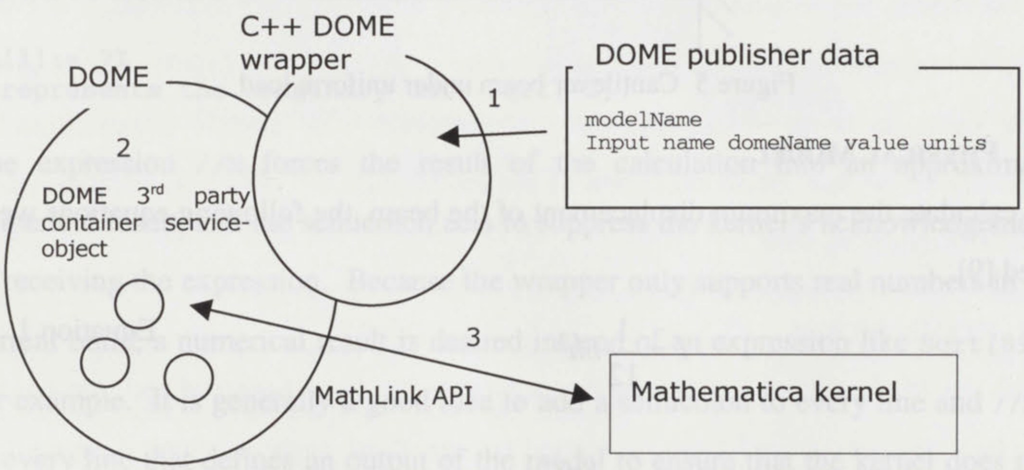

Figure 4 Schematic of implementation overview

4 Example model: Cantilever Beam Under Uniform Load

This chapter demonstrates the results of the wrapper implementation through a simple model of a cantilever beam under uniform load as shown in Figure 5. Although simple to model, the maximum displacement of the beam under load is potentially a product design criterion that must be optimized. The model is discussed in three domains: physical, Mathematica, and DOME. This chapter can also be considered the documentation for the wrapper since it steps through the process of creating a Mathematica model in DOME. A HTML version of the documentation can be found in Appendix D and on-line at http://cadlab.mit.edu/dome/doc/current/Mathematica/index.html.

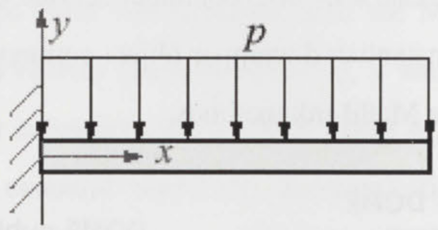

Figure 5 Cantilever beam under uniform load

4.1 Physical Model

To calculate the maximum displacement of the beam, the following equations were used [9].

$$I = \frac{1}{12} b h^3 \qquad \text{Equation 1}$$

$$\delta_{\max} = -\frac{1}{8}\frac{pL^4}{EI} \qquad \text{Equation 2}$$

Equation 1 expresses the moment of inertia I for a rectangular beam of width b and height h. Equation 2 defines the maximum displacement for a beam of length L under a pressure load of p. E is the Young's modulus of the material.

4.2 MATHEMATICA MODEL: DISPLACEMENT.M

In the Mathematica domain, Equations 1 and 2 are defined in a Mathematica expression file called `displacement.m` as shown in Figure 6.

```
displacement.m

i = 1/12 b h^3 //N;
w = - ( ( L^4 * p ) / ( 8*Y*(i/100000000) ) ) * 100 //N;
```

Figure 6 Mathematica expression file `displacement.m` to calculate moment of inertia *I* and maximum displacement *w*

The variable names must be carefully chosen since the model will not execute correctly if it contains variables that conflict with existing symbol names in Mathematica. One way to avoid variable collision is to type `[?variableName]` at the Mathematica kernel prompt. The message generated in the following interaction with the Mathematica kernel indicates that the symbol `I` is protected; it can not be used as a variable name.

```
In[1]:= ?I
I represents the imaginary unit Sqrt[-1].
```

The expression `//N` forces the result of the calculation into an approximate numerical result, and the semicolon acts to suppress the kernel's acknowledgement of receiving the expression. Because the wrapper only supports real numbers in its current build, a numerical result is desired instead of an expression like `Sqrt[89]`, for example. It is generally a good idea to add a semicolon to every line and `//N;` to every line that defines an output of the model to ensure that the kernel does not send back unexpected packets that might raise errors in the model.

4.3 PUBLISHER DATA FILE

To bring the cantilever beam model into the DOME domain a publisher data file must be defined in the correct format. Figure 7 provides the publisher file for the beam model.

```
displacement.m
Input    L   length             1.0            meter
Input    b   width              4.5            centimeter
Input    h   height             3.0            centimeter
Input    p   pressure_load      100            newtons_per_meter
Input    Y   Youngs_Modulus     70000000000    pascal
Output   i   Moment_inertia     0.0            centimeter^4
Output   w   Max_displacement   0.0            centimeter
```

Figure 7 Publisher file for cantilever beam displacement model

The first line of the publisher file is the name of the Mathematica expression file that resides in the `dome/bin` directory.

The following lines are divided into five columns and each field must be defined for each variable. The first column states whether the variable is an input or output of the model, the second defines the variable's name within Mathematica and the third its corresponding-more descriptive-name in DOME. The fourth column gives each value an initial value, and the fifth its unit. The default values must be carefully set so that initial execution of the model will not yield non-numbers such as in a division by zero.

4.4 DOME Model

Once the Mathematica model and the publisher file have been defined, the model can be integrated into DOME wrapped by the third party application container service-object. The service-object can be created as shown in Figure 8.

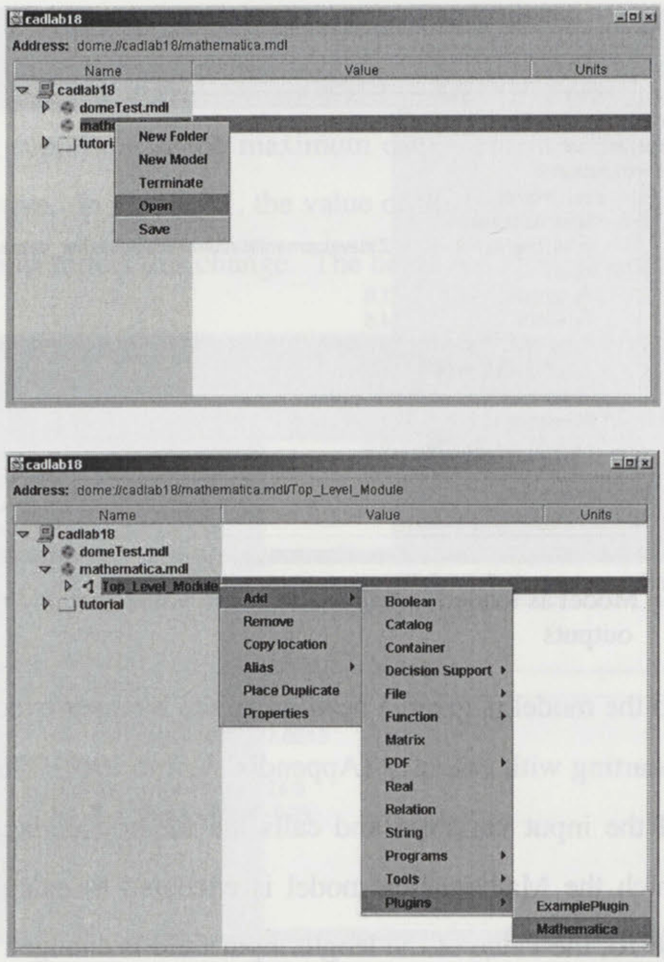

Figure 8 Opening and creating an instance of the Mathematica wrapper

When the publisher data file is typed in the PublisherFileName field as shown in Figure 9, the model is loaded with its input and output variables set to their initial values. There are six inputs and two outputs where the outputs are defined by Equations 1 and 2 from the physical model section.

When the model is loaded for the first time, the expression file `displacement.m` is not yet executed. The file is executed for the first time when the user changes an input value.

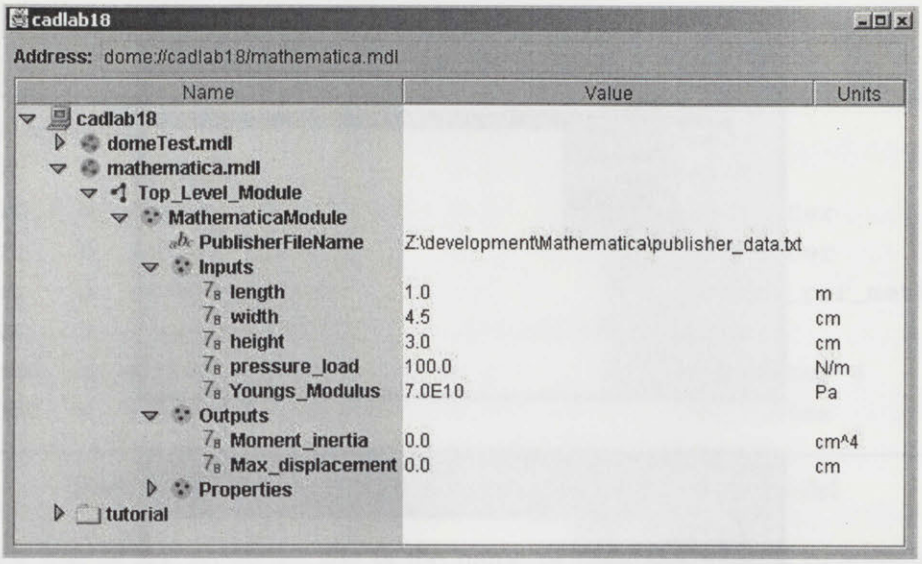

Figure 9 Model as loaded initially with initial values for all inputs and outputs

When an input to the model is given a new value, the wrapper executes a sequence of method calls starting with `react()` (Appendix A, line 190). This method resets the values of all the input variables and calls the method `updateAllOutputs()` (line 268) in which the Mathematica model is executed to calculate new output values. In Figure 10, the value of the length input field is changed to 1.5, and new output values are calculated as shown.

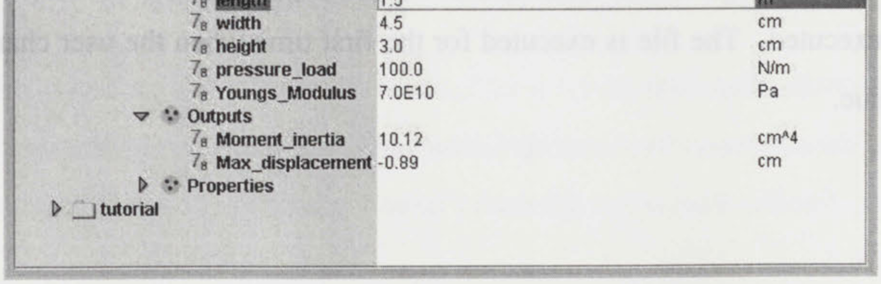

Figure 10 Changing the value of the beam length L to 1.5 meters.

Since ultimately, the goal of creating integrated simulations is to use them to explore alternatives by changing the values of design variables (Borland, Senin and Wallace, 2000), suppose that the maximum displacement was too large to satisfy the design objective. In Figure 11, the value of the beam height is increased to 4.0, and the two outputs reflect this change. The beam has a smaller displacement.

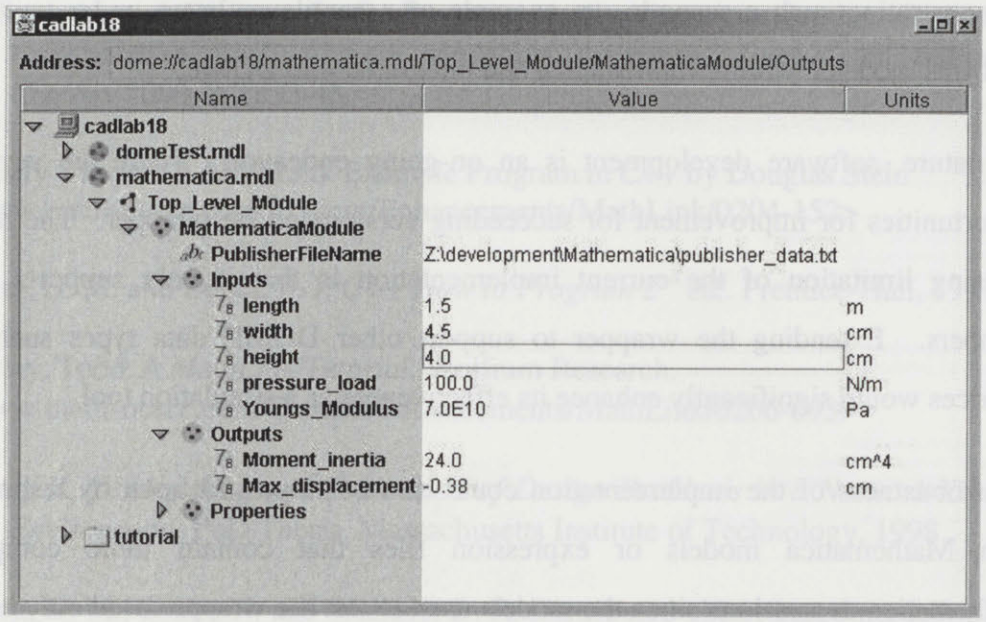

Figure 11 Beam height h is increased to 4.0 cm for a smaller maximum displacement of 0.38 cm

5 Conclusion and Opportunities for Improvement

A DOME wrapper to Wolfram Research's engineering analysis and modeling software Mathematica® was implemented as the main goal of this thesis. This software module will allow mathematical simulations defined in Mathematica to be easily interfaced with many other modeling environments and databases. Demonstrating such a scenario, an example of a cantilever beam under uniform load was modeled within Mathematica and integrated into DOME.

By nature, software development is an on-going endeavor. There are several opportunities for improvement for succeeding versions of the wrapper. The most pressing limitation of the current implementation is that it only supports real numbers. Extending the wrapper to support other DOME data types such as matrices would significantly enhance its effectiveness as a simulation tool.

The robustness of the implementation could also be improved upon by testing it with Mathematica models or expression files that contain more complex Mathematica expressions since the models used to test the wrapper used only basic arithmetic operations. The ability to render graphical representations of a simulation would also enhance the wrapper service.

REFERENCES

Abrahamson, S., Wallace, D.R., Senin, N. and Sferro, P. Integrated Design in a Service Marketplace. *Computer-Aided Design* 2000; 32:97-107.

Borland, N., Senin, N. and Wallace, D.R. Distributed Object-based Modeling in Design Simulation Marketplace. *ASME Journal of Mechanical Design*, 2000

Borland, N. and Wallace, D. R. Environmentally Conscious Product Design: A Collaborative Internet-based Modeling Approach. *Journal of Industrial Ecology* 2000; 3(2):33-46.

Density Viewer--A MathLink Example Program in C++ by Douglas Stein <www.mathsource.com/Content/Enhancements/MathLink/0204-152>

Deitel, H. M. and Deitel, P. J. *C++ How to Program 2nd ed.*. Prentice-Hall, 1997.

Gayley, Todd. *A MathLink Tutorial*. Wolfram Research. <www.mathsource.com/Content/Enhancements/MathLink/0206-693>

Pahng, K. F. *Modeling and Evaluation of Design Problems in a Network-Centric Environment*. PhD Thesis. Massachusetts Institute of Technology, 1998.

Pahng, K. M., Senin, N. and Wallace, D. R. Distributed object-based modeling and evaluation of design problems. *Computer-Aided Design* 1998; 30(6):411-423.

Specific Beam Loading Case: Cantilever Uniform Load. EFunda. <www.efunda.com/formulae/solid_mechanics/beams/casestudy_display.cfm?case=cantilever_uniformload#target>

Wang, Priscilla H. *Benchmarking a Collaborative, Concurrent Computer Design Tool*. BS Thesis. Massachusetts Institute of Technology, 1998.

Wolfram, Stephen. *The Mathematica Book, 4th ed.*. Wolfram Media/Cambridge University Press, 1999.

APPENDIX A: C++ SOURCE CODE

MathematicaDebug.h

```
/*------------------------------------------------------------
   MathematicaDebug.h
   Version 1.0
   Copyright (c) 2001 MIT All Rights Reserved
------------------------------------------------------------*/

#define MATHEMATICA_DEBUG_SWITCH false
#define MATHEMATICA_DEBUG(s) if (MATHEMATICA_DEBUG_SWITCH) cerr<<"Mathematica Debug: "<<s<<endl;
#define MATHEMATICA_ERROR(s) cerr<<"Mathematica Error: "<<s<<endl;
```

MathematicaModule.h

```
// MathematicaModule.h: interface for the MathematicaModule class.
//////////////////////////////////////////////////////////////////////

#if !defined(AFX_MATHEMATICAMODULE_H__9B86B222_3EA7_4538_B1B4_340C579B8744__INCLUDED_)
#define AFX_MATHEMATICAMODULE_H__9B86B222_3EA7_4538_B1B4_340C579B8744__INCLUDED_

#include <dome/dome.h>
#include <dome/pnames.h> // for the keywords in the read method

// Use the right streams library.
#if defined(USE_OLD_STREAMS)
#include <fstream.h>
#elseif defined(USE_ANSI_STREAMS)
#include <fstream>

#else
#ifndef NO_STREAMS
#define NO_STREAMS
#endif
#endif

#include <stdexcept>
#include <outputstreambuf.h>
#include "MathematicaDebug.h"
#include "mathlink.h"
using namespace std;  // don't remove
#if _MSC_VER > 1000
 #pragma once
#endif

#define MATHEMATICA_FILENAME "PublisherFileName"
#define MATHEMATICA_INPUTS        "Inputs"
#define MATHEMATICA_OUTPUTS       "Outputs"
#define MATHEMATICA_PROPERTIES      "Properties"
#define MATHEMATICA_IS_ONLINE       "Is_Online"
#define MATHEMATICA_IS_RUNNING      "Is_Running"
#define MATHEMATICA_TYPE_INPUT      "Input"
#define MATHEMATICA_TYPE_OUTPUT       "Output"

// __declspec(dllexport)
class MathematicaModule : public PContainerModule
{
public:
  MathematicaModule(const char* name=0);
  MathematicaModule(const MathematicaModule&);
  virtual ~MathematicaModule();

  virtual PService* clone(CloneMethod) const {return new MathematicaModule(*this); }
  virtual const char* ClassName() const {return "MathematicaModule";}
  virtual const char* classname() const {return ClassName();}

  virtual void react(const PSpeaker&, const PMessage&);
  virtual PStatus read(PString&);

protected:
  struct ServiceModule
  {
    PRealModule* service;
    string name;
    string domeName;
  };

  PSimpleArray<ServiceModule*> inputs;
  PSimpleArray<ServiceModule*> outputs;

  PStringModule* fileNameService;

  PContainerModule* inputContainer;
  PContainerModule* outputContainer;
```

```
    PContainerModule* properties;
     PBooleanModule* isOnline;
     PBooleanModule* isRunning;

    void updateAllOutputs();
    void readPublisherFile(const char* );

    void setValue(ServiceModule* );
    double getValue(string );
    void updateValue(ServiceModule*);

    void openLink();
    void closeLink();
    void loadModel(string );

private:
    void* ep;
    MLINK lp;
    string mathlinkFilename;
    void readReturnPacket (MLINK );
};

#endif // !defined(AFX_MATHEMATICAMODULE_H__9B86B222_3EA7_4538_B1B4_340C579B8744__INCLUDED_)
```

MathematicaModule.cpp

```
// MathematicaModule.cpp: implementation of the MathematicaModule class.

#include "MathematicaModule.h"

#define USE_WIN32_DSO
#if defined(USE_WIN32_DSO)
#define EXPORT __declspec(dllexport)
#else
#define EXPORT
#endif
#if defined(_MSC_VER)
#include <windows.h>

BOOL WINAPI DllMain( HINSTANCE, DWORD wDataSeg, LPVOID)
{
  switch(wDataSeg) {
      case DLL_PROCESS_ATTACH:
      DOME_REDIRECT_OUTPUT //important so that output is redirected to Java
        return 1;
        break;
    case DLL_PROCESS_DETACH:
      break;
      default:
        return 1;
          break;
    }
    return 0;
}
#endif //defined(_MSC_VER)

// These are the functions that will be exported in the shared library.  Their
// names must not be mangled (thus the extern "C").
extern "C" {
  EXPORT PService* DOMEPluginCreate() {
    return new MathematicaModule("MathematicaModule");
  }
  //always returns the class name
  EXPORT const char* DOMEPluginName() { return "MathematicaModule"; }
}

MathematicaModule::MathematicaModule(const char* name) : PContainerModule(name)
{
  MATHEMATICA_DEBUG("Entering MathematicaModule default constructor.");
}

MathematicaModule::MathematicaModule(const MathematicaModule& x) : PContainerModule(x)
{
  MATHEMATICA_DEBUG("Entering MathematicaModule copy constructor.");

  ep = NULL;
  lp = NULL;

  fileNameService = new PStringModule(MATHEMATICA_FILENAME);

  inputContainer = new PContainerModule(MATHEMATICA_INPUTS);

  outputContainer = new PContainerModule(MATHEMATICA_OUTPUTS);

  properties = new PContainerModule(MATHEMATICA_PROPERTIES);
  isOnline = new PBooleanModule(MATHEMATICA_IS_ONLINE, pTrue);
  isRunning = new PBooleanModule(MATHEMATICA_IS_RUNNING, pFalse);
  properties->addService(isOnline);
  properties->addService(isRunning);

  addService(fileNameService);
  addService(inputContainer);
  addService(outputContainer);
  addService(properties);

  startListening(fileNameService);

  MATHEMATICA_DEBUG("Leaving  MathematicaModule copy constructor.");
}

MathematicaModule::~MathematicaModule() {
  MATHEMATICA_DEBUG("Entering MathematicaModule destructor.");
  closeLink();
}

void MathematicaModule::react(const PSpeaker&speaker, const PMessage&)
{
  if (isOnline->value() == pFalse) return;

  const PService* speakerService = DYN_CAST(const PService*, &speaker);
  MATHEMATICA_DEBUG("Entering MathematicaModule::react for " << speakerService->parent()-
>name()<<"."<<speakerService->name());
```

```
  if (isRunning->value() == pFalse)
  {
    isRunning->value(pTrue);

    if (fileNameService == speakerService)
    {
      readPublisherFile(fileNameService->value());
    }
    else
    {
      for (int i=0; i<inputs.size(); i++)
      {
        if (inputs[i]->service == speakerService)
        {
          setValue((ServiceModule*) inputs[i]);
          updateAllOutputs();
        }
      }
    }
    isRunning->value(pFalse);
  }
} // end react()

PStatus MathematicaModule::read(PString& s) {
  MATHEMATICA_DEBUG("Entering MathematicaModule::read.");
  PContainerModule::read(s);
  PService* service;

  service = getServiceByName(MATHEMATICA_FILENAME);
  fileNameService = CON_CAST(PStringModule*, DYN_CAST(const PStringModule*, service));

  if (fileNameService == NULL)
  {
    MATHEMATICA_ERROR("fileNameService is null in read()");
    return P_ERR;
  }

  service = getServiceByName(MATHEMATICA_INPUTS);
  inputContainer = CON_CAST(PContainerModule*, DYN_CAST(const PContainerModule*, service));

  service = getServiceByName(MATHEMATICA_OUTPUTS);
  outputContainer = CON_CAST(PContainerModule*, DYN_CAST(const PContainerModule*, service));

  service = getServiceByName(MATHEMATICA_PROPERTIES);
  properties = CON_CAST(PContainerModule*, DYN_CAST(const PContainerModule*, service));

  if (properties)
  {
    service = properties->getServiceByName(MATHEMATICA_IS_ONLINE);
    isOnline = CON_CAST(PBooleanModule*, DYN_CAST(const PBooleanModule*, service));

    service = properties->getServiceByName(MATHEMATICA_IS_RUNNING);
    isRunning = CON_CAST(PBooleanModule*, DYN_CAST(const PBooleanModule*, service));
  }else
    MATHEMATICA_ERROR("Properties is null in read()");

  for (int i = inputContainer->nServices(); i>0; i--)
  {
    inputContainer->remService(i-1);
  }
  for (i = outputContainer->nServices(); i>0; i--)
  {
    outputContainer->remService(i-1);
  }

  readPublisherFile(fileNameService->value());
  MATHEMATICA_DEBUG("Leaving MathematicaModule::read.");
  return P_OK;
} // end read()

void MathematicaModule::updateAllOutputs()
{
  try
  {
    MATHEMATICA_DEBUG("Entering MathematicaModule::updateAllOutputs");

    loadModel(mathlinkFilename);

    for (int i=0; i<outputs.size(); i++)
    {
      updateValue( (ServiceModule*) outputs[i] );
    }
    MATHEMATICA_DEBUG("Leaving MathematicaModule::updateAllOutputs");
  } catch(...) {
    MATHEMATICA_ERROR("ERROR: Caught Exception in setValue()!!!!!!!!!!!!!!!!!!");
  }
}

// use "Set" function in MathLink to set the value of input
```

```
// discard returning packet which is simply echo of that value
void MathematicaModule::setValue(ServiceModule* module)
{
  if (lp == NULL) return;

  MATHEMATICA_DEBUG("Entering MathematicaModule::setValue");
  MATHEMATICA_DEBUG("name " << module->name.c_str());

  // API calls to set value of variable
  MLPutFunction(lp, "EvaluatePacket", 1);
   MLPutFunction(lp, "Set", 2);
     MLPutSymbol(lp, module->name.c_str());
     MLPutReal(lp, module->service->value());
  MLEndPacket(lp);

  //MLFlush() ensures that the packet is sent immediately
  MLFlush(lp);

  int pkt;

  // discarding all packets before a RETURNPKT
  while ((pkt = MLNextPacket(lp)) && pkt != RETURNPKT)
    MLNewPacket(lp);

  if (!pkt) {
    MATHEMATICA_ERROR(MLErrorMessage(lp));
    MLClearError(lp);
  } else {
    //readReturnPacket(lp);
    MLNewPacket(lp);
  }
  MATHEMATICA_DEBUG("Leaving MathematicaModule::setValue");
  return;
} // end setValue()

// use "Get" function in MathLink to get the value of input
double MathematicaModule::getValue(string name)
{
  if (lp == NULL)
  {
    MATHEMATICA_ERROR("Leaving getValue lp == NULL");
    return 0.123456789;
  }

  MATHEMATICA_DEBUG("Entering MathematicaModule::getValue for " << name);

  // API calls to get value of variable
  MLPutFunction(lp, "EvaluatePacket", 1);
    MLPutFunction(lp, "Get", 1);
      MLPutSymbol(lp, name.c_str());
  MLEndPacket(lp);

  //MLFlush() ensures that the packet is sent immediately
  MLFlush(lp);

  int pkt;
  while ((pkt = MLNextPacket(lp)) && pkt != RETURNPKT)
    MLNewPacket(lp);

  if (!pkt) {
    MATHEMATICA_ERROR(MLErrorMessage(lp));
    MLClearError(lp);
    return 0.123456789;
  } else {
    double result;

    // Ignore all expressions up to a REAL number
    // Ignoring the FUNCTION head and symbols

    while (MLGetNext(lp) != MLTKREAL) ;

    MLGetReal(lp, &result);

    /* for debugging

    char buffer[200];
    int j;
    j = sprintf( buffer, "Real: %f", result);
    MATHEMATICA_DEBUG(buffer);
    */

    MLNewPacket(lp);

    return result;
  }

  return 0.123456789;
} // end getValue()

void MathematicaModule::updateValue(ServiceModule* module)
```

```
{
  module->service->value(getValue(module->name));
}

void MathematicaModule::readPublisherFile(const char* filename)
{
  if (string(filename) == "") return;

  MATHEMATICA_DEBUG("Entering MathematicaModule::readPublisherFile for "<<filename);

  ifstream inFile(filename, ios::in);

  if( !inFile) {
    MATHEMATICA_ERROR("File " << filename << " not found!");
    return;
  }
  inFile.clear();

  char mathlinkFilenameChar[200];
  inFile.getline ( mathlinkFilenameChar, 200, '\n');
  mathlinkFilename = string(mathlinkFilenameChar);

  // check if model file exists
  ifstream modelFile(mathlinkFilenameChar, ios::in);
  if (!modelFile) {
    MATHEMATICA_ERROR("Model file " << mathlinkFilenameChar << " not found!");
    modelFile.close();
    return;
  }

  openLink();

  string type, varName, domeName, valueString, unit;

  while(inFile >> type >> varName >> domeName >> valueString >> unit)
  {
    ServiceModule* module = new ServiceModule;
    module->name = string(varName);
    module->domeName = string (domeName);

    double value = atof(valueString.c_str());

    // some string operations to convert '_' characters in unit to blank spaces

    for (int ix = 0; ix < unit.size(); ++ix)
      if (unit[ix] == '_' )
        unit[ix] = ' ';


    if (type == MATHEMATICA_TYPE_INPUT)
    {
      MATHEMATICA_DEBUG("Found input named: "<< varName.c_str());
      MATHEMATICA_DEBUG(unit.c_str());

      module->service = new PRealModule(domeName.c_str(), value, unit.c_str());
      inputs.add(module);
      inputContainer->addService(module->service);

      setValue(module);

      startListening(module->service);
    }
    else if(type == MATHEMATICA_TYPE_OUTPUT)
    {
      MATHEMATICA_DEBUG("Found output named: "<< varName.c_str());
      MATHEMATICA_DEBUG(unit.c_str());
      module->service = new PRealModule(domeName.c_str(), value, unit.c_str());

      outputs.add(module);
      outputContainer->addService(module->service);
    }
  }

  inFile.close();
  return;
} // end readPublisherFile()


// Opening Communication link with Mathlink in launch mode
void MathematicaModule::openLink()
{
  int argc = 4;
  char *argv[5] = {"-linkname", "c:\\Program Files\\Wolfram Research\\Mathematica\\4.1\\Math -mathlink", "-
linkmode", "launch", NULL};

  if(!ep && (ep = MLInitialize(NULL)) == NULL) {
    MATHEMATICA_ERROR("Error in MLInitialize");
    return;
  }
```

```
  if(lp == NULL)
    lp = MLOpen(argc, argv);

  MATHEMATICA_DEBUG("Link opened.");
  return;
}

// Closing Communication link with Mathlink
void MathematicaModule::closeLink()
{
  if(lp)
  {
    MLPutFunction(lp, "Exit", 0);
    MLClose(lp);
  }

  lp = NULL;
}

// use "Get" function in MathLink to load model
// also checking for warning or error messages from the kernel in case of
// syntax errors in model file

// Might still want to look into other file extensions
void MathematicaModule::loadModel (string modelName)
{
  if (lp == NULL)
  {
    MATHEMATICA_ERROR("Leaving loadModel lp == NULL");
    return;
  }

  MATHEMATICA_DEBUG("Entering loadModel for " << modelName.c_str());

  int pkt;

  MLPutFunction(lp, "EvaluatePacket", 1);
    MLPutFunction(lp, "Get", 1);
      MLPutString(lp, modelName.c_str());
  MLEndPacket(lp);
  MLFlush(lp);

  // 3 cases
      // 1. model loaded correctly and SYMBOL NULL is returned wrapped in a head
      // 2. model loaded correctly and returned the value of the output variable wrapped in a head
      // 3. model didn't load correctly and returned a SYMBOL of MessagePacket

  // have to check for MESSAGEPKTs, because there might be an error in loading the model

  while ( (pkt = MLNextPacket(lp)) && pkt != RETURNPKT && pkt != MESSAGEPKT )
    MLNewPacket(lp);

  if (!pkt) {
    MATHEMATICA_ERROR(MLErrorMessage(lp));
    MLClearError(lp);
  }

  // if kernel is sending back warning or error message
  if (pkt == MESSAGEPKT)
  {
    kcharp_ct name, tag;

    // hopefully a helpful error message

    MLGetSymbol(lp, &name);
    MLDisownSymbol(lp,name);

    MLGetString(lp, &tag);
    MLDisownString(lp,tag);

    MATHEMATICA_ERROR("Error in " << modelName.c_str() << " " << name << " " << tag);

    // to break out of loop set lp = NULL
    lp = NULL;

    return;
  } else {
    readReturnPacket(lp);
    MLNewPacket(lp);
    return;
  }

  MATHEMATICA_DEBUG("Leaving loadModel for "<<modelName.c_str());
  return;
} // end loadModel()


// helper function for debugging
// prints out the contents of the packet
void MathematicaModule::readReturnPacket (MLINK lp) {
```

```
kcharp_ct s;
char buffer[200];
int n;
long i, len, j;
double r;

switch (MLGetNext(lp)) {
case MLTKREAL:
  MLGetReal(lp, &r);
  j = sprintf( buffer, "Real: %f", r);
  MATHEMATICA_DEBUG(buffer);
  break;
case MLTKFUNC:
  MATHEMATICA_DEBUG("function: ");

  if (MLGetArgCount(lp, &len) == 0) {
    MATHEMATICA_ERROR(MLErrorMessage(lp));
  } else {
    readReturnPacket(lp);
    MATHEMATICA_DEBUG("[");
    for (i = 1; i <= len; ++i) {
      readReturnPacket(lp);
      if (i != len) MATHEMATICA_DEBUG(", ");
    }
    MATHEMATICA_DEBUG("]");
  }
  break;
case MLTKSYM:
  MLGetSymbol(lp,&s);
  MATHEMATICA_DEBUG("symbol: " << s);
  MLDisownSymbol(lp,s);
  break;
case MLTKSTR:
  MLGetString(lp, &s);
  MATHEMATICA_DEBUG("string: " << s);
  MLDisownString(lp,s);
  break;
case MLTKINT:
  MLGetInteger(lp,&n);
  MATHEMATICA_DEBUG(n);
  break;
case MLTKERROR:
default:
  MATHEMATICA_ERROR(MLErrorMessage(lp));
}
return;
} // end readReturnPacket()
```

Appendix B: MathLink Methods

MathLink is a library of functions that implement a protocol for sending and receiving Mathematica expressions. MathLink enables external programs to utilize the Mathematica kernel's computational and programming services. Compiled below is a list of MathLink calls that are used in the wrapper implementation.

```
MLEnvironment MLInitialize(NULL)

MLINK MLOpen(argc, argv)

 int argc = 4;
 char *argv[5] = {"-linkname", "c:\\Program Files\\Wolfram
    Research\\Mathematica\\4.1\\Math -mathlink", "-linkmode", "launch",
    NULL};

MLDeinitialize(MLEnvironment )

MLClose(MLINK )

MLPutFunction(MLINK , const char* packetHead, int argCount)

MLPut$dataType(MLINK , $dataType value)

MLEndPacket(MLINK )

MLFlush(MLINK ): ensures that packets is not buffered but sent
                 immediately to the kernel.

int MLNextPacket(MLINK ): opens next packet

MLNewPacket(MLINK ): discards open packet

MLGet$dataType(MLINK , $dataType*)

int MLGetArgCount(MLINK , long* length)

int MLGetType(MLINK )
```

Appendix C: Data Transfer in MathLink

This appendix illustrates the data transfer mechanism in MathLink. When the kernel is running in "mathlink mode", all communication takes place in the form of packets. The packet head or label conveys how its contents should be processed.

The set of MathLink calls in Figure 12 generate an error message from the kernel, because the symbol `I` is protected within Mathematica and can not be used as a variable name.

```
MLPutFunction(lp, "EvaluatePacket", 1);
  MLPutFunction(lp, "Set", 2);
    MLPutSymbol(lp, "I");
    MLPutReal(lp, 1);
MLEndPacket(lp);
```

Figure 12 MathLink calls to evaluate the expression `I = 1` in the kernel

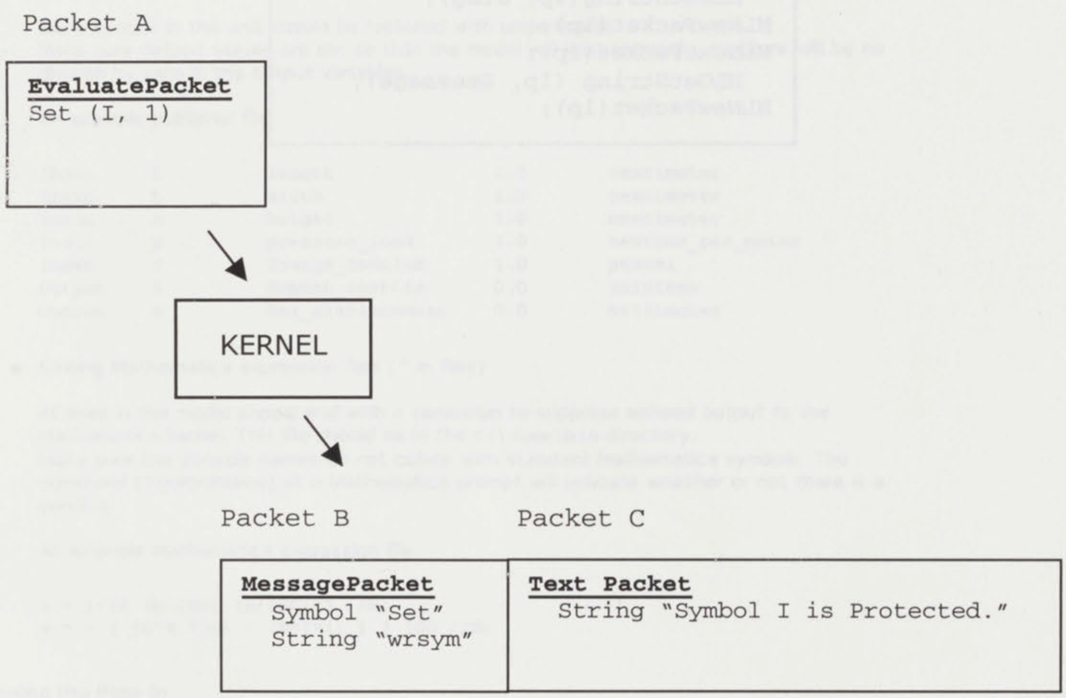

Figure 13 Packets created by the MathLink method calls in Figure 12

The packets created by the MathLink method calls from Figure 12 are shown in Figure 13. Packet A is an `EvaluatePacket` and contains the expression `Set[I,1]`, where `I` is a symbol and 1 is a real number. Upon receiving packet A, the kernel responds with two packets, packet B and C. Packet B is a `MessagePacket`, the packet type used to indicating warnings or error messages from the kernel. It contains a symbol and string outlining the error. The `MessagePacket` B is always followed by a `TextPacket` containing the actual error message.

To read the contents of packets B and C a series of MathLink methods need to be called, but first, the packet types must be determined. `MLNextPacket` returns a predefined integer constant that encodes the packet head. Packets sent from the kernel can be of five types: `InputNamePacket` and `OutputNamePacket`; `ReturnPacket`, `ReturnTextPacket`, and `ReturnExpressionPacket`; `MessagePacket`; `TextPacket`; or `DisplayPacket`. The only types of packets that the wrapper implementation expects are `ReturnPacket`, `MessagePacket`, and `TextPacket`. Once the packet type is determined, the contents of the packet need to be read out or discarded with a call to MLNewPacket. Figure 14 lists the sequence of method calls to read out the contents of packets B and C.

Figure 14 MathLink calls to read contents of returned packages

```
MLNextPacket(lp);
   MLGetSymbol(lp, &name);
   MLGetString(lp, &tag);
MLNewPacket(lp);
MLNextPacket(lp);
   MLGetString (lp, &message);
MLNewPacket(lp);
```

Appendix C: Wrapper Documentation (HTML)

DOME module documentation - Mathematica - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Documentation for Mathematica Module

Author: Kathy Lee
May 5, 2001

This module enables DOME to interact with the computing software package Mathematica. (tested on version 4.1) The module uses MathLink, the Mathematica API.

Setting Up

- Writing the Publisher Data Files

  The publisher file is an external text file that dictates how the Mathematica model is defined in the DOME domain. The first line of the publisher file contains the file name of the Mathematica model. Files with extension *.m are Mathematica expression files in plain text format. The following lines consist five columns with fields as defined below.

  Input/Output variableName(Mathematica) variableName(DOME) initialValue unit

  Blank spaces in the unit should be replaced with underscores.
  Make sure default values are set so that the model will load correctly. ie. there will be no division by zero in the output variables.

  An example publisher file

```
Input    L    length            1.0    centimeter
Input    b    width             1.0    centimeter
Input    h    height            1.0    centimeter
Input    p    pressure_load     1.0    newtons_per_meter
Input    Y    Youngs_Modulus    1.0    pascal
Output   i    Moment_inertia    0.0    unitless
Output   w    Max_displacement  0.0    millimeter
```

- Writing Mathematica expression files (*.m files)

  All lines in the model should end with a semicolon to suppress echoed output to the Mathematica kernel. This file should be in the `C:\dome\bin` directory.
  Make sure the variable names do not collide with standard Mathematica symbols. The command [?symbolName] at a Mathematica prompt will indicate whether or not there is a conflict.

  An example Mathematica expression file

```
i = 1/12 (b/100) (h/100)^3 //N;
w = - ( (L^4 * p) / (8*Y*i) ) * 100 //N;
```

Running the Plug-in

- Add a Mathematica Module: Add -> Programs -> Mathematica
- Double-click on the icon to open the user interface.
- Enter the full path and name of the publisher file.
  (eg. `C:\development\Mathematica\publisher_file.txt`)
- When closing remove the MathematicaModule under Top_Level_Module so that the destructor is called and there are no floating math.exe processes.

My Computer

2497-64