To the Graduate Council:

I am submitting herewith a thesis written by Delphy Jean Nypaver entitled "A Java Interface for the Integration of Legacy Numerical Software into the NetSolve System." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Jack Dongarra, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

Associate Vice Chancellor
and Dean of the Graduate School

A Java Interface for the Integration of Legacy Numerical Software into the NetSolve System

A Thesis

Presented for the

Master of Science Degree

The University of Tennessee, Knoxville

Delphy Jean Nypaver

May 1997

Acknowledgments

I would like to thank Dr. Jack Dongarra, my major professor, for giving me the opportunity to prepare this work. I would also like to thank both him and Henri Casanova for their patience and guidance through the entire project. Thanks also to my thesis committee, Dr. Michael Berry and Dr. Jim Plank, for their assistance and guidance in the preparation of this work.

Special thanks goes to my family for their never ending extension of love, support and encouragement through this and all my endeavors in life.

And in loving memory of my father. He has always been, and will always continue to be, a constant and significant driving force in my life.

Abstract

NetSolve is a client-server application that enables users to solve complex scientific problems remotely. The system accesses both hardware and software computational resources distributed across a network. NetSolve searches for computational resources on a network, chooses the best one available, solves a problem using retry for fault tolerance, and returns the answer to the user. A load balancing policy is implemented in the NetSolve system to ensure good performance by enabling the system to utilize available the computational resources as effectively as possible.

Some objectives of the NetSolve project are user-friendly interfaces, more productive use of system resources, and the integration of any numerical software as a resource into the NetSolve system. Numerous interfaces have been designed and implemented which enable users to access and use the system more easily. An agent-based design has been implemented to ensure more efficient utilization of the system's resources. Thus, the work presented in this paper focuses on the goal of arbitrary software integration into the NetSolve System.

One of the key characteristics of any successful software package is versatility. In order to ensure the success of NetSolve, the system has to be able to incorporate any piece of numerical software, regardless of its format, with relative ease. There can be no restrictions on the type of numerical software that can be integrated into the system.

The process of adding and updating NetSolve computational resources or servers is greatly enhanced and facilitated by the addition of a Java interface. Java is a new object-oriented programming language and environment that is secure, dynamic, architecture-neutral and multi-threaded. The NetSolve server interface is designed so that any scientific software can be translated and compiled into a common source code. The Java interface design is able to incorporate some of the tasks previously implemented in the NetSolve server compiler. The Java interface also guarantees that every formal language file presented to the compiler is always in a predetermined error-free format. Thus, the implementation of the Java interface resulted in the NetSolve server compiler being redesigned into a more efficient and effective tool.

With the use of the Java interface, the NetSolve resource administrator can create and dynamically change a NetSolve computational server simply and clearly. The administrator no longer needs to know about the complexity of the underlying formal descriptive language utilized by the system. The only information required is that of the software package being added to the system. So now with the development of the Java interface, in conjunction with the use and translation of the formal descriptive language used by the NetSolve server, any scientific numerical library can be integrated into the NetSolve system.

Contents

List of Figures

Chapter 1

Introduction

The motivation behind the NetSolve project was to devise a fast, efficient, easy to use mechanism to effectively solve large computational problems, regardless of the type of machine one happens to be using [2]. With today's networking capabilities, users should not be restricted by the hardware or software limitations of their local machine, nor should they get bogged down with programming requirements and restrictions. The ability to solve large scientific problems effectively and efficiently should only be a mouse click or function call away.

NetSolve has been designed to overcome hardware and software restrictions so that computational scientific software can be available to any user anywhere on a network [4]. It is a client-server application designed to solve any computational problem over a network. A user can access the NetSolve system through a number of easy to follow interfaces, including C, Fortran, MATLAB, and the World Wide

Web. NetSolve implements an agent-based design to optimize the best use of the computational resources available. The agent-based design includes load balancing and fault tolerance. Since simultaneous requests can be sent to the NetSolve server in parallel, the load balancing policy is employed to optimally map all requests to the server in a way that solves all problems most efficiently. NetSolve is a distributed system with many different server administrators. The fault tolerance scheme used by NetSolve ensures that the deficiency of one or more servers in the system does not cause a catastrophic failure. In addition, the number of side effects generated by a failure in the system is kept to a minimum in order to prevent a drop in performance.

As NetSolve progresses, improvements continue to be made to the system. These improvements are mainly in the area of facilitated use when accessing the NetSolve system through the different client interfaces and in the management of the NetSolve servers. The work presented here concentrates on the area of server management. In order to encourage server administrators to add all types of numerical software, the system has to be able to interpret and understand the software packages as they are, not requiring users to change the function calls to the various packages in any way. Through the addition of the Java interface and the design and implementation of a declarative descriptive language, the NetSolve system is able to integrate any scientific numerical software package. The interface automatically transforms the data entered into a source code file

that can be compiled on any NetSolve server. Once a software package has been added to a server, the functionality of that package can be added to any other NetSolve server by simply downloading, translating and compiling the generated source code file. The descriptive language NetSolve implements is not machine dependent. Therefore the numerical software described in these files can easily be added to any type of NetSolve server regardless of the hardware or software platform.

What started out as a text editing process that required patience and knowledge of the underlying system, has been transposed into a simple, easy to follow interface that only requires the server administrators to be informed about the scientific package that they are adding to the server. The administrators are led through a sequence of screens that allows them to setup and compile their server with ease. The interface automatically generates a source code file in the form of a descriptive language that is known to all NetSolve servers. Therefore software packages only need to be added one time to one server to produce the source code. Once the source code describing the numerical functionality of a software package is produced, it can be incorporated into any server on the system.

Chapter 2

Overview of the NetSolve System

NetSolve is a network server for solving computational science problems [1]. The NetSolve system enables users to remotely access computational resources over the network. NetSolve has the ability to find the best computational resource available, solve the problem and return the result to the user.

2.1 System Architecture

Figure 2.1 illustrates the concept behind the NetSolve system. The system is divided into three separate components - the client, the agent, and the pool of resources or servers. Each component is used to solve a problem, starting with

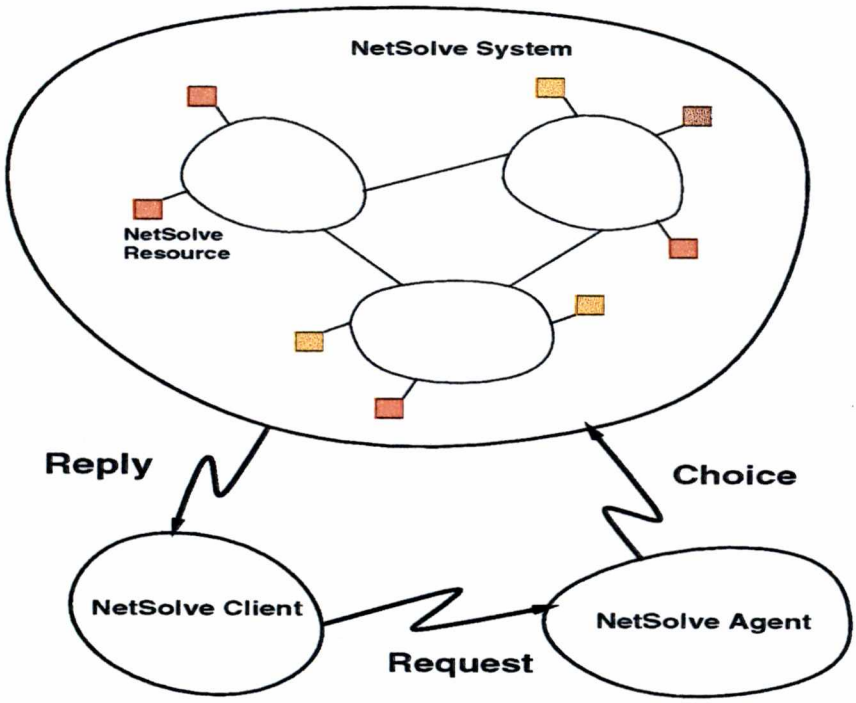

Figure 2.1: The NetSolve System

the client, who initiates the request. Then the agent uses its knowledge of the different computational servers to choose the "best" server and sends that server the client's request. The server uses the numerical libraries which have been installed to solve the problem and then returns the result back to the client.

TCP/IP sockets are used to communicate between the different layers of the NetSolve system. The machines in the system are connected through a network. This network can be a simple local-area network or a complex wide-area network. The machines in the network can be heterogeneous, which means that they can have different data formats, but still have access to all the computational servers.

A description of each component and the role they play in the NetSolve system is described below.

2.2 Client

A client is either a program or user interacting with one of the user-friendly NetSolve interfaces. Both the programming interface and the MATLAB interface can issue a request in the NetSolve system to perform a numerical computation. A user can access the client portion of the system by either calling the NetSolve system directly through C or Fortran function calls inside a user program or by accessing the system indirectly through one of the available NetSolve interfaces. This is where the user enters the NetSolve system. The client component is used

to send a request through the agent to a server to solve a problem.

Various means of sending a request for a problem to be solved by the NetSolve system are available to the user. Below is a brief description of the client interfaces accessible to a user.

2.2.1 Using the Client to Solve a Problem

Each interface provides the user with a way to define a problem for the NetSolve system. But first the user must know what problems are solvable through the NetSolve system. The World Wide Web can be used to find out which problems are handled by which servers and what servers are available. In addition, if the user is writing a program to access the NetSolve system, they are informed of the calling sequence for each available problem.

Once the user has the setup information for the problem to be solved, input data can be defined for a given problem. Once the input data is defined, a call to the NetSolve system will return a result.

2.2.2 Client Interfaces

The client interfaces have been designed to make the use of the NetSolve system as simple and straightforward as possible. A number of different interfaces have been developed and they are classified into two groups - the interactive interfaces and the programming interfaces. One of the interactive interfaces provided by

the NetSolve system is a MATLAB interface. When using this interface, the user can benefit from the MATLAB [7] formalism while accessing the computational power of NetSolve. The programming interface allows the user to access the NetSolve system directly through function calls inside their C or Fortran program. Both the MATLAB interface and the programming interface provide blocking and nonblocking calls to the NetSolve system. Since NetSolve uses a load balancing scheme and problems can be solved simultaneously on different servers, the nonblocking calls allow user-level parallelism.

A Java interface for the client is currently being implemented to provide easy access to resources in the NetSolve system [5]. This interface enables the user to define a problem, access an agent and then display the solution all within one graphical tool. User-level parallelism is implemented in the Java interface through multi-threading, thus providing the user with a speedup in computations.

2.3 Agent

The agent analyzes the request by the client and chooses the best available computational resource for the given problem. "Best" is defined in terms of network latency and bandwidth between communicating machines, size of the data to be sent, size of the result received, size of the problem, complexity of the algorithm to be used and the performance of the host to do the computation, which is based

on its workload and raw performance. The agent makes the system fault tolerant and is also the implementor of the load balancing strategy of the system.

2.3.1 Fault Tolerance

It is possible for failures to occur at any level of the NetSolve system. But generally failures are caused by a network problem, a server disappearing from the system or a server failure. When a server fails, the agent keeps a record in its database, but does not remove the server until after a given time. If the server is restarted and/or the failures do not continue, the server remains in the system.

In order to keep the number of side effects generated by a failure as low as possible, the agent is designed to minimize its requests to poorly performing servers. If a computation fails, the client is informed and another request is made to a different server, but the user is never aware of the problems. The agent updates its view of the NetSolve server when errors occur. If a computational server is performing poorly and errors start to accumulate for that server, it will eventually be dropped by the agent. Failures that the agent detects include the server's host not being reachable, the server not running, a time-out, and the server failing during a computation. Since the agent implements this type of fault tolerance mechanism, the client will receive a solution to its request unless every resource in the system has failed or is unavailable.

2.3.2 Load Balancing

NetSolve performs computations over a network. This network can contain a large amount of machines, each with a variety of characteristics. By executing a load balancing strategy, the agent is able to choose the best machine out of its pool of servers to solve the given problem.

The agent is continuously aware of the configuration and status of the NetSolve system. Based on the agent's knowledge of the number of computational resources available, the problems each resource or server can solve, the load of each resource and their distance over the network, the agent determines which resource is most suitable for each request. The agent makes every effort to solve problems as efficiently as possible.

2.4 Server

Every host in the NetSolve system implements a NetSolve computational server or resource. These servers have access and knowledge about the scientific packages (libraries or stand-alone systems) accessible from the host machine. The utilization of a Java interface helps to ensure the easy creation and management of NetSolve computational servers.

2.4.1 Creation

A Java interface has been implemented which ensures the ability to have and maintain an extensive application resource by facilitating the creation of new computational servers and the addition of new problems to already existing computational servers. Code generation for the server is based on an input configuration file which describes each problem in a formal way. The Java interface leads the server administrator through a series of screens that allows them to easily input the necessary information about their scientific software package and setup their computational server. The interface automatically generates a source file that describes the problem in a formal descriptive language that all NetSolve servers can interpret. The Java interface ensures that problems added to the server have the correct formal information. A special compiler is provided by the NetSolve system which takes this source code file and compiles or translates it into actual C code that can be compiled on any computational server. The server uses this code to solve the computational problem. Once the source code is translated and compiled, the problem is added to the server's database of solvable resources. Users of the NetSolve system have access to any scientific package available on any server in the system.

2.4.2 Management

Since the agent enforces a fault tolerance scheme, the NetSolve system can be modified without disruption to the entire system. Therefore, any NetSolve server can be deleted or created at any time. Updating the server's pool of resource information is also easily maintained. Individually compiled sources of the computational resources enables problems to be added or taken off the server without interruption to the NetSolve system. In addition, once the formal description of available resources has been defined, the source code file can simply be downloaded and compiled on any server in the NetSolve system. Since the formal language source file is not machine dependent and can be interpreted by any configured server, the numerical functionality of any software package can be obtained on any machine by merely creating it once. Server administrators on any type of machine only need to obtain the already created source file, translate and then compile it into their own NetSolve server system.

Chapter 3

Creating a Computational Server

Through the implementation of a Java interface, the creation of a NetSolve computational server has been greatly facilitated. The NetSolve server administrator no longer has to be concerned with the formal language requirements for specifying a problem. Through the enforcement of mandatory entry of data, extensive error checking and automation of file generation, scientific software packages can be added to computational servers with confidence and ease.

3.1 The Need for an Interface

Before the Java interface was developed, the server administrator was forced to have in-depth knowledge of the underlying NetSolve system and the declarative formal language used by the system to define numerical resources. In order to create and maintain a computational server the server administrator had to know

and understand the formal language specifications and requirements for adding scientific software packages. Now the server administrator can setup a computational server by inputting information about the scientific software packages through a straightforward, easy to follow interface. The server administrator is totally unaware of the complexity of the underlying NetSolve system.

3.2 The Challenge

In order to make NetSolve a versatile tool, the system had to be able to incorporate any scientific software package into its repository of resources. The system had to be able to take any arbitrary set of problems, interpret them in a way that could be used by NetSolve and make them accessible to any user of the NetSolve system. With the diversity of software that the system needed to accept, it was unrealistic to assume that these packages would be changed in any way to meet the requirements of the NetSolve system. A solution had to be adapted that would enable the NetSolve server to accept any scientific computational package and transform it into something that could be used by any server on the network, regardless of its platform. Ideally, the addition of software packages needed to be as effortless as possible for the server administrator to ensure the addition of a wide range of computational packages.

3.3 The Solution

In order to ensure that the NetSolve system could interpret any scientific computational software package, a formal descriptive language was developed that can be translated into C code and compiled on any server in the NetSolve system. A Java interface was developed to make the addition of computational packages to the NetSolve server a clear and precise task. The interface requires the server administrator to input information about the scientific package that is required by the NetSolve system. After the needed data is entered by the server administrator, the interface automatically generates a file in the formal declarative language describing the package.

A compiler or translator is provided by the NetSolve system which takes this source code and transforms it into C code that can solve the scientific problems represented in the software package. The formal descriptive language file can be taken to any NetSolve server and translated into C source code. Then the C source code can be compiled and added as a NetSolve computational resource. Thus, once a software package has been created on one NetSolve server, the functionalities of that package are easily obtained by any computational server in the system. All that is required is the NetSolve translator and a C compiler. The formal language source code file can be downloaded to any NetSolve server, translated, compiled and then added to that system's repository of resources. Figure

3.1 illustrates the process of creating a NetSolve computational server. Figure 3.2 illustrates the process of downloading a formal language file and creating a NetSolve computational server.

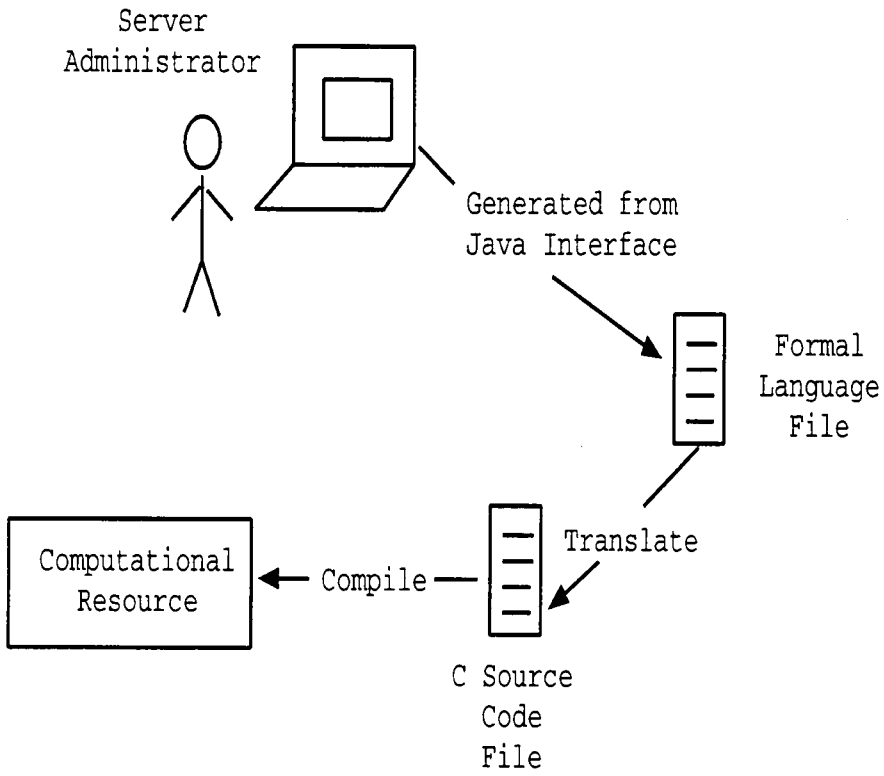

Figure 3.1: The Process of Creating a Computational Resource

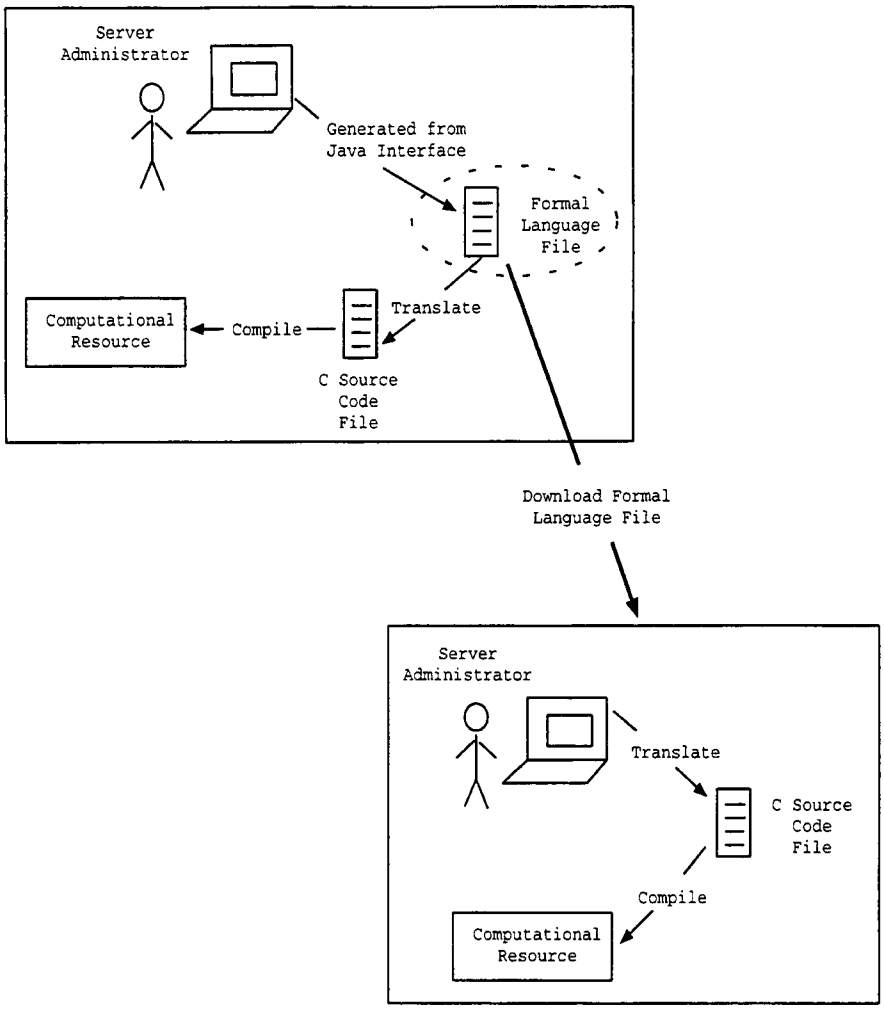

Figure 3.2: The Process of Downloading a Formal Language File and Creating a Computational Resource

3.4 Why Java?

The NetSolve server administrators needed to be able to develop their computational servers through graphical applications. Java applications or *applets* are interactive networked programs. They react instantly to user inputs and change

dynamically with their requests. In addition, by using Java, the interface to the NetSolve computational server can be incorporated into World Wide Web pages, which means that repositories of scientific software packages can be added from anywhere on the network.

Security restrictions enforced by Web browsers are upheld by Java applets. Java applets are inherently prohibited from gaining access to arbitrary system resources, either on the Web server or client machines, and are therefore reassuringly secure to use in the world of the internet [8].

Thus, by using Java to implement the interface to the NetSolve computational server, the NetSolve objectives of versatility, graphical enhancement, ease of use and clarity were maintained. Java applets automatically provided the tools needed through their dynamic graphical capabilities and their use and accessibility through the World Wide Web.

Chapter 4

Language Specification

In order to incorporate any scientific software package into the NetSolve system, problems need to be defined in a general, yet inclusive way. The problem definition needs to be general enough to include any arbitrary software package, but inclusive enough to allow the NetSolve system to obtain the information it needs to solve the problem. The formal declarative language used by the NetSolve system was developed so that the data needed to solve computational requests could be presented generically to the NetSolve server.

4.1 Overview

The Java interface creates a text file that is translated, compiled and then used by the NetSolve computational server to solve problems that are requested by the NetSolve client. These files represent the software repositories stored and

processed on the computational server. Therefore it is important that the information stored in these files defines the problem properly, and gives the NetSolve computational server all the data that it needs to process and solve the client's request.

4.2 The Implementation Need

NetSolve presented the challenge of coming up with a process to define any arbitrary problem and translate that problem into a form that the computational solver could understand. NetSolve had to be able to incorporate diverse scientific software packages into the computational resources of the system. The diversity of the software added another level of complexity: being able to understand and translate any type of scientific computational software. Also the server had to be able to manage a vast amount of problems. This required that the NetSolve server administrator be able to add scientific software packages to the NetSolve server in an understandable yet uniform way. The Java interface had to be able to translate the data on any given software package into a formal language that the NetSolve system could use. Then the server had to be able to transform the formal language file into a compilable C source code file so that the problem could be computed by the NetSolve server.

In addition to the formal language specification, NetSolve had to provide a

compiler or translator that could take the descriptive language source code file generated by the interface and produce something that the NetSolve server software could compile and use to solve the interpreted computation. The translated source had to be stored on the NetSolve server machine and be made accessible from any client machine, regardless of the hardware or software platforms. The objective was to keep the details of the NetSolve formal descriptive language and the complexities of the NetSolve system from overwhelming and confusing the server administrator. The server administrator should only be responsible for knowing and inputting the details of the software package they were installing.

Since the NetSolve server compiler generates C source code that can be compiled and used by any NetSolve server, only one person has to input the information about the computational software. The resources of the software can be added to any server in the NetSolve system by just downloading the formal language source code, translating it into C source code and then compiling the C code on the platform of each server machine. The Java interface ensures that the formal language code produced is error-free and translatable on any host machine. Once the formal language source code file has been generated for one server, the entire functionality of the scientific software package can be added to another NetSolve server in a matter of minutes.

4.3 Problem Description

A problem refers to a computational routine represented in any scientific software package. Each function or problem has a set of inputs and outputs and a predetermined calling sequence requirement. A declarative language was developed to formally describe any function.

The general description given to a problem in the NetSolve system is as a 3-tuple. The 3-tuple consists of the name of the problem, a list of input objects and a list of output objects. Input and output objects contain information pertaining to the structure of the data, which can be *MATRIX*, *VECTOR* or *SCALAR*. The type of the input and output data is also required. NetSolve supports the Fortran data types of *integer*, single and double precision *real*, single and double precision *complex*, and *character*. If the object type is a *VECTOR* or 1-D array, then the size of or number of elements in the vector is required as input. If the data type is a *MATRIX* or 2-D array, then the number of rows and the number of columns of the matrix and the leading dimension of the matrix are needed as inputs. Data structures requiring more than two dimensions are not available in this implementation of the problem definition.

The Java interface has been designed so that the server administrator does not need to know of the 3-tuple definition. All the server administrator needs to know is the definition of the problem they are adding to the NetSolve server.

The problem definition NetSolve uses is explained in Section 5.1.1. The problem definition is presented to the server administrator in a clear and precise way through the Java interface. The server administrator is required to enter a name for the problem and describe the inputs and outputs of the problem. The server administrator is presented with a pulldown menu selection of data structures and data types to choose from. The interface will not let the server administrator continue until all the required information defining the problem has been entered.

4.4 Calling Sequence

In addition to defining the inputs and outputs of the problem and their data types, the formal language describes and enforces the required calling sequence of the arguments to the C or Fortran library function call. One of the NetSolve objectives was to enable those already using C or Fortran scientific software packages to be able to switch to the NetSolve system without any modification to their existing code. For this reason, the server administrator is also required to define the format, as well as the definition, of the problem. In other words, the server administrator needs to define the C or Fortran calling sequence for the problem. This is the calling sequence that the NetSolve system will use and enforce when processing computational requests for the problem. The Java interface makes defining the calling sequence a logical, step-by-step graphical process.

The Java interface allows the server administrator to define the number of arguments that are required by the problem, and easily assign the inputs and outputs of the problem to arguments. In other words, the server administrator defines the parameter sequence to the function call that computes the problem. An argument in the calling sequence can also be defined as a *workspace* variable or an *external* function. If an input argument is *SCALAR* of type *integer*, it can be assigned to an array dimension variable.

The interface design enforces type checking. Objects of different types cannot be assigned to the same argument. The Java interface also ensures that each input and output object is assigned to an argument and that array dimensions are defined properly.

4.5 Objects

Objects are defined in the NetSolve system as either *MATRIX*, *VECTOR* or *SCALAR*. Object definitions are used for interaction with high-level interfaces, such as MATLAB. These objects are also used to encapsulate the definitions of arrays. For single dimensional arrays the *VECTOR* object includes the number of elements in the array. For two dimensional arrays the *MATRIX* object includes the leading dimension, the number of columns and the number of rows in the array.

When defining arrays for the calling sequence through the Java interface, a "*" can be used to define the number of rows in a 1-D array or the number of columns in a 2-D array. But when defining these arrays as *VECTOR* or *MATRIX* objects respectively, the "*" has to be resolved as either one of the input integer scalar arguments in the calling sequence, a constant or a formula. The number of rows in the 2-D array is not entered for the calling sequence, but is required in the *MATRIX* object definition. When defining a *MATRIX* object the number of rows is set to the leading dimension of the array, but this value can be changed to one of the input integer scalar arguments in the calling sequence, a constant or a formula.

4.6 Mnemonics

Mnemonics are a naming convention used to describe the inputs and outputs of a problem. These mnemonics are used when generating and translating the formal language source code file for the NetSolve server. The mnemonics for input objects start with *I* and are followed by the corresponding input number. For example, the mnemonic *I0* corresponds to the object pointing to *Input #0*. Output objects have a similar naming convention. The initial character is an *O* and is followed by the corresponding output number. If an argument is assigned to both an input and an output object, it will have two mnemonics associated with it, one for

each representation. The mnemonic for the number of rows of an input or output object begins with an m and is followed by the mnemonic of the object that it is associated with. For example, *mI0* corresponds to *the number of rows of input #0*. The number of columns of a matrix has the same format, with the m being replaced by an n. The mnemonic for the leading dimension of a matrix also uses the same format but replaces the m with an l.

Chapter 5

Java Interface Design

Two different types of classes were designed for the implementation of a Java interface to the NetSolve formal descriptive language. This language is needed by the NetSolve server in order to incorporate any arbitrary scientific computation into its resource repository. One type of class was needed to define the different parts of the declarative language. Another type was needed to define the different interactive screens in a way that would present the needed input data for the problem clearly and precisely to the server administrator. The design of each class used in the implementation of the Java Interface is described below.

5.1 Language Classes

A formal descriptive language was designed so that any piece of scientific computational software could be incorporated into the NetSolve system. Specific parts

of the language were separated into different classes. These classes include specifications for the problem, constants, formulas, variables, arithmetic expressions, mnemonics, input and output objects and lists of input and output objects.

5.1.1 Problem

The problem definition includes a problem name, problem path, problem description, language specification, the complexity of the problem and the number of arguments in the calling sequence. Lists of input and output objects, mnemonics, variables, constants and formulas used in the problem are also included. The initial portion of the problem definition is presented to the server administrator in the first screen of the Java interface.

The problem name tells the NetSolve server the name of the function being added. The problem path is a naming convention used by other NetSolve programs for displaying problems in the NetSolve system alphabetically. The problem description is used by the NetSolve client interface to help users of the NetSolve system get a better understanding of the functionality of the problem.

The language specification flag indicates whether the problem is a Fortran library function call or a C library function call. The complexity of the algorithm is considered by the agent when computing the time required on a given machine to perform the computation of the problem. This feature is sufficient to describe the complexity of a deterministic algorithm, such as a matrix multiply. However,

the server administrator must enter a "guess" for the complexity when dealing with algorithms whose complexity depends on the input data, such as an iterative algorithm.

Input and output objects are used to setup the calling sequence to the problem. Mnemonics are a naming convention used by the system to define variable names for the input and output objects (see Section 4.6). External functions, workspace variables, constants and formulas can also be used when defining a problem for computation.

5.1.2 NetSolveObject

The NetSolveObject class defines the input and output objects of the problem. The class definition includes the object type, data type and a description. The object type is the structure type of the object. Object types are defined as *NETSOLVE_MATRIX*, *NETSOLVE_VECTOR* and *NETSOLVE_SCALAR*. Workspace variables can only have a type of *NETSOLVE_VECTOR*. The data types of a NetSolve input/output object are integer, single precision real, double precision real, single precision complex, double precision complex, character.

Three methods were defined in the NetSolveObject class definition. The methods *datatype2str* and *datatype2Cstr* take the data type of an object and return the corresponding string describing the type. The method *objtype2str* takes the object type of an object and returns the corresponding descriptive string. These

methods are used by the Java interface for display purposes and when generating the formal language source code file.

5.1.3 Mnemonic

The Mnemonic class contains a string for the mnemonic's name. Methods are defined to determine if the given mnemonic is an input or output object. The *getDescription* method returns a character string describing the given mnemonic.

5.1.4 Variable

The Variable class contains information on workspace arguments. The definition includes a string for the variable name representing the workspace. A NetSolveObject, Constant and Formula class instance are also included in the definition. The NetSolveObject instance contains the structure and data type of the workspace variable. If the workspace definition contains a constant or formula, the Constant and Formula instances are used respectively to hold the input data. An integer variable is contained in the class definition for storing the argument number of the parameter representing the workspace dimension value. This argument is used to pass the size of the workspace in the calling sequence.

5.1.5 Constant

The Constant class contains a string for the mnemonic name assigned to the constant. A value variable is included in the class definition which contains the integer value of the constant. A Boolean flag is also in the definition to determine whether or not a value has been entered for the constant.

5.1.6 Formula

The Formula class contains a string for the name of the mnemonic the formula represents. A Boolean flag is also included in the definition to determine whether or not a formula has been entered. A *MathExp* class variable is used in the definition to define the *n-ary* tree representing the formula definition.

5.1.7 MathExp

The MathExp class represents the n-ary tree definition of a formula. The class definition contains a string for the name of the node and a left, right and middle MathExp variable. The *NextName* method returns the next variable name in the formula expression to be processed. The *buildTree* method takes the formula expression and recursively builds an n-ary tree definition. The *printStr* method is used to write the formula definition in the formal language source file. It is called recursively with the n-ary tree definition of the formula.

5.2 Interface Classes

The interface classes fill the language description classes with the data input by the NetSolve server administrator. This is the information needed to generate the formal descriptive language source code file. These classes present the problem definition, calling sequence, object definition, variable, formula and constant definitions and code generation tasks to the server administrator in a facilitated manner so that no knowledge of the underlying formal descriptive language is required by the server administrator.

5.2.1 NetSolveApplet

The NetSolveApplet class has two functions. First it initializes the Java applet and all of the underlying screens. It also presents the problem description information to the server administrator. This interface class takes the data input by the NetSolve server administrator and fills portions of the Problem class definition.

The method *check* verifies that all the required data is entered before allowing the server administrator to continue on to the calling sequence screen. The *clear* method clears all the input areas. The *setExample* method fills the input areas with data representing the example problem *dotproduct*, which is shown in Section 6.1. Figure 5.1 illustrates the problem description screen which is implemented in the NetSolveApplet class.

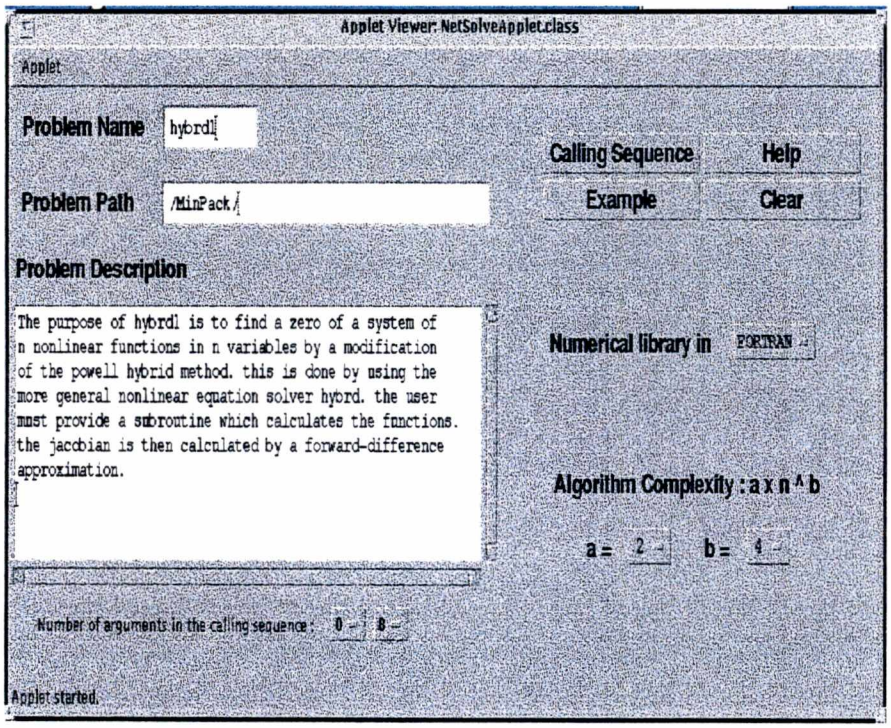

Figure 5.1: The Problem Description Screen Implemented in the NetSolveApplet Class

5.2.2 CallingSeqFrame

The CallingSeqFrame class defines the frame for entering the calling sequence of the problem. This class also maintains the input/output object list from the arguments defined in the calling sequence. The class definition consists of labels, panels, choice menus and buttons. The screen implemented in the CallingSeqFrame class is shown in Figure 5.2.

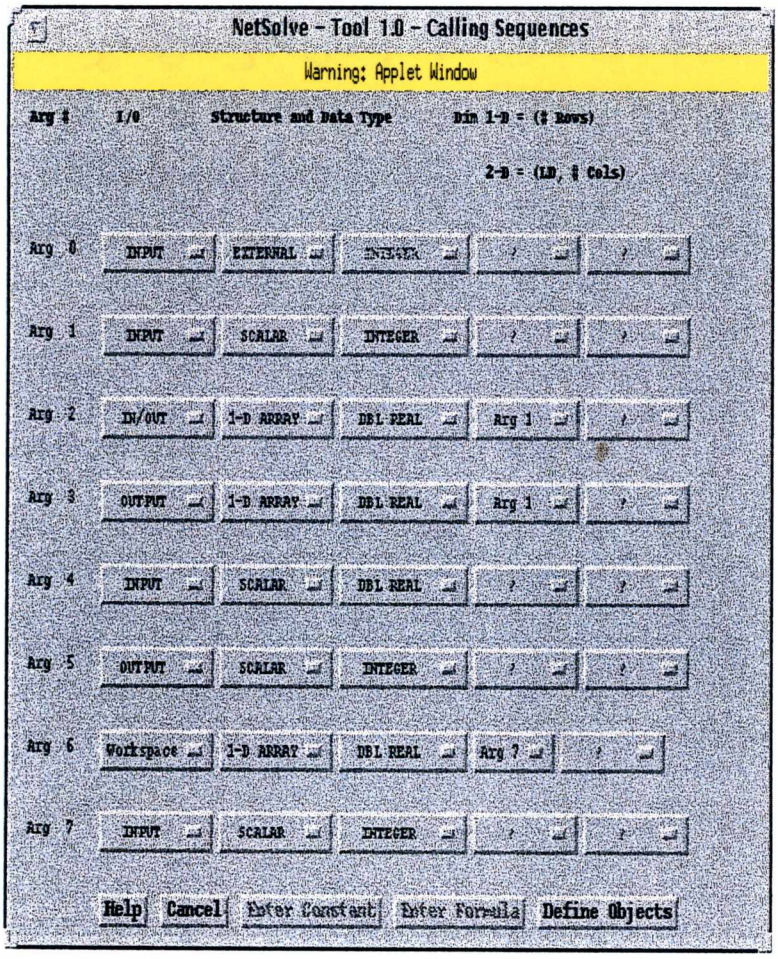

Figure 5.2: The Calling Sequence Screen Implemented in the CallingSeqFrame Class

The method *prepareProblem* establishes the problem mnemonics based on the data input for the arguments. The method *setExample* fills the screen with data from the example problem *ddot*. The screen is cleared through the *clear* method.

Several methods were defined in this interface class to deal with processing the data currently displayed on the screen. The methods *isOutput*, *isBoth*, *isInput*, *getMnemonic*, *getInputNo*, *getOutputNo*, and *getIOStr* analyze the input/output information. *isMatrix*, *isVector*, *isExtern*, *isIntVector*, *getTypeStr*, *getStructStr*, *getDataType* and *getStructType* manipulate structure and data type information. The methods *getIndexStr1*, *getIndexStr2*, *getItemStr1*, *getItemStr2*, *addDim*, *needsDim1*, *noDim1*, *noDim2*, *needsDim2*, *dimArgIndex* and *numDims* deal with data pertaining to array dimensions.

Information about the constants, formulas and variables defined in the problem is obtained through the *getTotConsts*, *getTotForm*, *getConstArg*, *getConstIndex*, *getFormArg*, *getFormIndex*, *getConstMnemonic*, *getFormMnemonic*, *getVarArg* , *isVar*, *isConst*, *isFormula*, *isVarConst* and *isVarFormula* methods. Error checking is performed in the methods *verifyDim*, *checkformulae*, *checkvars*, *checkconsts*, *needsStruct* and *needsData*.

5.2.3 ConstantsFrame

The ConstantsFrame class defines the frame used to input constant values. The frame is designed to change as input/output objects are put on or taken off the

problem's list of constants. A label is displayed for each object that requires a constant value. A text entry area is associated with each label for inputting the constant value for that object. A *buildList* method is defined to update the constant entry display area. The method *checkIntStr* is used to verify that all values entered for the constants are of type integer. The constants entry screen implemented in the ConstantsFrame class is shown in Figure 5.3.

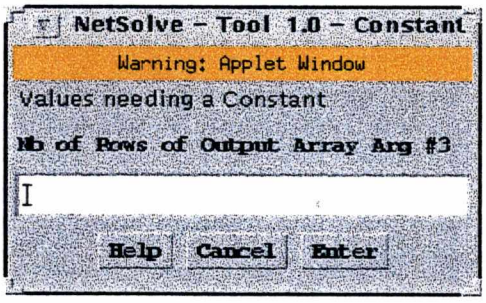

Figure 5.3: The Constant Entry Screen Implemented in the ConstantsFrame Class

5.2.4 FormulaFrame

The FormulaFrame class defines the frame used to input formulas. The formula entry area was designed to avoid free format text entry. Error checking is minimized if the server administrator has only valid entry options available, therefore all entry into the formula is done through buttons. Each formula has its own panel entry area which consists of valid arithmetic operations, parentheses, a right brace, an *if-then-else* conditional, conditional operations for the *if* statement, mnemonics that can be used in the formula expression and a constant entry area. Constants

are the only entry into the formula that have a free text entry format. Each formula has its own *Clear*, *Enter* and *Undo* buttons as well. All the formulas that need to be entered for the problem are displayed on the formula frame.

Only the mnemonics of type input integer scalar are displayed for entry into the formula. These mnemonics would include objects that represent the number of rows of a vector or matrix, the number of columns of a matrix, the leading dimension of a matrix and elements of an input integer vector object. Integer vector object buttons are labeled with the mnemonic of the integer vector followed by a left brace. The left brace is part of the mnemonic name, so there is no need for a single button to represent the left brace on the display. This is why only a right brace is offered as input.

The formula entry screen is updated as formulas are added to or taken out of the problem definition. The methods *buildList* and *buildPanels* are used to build and maintain the formula entry screen. Several error checking methods are in the class definition. When the *Enter* button is utilized the methods *checkIfs*, *checkIfThenElse*, *checkParenBalance*, *checkParenFormat*, *checkEmptyParen*, *checkBraceBalance*, *checkBraceFormat*, *checkEmptyBrace*, *checkCircularity*, *checkOperands*, *checkVars*, *checkMnArr*, and *checkFormulaStr* validate the formula. Errors that these methods look for include balanced *if-then-else* expressions, whether conditional operations alone are used in conditional statements, balanced parenthesis and braces, circularity, and proper format and use of parenthesis, braces,

arithmetic operations and variables inside arithmetic and conditional expressions. The formula entry screen implemented in the FormulaFrame class is shown in Figure 5.4.

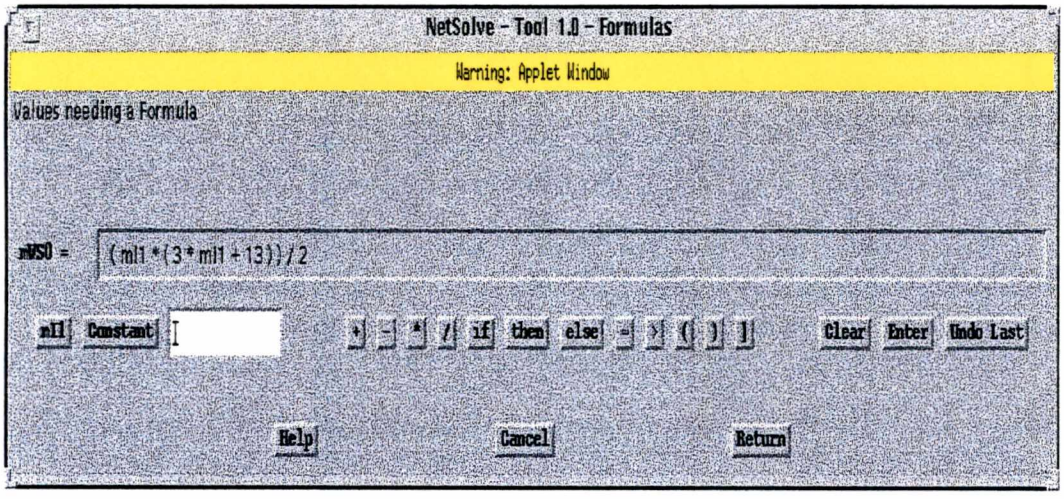

Figure 5.4: The Formula Entry Screen Implemented in the FormulaFrame Class

5.2.5 VariableFrame

The VariableFrame class implements the frame for defining workspace to be passed in the calling sequence. If arguments are assigned to workspace on the Calling Sequence window, this frame is displayed prior to the window for defining objects. The dimension of the workspace can be associated with an argument on the Calling Sequence window, but the actual size of the workspace is computed through the VariableFrame class. The dimension of the workspace is either assigned to a constant or a formula in the Workspace Definition or Variable screen.

The VariableFrame class definition contains labels and pulldown menus. A

label is displayed for each argument that requires a workspace assignment. The pulldown menus are for selecting either a constant or a formula definition for the dimension of the workspace.

The method *buildList* builds the NetSolve variable entry area. The methods *isConst*, *isForm*, *getFormVar*, *setNumConsts*, *setNumForms* manipulate the constant and formula data defined for the workspace variables. The implementation of the VariableFrame class is illustrated in Figure 5.5.

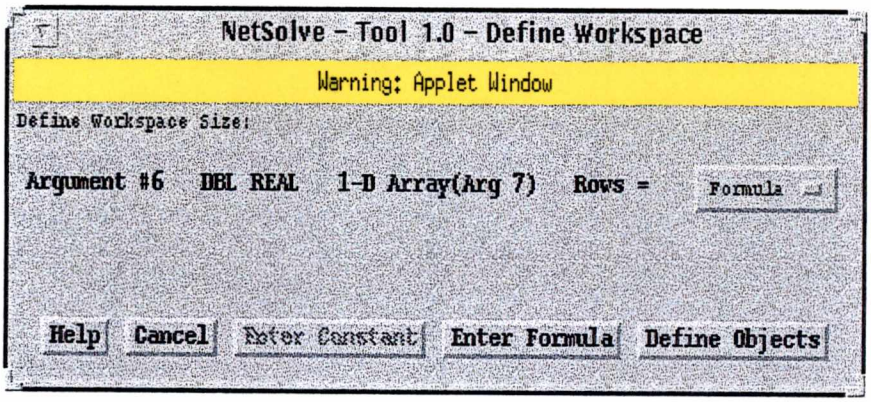

Figure 5.5: The Workspace Definition Screen Implemented in the VariableFrame Class

5.2.6 VarConstantsFrame

The VarConstantsFrame class extends the ConstantsFrame class. The functionality of the two classes are the same, but the VarConstantsFrame class is used to enter constants for a workspace variable definition. The constants entered in this frame are not a part of the list of constants defined for the problem. The constants entered through this frame are assigned to the Constant instance of the

Variable class (see Section 5.1.4).

5.2.7 VarFormulaFrame

The VarFormulaFrame class extends the FormulaFrame class. The relationship between the two classes is the same as that stated above for the VarConstants-Frame and the ConstantsFrame classes. The formulas entered through this frame are for workspace definitions and they assigned to the Formula instance of the Variable class (see Section 5.1.4).

5.2.8 DefineObjectFrame

The DefineObjectFrame class enables the server administrator to input object definitions. The class definition contains labels, choice menus, text areas, panels and buttons. Various integers and integer arrays are also included in the definition for maintaining an active list of objects, constants, formulas and variables. These arrays are used to correspond to the arguments defined in the Calling Sequence window, since the relationship between arguments and objects is not always one-to-one.

All input arrays and all output arguments are added to the DefineObjectFrame. One dimensional arrays are transposed to *VECTOR* objects. If the number of rows is defined as a "*" on the Calling Sequence window, then a menu is displayed for choosing either an input scalar integer argument, constant or formula for the

number of rows of the Vector. Otherwise, the selection for the number of rows is displayed as a label. Two dimensional arrays are transposed to *MATRIX* objects. The leading dimension selected on the Calling Sequence window is displayed as a label. If the number of columns is defined as a "*" on the Calling Sequence window, then a menu is displayed for choosing either an input scalar integer argument, constant or formula for the number of columns in the Matrix. Otherwise, the selection for the number of columns is displayed as a label. The number of rows of the Matrix is displayed as the selected leading dimension value, but menu options are also available for changing the selection. Only an input scalar integer argument, constant or formula can be chosen for the number of rows in the Matrix. An option to add input scalar integer arguments to the list of objects is offered through this class. The implementation of the DefineObjectFrame class is illustrated in Figure 5.6.

The method *buildList* builds a display of object data in terms of labels. *buildPanels* builds a display of object data for those objects that need dimension data defined. The input data required for the Problem class is stored in the *prepareProblem* method. The *getIndex* method returns an input or output object's index number. The *getArgIndex* method returns the index number of an argument. *getConstArg*, *getConstIndex*, *getFormArg*, *getFormIndex*, *getConstMnemonic*, *getFormMnemonic*, *getTotConsts*, *getTotForms*, *setNumConsts*, *setNumForms* are methods that manipulate constant and formula data.

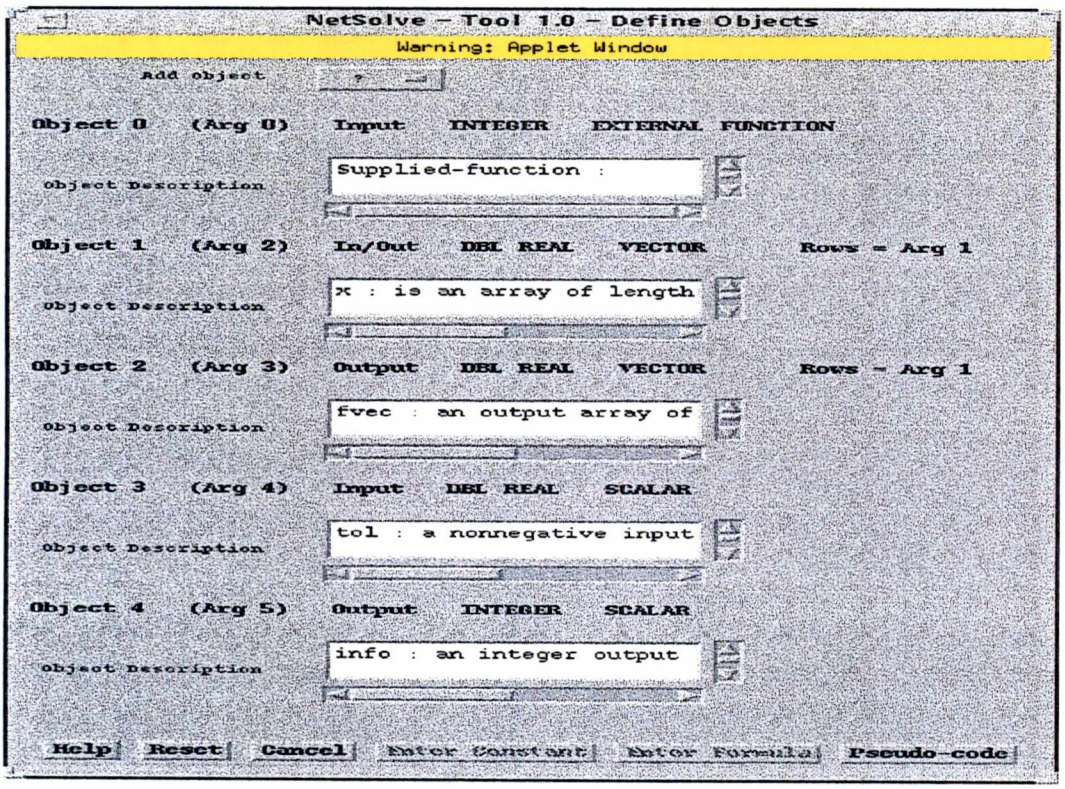

Figure 5.6: The Object Definition Screen Implemented in the DefineObjectFrame Class

5.2.9 GenerateCodeFrame

The GenerateCodeFrame class defines the frame for viewing and editing the pseudo-code generated by the interface for the given problem definition. The methods for generating the pseudo-code as well as the formal descriptive language source code are included in the class definition.

A text area is defined for the server administrator to view and edit, if necessary, the pseudo-code for the problem. The only check done on the pseudo-code is to verify that all the mnemonics referenced in the code are valid. The method *checkCodeStrs* checks the validity of the mnemonics.

The method *setDefault* generates and displays the pseudo-code for the input problem definition. The methods *GenCode*, *processedCode* and *writeFile* are responsible for creating the formal declarative language source file. Security restrictions enforced by Web browsers and Java applets required that a socket server be set up on the Web server so that the source code file could be temporarily written before being downloaded. The methods for reading and writing to the Web server socket are included in this class definition. The use of sockets for generating the source code file is explained in more detail in Section 8.2. The code generation screen, which is implemented in the GenerateCodeFrame class, is illustrated in Figure 5.7.

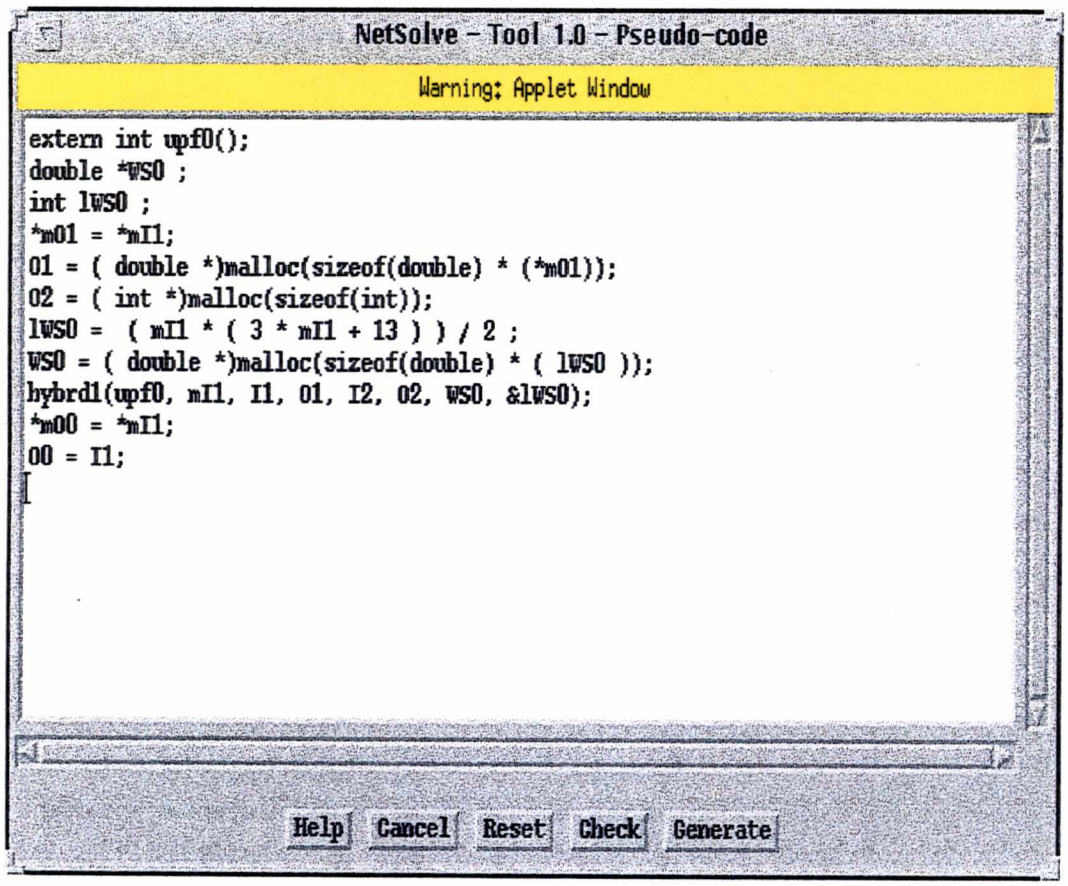

Figure 5.7: The Code Generation Screen Implemented in the GenerateCodeFrame Class

5.2.10 NetSolveWarning

The NetSolveWarning class defines the frame for displaying warning or error messages. The class definition contains a frame in which the error message is displayed and a button to verify that the error has been acknowledged by the NetSolve server administrator. Once acknowledged, the frame is dismissed and the server administrator is returned to the screen where the error occurred.

5.2.11 NetSolveHelp

The NetSolveHelp class defines the help windows used by the interface. The class definition contains a frame for displaying the help information and a button for closing the frame. The method *SetMessage* displays the help information for each frame in the interface.

Chapter 6

Using the Java Interface

The Java interface was designed to make adding scientific software packages to the NetSolve server an easy and straightforward process. The interface hides the complexity of the NetSolve system and the formalism of the NetSolve problem specification language from the server administrator. The only knowledge required by the server administrator is about the software packages being added to the system.

The interface automatically generates the formal language source code file for the software package entered by the server administrator. This is the source code that is to be translated into C code and then compiled by the NetSolve server. The interface does consistency error checking on the server administrator's input information and guarantees that the generated formal language source code is error-free. Once generated, the formal language file can be downloaded to any

NetSolve server for translation, compilation and use regardless of their hardware or software platforms.

6.1 Describing the Problem

The first screen of the Java interface allows the server administrator to describe the problem being added to the NetSolve server. Figure 6.1 shows the Problem Description screen of the interface for the example function, *dotproduct.*

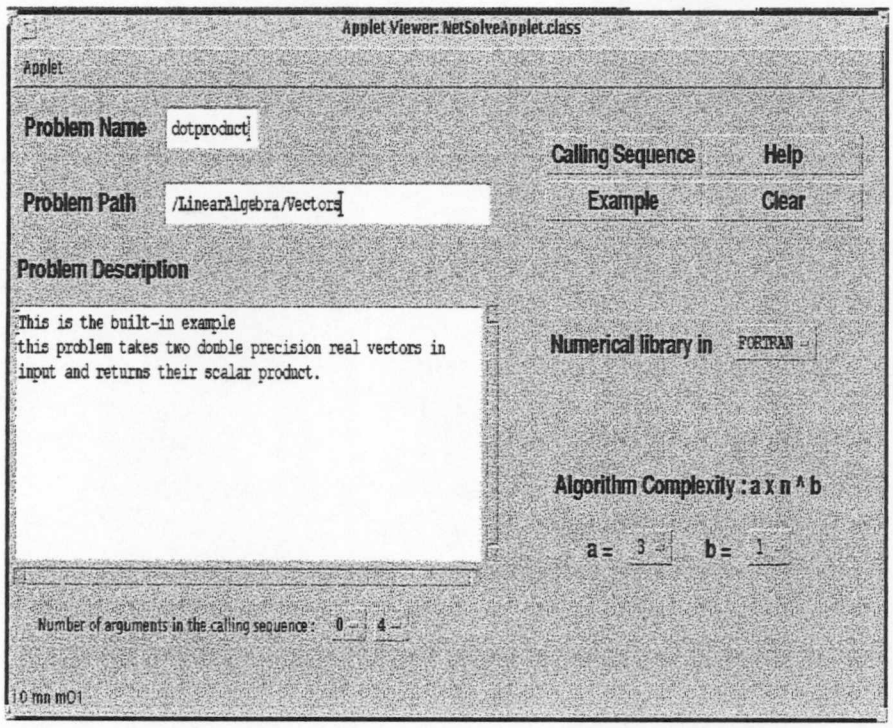

Figure 6.1: The Problem Description Screen

In order to specify a problem the server administrator is required to enter the problem name, path and description. The *problem name* is the same as the

name of the function being added. The *problem path* is a naming convention used by other NetSolve programs for displaying problems in alphabetical order. The *problem description* is used by the NetSolve client interface to help users of the system get a better understanding of the functionality of the problem.

The server administrator can specify the *numerical library* function calls as either Fortran or C library function calls. All function calls to the software packages made by NetSolve are in C, but the type of library routine determines how the data sent to these routines is stored and processed.

The *complexity of the algorithm* is also a required entry. This is used by the agent when computing the time required on a given machine to perform the computation of the problem. When dealing with a deterministic algorithm, such as a matrix multiply, this entry for complexity is sufficient. However, for algorithms whose complexity depends on the input data, such as an iterative algorithm, the server administrator must enter a "guess" for this field.

The *Number of arguments in the calling sequence* is entered through two choice menus located at the bottom of the screen. The menu on the left allows the server administrator to select the tens digit and the menu on the right is for selecting single digits. In this version of the interface, up to thirty arguments can be defined.

The buttons on the first screen allow the server administrator to display a *Help* file and to *Clear* or reset all the input areas. The *Example* button displays the *dotproduct* example function on the screen.

If the server administrator has input all the problem specification data represented on the screen properly, the *Calling Sequence* button can be used to go on to the next screen. The Calling Sequence screen is where the server administrator specifies the function parameters for the problem. The Calling Sequence screen is described below.

6.2 Defining the Calling Sequence

The second input screen allows the server administrator to define the C or Fortran calling sequence for the problem that the NetSolve system will need to use, enforce and describe to the client software. This screen is illustrated in Figure 6.2. The process of defining the calling sequence has been greatly facilitated by the Java interface. Instantaneous error checking is done to ensure that the server administrator cannot assign invalid types or values to the input or output objects. If errors exist, the server administrator cannot continue until the errors have been resolved.

6.2.1 Arguments

Arguments can be defined as either Input, Output, both Input and Output or a Workspace variable. The structure types available for input arguments include 1-D array, 2-D array, Scalar and External. The External type is used for defining an input argument as an external function. Some numerical packages, typically

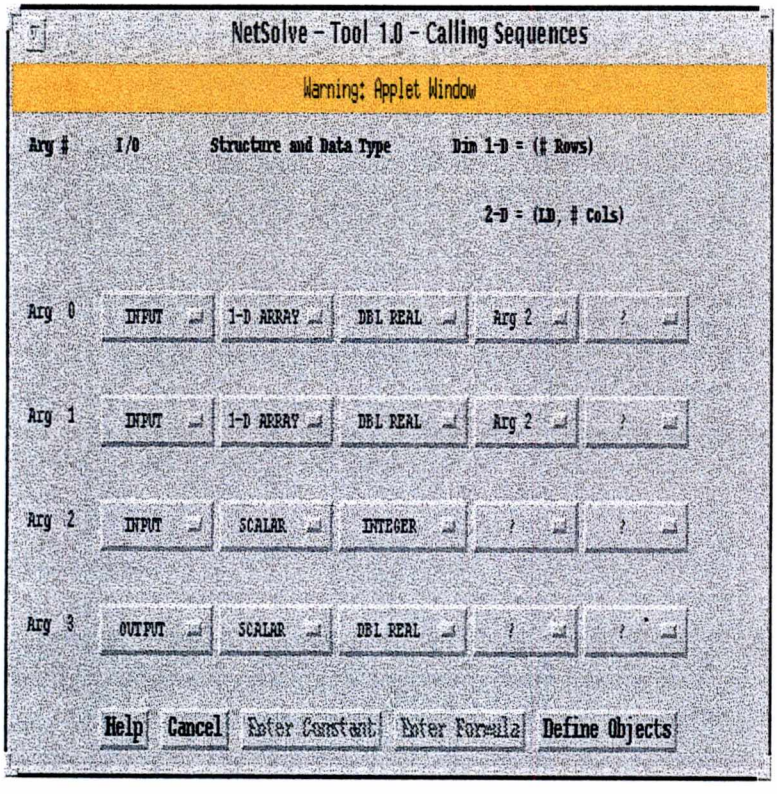

Figure 6.2: The Calling Sequence Screen

nonlinear software, require that the user provide a subroutine that computes the nonlinear function. The structure types available for output arguments include 1-D array, 2-D array and Scalar. The only structure type available for Workspace variables is a 1-D array. The data types of integer, real, double precision real, complex, double precision complex and character are available for all argument types. The data type for an external function defaults to integer, but this type is irrelevant to the NetSolve system. The structure and data type are required entries for all arguments.

If an argument is defined as a 1-D array, a choice menu is displayed for selecting the number of rows in the array. Selections for the number of rows include any input scalar integer argument, "*", Formula and Constant. If an argument is defined as a 2-D array, two choice menus are displayed. One for entering the leading dimension of the array and the other for entering the number of columns in the array. The selections for the leading dimension are any input scalar integer argument, Formula and Constant. The selections for the number of columns in the array are any input scalar integer argument, "*", Formula and Constant. Dimensional input is required for all arrays.

If a Constant is selected for a dimension the *Enter Constant* button is activated. If a Formula is selected for a dimension, the *Enter Formula* button is activated. For more information on entering Constants and Formulas see Sections 6.2.3 and 6.2.4 respectively.

The *Help* button displays help information on defining the calling sequence of the problem. The *Cancel* button closes the Calling Sequence window, zeros out the calling sequence data and activates the main Problem Description window. When the Calling Sequence is defined, the *Define Objects* button can be utilized to continue on to define the objects in the problem.

6.2.2 Variables

If an argument is defined as *Workspace* when the objects are to be defined, a screen is displayed so that the server administrator can define the size of the workspace as either a constant or a formula. All workspace dimensions must be assigned to a constant or formula value before proceeding to the Define Objects screen.

The dimension of the workspace was assigned to either an argument or a "*" on the Calling Sequence window. If an argument is assigned to the dimension of the workspace, this argument is used by the system to pass the constant or formula size of the workspace in as a parameter. If the "*" is selected, the size of the workspace must still be entered as either a constant or formula so that the workspace array can be allocated properly by the system. If the dimension is defined as a "*" on the Calling Sequence window, the size is not passed in as a function parameter. The system automatically generates a variable name for the workspace and its dimension variable. The screen for defining workspace is shown in Figure 6.3.

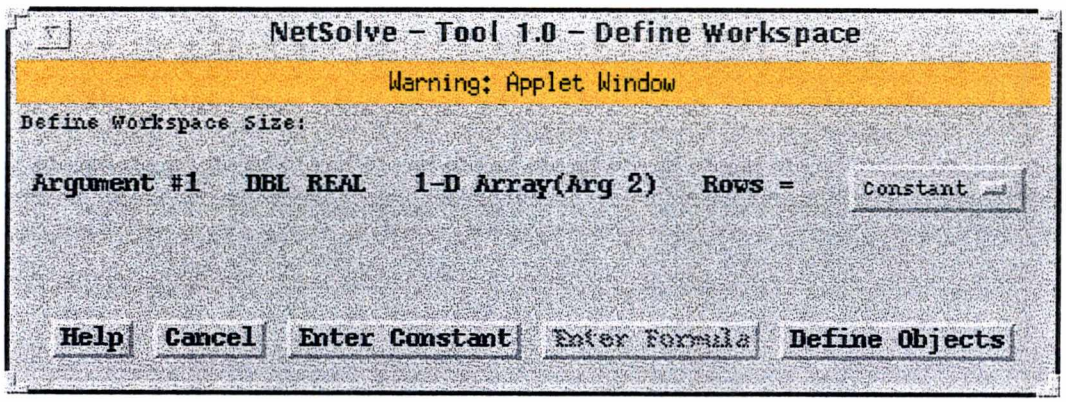

Figure 6.3: The Workspace or Variable Definition Screen

If a Constant is selected on the dimension menu, the *Enter Constant* button is activated. If a Formula is selected on the dimension menu, the *Enter Formula* button is activated. These buttons can be utilized to enter the constant and formula data, respectively.

The *Define Objects* button is used to store the workspace information and continue on to the Define Objects screen. Error checking is done to ensure that either a constant or a formula was defined for each workspace array. An error message is displayed in a dialog window if any errors are found and after acknowledging the error, the server administrator is returned to the Variable window. The *Cancel* button dismisses the screen and the workspace values are reset to their last stored values. Help can be obtained on entering workspace variables via the *Help* button.

6.2.3 Constants

Constants can be used to define the the number of rows in a 1-D array or Vector, and the leading dimension, the number of rows and the number of columns in a 2-D array or Matrix.

Constants can be entered from the Calling Sequence, the Define Objects and the Define Workspace Variable screens via the *Enter Constants* button. The screen for entering constant values for arguments or objects can be seen in Figure 6.4.

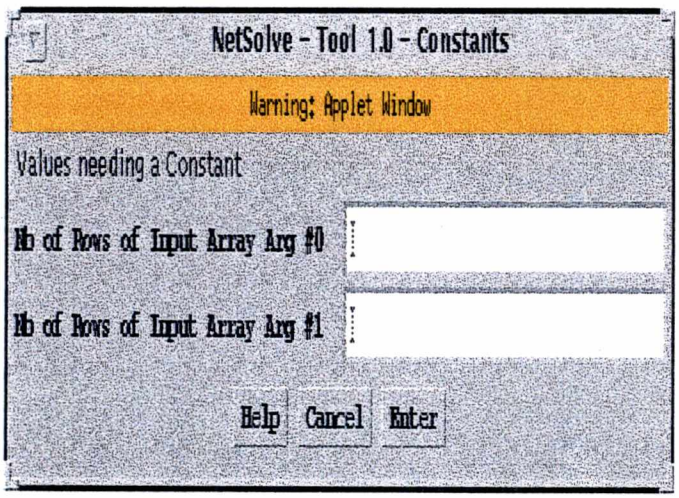

Figure 6.4: The Constant Entry Screen

All constants that have been designated for the problem, which are those that have been defined for arguments or objects, are displayed on one screen. The workspace constants are displayed on a separate Constants screen. A label is shown for each constant entry describing the argument, object or variable. The Constants window is updated as arguments, objects or variables are assigned to

or removed from the constants list.

Entries for constants are required to have integer values. If constant values have been assigned, they are displayed in the text area. The *Enter* button is used to store the constant values displayed. Error checking is done when the constant values are entered. If the entered values are not integers, an error message is displayed in a dialog window and after acknowledging the error the server administrator is returned to the Constants window. The *Cancel* button dismisses the window and the constant values are reset to the last stored set. Help can be obtained on entering constant values via the *Help* button.

6.2.4 Formulas

Formulas can be used to define the the number of rows in a 1-D array or Vector, and the leading dimension, the number of rows and the number of columns in a 2-D array or Matrix.

Formulas can be entered from the Calling Sequence, the Define Objects and the Define Workspace Variable screens via the *Enter Formulas* button. The screen for entering formula values for arguments or objects can be seen in Figure 6.5.

All formulas that have been designated for the problem, which are those that have been defined for arguments or objects, are displayed on one screen. The workspace formulas are displayed on a separate Formulas screen. A label is shown for each formula entry describing the argument, object or variable. The Formulas

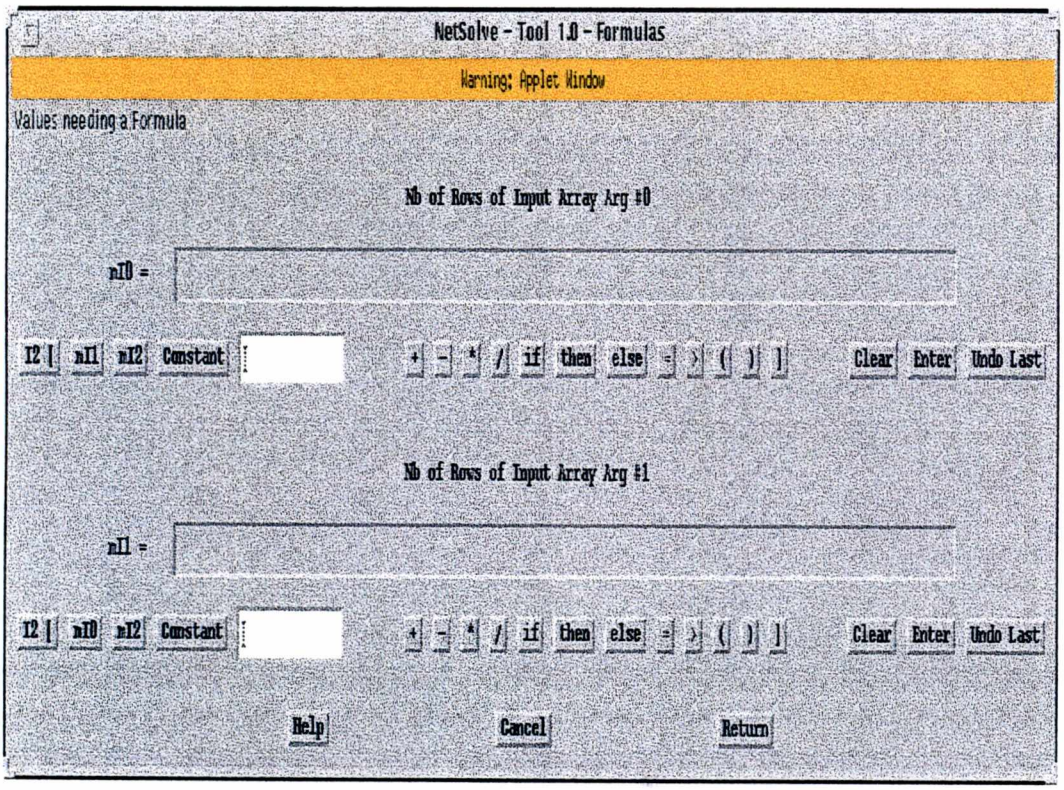

Figure 6.5: The Formula Entry Screen

window is updated as arguments, objects or variables are assigned to or removed from the formulas list.

Formulas must result in an integer value. Therefore variables used in the formula are required to have an integer type. All the available options for variables to be used in the formula are presented to the server administrator as buttons below each formula display area. Constants can also be used in the formula. The constant entry area for a formula is displayed below each formula display area. Constants are entered by typing an integer value in the constant text area just to the right of the *Constant* button and then by pressing the *Constant* button. When the constant value is entered, error checking is done to ensure that the entry is an integer. If the entered constant value is valid, then it is displayed in the formula display area. If an error occurred, an error dialog window is displayed to the server administrator explaining the error.

The arithmetic operations of addition (+), subtraction (-), division (/) and multiplication (*) can be used in the formula definition. The arithmetic operations are presented to the server administrator as buttons. Any mnemonic that has an integer type is displayed as an option for input into the formula in the form of a button. Left and right parentheses are also valid entries in the formula. A right brace is also available. This brace is used to close an array entry. For example, if *input #0* is an integer vector, then *I0[* would appear as an entry option or button to the server administrator. If the server administrator pressed this button, it

would appear in the formula display area. The server administrator would then be expected to enter an element location, either as a constant value, any mnemonic variable button displayed as an option, or an arithmetic expression. After entering the element location, the server administrator would close the vector element entry with a right brace.

An *if-then-else* clause is also available for entry into the formula. All three parts of the clause are required entry. The equal (=) and the greater than (>) signs are available for setting up the conditional *if* expression and are only active if an open *if* conditional currently exists in the formula definition. The *if-then-else* clause returns an integer value and therefore can also be used as part of an arithmetic expression.

The *Undo Last* button can be used to undo the last entry. The server administrator can continue to use the *Undo Last* button until the formula displayed is back to the needed form. The *Clear* button clears the formula display area it is associated with. The *Enter* button stores the formula that is associated with that formula display area. All formulas that have been entered or saved are displayed in a bold font. Error checking is done when the formula is entered to ensure that vector elements are defined properly, parentheses are balanced, there are no circularity problems and that the formula has a valid arithmetic format. Circularity errors occur when interdependencies exist between formula definitions, i.e. *formula#1* cannot be resolved until *formula#2* is solved and vice versa.

6.3 Defining Objects

The screen for defining objects is shown in Figure 6.6. By default objects are arguments defined on the Calling Sequence screen as either arrays or output arguments. However, the server administrator can add any argument to the object list by selecting the needed argument from the *Add Object* menu. The display is updated to reflect any additions to the Object list.

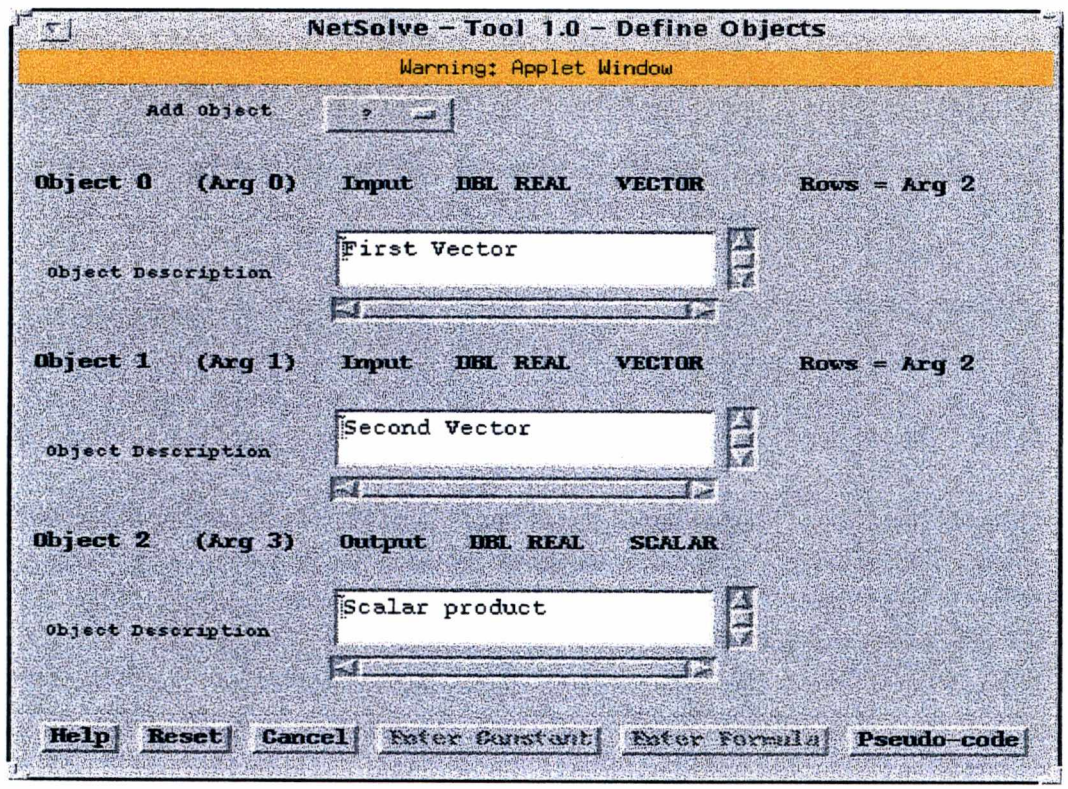

Figure 6.6: The Object Definition Screen

An object description is required entry for all objects displayed. These descriptions are made available to the users of the client interface to help them get

a better understanding of the problem.

If a Constant is selected for a dimension the *Enter Constant* button is activated. If a Formula is selected for a dimension, the *Enter Formula* button is activated. For more information on entering Constants and Formulas see Sections 6.2.3 and 6.2.4 respectively.

The *Help* button displays help information on defining the objects in the problem. The *Cancel* button closes the Define Objects window, zeros out the object data and activates the Calling Sequence window. The object display can be reset to the default set of objects defined by the calling sequence via the *Reset* button. When the objects are defined, the *Pseudo-code* button can be utilized to automatically generate the pseudo-code for the problem.

6.3.1 Vector Objects

One dimensional arrays are transposed into *Vector* objects. The Vector object contains the structure and data type of the array, the corresponding argument number and the number of rows defined for the array. If the number of rows of a 1-D array was defined as "*" on the Calling Sequence screen, a choice menu is displayed for selecting the number of rows in the Vector. Otherwise, a label is displayed showing the selection. Dimensional menu selections are any input scalar integer argument, constant or formula.

6.3.2 Matrix Objects

Two dimensional arrays are transposed into *Matrix* objects. The Matrix object contains the structure and data type of the array, the corresponding argument number and the leading dimension and number of columns defined for the array, plus the number of rows in the Matrix. If the number of columns of a 2-D array was defined as "*" on the Calling Sequence screen, a choice menu is displayed for selecting the number of columns in the Matrix. Otherwise, a label is displayed showing the selection. A choice menu is displayed for selecting the number of rows in the Matrix. The initial selection for the number of rows in the Matrix is set to the leading dimension selection, but can be changed by the server administrator. Dimensional menu selections are any input scalar integer argument, constant or formula.

6.4 Code Generation

All arguments, objects, variables, constants and formulas that are defined for the problem must be entered before the pseudo-code or formal language definition file can be generated. If the objects, arguments or variables have constants or formulas assigned to them, they cannot be blank. The Code Generation screen is shown in Figure 6.7. This is the screen that displays the pseudo-code generated by the Java interface. The server administrator has the option of editing this

code. The mnemonics assigned by the interface are used in the pseudo-code when referencing the input/output objects of the problem. The variable names for the workspace and their dimensions used in the pseudo-code are generated by the interface. External function names are also generated by the interface.

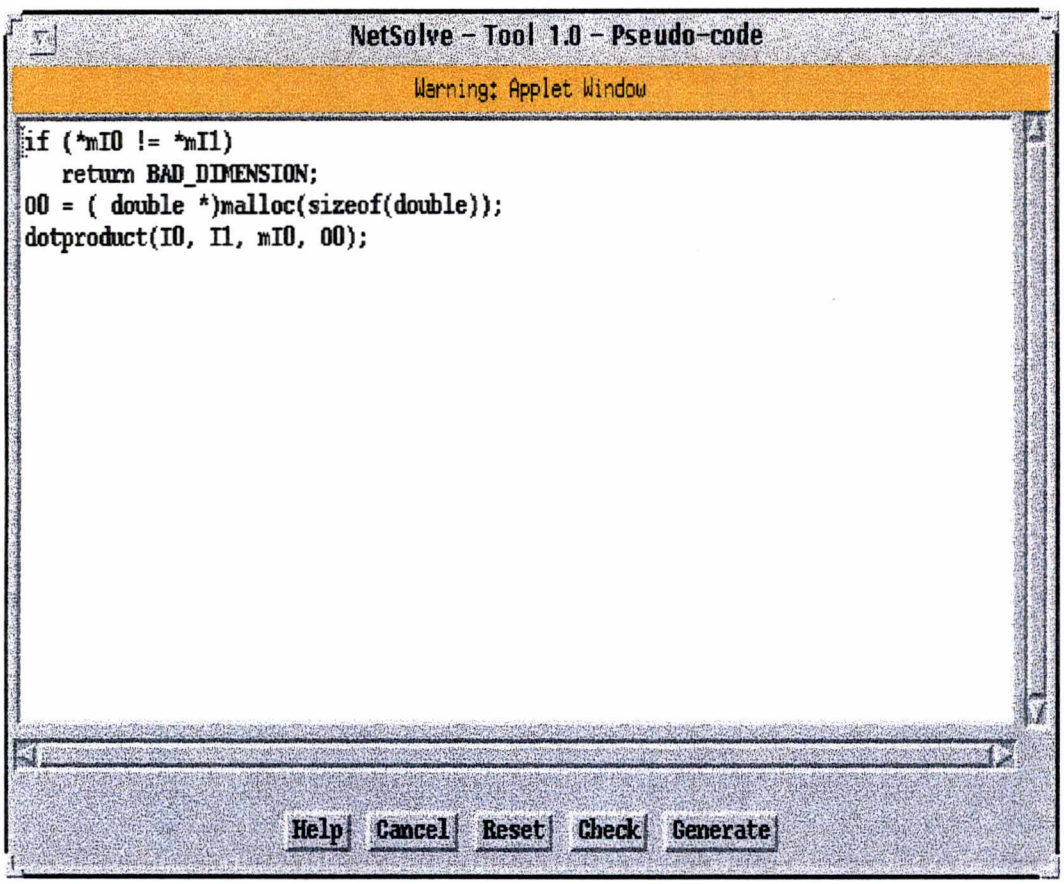

Figure 6.7: The Code Generation Screen

The server administrator can get help on code generation by utilizing the *Help* button. The *Reset* button resets the code entry area with the generated pseudo-code. The *Cancel* button dismisses the window, zeros out the object data and activates the Calling Sequence window.

When the server administrator is ready to store the pseudo-code and generate the formal language file for the compiler, the *Generate* button should be utilized. The Java interface uses all the input and output data information, the calling sequence data, the object definitions and the pseudo-code to generate the declarative formal language source code file. This is the file that will be translated into C code and compiled on the NetSolve computational server. This process enables the problem represented in the formal language source code file to be presented to the client and solved by the server. The generated formal language source code file is displayed to the server administrator as shown in Figure 6.8. To save the declarative formal language source code file, the server administrator can use the *Save As* option under the *File* menu of their Web browser.

6.5 Clarifying Data

Great effort was put into the design of the Java interface for the NetSolve server to make it as straight forward and user-friendly as possible. In addition to on-line help, extensive error checking and easy to follow screens, dynamic menu updates were added to the interface to aid the server administrator in setting up the NetSolve scientific computational server.

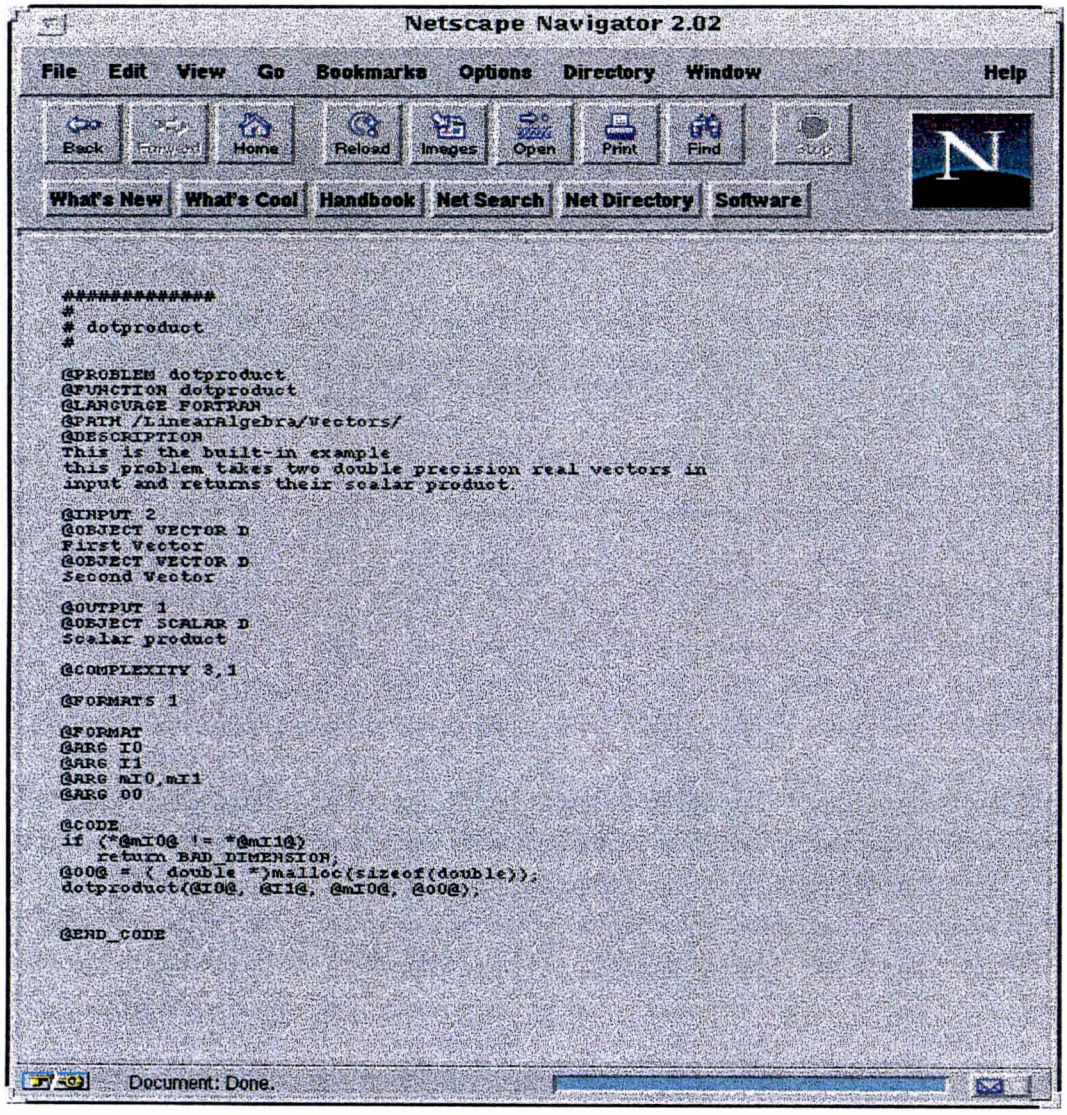

Figure 6.8: The Formal Language Source Code Display

6.5.1 Dynamic Menu Updates

In order to minimize errors, menu options are updated dynamically to ensure that only valid selections are presented to the server administrator. In addition, by clarifying menu selections, the server administrator will not be confused by invalid menu options. As options are selected on one choice menu, invalid items for that choice are taken off other menus. For example, the interface never has to be concerned with an output argument being assigned to an external function. Nor does it have to be concerned with the structure type of a workspace variable being anything other than a 1-D array. The dynamic menu updates guarantee that the server administrator will not choose invalid options, which eliminates the need for certain error checks.

6.5.2 On-line Help

On-line help options were added to the system to help answer questions server administrators may have while using the interface. Each screen in the interface is equipped with a *Help* button. The information available is an in-depth explanation of the data represented on the screens and how each screen works with the system.

6.5.3 Error Checking

Thorough error checking is performed on all input data before the server administrator is allowed to proceed to the next interface screen. Error checks are done

for mandatory data entry, type checking, assignment of all arguments and on the objects, variables, constants, formulas and pseudo-code in the problem definition. If any error is detected by the system, an error dialog box is displayed informing the server administrator of the problem. Once the server administrator acknowledges the error via the *OK* button, the system returns the server administrator to the screen where the error was found so that it can be resolved.

Chapter 7

The Compiler

The translator or compiler for the NetSolve server takes the formal language description file and produces C source code. The formal descriptive language file can be downloaded to any computational server in the NetSolve system, translated, compiled and added to the server's repository of numerical software. The formal language description file only needs to created once. Then the numerical functionality of the scientific software package represented in the formal language file can easily be duplicated on any NetSolve computational server.

7.1 Formal Declarative Language File

The formal declarative language file can generically describe any problem in an arbitrary scientific computational software package. The formal language source code files contain information about the problems in the computational scientific

library. This information is used by the NetSolve server to accurately calculate requests from the client. The formal language file is created by the Java interface and is guaranteed to be in the proper format so that the translator does not have be concerned with syntactic errors. This file contains information about input and output data requirements of the problem as well as the calling sequence restrictions of the problem.

Key words used by the translator are preceded by the @ symbol and are written in all upper case characters. The translator searches for key words and transforms the data following them into C source code that represents the problem. The compiled C code is used by the computational server to solve the problem requests of the NetSolve client. Mnemonics represented in the pseudo-code input by the server administrator are preceded and postfixed by the @ symbol.

The formal declarative language files produced by the Java interface represent the problems contained in a scientific computational library. The server administrator can either concatenate all the formal language files produced by the interface into one file or they can list all the files that are to be added to the server on the command line when they start the NetSolve compiler. The translator creates function calls for each problem represented in the formal declarative language file and stores them in one C source code file. The C source code file can be compiled and added to any NetSolve server's repository of numerical resources. An example of a NetSolve formal declarative language file can be found in Appendix A.

7.2 C Source Code File

The C source code file produced by the compiler contains one function call, which represents the scientific library's name. The parameters of the function call are the problem name, the size of the input and output data, followed by pointers to the input and output data. A block of code, which is the translated pseudo-code generated by the Java interface, is in the C source code file for each problem included in the formal language description file of the library. If the problem name passed in as a parameter matches one of the problems in the scientific library, the block of code representing that problem name is executed. An example of a C source code file generated by the NetSolve server compiler can be found in Appendix B.

The translation process sets up a routine based on the input problem name. Then it produces C code that allocates the input and output data variables based on the input sizes and the types defined for each through the formal language definition. The C source code contains the pseudo-code entered by the server administrator, but all the mnemonic representations are transposed into C syntax.

7.3 Implementation

The name of the formal language source code file containing the problem definition to be translated should be entered on the command line. The compiler is able

to take multiple formal language filenames and translate them into one C source code file. If at least one formal language source code filename is not given on the command line, the program displays an error to the server administrator and exits.

The translator then checks to make sure that the *NETSOLVE_SERVER_ROOT* environment variable is set. This environment variable tells the compiler where to store the C source code that is produced. If the variable is not set, the program displays a message to the server administrator and quits.

The translator opens the formal language file and is guaranteed that the keywords representing the problems in the scientific computational software package are in a certain sequential format. The translator goes into a *while* loop and prepares to translate each formal language file input on the command line. If one of the formal language files cannot be opened by the compiler, a message is printed for the server administrator and the compiler exits.

The input formal language source code file is parsed for keywords. All of the data read in from the formal language source file is stored in a data structure. After all the data has been read in, the data structure defining the problem is used to create the C source code file.

First, the problem and function names are read and stored. Then the language specification of the computational library is read and stored in the data structure. The input data definition follows in the formal language definition file and is

preceded by the output data definition. Each is read by the compiler and stored in the problem description data structure. Next, the calling sequence requirements for the problem are read in and stored appropriately. If any constants or formulas have been defined for the problem in the formal language specification file, they are then read in and stored. Finally, the pseudo-code is read in and processed.

The compiler takes the input data that is stored in the problem description structure and writes the necessary *malloc* calls to the C source code file. The compiler also transforms the mnemonics in the pseudo-code into the appropriate variable definitions, ensuring that the needed calling sequence is maintained and all the data is allocated properly based on the data types and sizes read in from the formal language source code file.

The compiler continues to translate formal language definition files until all those specified on the command line have been processed. Once the compiler has completed the translation process, a C source code file is stored on the NetSolve server that represents the scientific computational software package input by the server administrator.

Once the C source code file is produced that represents the scientific computational library of the NetSolve server, the server administrator needs to compile the C source code. The NetSolve system provides a *Makefile* that takes the C source code generated by the NetSolve compiler and actually builds the NetSolve computational server. The generated executable is called *server* and once com-

piled is ready to be started. When the *server* executable has been started on a machine, a NetSolve computational server has been created.

7.4 Tasks Taken Over by the Java Interface

When the Java interface was added to the NetSolve server system, the job of the compiler was greatly reduced. The compiler no longer has to be concerned with the sequence of the keywords in the formal language specification file. The Java interface guarantees that the keywords found in all the formal language specification files will always be in the same sequence.

In addition, the compiler no longer has to worry about errors in the formal language description of the problem. The Java interface ensures that all the formal descriptive language files that it produces will always have the correct formal language format. The need for formal language error checking was alleviated from the compiler when the Java interface was implemented in the NetSolve server system.

Why doesn't the Java interface eliminate the entire translation process? Why does the Java interface even need to create a formal declarative language file? Shouldn't the interface produce the C source code file in the first place? Granted, the interface could just as easily produce a C source code file as a formal declarative language file. But there is work being proposed that will use these formal

declarative files produced by the Java interface, not just for setting up NetSolve computational servers, but also for the development of numerical tools that will use these descriptive language files to provide users with uniform, dynamic access to repositories of computational software libraries [3].

Chapter 8

World Wide Web Access

In order to make the NetSolve Java interface for the creation and maintenance of a computational server available to anyone on the internet, it was made accessible through the World Wide Web (WWW). Incorporating the interface to the NetSolve server into a Web page presented several challenges when dealing with the security restrictions upheld through Web browsers and Java applets.

8.1 File Creation Problems

Before a NetSolve computational server can be implemented, a formal language source code file has to be created. This requires that a file be written to a disk somewhere. Because of the security restrictions enforced by Web browsers and Java applets, which prevent file creation on both Web client and server machines, the interface design had to incorporate a solution that would allow it to create and

write the formal language source code file somewhere. Since the Java interface has no control over the machine accessing the Web site, the Web server running the Java interface had to be able to temporarily write the formal language file locally before it could be downloaded to the Web client machines. Web browsers provide mechanisms for downloading already existing files. So once the formal language file is created, the server administrator can use the mechanisms inside their browser to actually download the file to their local disk.

8.2 Solution Implemented for File Creation Restrictions

Most Web browsers enable users to download already existing files. But with the implementation of the Java interface for NetSolve servers, a file had to be created first and then downloaded. Security restrictions implemented in Web browsers and enforced in Java applets prevent Web applications from easily creating files on a Web server or a Web client. The need for these restrictions is obvious, but created an extra obstacle in the interface implementation.

There are some options available for creating files inside Web applications. One is the use of the Common Gateway Interface (CGI) scripts [9] on the Web server and another is through TCP/IP communication. Java provides a *ServerSocket* class for TCP/IP socket implementation, so this option was taken for the first version of the interface. The implementation of a server socket will most likely be

changed or replaced later on in the development of the NetSolve server interface. The final solution and implementation will be one that is more easily incorporated into the NetSolve Web site. But for this version of the interface, a separate process needs to be run on the Web server that waits for requests from clients to write their formal language source code file. When a request is sent by the interface to the socket server, the formal language file is temporarily created on the Web server. When this file is either downloaded or discarded by the Java interface, it is removed from the Web server's disk.

Once a file is created on the Web server, it can easily be downloaded by using the options already available in the Web browser. Inside the NetSolve server interface, the formal language source code file is displayed to the server administrator inside a Web browser frame, as described in Section 6.4. If the server administrator chooses to download the file, they can do so by using the *Save As* option under the *File* pulldown menu. As soon as the formal language file is downloaded, the NetSolve server compiler can be used to translate it into C code and then this C code can be compiled into a computational resource for a NetSolve server.

8.3 Accessing the NetSolve Server Interface Through the WWW

The Java interface for adding scientific computational software packages to NetSolve servers will be accessible over the WWW through the NetSolve home page [2]. By making the interface available throughout the internet, the versatility of NetSolve is greatly enhanced. Because of the easy access and use of the NetSolve server interface, the varieties of computational scientific software packages that can be added to the NetSolve system increases dramatically.

By making the interface accessible through the WWW, only one version of the interface will have to be maintained. Server administrators will not have to worry about obtaining upgrades of future releases of the NetSolve server interface. They can be assured that they will always have access to the latest version of the NetSolve server interface. As a new version of the Java interface is developed and added to the NetSolve system Web site, it automatically becomes available to all NetSolve server administrators in the NetSolve system.

Chapter 9

Conclusion

The work presented in this paper focuses on the integration of computational scientific software packages into the NetSolve system. The purpose of the work described here is to ensure that any computational scientific software package can easily be added to the NetSolve system.

The NetSolve system needs to be versatile, in that it has to be able to integrate any arbitrary computational numerical software package, regardless of the structure of its calling sequence or format of its input and output data. It was unrealistic to assume that current users of any scientific software package would be willing to change the calling sequence or data layout of their existing code in order to use the functionality of the NetSolve system. Therefore, no restrictions could be placed on the numerical software that would be integrated into the NetSolve system. The Java interface provides a mechanism for presenting any scientific

software package being added to the NetSolve system in generic terms, so that the system can recognize and process any numerical computation.

9.1 Benefits of the Java Interface

The Java interface for the NetSolve computational server has had an impact on both the NetSolve system and the NetSolve server compiler. Through the use of the Java interface the NetSolve resource administrator can create and dynamically change a NetSolve computational server with ease and clarity. The server administrator no longer needs to know about the complexity of the underlying formal descriptive language used by the NetSolve system and the NetSolve compiler no longer has to be concerned with the format or the integrity of this file.

9.1.1 Impact on the NetSolve System

The process of adding and updating NetSolve computational resources or servers has been tremendously improved and facilitated by the implementation of a Java interface. The development of the Java interface, in conjunction with the use of the NetSolve formal descriptive language, ensures that any arbitrary scientific software package can be integrated into the NetSolve system.

The Java interface automatically makes the NetSolve system more versatile and diverse by simplifying the creation, maintenance, use of and access to the computational server. The Java interface transformed the creation and mainte-

nance of NetSolve computational resources into a straightforward, easy to follow process. By making the interface available throughout the internet on the WWW, the versatility of NetSolve is greatly enhanced. Because of the easy access and use of the NetSolve server interface, the diversity of computational scientific software packages that can be added to the NetSolve system increases dramatically. And finally, the implementation of the Java interface maintained the NetSolve objectives of versatility, graphical enhancement, ease of use and clarity.

9.1.2 Impact on the NetSolve Compiler

The design of the NetSolve server compiler that translates the formal descriptive language source code file into C code was able to be facilitated due to the implementation of the Java interface. The Java interface incorporates tasks previously done by the NetSolve compiler, thus allowing the compiler to become a more efficient and focused tool.

The Java interface ensures that the format of the formal language description file that it produces will be in a predetermined sequence. The interface also alleviates the compiler's job of in-depth error checking. The server compiler is assured that the formal language description represented in the source file provided by the Java interface has been defined properly and without error.

9.2 Future Work

There are a few areas in the Java interface design that could be improved. These improvements range from incorporating extra features in the interface design to overcoming problems caused by Java immaturities. Some brief suggestions about these enhancements are given below.

9.2.1 Pseudo-code Parsing

An area of the Java interface that could be enhanced is that of the interpretation of the pseudo-code. Ideally, the compiler should be assured that the pseudo-code edited by the server administrator is free of both compile and runtime errors.

The Java interface should parse the pseudo-code for errors. The free format text entry screen for the pseudo-code increases the possibility of errors. If any errors exist in the formal language description file, they would be in the presentation of the pseudo-code. The pseudo-code that the Java interface generates is error-free, however the server administrator is allowed to edit the code anyway they choose. Therefore, in order to make the Java interface as fool proof as possible, a better mechanism for checking the pseudo-code should be implemented.

9.2.2 Automation of the Translation Process

Another feature that could be added to the Java interface is the automation of the translation of the formal language file. The server administrator should have

the option of either creating and downloading a formal language file or a C source code file. While there are proposals for using the formal language file, the server administrator should have the ability to produce either type of file. Implementing this feature, however, would require changing parts of the NetSolve system that are outside the scope of this work.

9.2.3 Java Inefficiencies

Since Java is in the early stages of development, there are areas of weaknesses in the Java API library routines [6]. These weaknesses are especially noticeable to the server administrator when trying to enter data in a text area. Users of graphical tools that expect text entry to be a smooth and easy process are going to be disappointed with their use of Java text areas.

There are also problems with different versions of Java and Web browsers that cause the Java Lists in the interface to be displayed improperly. But as the Java libraries continue to improve and upgrade, so will the interface to the NetSolve computational servers.

Bibliography

Bibliography

[1] S. Browne, H. Casanova, and J. Dongarra. Providing access to high performance computing technologies. In J. Wasniewski, J. Dongarra, K. Madsen, and D. Olesen, editors, *Lecture Notes in Computer Science 1184*, pages 123–134, Berlin, 1996. Department of Computer Science, University of Tennessee, Knoxville, Springer-Verlag.

[2] H. Casanova and J. Dongarra. The NetSolve Project accessible from the NetSolve home page "http://www.cs.utk.edu/netsolve".

[3] H. Casanova and J. Dongarra. Providing Uniform Dynamic Access to Software Repositories.

[4] H Casanova and J. Dongarra. NetSolve: A network server for solving computational science problems. In *Proc. of Supercomputing'96, Pittsburgh.* Department of Computer Science, University of Tennessee, Knoxville, 1996.

[5] H. Casanova, J. Dongarra, and K. Seymour. Client user's guide to netsolve.

Technical Report CS-96-343, Department of Computer Science, University of Tennessee, 1996.

[6] C. Fraizer and J. Bond. *JAVA API Reference.* New Riders, 1996.

[7] The Math Works, Inc. *MATLAB Reference Guide.* 1992.

[8] P. Naughton. *The JAVA Handbook.* Osborne McGraw-Hill, 1996.

[9] Gaither M. Hassinger S. Tittel, E. and M. Erwin. *Foundations of World Wide Web Programming with HTML and CGI.* IDGE Books Worldwide, Inc., 1995.

Appendix

Appendix A

Formal Descriptive Language

Source Code

```
#
# iqsort
#
@PROBLEM iqsort
@FUNCTION iqsort
@LANGUAGE C
@PATH  /QuickSort/Integer/
@DESCRIPTION
Quicksort
@INPUT 1
@OBJECT VECTOR I
Vector of integers to Sort
```

```
@OUTPUT 1

@OBJECT VECTOR I

Sorted Vector

@COMPLEXITY 1,1


@FORMATS 1

@FORMAT

@ARG mI0

@ARG I0,O0


@CODE

extern intcompare();

qsort(@I0@,*@mI0@,sizeof(int),intcompare);

*@mO0@=*@mI0@;

@O0@=@I0@;


@END_CODE


#

# dqsort

#
```

```
@PROBLEM dqsort

@FUNCTION dqsort

@LANGUAGE C

@PATH  /QuickSort/DoublePrecision/

@DESCRIPTION

Quicksort

@INPUT 1

@OBJECT VECTOR D

Vector of integers to Sort

@OUTPUT 1

@OBJECT VECTOR D

Sorted Vector

@COMPLEXITY 1,1


@FORMATS 1

@FORMAT

@ARG mI0

@ARG I0,O0


@CODE

extern doublecompare();
```

```
qsort(@I0@,*@mI0@,sizeof(double),doublecompare);

*@mO0@=*@mI0@;

@O0@=@I0@;


@END_CODE
```

Appendix B

Compiler Generated C Source Code

```
/*****************************************************************/
/*        linear_algebra.c                                       */
/*        Version 1.0                  Henri Casanova            */
/*        Version 2.0                  Delphy Nypaver            */
/*   linear_algebra()                                            */
/*****************************************************************/

#include "server.h"
#include "fortran.h"

/*
 * This function is the gateway to the linear algebra
```

```
 * libraries. It is called within compute() and just
 * performes the computations. No NetSolve communication
 * is involved here.
 */

linear_algebra(pb_desc,sizes_input,sizes_output,data_input,data_output)
problem_desc *pb_desc;
input_sizes *sizes_input;
input_sizes *sizes_output;
void **data_input;
void **data_output;
{
char *buf1;

buf1 = (char *)strdup(pb_desc->code);
uppercase(buf1);
#if (defined(flag_iqsort)||(!defined(UPF)))
/***************************/
/* Problem : iqsort */
/***************************/
```

```
if (strcmp(buf1,"IQSORT") == 0)
{

extern intcompare();
qsort(((int *)(data_input[0])),*((int *)(&(sizes_input[0].m))),
sizeof(int),intcompare);
*((int *)(&(sizes_output[0].m)))=*((int *)(&(sizes_input[0].m)));
(data_output[0])=((int *)(data_input[0]));

}

#endif
#if (defined(flag_dqsort)||(!defined(UPF)))
/**************************/
/* Problem : dqsort */
/**************************/
if (strcmp(buf1,"DQSORT") == 0)
{

extern doublecompare();
```

```
qsort(((double *)(data_input[0])),*((int *)(&(sizes_input[0].m))),
sizeof(double),doublecompare);
*((int *)(&(sizes_output[0].m)))=*((int *)(&(sizes_input[0].m)));
(data_output[0])=((double *)(data_input[0]));


}


#endif




return 1;
}
```

Vita

Delphy Nypaver graduated from the University of Tennessee in June of 1985 with a Bachelor's Degree in Computer Science and a minor in Mathematics. She expects to complete her Master of Science degree in Computer Science in May of 1997.

She began working at the Oak Ridge National Laboratory in August 1985 in the Instrumentation and Controls Division (I&C), where she continues to work today. Currently she is working in the Energy Technologies Group of the Controls and Systems Integration Section of the I&C Division. Her current projects include the design and implementation of interfaces for generic reactor simulation codes and neutronics codes that compute the isotopics of various reactor types.

Delphy enjoys water sports, cycling, weight training and playing her trumpet in her spare time. She is a member of the Association of Computing Machinery and is involved in the Ladies of Charity organization. Delphy presently resides in Knoxville, Tennessee.