Bullfrog: An Extensible, Modular Toolkit for the Construction of NuMesh Applications

by

Michael W. Connell

Submitted to the Department of Electrical Engineering and Computer Science on January 19, 1996 in partial fulfillment of the requirements for the Degree of Master of Science in Electrical Engineering and Computer Science.

Abstract

The current generation of NuMesh, a communication substrate for scaleable parallel processing systems composed of arbitrary processing elements, requires an integrated development environment to enable NuMesh programmers to construct applications rapidly, flexibly, robustly, and conveniently. A design and implementation of the Bullfrog Network Description Language (Bullfrog NDL)—a very-high-level language for writing NuMesh network applications—and a Bullfrog NDL compiler is presented, and a framework based on this design and intended as the core component of an integrated development environment is implemented and documented. The Bullfrog compiler reads functional network descriptions written in the Bullfrog NDL and outputs communication information in the form of NuMesh virtual stream specifications, along with compiler directives (including compile-time constants calculated from the network description) for compiling the source code to be run on the network hardware. The software written for this project supports an abstraction layer that was previously nonexistent between the physical NuMesh network topology and the NuMesh programmer, allowing for the construction of much more complex NuMesh applications than was previously practical.

Thesis Supervisor: Professor Stephen A. Ward

Title: Professor of Electrical Engineering and Computer Science
Acknowledgments

I'd like to take this opportunity to thank some of the people who have given help in some form or another during the long and sometimes torturous journey that has finally landed me here.

Thanks to Charlie for teaching me to drink bitter tears and laugh at the same time.

Thanks to Kimball for helping me build a robotic chicken leg late one night when he had nothing to gain by it.

Thanks, Gill Pratt, for hanging in there even when I bolted for New York.

I owe Shoe gratitude on far too many counts to go into it here. Let me just say thanks for conspiring to get me back to Boston and for helping keep the gossamer thread of sanity from snapping once I arrived.

Thank you, Steve Ward and Alan Turing, for making Computer Science a respectable field. Oh, and thanks, Steve, for the Camp Minnetonka trip and the Tomb of Horrors and all. And the thesis advisement.

Thanks to Nietzsche, Bertrand Russell, The Tick, Kurt Vonnegut, Albert Einstein, and Elvis for various and sundry kindnesses over the years.

Thanks to my family for being the best.

Thanks to Shoe's parents for having a daughter.

Thanks to Shoe's parents' daughter for agreeing to marry me and for being tremendously supportive and generally delightful. Smart, too.

Finally, a great big thank you to all of the women at the graduate office who have been so helpful and pleasant through all of this. And a special thank you to Marilyn Pierce, who has pulled me out of more than one metaphorical snake pit.
For My Mother,  
Darlene  

Who Has Taught Me  
So Many Important Things  
by Example  

With All My Love  

February 1, 1996
# INTRODUCTION

- BACKGROUND AND MOTIVATION .......................................................... 6
- PROJECT OVERVIEW .............................................................................. 8

# THE NUMESH PROGRAMMING MODEL

- NUMESH CONCEPTS AND DEFINITIONS .................................................. 13
- NUMESH PROGRAMMING IN THE DARK AGES (BEFORE BULLFROG) .......... 17
- NUMESH PROGRAMMING IN THE AGE OF BULLFROG .............................. 21

# BULLFROG NETWORK DESCRIPTION LANGUAGE

- OVERVIEW ............................................................................................... 24
- LANGUAGE SPECIFICATION ..................................................................... 28
  - Comments .............................................................................................. 28
  - Keywords ............................................................................................... 28
  - define-function ...................................................................................... 29
  - define-network ...................................................................................... 32
  - Application Entry Point ......................................................................... 34
  - Language restrictions ........................................................................... 34
  - Example ................................................................................................. 35
- PARSER ..................................................................................................... 36
  - PARSER COMPONENTS ......................................................................... 37
    - Tokens ................................................................................................. 37
    - Atomic Objects .................................................................................... 39
    - Composite Objects .............................................................................. 43
  - LISP READER (LEXICAL ANALYZER) ...................................................... 45
    - Example ............................................................................................... 46
  - GARBAGE COLLECTION ........................................................................ 48
- INTERNAL REPRESENTATION ................................................................. 50
  - OVERVIEW ............................................................................................. 50
  - NAMING CONVENTIONS ....................................................................... 54
  - OBJECT DESCRIPTIONS ....................................................................... 55
- BULLFROG OUTPUT GENERATION - (EXPANDER & EXTRACTOR) ............. 64
  - NETWORK EXPANSION ....................................................................... 65
  - DATA EXTRACTION .............................................................................. 66
  - PARAMETER EVALUATION ................................................................... 68
  - BULLFROG OUTPUT FILES ................................................................... 72
    - Nodes.bfg ........................................................................................... 72
    - Streams.bfg ....................................................................................... 73
    - Compiler.bfg ...................................................................................... 75
- GRAPHICAL USER INTERFACE ............................................................... 77
- CONCLUSION .......................................................................................... 79
  - OTHER TOOLS ..................................................................................... 79
  - FUTURE BULLFROG ENHANCEMENTS ............................................... 80
  - WHAT'S NEXT ....................................................................................... 80
- BIBLIOGRAPHY ....................................................................................... 82
Introduction

Background and Motivation

NuMesh is a communication substrate for high-bandwidth communications among arbitrary processing elements arranged in user configurable lego-like lattices in three-space. NuMesh includes packaging and interconnect specifications, a hardware and software specification of a communication finite state machine (CFSM) to be included on each node, and specifications for a set of internode and intranode communication protocols. The NuMesh technology is expected to yield the largest performance gains when executing programs for which most or all of the communication patterns can be computed statically (at compile time), although more general dynamic communication patterns can be accommodated, and the issues involved in the latter are currently being studied [12].

The philosophy driving the NuMesh project is underpinned by the belief that there are significant advantages to be realized by decoupling the communication interface of a processing system from a specific architecture and from the processing elements themselves. Several advantages can be secured in this way over other designs, including modularity, robustness, cost-effectiveness, opportunity for optimization in accordance with the characteristics of a specific problem, and wide applicability. This design is in contrast to the typical backplane bus design found in most single-processor computers today, as well as the various bus designs implemented for specific (mostly multi-processor) projects. The NuMesh technology is intended as a general-purpose communication substrate specification which can be used to construct a processing network out of whatever processing or peripheral elements are most suitable to a particular task, and is not tied to any specific architecture.
Previous generations of the hardware (and simulations of the hardware) required the NuMesh programmer to use a set of ad hoc, low-level, and disparate tools, utilities and procedures to write, compile, link, and load the program onto the NuMesh network, and node intercommunications had to be specified manually by the programmer. This worked fine in the early stages of development, where the main focus was specification of the NuMesh hardware and protocols, when the requirements of a development environment were not yet known and the sizes of the networks being dealt with were limited to a few nodes. As the hardware specifications become more stable, however, and the focus shifts to the problem of programming a sizable NuMesh network to do useful work, the complexity of the programming task is quickly becoming unmanageable with the current tools. Some of the problems facing the NuMesh programmer at this stage are described in the following paragraphs.

The lack of an integrated development environment leads to problems of compatibility and consistency among the different tools used in the process of developing a program to be run on a network. There is no common interface and no common set of documents explaining to a newcomer how to use the system. Also, as tools change, there is no good way of notifying everyone affected by the changes, nor is there a natural way of making sure the tools still work together since there is no specification describing interfaces or functionality. Furthermore, there is no clear division of responsibility among the different tools, and error checking for some stages is currently non-existent. Finally, much of the work which could be automated is currently being done by hand; for instance, virtual stream specifications can be automatically calculated from a programmatically simple high-level network specification, but are currently being produced manually by the programmer—a tedious and error-prone process.
The lack of a mechanism for automatic specification of communication streams means, for one thing, that functional units involving more than one processor node cannot be abstracted as a single functional element and cannot, therefore, be reused—which means there is no support for controlling complexity; i.e., the programmer has had to specify every communication stream in the network, even if a previously specified subnetwork is being used repeatedly within a larger network. As a result of this lack of an abstraction mechanism, the task of programming, debugging, and maintaining a program is prohibitively—and unnecessarily—complex.

Previous attempts at creating system software or software specifications ([4], [5], [8], [11]) have been hampered in part by the inevitably significant differences among early hardware revisions and the fact that the specifications and requirements of such software were still under investigation by members of the NuMesh research group. The current generation NuMesh prototype represents a significant step in the evolution of the NuMesh system, especially with regard to performance and hardware support for the specified communication protocols. As the hardware specifications stabilize, sophisticated system software is becoming increasingly well-defined and the necessity of such software is growing.

**Project Overview**

The work embodied in this thesis represents an attempt to lay the foundation for an integrated development environment for the NuMesh system that will eliminate the problems described above, enormously simplify the programming process, and provide a framework upon which additional tools and utilities can be built in such a way as to leverage the features of the existing components, thereby reducing redundant work and increasing the capabilities of each new component. A design is presented
for the front end of a NuMesh system compiler, and the current implementation of the core functionality described in that design is discussed. The implementation is intended to be an extensible, portable framework upon which the rest of the components described in the design (and, undoubtedly, others which have yet to be specified) can be built over time. At the present time, work is being completed on the virtual stream compiler by another member of the NuMesh research group [7], and work has been initiated on a module for the automatic calculation of an efficient mapping of high-level code modules to hardware processors based on the characteristics of each individual network application developed using the Bullfrog software. These and other modules will eventually be integrated seamlessly with the Bullfrog system to realize the goal of an integrated NuMesh development environment.

This initial design describes four interdependent but clearly definable modules that make up the core of the NuMesh application development environment. The ultimate goal is to have one integrated package that allows the NuMesh programmer to rapidly and flexibly construct network applications using a "very high level" graphical language (see [5] and [11]) similar to those being developed for other research projects ([6], [10], [13]), and which generates, links, and loads the executable and communication code onto the processor nodes in the network after calculating an efficient network configuration and code layout, so the programmer does not have to deal with intermediate files or perform steps manually which could be handled automatically.

The four modules analyzed and implemented in this work are the following:

- A specification of the Bullfrog Network Description Language
- A parser for tokenizing Network Description Language files
- A flexible, portable, extensible, modular, and efficient internal representation of the network with hooks to support future functionality
- An output module that generates compiler directives and communication information from the internal representation that can then be processed further by other NuMesh tools

In addition, characteristics of a graphical user interface will be discussed throughout this work where appropriate, and several of the references in the bibliography point to other works that discuss this component in much greater detail [5] [11].

Figure 1.1 (below) depicts the system composed of these four elements and illustrates how they interconnect. Two modules are included in the diagram that have not yet been implemented: a graphical user interface and a module that saves an internal representation to a permanent file using Bullfrog NDL syntax. These modules are connected to the existing system components using dotted lines. The file-saving module also has an arrow feeding back into the parser, indicating that the files created by it can then be read by the parser the same way a file written by a human programmer can.
Figure 1.1 Block Diagram of the Bullfrog system.

Dotted lines indicate components that do not yet exist but that were anticipated in the design and are expected to be added soon.
The internal representation has been laid out in such a way as to support, in what should be a straightforward manner, the addition of a graphical user interface for manipulating the network once it has been parsed, for entering networks into a library as reusable modules, and for outputting a compact text representation of a network constructed or modified using the graphical user interface. Furthermore, related projects currently being implemented should be straightforward to integrate with this work; specifically, the virtual stream compiler currently being completed by Pat LoPresti of the NuMesh group [7], along with the normal language compilers (for C, C++, LISP, etc.) will comprise the back end of the compiler system and should be straightforward to integrate with this package, and a module for generating a reasonable mapping of executable code to processor nodes (work currently being initiated by Greg Spurrier of the NuMesh group) can operate upon either the internal representation directly, or upon the output of the extraction module included in this work to automate another of the difficult tasks currently faced by the NuMesh programmer.

A single example, a NuMesh application that calculates the sixteenth root of its input, will be traced throughout this work to illustrate how each module in the Bullfrog system transforms its input into its output.

NOTE: In the following pages, the term "NuMesh programmer" is used to indicate a person using the Bullfrog system to construct applications for a NuMesh network. The term "Bullfrog programmer" will be used to refer to a person extending or maintaining the Bullfrog software itself—that is, the software that is implemented as part of this thesis and that serves as a development system used in the construction of NuMesh applications.
The NuMesh Programming Model

In an effort to make explicit the problems being addressed by the work embodied in this thesis, an overview of the NuMesh programming model before the advent of the Bullfrog system will now be presented, and some concepts used throughout the thesis will be defined and/or explained. Following that, the programming model supported by the Bullfrog system will be presented and contrasted to the previous model.

NuMesh Concepts and Definitions

A NuMesh network is a (possibly heterogeneous) set of processing elements mounted on boards designed according to NuMesh specifications and connected to one another through NuMesh-compliant connectors to form a hardware network in three-space. The network may be configured in any three dimensional topology desired; ideally, a topology will be selected which is most suitable for maximizing the important performance parameters of a particular application running on a particular network. The topology and processing element characteristics are left unspecified by the NuMesh technology. What the NuMesh specification does define are the internode and intranode communication protocols, the characteristics of the boards on which the processing elements are mounted, and the node interconnection characteristics. In addition, a communication finite state machine (CFSM) is specified which is included on each node to process node communications [12].

The NuMesh execution model is different from the model associated with the majority of other computers. Conventional single and multiprocessor computer systems generally run a single application at a time. Even when multiple applications are running, each application usually runs as if it
were the only one; i.e., they have little or no intentional interaction with one another. In the case of multiprocessor computers, a single application is often replicated across all processors and the data is decomposed into modules to be processed on different nodes, and when each node's processing is complete, the results are gathered to synthesize the final output [3]. In contrast, a NuMesh network typically consists of a variety of different programs running on separate nodes simultaneously and passing data between them via virtual communication streams. For example, a single NuMesh network might be composed of hundreds or thousands of nodes, some containing digital signal processing elements, some containing RISC processors, some CISC processors, and each one of them performing a distinct operation on its data.

Simply put, a virtual stream (in the NuMesh world) is a stream of data which flows from the output of one processor to the input of another, possibly passing through any number of nodes in between its source and destination. The intermediate nodes do not operate upon the data, they merely pass it along. Ideally, the programmer specifies the endpoints of such a stream and the software will automatically calculate a reasonable path for the stream to take. Additionally, the network should be able to operate in the presence of node failures, unbalanced communication loads, and other singular conditions by re-routing the virtual streams dynamically. This highlights an important and unusual feature of the NuMesh technology—its communication paths are not explicitly defined by the architecture [12]. Although one topology might be more efficient than another for a specific application because it allows critical communications to happen more quickly, a particular topology is not, in general, required for a particular application to run.
Low-level code is conventional assembly or machine language code that runs on a physical processor. It can be binary or mnemonic, but there is a one-to-one mapping between software instructions and the instructions actually executed on the hardware. This type of code is processor specific.

High-level code is code written in a conventional high-level programming language, such as C++, Java, LISP, or any one of the many languages in existence. One high-level primitive instruction generally compiles into several low-level instructions. This level of code supports a programming model that is easier for a human programmer to work with than is low-level code, as it helps control software complexity through support for data, function, and iteration abstraction. The code that runs on the individual NuMesh nodes is written in a high-level language. This level of code is, in general, portable across platforms in the same architecture category, but specific to the topology or category of hardware on which it is being executed; i.e., high-level code for a parallel processing architecture will not run on a single-processor architecture, and vice-versa (it won’t take advantage of the parallel architecture in the latter case, at any rate).

A very-high-level language (in the case of the NuMesh) is a language using entire high-level modules as its primitives, and the Bullfrog Network Description Language is such a language designed specifically for writing NuMesh applications. This language supports an additional type of abstraction—topological abstraction—that provides analogous benefits to the NuMesh programmer that a programmer of a conventional computer architecture enjoys by utilizing a high-level language, as explained in the following sections.

Figure 1.2 illustrates the relationships between low, high, and very-high level languages.
Very-high-level language (Bullfrog NDL)

Primitives: Functions or modules defined in high-level languages, wrapped in Bullfrog interfaces (via `define-function`).
Support for abstraction of user-defined composite functional blocks, which are then viewed by the programmer the same way a primitive function is (via `define-network`).
Mechanisms for connecting Bullfrog functional blocks together.

Example:
```
(define-function (sqrt
    in = (input real)
    out = (output real)
    source-code = "sqrt.cc")

(define-network (fourth-root
    in = (input real)
    out = (output real)
    (streams (baz real))
    (sqrt in = in out = baz)
    (sqrt in = baz out = out)))
```

There is a one-to-many mapping between Bullfrog functional blocks and high-level code modules. When Bullfrog NDL source is compiled, the result is a set of code modules compiled from high-level source files with specific compile-time parameters and information specifying module relationships and communications.

High-level language (C, LISP, etc.)

Primitives: Fundamental data types (float, int, char, etc.)
High-level constructs representing computational concepts such as abstraction, flow-control, and support for user-defined aggregate data types

Example:
```
for (i = 0; i < 10; i++) ...
```

There is a one-to-many mapping between high-level statements and low-level instructions. Set of available operations is generally specified by some standard and (ideally) portable across platforms. When source is compiled, the result is hardware-specific low-level machine code.

Low-level language (Assembly language)

Primitives: Machine opcodes supported by the hardware and associated binary arguments

Example:
```
LDAB 0xFFFF ➔ 01010100011001111111111111111111
```

There is a one-to-one mapping between language statements and machine-executable instructions, and set of operations is hardware specific and not portable.

Figure 1.2
Relationship between low-level, high-level, and very-high-level programming languages.
NuMesh Programming In the Dark Ages (Before Bullfrog)

Until now, the NuMesh programmer has had to work at a level comparable to traditional assembly language programming, but without the support for subroutines generally offered at that level. Before Bullfrog, the NuMesh programmer had to go through something resembling the following process in order to get an application up and running on a NuMesh network.

- Write, debug, and compile the set of source files containing code to be run on the network nodes (processors). These might be written in C, LISP, Assembly, or any other programming language or combination of languages.

- Specify a topology for the hardware. This had to be done before the application was actually written, because the program was dependent on it.

- Write the NuMesh application (which probably entailed drawing a block diagram on paper), using the functions created in the first step as primitives.

- Assign functional blocks to processor nodes in the target network, and obtain or write the CFSM code to handle communications.

- Load the code onto the appropriate processors (as determined in the previous step) and load the CFSM code into CFSM memory.

- Specify each communication stream connecting nodes in the target network, and compile the communication stream information using the NuMesh stream compiler.
• Debug the target application, including high-level code source files, if necessary.

• Iterate.

Note that the programming model has no mechanisms for supporting abstraction at the level of a NuMesh application. The only functions that can be used in such an application are those defined in the high-level source files—no composition is possible, and no reuse of a particular network structure is possible. Furthermore, notice that the NuMesh programmer had to locate, configure, and figure out how to use several scattered tools and technologies (language compilers, NuMesh stream compiler, CFSM code, etc.), and he must remember to (and how to) perform each of the many steps involved. The programmer spent the bulk of his time attending to network-dependent details, such as keeping track of and specifying each stream's sources, destinations, and parameters, and then verifying their correctness, or assigning functions to nodes, rather than focusing on constructing a functional network with the appropriate behavior. Finally, note that a NuMesh application written in this way is not portable across either networks containing the same processing elements but having non-isomorphic topologies, or across topologically isomorphic networks that contain different processing elements. To have the same application (viewed from a functional standpoint) run on two networks differing in any way required some amount of additional work on the part of the NuMesh programmer to port it.

Here is a simple example of a NuMesh application that calculates the sixteenth root of a stream of numbers, using the square root function as the only NuMesh primitive.
Source file for code to run on processor nodes:

```c
#include <math.h>
void NMSqrt()
{
    long int x = ReadNuMesh();
    WriteNuMesh( sqrt( x ) );
}
```

(compile this with appropriate switches)

NuMesh application block diagram (generated by hand by NuMesh programmer):

![NuMesh application block diagram](image)

Communication stream specification (generated by hand by NuMesh programmer):

```plaintext
( stream1 ( src host )
    ( dest sqrt0 )
    ( type double )
    ( bandwidth 0.01 )
    ... )

( stream2 ( src sqrt0 )
    ( dest sqrt1 )
    ( type double )
    ... )

( stream3 ( src sqrt1 )
    ( dest sqrt2 )
    ... )

( stream4 ( src sqrt2 )
    ( dest sqrt3 )
    ... )

(stream5 ( src sqrt3 )
    ( dest host )
    ... )
```
As the example shows, the programmer would have had to specify five streams by hand, enforcing type consistency and program validity without the aid of any tools to reduce complexity. Even this trivial example presents many opportunities for introducing bugs. In a real-world application, there might be hundreds or thousands of primitives and/or nodes to be managed. Furthermore, in this scheme there is no way to have the computer calculate a reasonable topology or an efficient distribution of code to processor nodes, since there is no intermediate representation of the NuMesh program in this programming model from which to extract this information. Whereas an assembly language programmer of a conventional computer system is tied to a particular processor, the NuMesh programmer has had an analogous problem in being tied to a particular network topology. Similarly, whereas an assembly programmer on a single processor architecture has to spend time on mechanistic details of how a program gets things done rather than being free to work with more simple and abstract computational elements to solve a problem (e.g., having to have an addition loop rather than being able to execute a simple multiplication operation), the NuMesh programmer has, until now, had to worry about how physical processing nodes are interconnected rather than on how to build on a set of functions to do a particular job. Bullfrog provides the means to make this transition from what amounts to NuMesh assembly programming to a simpler, more conceptual, more flexible, and more powerful level of programming abstraction.
NuMesh Programming in the Age of Bullfrog

Using the Bullfrog language and compiler, the same program for calculating sixteenth roots would look like this:

Source file for code to run on processor nodes (essentially unchanged from previous example):

```c
#include <math.h>
void NMSqrt()
{
  long int x = ReadNuMesh( 'in' );
  WriteNuMesh( 'out', sqrt( x ) );
}
```

Bullfrog Network Description Code:

```
(define-function (sqrt in = (input real)
  out = (output real)
  language-type = C
  source-code = "sqrt.c"
  object-code = "sqrt.o")

(define-network ( fourth-root in = (input real)
  out = (output real))

  ((streams (foo real))
   (sqrt in = in out = foo)
   (sqrt in = foo out = out)))
```

The second expression creates a composite function, fourth-root, that can now be used as if it were a primitive function:

```
(define-network (sixteenth-root in = (input real)
  out = (output real))

  ((streams (baz real))
   ( fourth-root in = in out = baz )
   ( fourth-root in = baz out = out )))
```

This last expression creates a composite function containing two copies of the fourth-root composite function (shown below). The result can now be
used in other definitions as if it were a primitive. This can be repeated ad
infinitum (or at least until memory runs out).

Clearly, this approach has enormous advantages over the original, and the
advantages become more profound as the networks involved become more
complex. To take a simple case, if the example above were extended to
calculate the 256th root, the first method would require the addition of four
more nodes and four more streams—doubling its size and complexity, while
the Bullfrog example would require only one new **define-function**
expression.

The Bullfrog system should (eventually) calculate a suitable NuMesh
topology (or take one as input from the programmer), compile the final
composite network down to the module level, calculate a distribution of code
to network nodes, generate and execute compiler directives for the source
code modules, generate stream information and compile it using the virtual
stream compiler, and enforce type constraints on the NuMesh application.
Moreover, the introduction of the Bullfrog Network Description Language
separates the programming model presented to the NuMesh programmer
from the underlying hardware on which the application is running, which
frees the programmer to concentrate on the application's functionality
rather than the details of how to map the code modules onto the network
and how to interconnect them with virtual streams to get the correct output.
The programming model has changed dramatically. No longer is the model
focused on blocks and streams and their interconnections; now the model
focuses on functional units and their relationships to one another, independent of the underlying hardware.
Bullfrog Network Description Language

Overview

The Bullfrog Network Description Language was developed to be a very-high-level language for constructing NuMesh applications rapidly, flexibly, and robustly. A NuMesh application consists of a set of functional blocks that are connected by virtual streams of data and that perform some set of calculations or transformations on the data. The application runs on a hardware network composed of processing elements, wires, and communication finite-state machines [12]. Until now, the NuMesh programmer has only been able to do the NuMesh equivalent of assembly language programming on a conventional CISC or RISC single-processor computer. That is, the NuMesh programmer has had to work directly with and take into consideration the actual elements of the underlying hardware network when writing a NuMesh application. He has had to calculate or at least be familiar with the topology of the hardware network, calculate a distribution of low-level code to processor nodes, specify every parameter of every communication stream, enforce data type and other kinds of constraints, and debug the NuMesh application with very little in the way of supporting tools or utilities.

The Bullfrog Network Description Language was designed with the goals of extensibility and flexibility in mind. In addition, the language was influenced by the requirement of being easy for both a human and a computer to parse, while remaining compact and avoiding unnecessary syntactic complexity and overhead. [4][8]. The structure of the language reflects the physical structure of the NuMesh network, but at a more abstract level—for example, there are ways to specify code blocks, to abstract functional groups and use them as a single block, and attach streams to input and output ports on both kinds of block. The Bullfrog NDL
allows a NuMesh programmer to work with NuMesh network elements while at the same time separating him from the underlying topological and processor-specific details. The Bullfrog Network Description Language is based upon work done by Chris Metcalf on the NuMesh Interchange Format for Text (NIFT) [8], and the work described in [4].

**Extensible**

The Bullfrog language is extensible in several ways on various levels.

- The primitive environment used to preprocess and/or postprocess the input file can be extended by exposing primitive operators from C and adding arbitrary user-defined operators; these would be primitive (built-in) operators from the point of view of the Bullfrog programmer.

- Mechanisms can be added to enrich the Bullfrog NDL programming environment itself. For instance, the keyword `define` could be added to allow a Bullfrog NDL programmer to bind symbols to expressions and then use the symbol in place of the more complex expression in other expressions. Also, the CLispClosure functionality could be exposed to the Bullfrog NDL programmer to allow him to extend the primitive environment on the fly.

- The syntax of existing expressions can be extended to include additional parameters, or new keywords can be introduced by adding functions for processing those parameters and adding an entry to a dispatch table in the Bullfrog source code.

One anticipated future addition to the language is the ability to check specified data types against an external list of valid types, which may be different among applications or even among expressions within the same application. This way, if the source file pointed to by a define-function is a C file, for instance, the data types of ports and streams can be checked against all the types which are valid in the C language. This mechanism
could be made general enough that a function or network parameter could point to the file of type information, and then new files could be introduced as needed, without further change to the Bullfrog system's source code.

**Flexible**
The primitives are defined as needed, by creating a Bullfrog function out of an external high-level code module. The Network Description Language merely provides a framework for interconnecting modules and enforcing generic constraints; the functionality and interface of any primitive is dependent on the high-level module being wrapped. The Bullfrog NDL simply provides mechanisms for abstracting high-level modules so that the application programming model is separate from the details of the underlying physical network. Just as a programmer on a conventional computer gains flexibility and control over complexity through the use of a high-level language, the NuMesh programmer gains similar benefits by using a very-high-level language—the Bullfrog NDL. The language doesn't actually define the functions, and the parser and internal representation do not depend on what functions are being used. This makes it a very flexible environment to work in, because the programmer can extend the range of functionality incorporated into a NuMesh application as needed by adding functions using `define-function` and by adding networks using `define-network`—without having to change a single line of code in the Bullfrog sources.

**Ease of parsing**
The Bullfrog language uses a LISP-like syntax, for several reasons. For one, it is straightforward to parse a LISP expression into tokens and lists, the latter of which generally represent meaningful constructs in the language, and variable length lists of arguments are easy to handle. LISP
syntax is already known to many people, especially in the research community, where this software is expected to be used, at least initially. It is also easy to extend the set of parameters allowed within a specific expression. A very compelling reason for using this type of syntax is that we can also take advantage of existing LISP evaluators to preprocess the input file before operating upon it, thus allowing, for instance, general mathematical expressions as parameters rather than requiring hard-coded numbers there. Furthermore, the basic

\[
(operator_1 \ operand_1 \ operand_2 \ldots \ operand_n)
\]

format for expressions itself allows for a great deal of flexibility, in that when a new type of operand or operator is added to the language, the parser can be left unchanged—only the semantic analyzer needs to be expanded to handle the new element, and that addition in most cases can be accomplished simply by adding the function(s) necessary to process the construct and by adding a line or two to the appropriate dispatch table(s). Finally, complex expression evaluators, regular grammars, and other brittle constructs are not necessary because precedence is made explicit by the Bullfrog NDL programmer. Expressions in this format are not as easy for a human to work with, granted, but it becomes second nature after working with it for a while; also, the complexity of mathematical expressions used in Bullfrog NDL programs is not expected to be very great.
Language Specification

- **Comments.**
  Comments in a Bullfrog source file begin with a semicolon (";") and end at the next newline character.

- **Keywords.**
  The following keywords are recognized when they appear as the first symbol in a Bullfrog NDL expression.
  
  define-function
  define-network

  The following words and symbols are also keywords, but are only recognized as such in the context of certain subexpressions (described below).
  
  input
  language-type
  main
  object-code
  output
  real
  source-code
  streams
  +
  -
  *
  /

  +
  -
  *
  /
• **define-function**

The Bullfrog NDL does not contain any primitive operators in the sense of pre-defined functions that execute on network nodes. Instead, the **define-function** keyword allows for the introduction of specific primitive functions into the programming environment that are relevant to a particular application. It is expected that in time a library of such functions will be written or accumulated and that mechanisms for importing functions from and adding functions to such a library will be added to the Bullfrog NDL programming model. In other words, the basic building block in the Bullfrog NDL is the **define-function** expression, which specifies pointers to source, object, and/or executable versions of a function to be run on a NuMesh node, along with expressions defining the inputs and outputs for the function, types of and default values for parameters, and similar information. The **define-function** expression is used by the semantic analyzer to construct a corresponding CNMCodeBlock object (see **Internal Representation**).

The **define-function** expression has the following syntax:

```
(define-function ( <function-name>
   <port name> = ( input <port data type> ) ; input port specification
   <port name> = ( output <port data type> ) ; output port specification
   language-type = <expr1> ; C, LISP, FORTRAN, etc.
   source-code = <expr2> ; these may be code or a pointer to it
   object-code = <expr3> ; (currently, these are expected to be path names)
   ... ;
   ... ; include additional parameters here
   ... ; with syntax described below.
   <parameter-name> = <exprn>) ; arbitrary function parameter
```

Except for the function name, none of the parameters to **define-function** is position dependent.
<function name> is the name of the Bullfrog NDL function being defined. The name entered in this position will be used in other Bullfrog NDL expressions to reference this construct.

The syntax

\[ <\text{port name}> = ( \{ \text{input} | \text{output} \} <\text{port data type}> ) \]

is the manner in which a data port is specified for communicating with the function being defined. The <port name> should correspond to the name specified in the actual source file, so the appropriate inputs and outputs can be matched between the Bullfrog function and the source code function it is abstracting. The <port data type> value can be any data type; the compiler uses the textual symbol to enforce type constraints, without regard for the meaning of the symbol. The compiler could easily be made to recognize certain data types, however, if that becomes necessary for additional processing or some other reason.

language-type indicates the language the source code is written in (C, C++, LISP, etc.). This information will be used to generate compiler directives from the compiler output of the Bullfrog system.

source-code specifies the path to the source code to be compiled and run on the network hardware.

object-code specifies the path to the file containing the object code to be run on network hardware.

The syntax

\[ <\text{parameter-name}> = <\text{expr}> \]
is how arbitrary parameters to the function are specified. It is expected that
the source code will require certain compile-time constants to be defined,
and this is the mechanism that will allow for that, so the `<parameter-name>`
value will probably have to match what the source file is expecting. Other
annotations could be added in this way for whatever purposes might become
necessary in the future. These could be used for enforcing additional
constraints, for generating additional output, for entering debug
information, etc. Currently, the only expression recognized has the form

```
<parameter-name> = (real <numerical expression>)
```

where the `<numerical expression>` could be any arbitrary mathematical
expression recognized by the Bullfrog compiler (currently, any LISP-like
mathematical expression involving +, -, *, and / could be entered in this
position), and may include symbols as operands that are defined in an
enclosing environment. Parameters of this type are evaluated only once,
and evaluation is delayed until the network description is compiled and the
output generated. This expression provides a default value for the
parameter, which may be overridden when a function block is instantiated
(see `define-network`, below).

The specific key words shown in bold above are only examples of the
possible types of information that can be used to annotate the code block.
For instance, a full-blown compiler command could be specified instead of
the language type, or a list of flags could be entered in addition to the
language name, or whatever information turns out to be most useful and
convenient can be entered as parameters. The keywords explicitly shown
here are the ones currently recognized by the Bullfrog system and included
in the compiler output. Support would have to be added to the Bullfrog
sources for any other parameters, but this should be a straightforward task,
using the code for the existing parameters as a guide.
- **define-network**

  Functional abstraction is supported in the Bullfrog NDL using the `define-network` keyword, which is analogous to the `define-function` primitive in many ways. A network definition contains a list of parameters and inputs/outputs for the composite functional block being defined, in addition to a list of instantiations of the sub-blocks contained in the network and information specifying how they are interconnected. The sub-blocks may be either primitive functions or other, previously defined networks. Each `define-network` expression results in the creation of a corresponding CNMNetworkBlock in the internal representation.

  The network block has the following syntax:

  ```
  (define-network ( <network-name> 
   <parameter_name1> = <expr1> 
   <parameter_name2> = <expr2> 
   . 
   <parameter_name_n> = <expr_n> ) 
   ( ( streams ( <strm1_name> <attr1> <attr2> ... ) ( <strm2_name> ... ) ... ) 
   <block_instantiation1> 
   <block_instantiation2> 
   . 
   . 
   <block_instantiationN> ) )
  ```

  `<network-name>` is the name of the network being defined, and it is the only position dependent parameter.

  The syntax

  ```
  <parameter_name1> = <expr1>
  ```
is the same as for **define-function**, above.

The ( **streams** ... ) expression is analogous to a LISP **let** construct, and names streams internal to the network. Currently, every port in an instantiated function must connect to exactly one stream or other port, and every internal stream must connect to exactly two ports. It is expected that this requirement will change in future versions of the Bullfrog software (when support for broadcast and multicast communications are added, for example). The syntax for the **streams** expression is

\[
\text{(streams} \quad (<\text{stream name}_1> \ <\text{stream}_1 \text{ data type}>) \ldots \\
\quad (\ <\text{stream name}_n> \ <\text{stream}_n \text{ data type}>)) \).
\]

The **<block_instantiation>** expression is an instantiation of a function or network definition and takes the form:

\[
\text{(}<\text{function or network name}> \ <\text{port name}_1> = <\text{port or internal stream}_1> \\
\quad <\text{parameter name}_1> = <\text{parameter value}_1> \\
\quad \ldots \\
\quad <\text{port name}_m> = <\text{port or internal stream}_m> \\
\quad <\text{parameter name}_n> = <\text{parameter value}_n> \ )
\]

where:

- **<port_name>** is the name of a port in the definition of the function or network being instantiated,
- **<port or internal stream>** is the name of a port on the network being defined (not the one being instantiated) or the name of an internal stream specified in the **streams** expression
- **<parameter_name>** is the name of a parameter in the definition of the function or network being instantiated
- **<parameter_value>** is an expression of the form
(real <LISP-like mathematical expression>).
The operands used in the <parameter_value> expression may be numbers, or they may be names of parameters specified in the function or network being defined. In the latter case, when the <parameter_value> expression is evaluated, the symbol will take on the default value of the parameter if no overriding value is specified in the applicable instantiation, otherwise, the symbol will take on the overriding value of the parameter.

- **Application Entry Point.**
The entry point for a Bullfrog NDL application is the network with the distinguished name main.

There must be exactly one main network defined in the Bullfrog NDL application being compiled. This is the top-level network that implicitly attaches to the single host node. All inputs to main are attached implicitly to outputs on the host, and all outputs on main are implicitly attached to inputs on the host.

- **Language restrictions.**
A network definition can only reference previously defined networks and functions.
The ‘$’ character should be reserved for system names, as it is currently being used to separate the two parts of an object identifier (see CNuMeshID in chapter entitled Internal Representation).

---

34
Example

The following network description defines a network to calculate the sixteenth root of the values on an input stream from the host:

```lisp
(define-function (sqrt
    in = (input real)
    out = (output real)
    prec = (real 0.001)
    language-type = lisp
    object-code = "/projects/test.o"
    source-code = "/projects/test.lsp"))

(define-network (fourth-root
    in = (input real)
    out = (output real)
    prec = (real 0.002))
  ((streams (foo real))
    (sqrt in = in out = foo prec = (real (* 123 prec)))
    (sqrt in = foo out = out)))

(define-network (main
    in = (input real)
    out = (output real))
  ((streams (baz real))
    (fourth-root in = in out = baz)
    (fourth-root in = baz out = out prec = (real 1.0))))
```

; primitive function
; input port
; output port
; function parameter: default val = 0.001
; sqrt source language type
; sqrt object code path
; sqrt source code path

; composite function
; input port
; output port
; function parameter: default val = 0.002
; internal streams definition
; primitive instantiations; override sqrt
; default prec val with prec = 0.002
; or whatever value is given for prec
; when fourth-root is instantiated

; top-level network
; input port
; output port
; internal streams definition
; network instantiations; override prec
; default val with constant values
Parser

The parser is based on a C++ port of Professor Ward's CLisp package. The CLisp package is a set of functions and data structures capable of reading in LISP-like expressions, parsing them into tokens, and evaluating the resulting expressions in a LISP-like environment using an eval-apply loop similar to that also used in LISP [1][9]. The CLisp package also includes a simple garbage collection mechanism. The package allows C programmers to incorporate some of the flexibility of LISP into their C programs.

The Bullfrog parser is a C++ version of the CLisp package called C++Lisp (or CPPLisp), with extensions and modifications. Originally created as a general-purpose parsing utility, it has been tailored somewhat to the specific needs of the Bullfrog project. Specifically, the set of expressions it can parse has been expanded to include the syntax

\[ <\text{parm}> = <\text{value}>, \]

where \( <\text{parm}> \) is a parameter and \( <\text{value}> \) is a default value for \( <\text{parm}> \) (see the chapter on the Bullfrog Network Description Language). Also, the garbage collection mechanism has been replaced with a much simpler one. A more complex scheme could and should be implemented to replace what has been done for this work, as it keeps information in memory much longer than it needs to and places a couple of requirements on the Bullfrog programmer that are error-prone.

The parser converts the textual input into tokens organized in trees, with elements at the same level of nesting in the input expression located at the same depth in the tree, and tokens within the same parenthesized subexpression (or "list", to borrow the LISP term for it) found hanging in a chain from a common node. The figure below illustrates the parsing process graphically.
The output of this stage (trees of tokens representing expressions) can then undergo preprocessing—if mathematical expressions need to be evaluated at this point, for instance—or it can be passed to the semantic analyzer for construction of the internal network representation. This method of parsing, combined with the structure of the network description language, provides flexibility to the Bullfrog programmer and leaves room for later modification and extension. The parser aids in the production of robust NuMesh applications by detecting and reporting syntax errors in the input.

**Parser Components**

The hierarchy of classes that make up the parser is shown in Figure 4.1.

- **Tokens**
  
  This section describes the set of tokens and constructs into which the parser converts the input. The set of available tokens can be extended by deriving new objects from those found here, as in the case of CLispParameter. Implementation specific information about the tokens is included below for clarity, although each could be implemented in various ways.
a) Inheritance Diagram of C++ Lisp objects which comprise the set of Bullfrog lexical tokens which are output by the Bullfrog reader.

b) Inheritance Diagram of C++ Lisp classes that read Bullfrog text expressions and perform the conversion to lexical tokens.

Figure 4.1. Components of the Bullfrog parser.
Dotted lines represent Bullfrog-specific extensions to the CLisp package.
Arrows point toward superclasses.
The **CLispObject** class implements the basic functionality of the set of CPPLisp token objects, and all tokens are derived from it. It contains default implementations for functions that are available to more than one type of token, such as Eval() and Cons(), and for functions that need to report an error if called on the wrong type of object, such as Apply(). The CLispObject class also maintains bookkeeping information for the set of CLisp tokens created, and contains initialization functions used by the entire hierarchy or used for management of the entire set, such as InitializeSpecials() and FreeLispObjects(). Viewed as an abstract token, this object contains only type information. CLispObject is an abstract type and an object of this type cannot be instantiated directly; only a class derived from CLispObject and providing implementations for its abstract members can be instantiated.

- **Atomic Objects**
  This section describes the classes implementing the atomic tokens, which are the primitive types.

The **CLispInteger** object provides an immutable token representing an integer. Its type is CL_INT, and its value is an integer that can only be set at the time of creation. It is expected that in the future an integer table will be created so that each integer token is only created once, stored in the table and shared by all constructs that use it.
**CLispDouble**

<table>
<thead>
<tr>
<th>Object Type</th>
<th>CL_DOUBLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>&lt;floating point value&gt;</td>
</tr>
</tbody>
</table>

The **CLispDouble** object is an immutable token representing a floating point number. Its type is CL_DOUBLE, and its value is a floating point number that can only be set at the time of creation. It is expected that a table of floating point numbers will in the future be created similar to the one described above for CLispInteger.

**CLispSymbol**

<table>
<thead>
<tr>
<th>Object Type</th>
<th>CL_SYMBOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>&lt;alphanumeric symbol name&gt;</td>
</tr>
</tbody>
</table>

The **CLispSymbol** object is an immutable token representing an atomic alphanumeric symbol. A symbol is a non-literal string of alphanumeric characters (beginning with a non-numeric character) that generally (but not always) evaluates to something else, such as a procedure object, a number, or a subroutine. The type of a CLispSymbol is CL_SYMBOL, and its name is its alphanumeric string representation, which cannot be changed after the symbol is created. It is expected that in the future a symbol table will be created to keep track of all symbols used, that each symbol will only be created once, and that all uses of a symbol will point to the same object, as described above for CLispInteger.
The **CLispParameter** object is a mutable derivative of CLispSymbol. It was added to support the syntax "<parm> = <default value>" in the Bullfrog Network Description Language, which allows the Bullfrog NDL programmer to specify an overridable default value for a function parameter. A CLispParameter's name cannot be changed after creation, but its value can. The type of a CLispParameter object is CL_PARM, its name is its alphanumeric string representation (inherited from CLispSymbol), and its value is another CLispObject, which can be any type (although whatever type it is must correspond to the type expected by the function being defined).

The **CLispString** object is an immutable object encapsulating a literal string. Its type is CL_STRING and its name is the literal string which it represents. Its name cannot be changed after creation. It is expected that a string table will be added in the future, analogous to the tables described for CLispSymbol, CLispInteger, and CLispDouble so that every use of a particular string literal points to the same immutable object.
CLispSpecial

Object Type = CL_SPECIAL
Name = <string representation of special's token name>

The CLispSpecial class provides for a set of immutable system-defined symbols with reserved meanings. Many are used to flag error conditions. The following list enumerates the entire set of specials currently defined and explains the meaning of each one:

- **CL_End** Unmatched right parenthesis encountered.
- **CL_EOF** End of file token.
- **CL_Error** General error condition encountered.
- **CL_Unbound** Unbound object error condition encountered.
- **CL_Env** This token is placed at the head of a list to identify the list as an environment object.
- **CL_True** Token representing the boolean value TRUE.
- **CL_Nil** Token representing the boolean value FALSE; also used to mark the end of a list and anywhere a terminator symbol is required.

The value of a special is a string representation of the symbol’s name; CLispSpecials are unique in the sense that only one instance of each is created and they cannot be changed after creation.

The set of CLispSpecial objects should be global, as they currently are, but their constructors should probably be made private to help ensure that each is only created once (the constructor is currently public). Currently, the Bullfrog programmer must call CLispObject::InitializeSpecials() before the first use of these objects; this can probably be handled in a cleaner way automatically. Perhaps this call could be added to the CNMRootBlock constructor if the symbol table were added to the CNMRootBlock representation as described in the section below discussing garbage collection.
• **Composite Objects**

This section describes the classes that support organization of tokens into semantically meaningful structures.

### CLispSubroutine

<table>
<thead>
<tr>
<th>Object Type</th>
<th>CL_SUBR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>&lt;subroutine name (string)&gt;</td>
</tr>
<tr>
<td>Function</td>
<td>&lt;pointer to C function&gt;</td>
</tr>
<tr>
<td>Number of Args</td>
<td>&lt;number of args required by function&gt;</td>
</tr>
</tbody>
</table>

The **CLispSubroutine** object allows the Bullfrog programmer to package a C function as a LISP procedure that can be bound in an environment and applied to a list of arguments just like a LISP closure. It is used mainly for defining primitive operators to be bound in the primitive environment. Its type is CL_SUBR. It is immutable—it cannot be modified after construction. Its name is the name of the subroutine (mainly used for display purposes when dumping an environment or printing out the value of a subroutine object), its function is a pointer to the underlying C function, and its number of arguments is the number of arguments required by the function, where a zero specified here means either zero arguments or a variable number of arguments are required.

### CLispClosure

<table>
<thead>
<tr>
<th>Object Type</th>
<th>CL_CLOSURE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formals</td>
<td>&lt;pointer to list of formal parameters&gt;</td>
</tr>
<tr>
<td>Body</td>
<td>&lt;pointer to body of procedure (executable content)&gt;</td>
</tr>
<tr>
<td>Environment</td>
<td>&lt;Pointer to environment in which body is evaluated&gt;</td>
</tr>
</tbody>
</table>

The **CLispClosure** object provides an abstraction mechanism for composing new operations out of operations already defined in an environment. Its type is CL_CLOSURE and it is immutable. It contains a list of formal parameters that are evaluated at the time of execution in an environment passed to the closure, a body represented as a list of CLispObjects (possibly some of which are other CLispLists that may
contain CLispClosures) that specifies the operations the closure should carry out, and an environment in which the body should be evaluated.

**CLispList**

<table>
<thead>
<tr>
<th>Object Type = CL_LIST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car = &amp;CLispObject</td>
</tr>
<tr>
<td>Cdr = &amp;CLispList</td>
</tr>
</tbody>
</table>

The **CLispList** class provides the backbone out of which all token trees are constructed. It is mutable, as its Car and Cdr may be set; this may change in the future, as this kind of mutation can cause problems in some situations. Often, two sets of functions are provided in a LISP implementation for setting these values—one that mutates the list and another that copies it and modifies the copy—so the user can perform the appropriate operation in each case. This would be simple to add to this implementation if necessary in the future. A list can contain other lists, which is how nested expressions are represented. Every list is terminated with the null list (represented by the CL_Nil CLispSpecial object).

**CLispEnvironment**

<table>
<thead>
<tr>
<th>Object Type = CL_ENV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Car = &amp;CLispObject</td>
</tr>
<tr>
<td>Cdr = &amp;CLispList</td>
</tr>
</tbody>
</table>

A **CLispEnvironment** is, in this implementation, a derivative of the CLispList class, though it doesn’t have to be (see CNMEnvironment, under Internal Representation). It is, conceptually, a map of <key, value> pairs where the key is a CLispSymbol and the value can be any type derived from CLispObject. Environments can be chained (referred to as one environment enclosing another), such that a binding in one environment can shadow another binding with the same key in an enclosed environment. This object is mutable.
CLispPrimEnv

The **CLispPrimEnv** object is a specialized CLispEnvironment. It is an implementation of the primitive environment, which is the environment enclosing all other environments and containing the basic ("primitive") operations provided by the system (such as the arithmetic operators: +, -, /, *). There can only be one instance of the CLispPrimEnv class in an application.

**LISP Reader (Lexical Analyzer)**

The Lisp reader (lexical analyzer) is the module that reads in text and converts it into the appropriate tokens and organizes them into the appropriate structures.

**CLispReader**

The CLispReader reads text from a source file and converts the white space delimited symbols into the tokens and structures described in the previous section.

**CBullfrogReader**

The CBullfrogReader is very similar to the CLispReader, except it has been specialized for use in the Bullfrog system through the addition of specific support for the CLispParameter syntax

\[ <parm> = <expr> \].
Example

Figure 4.2 illustrate the way the parser transforms its input (a Bullfrog NDL text file) into its output (a tree of CPPLisp tokens), using the same sample NuMesh application from the previous chapter that calculates the sixteenth root of its input.
(define-function (sqrt
  in = (input real)
  out = (output real)
  prec = (real 0.001)
  language-type = lisp
  object-code = "/projects/test.o"
  source-code = "/projects/test.lsp")

; primitive function
; input port
; output port
; function parameter: default val = 0.001
; sqrt source language type
; sqrt object code path
; sqrt source code path

Figure 4.2 a.

Figure 4.2. a) Bullfrog parser input. b) Bullfrog parser output.
Garbage Collection

Garbage collection is currently being handled in a relatively crude manner. As CLispObjects are created, they are added to a list containing all previously created CLispObjects. The Bullfrog programmer must be sure to call CLispObject::FreeLispObjects() before the program terminates in order to reclaim that memory. This also means that all objects derived from CLispObject must be allocated using the C++ operator `new`; use of automatic (local or stack) variables will result in a memory violation when the call to FreeLispObjects() tries to delete an object that was already deleted by the system when it went out of scope. The call to FreeLispObjects() should not be made until just before program termination, as the CNMEnvironment objects contain CLispObject-derived objects that must remain valid until the network is compiled and the pertinent information is output to files. Figure ??? illustrates the structure of the symbol table.

This garbage collection scheme can and should be replaced in future versions of the software with a more sophisticated scheme. For instance, separate tables could be maintained for the different types of CLispObjects, and then things like a particular integer or string could be shared by all users rather than creating duplicates when one instance of such an object is already in existence, and certain tables could be deleted when the parser finishes reading the input file rather than being kept around until termination of execution, when FreeLispObjects is called. Also, FreeLispObjects() could be added to the CNMRootBlock destructor, perhaps, so the programmer doesn't have to make the call explicitly (the symbol table, which is currently a global object, would have to be added to the CNMRootBlock class representation to support this behavior). The insertion of objects into the symbol table could be limited to dynamically allocated objects so that automatic objects are permitted without causing
problems. Finally, any of a number of standard garbage collection schemes could be implemented [1].

Figure 4.3. CLispObject table used for garbage collection.
Internal Representation

Overview

The core of the Bullfrog system—and of the NuMesh application development system overall—is the set of data structures comprising the internal representation of the NuMesh network application. This representation must be compact, robust, and flexible; it must support dynamic functional abstraction; it must be modular; and it must contain hooks for extending the system in several ways.

**Compact.** The internal representation must be compact so that it does not use an inordinate amount of system resources—primarily RAM and disk space—so that it can accommodate large NuMesh applications. Also, a programmer must be able to save Bullfrog programs to disk in a format that is human readable and that conforms to the syntax and semantics of the Bullfrog language, which itself has a compact notation and could not be reasonably synthesized from a fully expanded representation.

**Robust & Flexible.** The intermediate representation must be constructed in such a way that it prevents programmer bugs (whether the programmer is working with static text files or—more importantly—a dynamic graphical user interface) and allows objects to be added to, removed from, or reconnected within the evolving structure dynamically without difficulty. Also, since this is a first version of the evolving Bullfrog language and NuMesh development environment, the internal representation must be flexible enough to handle unexpected changes that will undoubtedly occur in future revisions of the NuMesh hardware and software. This flexibility is provided in part by the modular and extensible structure of the internal representation, and also by the flexible way in which constraints (such as
the constraint that streams must have the same type as all the ports to which they are attached) are specified and enforced, and the alternate ways in which they could easily be specified or enforced.

**Modular.** The programmers extending and maintaining the Bullfrog software itself must be able to change the specification of one component (such as the CNMPort object) without having to do a lot of work changing the other objects that rely on it. The internal representation must also be insulated from the input and output modules so that their structures may be changed independently without affecting one another adversely.

**Support for Dynamic Functional Abstraction.** If an application of any significant complexity is to be constructed, the NuMesh programmer needs to be able to build complex objects out of simpler objects and then use those complex objects the same way that he uses simple objects in other constructs or save them in a library for future use.

**Extensible.** The internal representation allows for extensibility in several ways. The data structures of which the representation is comprised are self-contained objects containing both data and associated methods. It is easy to add objects to this set and derive more complex objects from the objects that exist now; for example, a set of data objects representing different processor types could be added and used to manipulate a network application for some specific purpose. Changing the existing objects is straightforward in many cases, as long as the interface through which other objects interact with an object is left intact (methods can be added to an object without any trouble, but changing or removing parts of the interface requires updating all objects utilizing that interface). Each object has associated with it a generic string map in which arbitrary information (such as arbitrary type information) can be stored, and certain objects have an environment that can hold any object.
derived from the appropriate class (described below). This environment has methods for binding and evaluating such objects, and is very flexible as to the kinds of operations that it can be made to support. Currently, parameters to function blocks can be stored as complex mathematical expressions and evaluated only when needed. These expressions can contain values that are defined in another environment, and so on.

The hierarchy of classes making up the core of the internal representation is shown in figure 5.1.
Figure 5.1 – Inheritance diagram of classes in the Bullfrog internal network representation. (Arrows point toward superclasses.)
The objects that make up the internal representation are analogous to the physical components in the hardware network itself. Not only is this an obvious way to decompose the system into modules, but it is also the decomposition allowing for the most straightforward extension of the representation as new components are added in the future (such as different kinds of processor nodes or specific types of ports), and it is thought to be the design that will support a graphical user interface in what seems to be the most straightforward manner. Since the objects the user will be working with in the GUI will be function blocks, virtual streams, nodes, ports, and the other components of the NuMesh programming model, it clearly makes sense to have these be some of the objects of the internal representation. The most important and complex objects are the terminal nodes in the inheritance graph.

Following is a description of each element and some explanation of its role in the framework. For clarity, not all information associated with each object is listed here—only those items that implement the representation of the abstract object under consideration.

**Naming Conventions**

A couple of conventions are loosely adhered to in naming the different objects in the system. The initial C, as in CNMPort, indicates the object is a class. The letters NM stand for NuMesh, to indicate that the object is a part of the NuMesh package, as opposed to some other package (e.g., CLispObject).
Some of the diagrams below illustrating the structure of the different objects have colored tags to the left of one or more entries. These tags have the following meanings:

- A solid black square indicates the specified object is a static member.
- A solid gray square indicates the specified member is inherited from a superclass.

Object Descriptions

**CNuMeshIDServer**

*CNuMeshIDServer* is a factory object which, when queried for an object identifier, returns a *CNuMeshID* guaranteed to be unique among all identifiers produced by that *CNuMeshIDServer* instance.

**CNuMeshID**

<table>
<thead>
<tr>
<th>String: &lt;identifier root&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int: &lt;a number to concatenate with the root to make ID unique&gt;</td>
</tr>
</tbody>
</table>

The *CNuMeshID* object supports a naming system for NuMesh objects that allows each object to have a globally unique handle but also allows two objects to be compared for similarity. The root is used to compare two objects for similarity (when an object is cloned, for instance, the two objects are said to be “similar” as opposed to “equal” and their common ID root reflects this), while the object handle created by concatenating the ID’s root string to its integer component as <root string>$<integer>$ provides a handle which can be guaranteed to be globally unique (globally within an application) if all ID’s are obtained through the same *CNuMeshIDServer* object.
CNMOBJECT

CNMObject is the generic root class from which all NuMesh objects are derived. It contains default definitions for overridable functions common to all NuMesh objects, and serves as a repository for information common to all such objects. The string map is used to hold the attributes of the CNMObject and other annotations pertaining to an individual CNMObject. One function of the string map is to serve as a mechanism for enabling the enforcement of generic constraints, such as data type constraints, through the comparison of <key,value> pairs associated with different objects, without regard for the meaning of the key. The contained CNuMeshIDServer is a static object shared by all CNMObject instances and publicly accessible outside the class. All object ID's must be obtained through this server to ensure uniqueness, which is critical.

CNMStream

A CNMStream is an abstraction of a NuMesh communication stream. It can connect to any number of CNMPort objects, and the ports connected by a particular stream are contained in its CNMPortManager object. The CNuMeshID and CStringMap objects are inherited from CNMObject.
The **CNMPort** object is an abstraction of a data port in the NuMesh world. Through a CNMPort is the way inputs and outputs of a CNMBlock and its derivatives are specified. A CNMPort object can currently be connected to only one CNMStream at a time, but the stream can be connected to an unlimited number of other ports. Each CNMPort is associated with a single block (its owner), to which it maintains a pointer. Each port also maintains a bit vector (called a PortAttrVector) that determines the type of the port (input, output, etc.)

### CNMPort

<table>
<thead>
<tr>
<th>CNuMeshID: &lt;unique object ID&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>CStringMap: &lt;port attributes: port data type, etc.&gt;</td>
</tr>
<tr>
<td>CNMBlock: &lt;pointer to block which owns this port&gt;</td>
</tr>
<tr>
<td>CNMStream: &lt;pointer to stream to which this port is connected&gt;</td>
</tr>
<tr>
<td>PortAttrVector: &lt;bit vector of port attributes: input, output, etc.&gt;</td>
</tr>
</tbody>
</table>

### CNMBlock

<table>
<thead>
<tr>
<th>CNuMeshID: &lt;unique object ID&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>CStringMap: &lt;miscellaneous annotations or attributes&gt;</td>
</tr>
<tr>
<td>CNMPortManager: &lt;set of input/output ports attached to this block&gt;</td>
</tr>
<tr>
<td>CNMEnvironment: &lt;set of (symbol, value) pairs (parameters, etc.&gt;)</td>
</tr>
</tbody>
</table>

A **CNMBlock** is an abstraction of a functional block. It provides a uniform interface for all the different types of block objects (see below) and allows each kind of block to be viewed as a set of consistent block properties and a set of ports through which the block can be connected to streams (and, possibly, other entities), whether it is a primitive code block or a composite block composed of a network of many other blocks, or some other kind of object. **CNMBlock** is the base class from which all NuMesh block classes will be derived. The contained port manager manages the data ports attached to this block, and the environment stores all of the parameters associated with the block, as well as the parameter types and default values. Other information could be stored there as well, as the need arises.
A **CNMClosure** supports functional abstraction; it is a type of block that encloses another block. It is used to instantiate a block previously defined, possibly overriding default parameters specified in that block's definition. Its ports are copies of the ports on the original block, and it holds a pointer to the enclosed block. The closure's environment shadows the enclosed block's environment. This is so that lookups will look first in the closure's environment, where parameters would have been overridden, and then in the original block's environment if the symbol is not bound in the closure's environment.

A **CNMClosure** supports functional abstraction; it is a type of block that encloses another block. It is used to instantiate a block previously defined, possibly overriding default parameters specified in that block’s definition. Its ports are copies of the ports on the original block, and it holds a pointer to the enclosed block. The closure’s environment shadows the enclosed block’s environment. This is so that lookups will look first in the closure’s environment, where parameters would have been overridden, and then in the original block’s environment if the symbol is not bound in the closure’s environment.

The **CNMCodeBlock** class provides the mechanism for wrapping source code modules to be run on the network nodes in a Bullfrog interface so they can be manipulated as blocks. A code block’s ports do not get attached to anything in the internal representation, since there is a single copy of each code block in that representation. When a code block is instantiated in a *define-network* expression, a CNMClosure is created that encloses the code block and the new closure’s ports are connected to streams as specified by the instantiation parameters. This not only leads to a compact internal representation, but it also allows a code block to be modified dynamically.
(using a GUI, for instance) without causing problems for the objects that interact with and depend on it (such as composite functions defined in terms of the code block being changed, for instance). The code block's string map contains the pathname of its source file, the source language type, and whatever other information is needed by the compiler that is actually going to compile the source file into executable code.

**CNMEnvironment**

<table>
<thead>
<tr>
<th>CNuMeshID: &lt;unique object ID&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>CStringMap: &lt;Miscellaneous information or attributes&gt;</td>
</tr>
<tr>
<td>CNMEnvironment: &lt;enclosing environment&gt;</td>
</tr>
<tr>
<td>CNMEnvironment: &lt;enclosed environment&gt;</td>
</tr>
<tr>
<td>CLispObjectMap: &lt;map of (String, CLispObject) symbol bindings&gt;</td>
</tr>
</tbody>
</table>

The **CNMEnvironment** class is an abstraction for a container class containing a set of CLispObjects indexed by strings, along with a pointer to the enclosing and enclosed CNMEnvironment objects (for purposes of chaining CNMEnvironments). It is analogous to a LISP environment (in fact, it is an alternate implementation to the CLispEnvironment class that implements the same abstract object).

**CNMManager**

<table>
<thead>
<tr>
<th>CNuMeshID: &lt;unique object ID&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>CStringMap: &lt;Miscellaneous information or attributes&gt;</td>
</tr>
<tr>
<td>CNMOBJECTMap: &lt;map of (String, CNMOBJECT) pairs&gt;</td>
</tr>
</tbody>
</table>

**CNMManager** is a class that represents a set of data objects indexed by string values, along with a set of functions to operate on, iterate through, and otherwise simplify the handling of the set of objects as a group. It is the generic base class for all of the specialized managers that handle objects derived from CNMOBJECT. This class generally should not be instantiated directly; rather, it should be used to derive a type-safe subclass that can be instantiated directly.
CNMBlockManager

| CNuMeshID: <unique object ID> |
| CStringMap: <Miscellaneous information or attributes> |
| CNMObjectMap: <map of (String, CNMBlock) pairs> |

The CNMBlockManager class is a type-safe manager class. It inherits everything from CNMManager, but enforces constraints on objects added to and removed from the set of objects it manages, making sure they are CNMBlock objects (or objects derived from CNMBlock).

CNMClosureManager

| CNuMeshID: <unique object ID> |
| CStringMap: <Miscellaneous information or attributes> |
| CNMObjectMap: <map of (String, CNMClosure) pairs> |

The CNMClosureManager class is a type-safe manager class. It inherits everything from CNMManager, but enforces constraints on objects added to and removed from the set of objects it manages, making sure they are CNMClosure objects (or objects derived from CNMClosure).

CNMFunctionNetwork

| CNuMeshID: <unique object ID> |
| CStringMap: <Miscellaneous information or attributes> |
| CNMObjectMap: <map of (String, CNMClosure) pairs> |
| CNMStreamManager: <set of internal streams connecting closures> |

A CNMFunctionNetwork is an abstraction of a network of instantiated function blocks connected by streams. Each block instantiation is represented by a CNMClosure object.
The **CNMPortManager** class is a type-safe manager class. It inherits everything from CNMManager, but enforces constraints on objects added to and removed from the set of objects it manages, making sure they are CNMPort objects (or objects derived from CNMPort).

The **CNMStreamManager** class is a type-safe manager class. It inherits everything from CNMManager, but enforces constraints on objects added to and removed from the set of objects it manages, making sure they are CNMStream objects (or objects derived from CNMStream).

**CNMNetworkBlock** is the class the Bullfrog system uses to support functional abstraction. It contains a set of CNMClosure objects, as well as a set of CNMStream objects connecting the closures. The CNMNetworkBlock serves as an interface shell that allows one to create a network out of other blocks connected by streams and make it look to the user like a simple
block. The stream manager that is not inherited manages the streams that maps the network block’s ports to the ports of its contained closures.

**CNMRootBlock**

<table>
<thead>
<tr>
<th>CNuMeshID: &lt;unique object ID&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>CStringMap: &lt;miscellaneous annotations or attributes&gt;</td>
</tr>
<tr>
<td>CNMPortManager: &lt;set of input/output ports attached to this block&gt;</td>
</tr>
<tr>
<td>CNMEnvironment: &lt;set of (symbol, value) pairs (parameters, etc.)&gt;</td>
</tr>
<tr>
<td>CNMStreamManager: &lt;set of internal streams connecting closures&gt;</td>
</tr>
<tr>
<td>CNMObjectMap: &lt;map of (String, CNMClosure) pairs&gt;</td>
</tr>
<tr>
<td>CNMStreamManager: &lt;set of streams connected to block’s ports&gt;</td>
</tr>
</tbody>
</table>

The CNMRootBlock is an abstract representation of the distinguished root block in a network, when the network is viewed as a tree with code blocks as leaves and network blocks as intermediate nodes. There must be exactly one root block in a network, which encapsulates the entire set of blocks and streams making up the NuMesh application being run on the hardware.

The internal representation for the sample Bullfrog NDL program that calculates sixteenth roots is shown in Figure 5.2.

This is a convenient representation for manipulating or building a network application, since a change made to the sqrt code block, for example, will be reflected immediately in everything that depends on sqrt, the representation is compact, and there is a unique Bullfrog NDL description of the network, which makes it easy to store the representation as an intermediate Bullfrog NDL text file. At run-time, on the other hand, there must be information mapping executable code to network nodes. The Bullfrog compiler would compile the above structure into a degenerate network that could be mapped directly to processor nodes. The next chapter explains the process carried out in compiling the internal representation to generate the Bullfrog system’s output.
Figure 5.2 Internal Representation of the sixteenth-root example.
Bullfrog Output Generation - (Expander & Extractor)

While a network representation is being manipulated through a GUI or some other means, and in order to enable a programmer to store a network description to a permanent file in the form of Bullfrog Network Description Language syntax, the internal representation has to be quite compact and has to maintain certain explicit relationships which are a part of the abstract programming model but not necessarily part of the run-time model of the application. An example of such a representation that is utilized by the Bullfrog system was shown at the end of the previous chapter. On the other hand, this abstract representation of the application has to be compiled into another form before it can be loaded onto the hardware. This section describes the conversion of one form to another and the extraction of information that is then output for use in other stages of the NuMesh programming process. The form and content of these files will be described below.

The procedure the Bullfrog compiler executes in converting the compact and abstract internal representation of a network into the more fundamental degenerate network that can be mapped onto the network hardware is as follows:
Network Expansion

- **Clone the abstract network, fully expanding each node so that each closure has its own copy of the enclosed block:**

![Diagram](image)

**Figure 6.1** Internal Representation of the sixteenth-root example (expanded clone).
Data Extraction

- Starting at the root block, iterate through the enclosed blocks and propagate each internal stream down to the code blocks it connects. Perform this step recursively for each block encountered. Internal streams are those streams contained in a network block that connect only contained closures—not the streams connected to the ports on the block being examined.

Figure 6.2 Internal and Port streams.

Figure 6.3 Internal Representation of the sixteenth-root example, after internal streams propagated.
- Propagate the root node's port streams down to the code blocks to which they connect.

**Figure 6.4** Internal Representation of the sixteenth-root example, after connecting port streams.
The result is equivalent to the structure shown below, composed of only code blocks (in this case, all of which are sqrt code blocks) and the streams connecting them (note that this could be mapped to a variety of network topologies—this degenerate network merely provides the network application represented in terms of the fundamental elements that are going to be mapped directly to the hardware):

As the information depicted in the figure above is extracted from the internal representation, it is output to three files. The degenerate network could certainly be assembled and stored in memory to be operated upon further, but this is not currently happening—as a stream is propagated down to the level of code blocks, for instance, the stream information is dumped to the appropriate file and the stream in memory is discarded.

**Parameter Evaluation**

The evaluation of parameters in Bullfrog NDL functions and networks is delayed until the network is compiled, so that the values of parameters could potentially be changed dynamically (i.e., through a GUI) and everything using the value of the changed parameter would reflect that change at compile time. During compilation, the following procedure is executed to calculate all the parameter values that are dumped to the compiler.bfg file (see next section).

- *Starting at the root block, iterate through the closures it contains.*
- For each closure contained in the current network block, evaluate the value of each parameter in the closure's environment with respect to the environment of the current network block. This causes the right hand side of parameter specifications in instantiations inside the network block to use the value of the parameter defined in the network block.

- For each parameter in the current closure's environment, insert a copy of the parameter into the environment of the block enclosed by the closure, overriding any existing parameter with the same name. This forces any parameters that were overridden in an instantiation to take the overriding value rather than the default value they were given.

- Recurse through the entire sequence for any network block encountered.

- When a code block is encountered, dump the contents of the environment to the output (see next section).

Note that the procedure is irreversible, in the sense that all unevaluated expressions are evaluated and replaced with the result of that evaluation, and default values of parameters are replaced destructively with their override values.

Turning once again to our 16th root example, the process would be carried out as follows (the text of the example is reproduced here for convenience):

```lisp
(define-function (sqrt
    in = (input real)
    out = (output real)
    prec = (real 0.001)
    language-type = lisp
    object-code = "./projects/test.o"
    source-code = "./projects/test.lsp")
```
(define-network (fourth-root
  in = (input real)
  out = (output real)
  prec = (real 0.002))
((streams (foo real))
  (sqrt in = in out = foo prec = (real (* 123 prec)))
  (sqrt in = foo out = out)))

(define-network (main
  in = (input real)
  out = (output real))
((streams (baz real))
  (fourth-root in = in out = baz)
  (fourth-root in = baz out = out prec = (real 1.0))))

This program creates the structure shown in figure 6.5. Only relevant information is included in the diagrams.

Figure 6.5. Sixteenth root structure before parameter evaluation.
During compilation, there is a flow of data from the top of the tree to the bottom, according to the following two rules:

1. A closure’s parameters are evaluated in the environment of the surrounding network block
2. After evaluation, a closure’s parameters are pushed down the tree by being bound in the environment of the enclosed network block

The result of applying these rules recursively to the sixteenth root example is shown in figure 6.6.

Figure 6.6. Sixteenth root structure after parameter evaluation.
Comparing the parameters in the code blocks’ environments with the compiler.bfg output from next section shows that these are indeed the results obtained by compiling the sixteenth root example.

**Bullfrog Output Files**

The output files generated by the Bullfrog compiler contain stream information, node information, and information needed to appropriately compile the high-level source code as explained in the following three sections. These files will be referred to as streams.bfg, nodes.bfg, and compiler.bfg, respectively. These are the default names for the output files, but they can be overridden by using the appropriate switches on the command line when invoking the Bullfrog compiler.

- **Nodes.bfg**
  The nodes.bfg file contains name and address information for each node in the hardware network. A node address is a cartesian 3-tuple giving the (x, y, z) coordinates of the node in the three-dimensional network topology. The host is currently placed at address (0, 0, 0) and the rest of the nodes are placed in a linear mesh with varying x value. A module for calculating a more optimized topology and mapping code onto it is expected to be implemented in the future. Such a module could either operate upon the internal representation directly, or it could parse the nodes.bfg file and work with the data in that form. Each method would have its particular advantages.
The node definitions have the syntax:

\[(\text{node} \ <\text{node name}> \ (<\text{addr} \ <x \text{ coordinate}> \ <y \text{ coordinate}> \ <z \text{ coordinate}>))\]

The nodes generated by the sixteenth root example are as follows:

\[(\text{node} \ \text{host} \ (<\text{addr} \ 0 \ 0 \ 0))\]

\[(\text{node} \ \text{sqrt}$4$ \ (<\text{addr} \ 1 \ 0 \ 0))\]

\[(\text{node} \ \text{sqrt}$6$ \ (<\text{addr} \ 2 \ 0 \ 0))\]

\[(\text{node} \ \text{sqrt}$8$ \ (<\text{addr} \ 3 \ 0 \ 0))\]

\[(\text{node} \ \text{sqrt}$10$ \ (<\text{addr} \ 4 \ 0 \ 0))\]

- \textit{Streams.bfg}

The streams.bfg file contains definitions of the communication streams connecting the function blocks defined by the high-level code modules used as basic building blocks in the NuMesh application. A stream definition has the following form:

\[(\text{stream} \ <\text{stream name}> \]
\[\quad (<\text{src} \ <\text{source node}> \ <\text{port on source node to which stream is connected}>))\]
\[\quad (<\text{dest} \ <\text{destination node}> \ <\text{port on destination node to which stream is connected}>))\]

The \textit{src} and \textit{dest} expressions are not position-dependent. The \textit{<stream-name> is} the name of the stream being defined. The source and destination node names refer to nodes defined in the node information file,
described in the next section. The port names refer to a specific port on the node, as specified in the define-function expression.

The streams for the sixteenth root example look like this:

(stream baz$1
  (dest sqrt$8 in)
  (src sqrt$6 out))

(stream foo$1
  (dest sqrt$6 in)
  (src sqrt$4 out))

(stream foo$2
  (dest sqrt$10 in)
  (src sqrt$8 out))

(stream stream$8
  (src host out0)
  (dest sqrt$4 in))

(stream stream$9
  (src sqrt$10 out)
  (dest host in0))

The degenerate network that these streams came from would have looked like this:

```
            stream$8
                Host
                stream$9
                   sqrt$4
                      foo$1
                   sqrt$6
                      baz$1
                   sqrt$8
                      foo$2
                   sqrt$10
```

74
These stream definitions would then be compiled by the Tadpole Stream Compiler [7] to produce executable CFSM code.

- *Compiler.bfg*

The third file, compiler.bfg, contains information that can be used to compile the source files encapsulated by the define-function expressions in the Bullfrog NDL code.

The compiler information will be used to compile the source files with specific compile-time constants calculated from information in the Bullfrog NDL source, such as values for function parameters. The information for a particular node includes the name of the source language (if specified in the function definition), so an appropriate compiler can be used on it; it also includes the source file location, the node name, and any compile-time constants that need to be defined. The syntax for a node’s compiler information block is the following (this can be changed easily by adding appropriate code to the Bullfrog sources to dump whatever information is required):

```
(node <node name>)
[ (source <source file path name>) ]
[ (file-type <language source file is written in, as specified in the define-function expression>) ]
[ <dump of the parameters defined for the function block, calculated from the Bullfrog NDL file> ]
```
Information for different nodes is separated by a blank line. The compiler.bfg file generated by the sixteenth root example is shown here:

(node sqrt$4)
(source /projects/test.lsp)
(file-type lisp)
prec=0.246000

(node sqrt$6)
(source /projects/test.lsp)
(file-type lisp)
prec=0.001000

(node sqrt$8)
(source /projects/test.lsp)
(file-type lisp)
prec=123.000000

(node sqrt$10)
(source /projects/test.lsp)
(file-type lisp)
prec=0.001000
Graphical User Interface

The graphical user interface will be a very important component of the NuMesh integrated development environment (IDE) when it is added in the future. It is expected that a graphical user interface will enormously simplify the task of creating NuMesh applications by allowing a NuMesh programmer to drag visual icons representing entities such as function blocks from a library of objects, drop them on a worksheet, and connect them to other such elements by clicking on the desired block ports and other hot spots. Double clicking on a function block might display the block’s properties, parameters, and internal structure (in the case of a network block), and allow these properties to be edited. If a text editor were incorporated into the GUI, then clicking on a function block could bring up the source code for the function and allow it to be edited and then recompiled without ever leaving the NuMesh development environment. An entire application could be constructed in this manner without the programmer ever having to use a tool outside of the NuMesh IDE or deal with a single line of Bullfrog NDL source code. With this in mind, great care was taken in this project to provide support for the future addition of such a GUI in the most straightforward manner possible. Previous incarnations of the NuMesh application development GUI can be found in [5] and [11].

One possible method for implementing the GUI would be to add a CNMGraphicalObject class to the internal representation and then add this object as a superclass of each existing class that has a graphical counterpart. This class would contain screen coordinates, a pointer to an icon, and other information needed to keep track of and display the graphical icons being manipulated by the programmer. It would also have member functions for manipulating these data, for displaying the object, and for updating the non-graphical data associated with the object when the
A graphical object was modified (by attaching a port to a stream, for example). Adding just the functionality described here (which does not involve a great amount of additional work) would allow for the implementation of a rudimentary GUI that would support most of the elements of the text-based version of the current Bullfrog Network Description Language.
Conclusion

Other Tools

The NuMesh project is entering an exciting phase of its evolution characterized by rapid progress on both the hardware and software fronts. The software tools and specifications produced as part of this thesis will hopefully represent a significant step toward the goal of a robust, integrated, user-friendly, and powerful development environment for the production of NuMesh applications by providing a framework upon which additional tools and modes of interaction can be built, such as the module currently under development by Greg Spurrier of the NuMesh group that will operate upon the internal representation described above to calculate a distribution of executable code to processor nodes that is balanced and that runs a specific application efficiently. Research is also being done on the types of applications most suited to the computational model supported by NuMesh and demonstrations are being constructed to showcase and explore its potential. In addition to the module described above, a stream compiler is near completion that will operate upon the communication output from the Bullfrog compiler and produce code executable by the nodes' communication finite state machines [7]. Compilers already exist for the high-level languages in which the executable functions' source code is written, and these compilers can be integrated with the Bullfrog tools either using a batch file or script (immediately) or by merging the tools at the source level (over time). Clearly, many of the necessary development tools are now in place and several more have become more or less defined during the course of this effort such that a relatively sophisticated and coherent NuMesh software system is coalescing rapidly.
Future Bullfrog Enhancements

During the course of this project, improvements were made in the available C++ compilers that will enable simplification and extension of the Bullfrog software in future revisions. For example, the recent introduction of C++ templates has eliminated the need for separate classes to represent the different types of manager objects (CNMBlockManager, CNMStreamManager, etc.) since a single template could be written that would support them all. This generality could also be exposed to the user, allowing him to produce new managers on the fly for new objects—such as an object to represent a specific kind of processing element—also created on the fly. Furthermore, the recent addition of dynamic type information available to the C++ programmer at run-time would eliminate the need for the code included in this project to support the minimal level of type information necessary to differentiate between types of blocks, for instance. These are just a couple of the obvious changes (and a few of their possible implications) that have occurred in the short time since the initiation of this particular part of the project—many more are no doubt imminent that will enable further streamlining of the system and expansion of the set of tools and functions it exposes to the NuMesh programmer.

What’s Next

The next logical component that should be added to the set of NuMesh system tools is a graphical user interface that sits on top of the work presented here to allow for the visual construction and editing of a network application. Research has already been done on this component [5] [11] which can serve as a starting point for this effort, providing designs, ideas, and perhaps even some reusable software components that would prove useful. It is expected that this will be relatively straightforward to integrate with the Bullfrog tools. It will also become important once a GUI
is added to be able to save a network description to permanent storage as a Bullfrog Network Description Language file; this has been anticipated and should also be straightforward without change to the data structures comprising the internal representation. Work will also probably be initiated at some point on a module for calculating a suitable topology to use for a particular application. Support for multicast, broadcast, and convergence (many-to-one) communications will no doubt be added to the programming model at some later date; the Bullfrog data structures currently support these types of communications, but the expansion, extraction, and parser modules do not know how to handle them. Adding support for these communication patterns should entail mainly adding functionality rather than changing existing functions.

Clearly, there is much work yet to be done. As the first coherent set of integrated tools for the NuMesh system is completed, however, the focus can shift from the task of defining and constructing the system to the task of extending and refining it—arguably a more rapid and less painful process in many ways. Bullfrog and the other NuMesh tools now in development will hopefully prove to be the components of that important milestone in the very near future.
Bibliography


