

1992

Object oriented simulation of systems with examples in structural design and parallel processing

David A. Bader
Lehigh University

Follow this and additional works at: <https://preserve.lehigh.edu/etd>



Part of the [Electrical and Computer Engineering Commons](#)

Recommended Citation

Bader, David A., "Object oriented simulation of systems with examples in structural design and parallel processing" (1992). *Theses and Dissertations*. 5533.
<https://preserve.lehigh.edu/etd/5533>

This Thesis is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.

OBJECT ORIENTED SIMULATION
OF SYSTEMS
WITH EXAMPLES IN
STRUCTURAL DESIGN
AND PARALLEL PROCESSING

by
David A. Bader

A Thesis
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of
Master of Science
in
Electrical Engineering

Lehigh University
1991

© Copyright 1992 by David A. Bader
All Rights Reserved

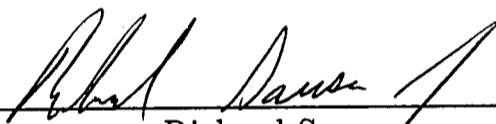
This thesis is accepted in partial fulfillment of the requirements for the degree of
Master of Science.

12 / 11 / 91

(Date)



Meghanad D. Wagh
Advisor in Charge



Richard Sause
Secondary Advisor in Charge



Kenneth K. Tzeng
CSEE Department Chairperson

Acknowledgments

I would like to thank my advisors, Dr. Meghanad D. Wagh, Associate Professor of Computer Science and Electrical Engineering, and Dr. Richard Sause, Assistant Professor of Civil Engineering, for encouraging me to pursue my academic and research interests. I would also like to thank the National Science Foundation (NSF) sponsored Center for Advanced Technologies of Large Structural Systems (ATLSS) at Lehigh University for supporting my graduate education and research project.

Contents

Acknowledgments	iv
Abstract	1
1 Introduction	2
1.1 Introduction to Problem	2
1.2 Motivation	3
1.3 Literature Survey on Object Oriented Simulation	3
1.4 Organization of thesis	4
2 Theoretical Foundations	5
2.1 Introduction	5
2.2 Notion of Abstract Modeling	5
2.3 Object Oriented Programming Concepts	7
2.3.1 an object	7
2.3.2 data abstraction	9
2.3.3 hierarchy	10
2.3.4 inheritance	11
2.3.5 client/supplier relationships	11
2.3.6 polymorphism	12
2.3.7 virtual functions	13
2.3.8 reuse of code	14
2.4 Creating the Object Oriented Process Model	15
2.5 Managing Collections of Objects	16

2.5.1	Overview of a Manager	16
2.5.2	Design of a Manager	17
2.5.3	Functions of Managers	19
2.5.4	Heterogeneous Collections	20
2.5.5	Polymorphism of Objects	20
2.5.6	Memory Allocation ⁸⁾	21
2.6	Problems with C++	21
2.6.1	Using Multiple Libraries	21
2.6.2	encapsulation of library objects	23
2.6.3	The need to know implementation of library objects	24
2.6.4	Type Casting	25
2.6.5	Copying Objects	26
2.6.6	Garbage Collection and Memory Management	27
2.6.7	Documentation	28
2.7	Interface between Simulation and External World	31
3	Project: FIDS	33
3.1	Overview	33
3.2	Modeled Objects	33
3.2.1	GridLine	35
3.2.2	GenericPoint	37
3.2.3	Compass	42
3.2.4	Load	43
3.2.5	LoadList	46
3.2.6	LoadEvent	48
3.2.7	ClearSpace	50
3.2.8	LOG (List of Objects on a GridLine)	52
3.2.9	Offset	54
3.2.10	RelativePosition	56
3.3	Managing Objects	58
3.3.1	Grid Holder	60

3.3.2	Grid Manager	62
3.3.3	LoadEvent Manager	65
3.3.4	Load Manager	67
3.3.5	ClearSpace Manager	69
3.3.6	LOG Holder	71
3.3.7	Offset Manager	73
3.3.8	RelativePosition Manager	75
3.3.9	Geometric Constraint Manager	77
3.3.10	Manager	79
3.4	Problem Formulation	84
3.4.1	Interface between Manager and User Interface	84
3.4.2	Stripped Problem Formulation Interface	85
4	Parallel Simulator	87
4.1	Overview	87
4.2	Introduction to Parallel Computation	87
4.3	ParSim Engine Description	89
4.3.1	Activities Menu	89
4.3.2	Connectivity Matrix	89
4.3.3	Connectivity Matrix File	90
4.3.4	A Sample Connectivity Matrix File	90
4.3.5	Topologies of Connectivity Networks	91
4.4	Algorithms used in ParSim	91
4.4.1	Runtime Creation of a Connectivity Matrix	91
4.4.2	Mapping Algorithm	92
4.4.3	Finding the Minimum Distance/Paths between two PE's	93
4.4.4	Mutual Partitions	93
4.5	General Execution Steps of Simulation	94
4.5.1	Example of Data Flow Simulation	94
4.6	Controller Object	100
4.7	Nodes	100

4.8	Links	101
4.9	Message	101
4.10	Architecture	102
4.11	Mapper	102
4.12	Conclusions	102
5	User Interface	103
5.1	Accessing the interface	103
5.2	Layout of workspace	104
5.3	Getting input from user	114
6	Conclusions	115
A	The NIH C++ Class Library Hierarchy	117
B	The NIH C++ Collections	120
C	NIH Template header file	124
D	NIH Template implementation file	126
E	Complete PF Header File (PF.h)	133
F	Stripped PF Header File (PF.g)	139
G	ParSim Definition Language Overview	144
	Bibliography	147
	Vita	151

List of Tables

4.1	ParSim Activity Menu	89
4.2	Algorithmically generatable topologies	91

List of Figures

2.1	Animal Class Hierarchy	8
2.2	Derived Class MyObject	9
2.3	Cat Class Member Variables	12
2.4	Example of Arrow and Related Class Hierarchy	13
3.1	The Manager Structure	59
3.2	Problem Formulation Interface	84
4.1	Connectivity Matrix	89
4.2	Linear Array with 4 PE's	91
4.3	Example Data Flow Network	95
5.1	FIDS-GUI File Menu	105
5.2	FIDS-GUI Activity/Tools Menu	106
5.3	FIDS-GUI GridLine Form	107
5.4	FIDS-GUI Duplicate Label Error Window	108
5.5	FIDS-GUI Relative Position Form	109
5.6	FIDS-GUI Offset Form	110
5.7	FIDS-GUI ClearSpace Form	111
5.8	FIDS-GUI Distributed Load Form	112
5.9	FIDS-GUI Point Load Form	113

Abstract

This thesis describes the design procedures for the constructing an object oriented simulation. The object oriented programming technique enables simulations of real world events to be modeled with ease. The gateway between the internal simulation and external user interface is described, and the appearance of the user interface for simulations is discussed. A prototype system for designing 2D steel structural frames given initial structural engineering constraints has been built and will be reviewed. Also, an example simulation of a parallel processing network will be given.

Chapter 1

Introduction

1.1 Introduction to Problem

A systematic approach for designing simulations using object oriented techniques encourages the use of abstract data packages for modeling real world occurrences. In this thesis, I will discuss the methodologies for creating a dynamic simulation package, and apply it to two examples: the simulation of large, steel structures for the construction industry, and the simulation of parallel processing networks for computer architects.

Powerful computer workstations are being developed at a rate faster than applications can be written for these new machines. I will attempt to shed some light on the software engineering tasks involved for creating large simulation packages for such workstations.

A new concept of **Object Oriented Design** is also integrated in this thesis. Object orientation implies that **Objects** in a model contain both data and functions for each instance of an object. Classic programming languages, as a reference, only provide for data structures and functions which can act on the data. Object oriented design has already proven useful in prototyping languages, such as SmallTalk, in which ideas are rapidly transformed into an implementation.

Object Oriented Libraries contain ready-made classes from which a software engineer can derive his object classes. By using abstract data hierarchies, inheritance,

CHAPTER 1. INTRODUCTION

polymorphism, and reuse of code, the application's engineer can quickly and powerfully model his problem using the expertise of the class library's software engineer.

Object Oriented Programming has been used extensively in industries to prototype user interfaces, but I will show that it is also applicable to the modeling of objects for simulation purposes.

1.2 Motivation

Recently, programming languages have been developed in which powerful functions are combined with object oriented capabilities, allowing a researcher to program in the problem domain and not the implementation of basic data structures. This thesis will describe how tools have been created, harnessing the high computational abilities of currently available workstations.

1.3 Literature Survey on Object Oriented Simulation

Various programming languages allow for object oriented constructions: **C++**, **Eiffel**, **SmallTalk**, Borland's **Turbo Pascal v6.0**, etc. In addition, there are general purpose object oriented libraries containing abstract data type hierarchies available for some of these platforms: Texas Instruments' **OATH** [20] and **COOL C++** [35] libraries, National Institutes of Health's **C++ Class Library** [14], ParcPlace Systems' **SmallTalk** class hierarchy [11], Borland's **Turbo C++ Libraries** [18], etc.

Other special purpose libraries have also been written: New York University's object oriented parallel programming language **ALLOY** [28], Andrew Grimshaw's **Mentat** [15] runtime system which allow for distributed parallelism and dataflow, and **ROSE**, an object oriented database management system.

Object oriented graphical libraries, such as Stanford's **InterViews** [37] and Cornell's **HOOPS** are also helpful.

CHAPTER 1. INTRODUCTION

1.4 Organization of thesis

This thesis will first cover the theoretical foundations in the use of object oriented programming concepts, and the organization of computer-based simulation. The Framework for Integrated Design System, (**FIDS**), a structural engineering simulation package written in C++ using a Sun Sparc Workstation, the NIH C++ Class Library, and Stanford's InterViews C++ User Interface Routines, is an example of one such simulator engineered by the author. **ParSim** [2], a concept for a parallel processing simulation of parallel interconnection networks, will also be discussed. Finally, I will present guidelines for constructing special purpose user interfaces for computer-based simulations.

Chapter 2

Theoretical Foundations

2.1 Introduction

This chapter introduces the reader to the theory behind object oriented design. First, the concepts needed to begin object oriented decomposition of a problem are discussed. Then the relationship between the mechanisms provided in an object oriented language and their benefit to the theoretical model of a simulation will be shown. Also, the necessary in-depth techniques for designing an object oriented simulation package from beginning criteria to a finished product are examined.

2.2 Notion of Abstract Modeling

Object Oriented Programming (OOP) relies on the principle that the behavioral and physical characteristics of a real-world entity can be conceptualized in an abstract model, called the **object**. The degree to which the **abstraction** models the entity will be directly proportional to the complexity of the simulation. If the problem to be simulated can be modeled very precisely, by including a lot of detail into the representation for each entity, the results will be far more accurate than a simple model could provide. The task of designing an object relies on the assumption that one knows the behavior of an entity under certain conditions, which is not always the case. Also, one might not wish to have such a fine grained simulation. The

CHAPTER 2. THEORETICAL FOUNDATIONS

abstraction **encapsulates** data structures and functions which describe the behavior and characteristics of any instance of the entity type involved in the simulation. By lumping the data and functionality together, we create a truer sense of an **object** than classical programming techniques, which separate coding into two distinct portions: data structures, and functions operating on these data structures.

In most simulation applications, many entities being simulated are "similar". Further, they may be grouped into layers of classes that may be similar themselves. For example, for a simulation of a parallel processing environment, one may have entities of type **ALU** (Arithmetic Logic Unit), **CL** (Control Logic), and memory. The ALU and CL together may form the entity called the **CPU** (Central Processing Unit), and the CPU and memory together may form each **PU** (Processing Unit). It is important to note that all the ALU's are similar as are all the CL's or memories. Further, all the clusters of these basic units, such as the CPU's or PU's are also similar to others of the same kind.

Object oriented programming successfully captures this idea by creating models which can have data and functionality for a particular species. In this scheme, objects belong to object **classes**, and these classes are arranged in a hierarchical structure. Each derived class can be **inherited** from a previously defined base class, and the new class provides a superset of the base class' functionality. If such a **class hierarchy**, or class library, is available, the software engineer need not spend his time recreating the wheel, per se, or creating the basic data structures for every application but spend his valuable time in the problem domain. **Reuse** of code, due to efficient hierarchical schemes, also reduces the number of lines of code an engineer must maintain, and in effect, drastically reduces the amount of time the engineer must spend in debugging his routines.

This thesis, although applicable to most object oriented languages, will be presented with specific references to **C++**, an object oriented programming language and superset of C.

2.3 Object Oriented Programming Concepts

2.3.1 an object

Amongst the new concepts of object oriented languages, the one that is of the most fundamental importance is the concept of object classes. Most programming languages provide the user with a selection of **fundamental data types** (FDT) from which all other types can be created. For instance, most languages have predefined types for integers, floating point reals and doubles, characters, strings, etc., which are standards of the language and portable to all hardware platforms for which the language is implemented. In most languages, a software engineer may define new types constructed as compounds of the FDT's. Records holding fields of FDT's are also standard forms for grouping data.

In object oriented languages, **abstract data types** (ADT), called **object classes**, organize both data structures and functions. An object class resembles a record, only in addition to holding data fields, the object class also holds functions to access and modify its data. Each class specifies its relation to existing classes by supplying a reference to the **base class**, or class from which it is derived¹. The object class is used similar to a conventional type, allowing instances of its class to be instantiated at the runtime of the program.

An object **class hierarchy** tree grows as new classes are added to the system. Since each new class is a superset of its base class, it **inherits** selected data and functions, meaning that the derived class, when given access by the inheritance rules, can call any of its base class' functions, or modify any of the data.

C++ is a **strongly typed** language, implying that at compile time, every literal in the code must know to which type it belongs. For example, the base class can be an **Animal** and it might have derived classes of **Mammal** and **Fish**, and **Mammal** has derived classes of **Dog**, **Cat**, and **Elephant**, and the **Elephant** might have a derived class for the **Gray Elephant**, as in Figure 2.1. Then at compile time, we must know what type of **Animal** an object is (it might just be an **Animal**). But at runtime, it

¹See Appendix C and Appendix D for Templates provided by the NIH Class Library for designing ADT's.

CHAPTER 2. THEORETICAL FOUNDATIONS

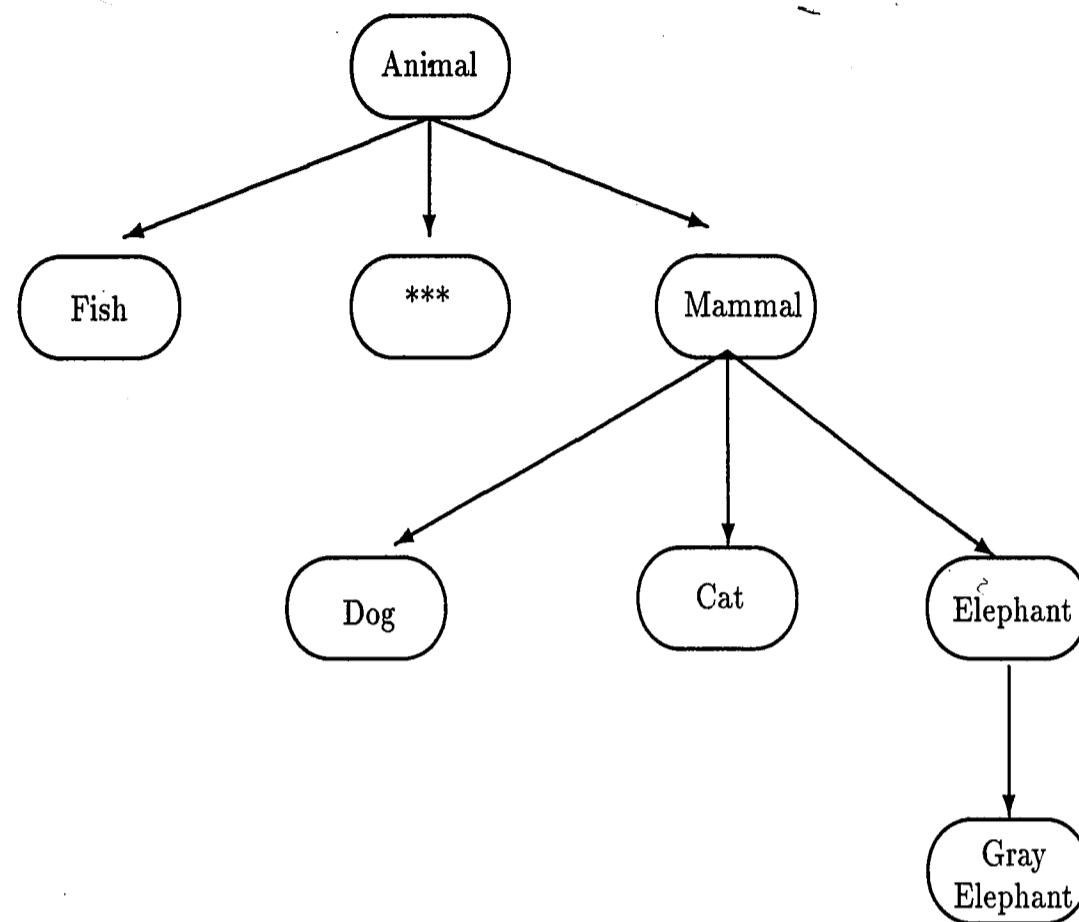


Figure 2.1: Animal Class Hierarchy

would be nice if we could ask an **Animal**: “What type are you?” and have it respond with “I’m a Cat!” or “I’m a Gray Elephant!” The root of the class hierarchy, (in this example, the **Animal**), must have those unique functions defined to answer *all* such questions for which objects need to respond. Also, functions which affect the entire class hierarchy may be implemented in this root class. An example of one type of such a function is an **overloaded operator** which supersedes the previously defined functionality for standard operators such as arithmetic, logical, conditional, indexing, pointer, and allocation operators. The operator can be given a new implementation when called for a specific class with varying arguments, and can be redefined at any point lower in the class hierarchy, too.

When designing an object found in a class library, a software engineer derives objects from a predefined class **Object**. Usually the engineer would like these objects to answer common messages which are not implemented in the library’s root **Object** class. In this case, the software engineer should make a derived class of **Object**, for example: **MyObject**, and derive all classes from **MyObject** that would normally be

CHAPTER 2. THEORETICAL FOUNDATIONS

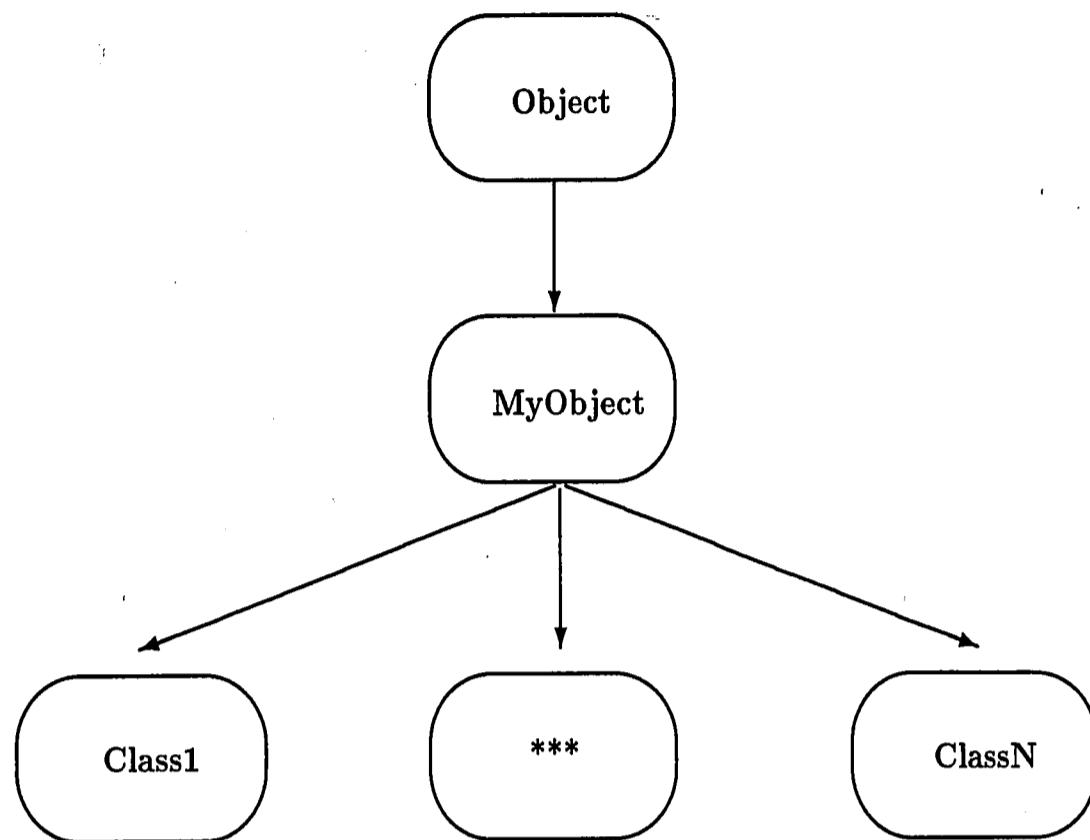


Figure 2.2: Derived Class MyObject

directly derived classes of Object, as shown in Figure 2.2.

2.3.2 data abstraction

Before any coding is done, a study must be performed on the entity to be modeled such that its properties are adequately contained in a class representation. The properties can be divided into two areas: **behavior** and **physical attributes**. Based on the behavioral characteristics, this work decomposes Objects into three main types:

Physical Objects - actual entities being simulated

Container Objects - dynamic holders for the physical objects

Process Objects - control and transformation processes on the physical objects

Various philosophies of programming exist for creating classes. In **Top Down** design, one creates all the classes for these objects, then fills in the functionality as needed. In **Bottom Up** design, the lowest level functions are written and then combined to create the classes. In this thesis, I propose that **Middle Out** design is

CHAPTER 2. THEORETICAL FOUNDATIONS

the only practical method by which a software engineer can create an object oriented system. Middle Out design is the combination of the two previous methods, whereby the designer fleshes out classes as he needs them, sometimes modifying other classes when a better design is discovered. When the designer attempts to think in terms of the implementation and begins to write the member functions, he becomes either more secure of his original design, or finds that the object class would be better suited with some other structure. As one proceeds in this fashion, the object classes constantly evolve.

2.3.3 hierarchy

The National Institutes of Health's Class Library² (NIH CL) provides a class **Object** which handles queries on an Object's class, name, initial assignments of all operators on the class, etc. so that the properties shared by all objects are represented by this class. All object classes are then created as derived classes of this Object class. By having a root base class, all instances of all object classes can be passed around the code as the type **Object**, eliminating the restriction of strong type checking for C++. Typecasting is a feature that allows a software engineer to explicitly change the apparent type of an object to another. This is very useful when the engineer wishes to handle objects similarly without knowing the explicit classes of the individual objects. At runtime, an object gets allocation of memory space sufficient to hold the class' member variables. The pointer to this memory space, however, may be of any type, masking the appearance of the object held. Typecasting is merely changing the class associated with the pointer without affecting the memory location. Typecasting is dangerous, because the engineer may change this pointer to a class other than a base class; when an access to the object is made using the pointer, an error will occur.

For example, referring back to Figure 2.1, one may instantiate a Gray Elephant, and pass the Gray Elephant to a routine written for any derived class of Animal. If this routine then wishes to call a function only available to Mammals, it must typecast the Animal object to a Mammal first. In order to classify the Animal object as a Mammal

²See Appendix A for the NIH Class Library hierarchy.

CHAPTER 2. THEORETICAL FOUNDATIONS

instead of a Fish, for instance, requires every Animal to know its derived class so that the software engineer can check this class and typecast the object accordingly. If the engineer mistakenly typecast this Gray Elephant, held as an Animal, as a Fish, the routine would fail at runtime.

2.3.4 inheritance

In decomposing an entity to be modeled, one must determine the data unique to the current object class, and concurrently, find similarities to the other existing object classes such that similar, or sibling, object classes can be generalized from a single base class containing all of the similarities. From experience in this procedure, similarities are found only after the first object class already has been modeled. Then a compromise is struck to create the object class that can be the abstract base class from which the siblings may be derived. An instance of a derived class is also an instance of its base class, holding all the instance variables and functionality associated with the base in addition to the newly derived extensions.

2.3.5 client/supplier relationships

An object may contain other objects in any combination of the following methods: inheritance and client/supplier relationships. As explained previously, inheritance creates a derived class that is a superset of an existing object class. An instance of a derived class holds an object from the base class to access the properties and behavior of that base class. The second method is via client/supplier relationships, whereby an object holds several **member variables** that are objects from unrelated object classes. These member variables are completely contained inside this modeled object.

When a class is designed, one must determine whether the relationship between the class and existing classes is of the inheritance or a client/supplier type. Keith Gorlen [14] explains that when a class is inherited from another class, like the Cat from the Animal class, it answers the question, “Is a?” as in “The Cat **Is** an Animal?” In client/supplier, the question, “Has a?” is answered. For example, a class is created, see Figure 2.3, to hold the Cat’s name, call it the **Name** class, then “The Cat **Has**

CHAPTER 2. THEORETICAL FOUNDATIONS

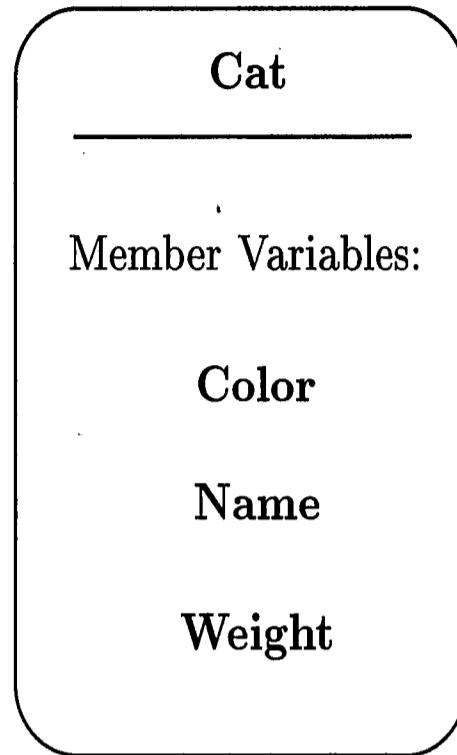


Figure 2.3: Cat Class Member Variables

a Name?"

In certain cases, the determination between these two methods is not as clear, such as an example with geometric shapes. If one creates a class for **Circles**, and wishes to create a class for **Arcs**, “Is an Arc a Circle?” or “Is a Circle an Arc?” Also, “An Arc **Has** a Circle?” and “A Circle **Has** an Arc?” Clearly, these questions are all ambiguous, and the task of choosing the relationship is then left up to the better judgment of the designer.

2.3.6 polymorphism

One advantage of object oriented programming is that objects can be treated as **polymorphisms**, meaning any object may be referenced or passed by the type of any object class above it in the inheritance tree, or, identically, by its actual type. For instance, referring back to the animal class hierarchy given in Figure 2.1, a **Gray Elephant** can be passed as a **Gray Elephant**, an **Elephant**, or an **Animal**, and a **Dog** can be passed as an **Animal**, too. By allowing polymorphism, objects truly can be handled without concern for its actual class. Polymorphism can simplify the problem of handling the return types of functions which can return objects of more

CHAPTER 2. THEORETICAL FOUNDATIONS

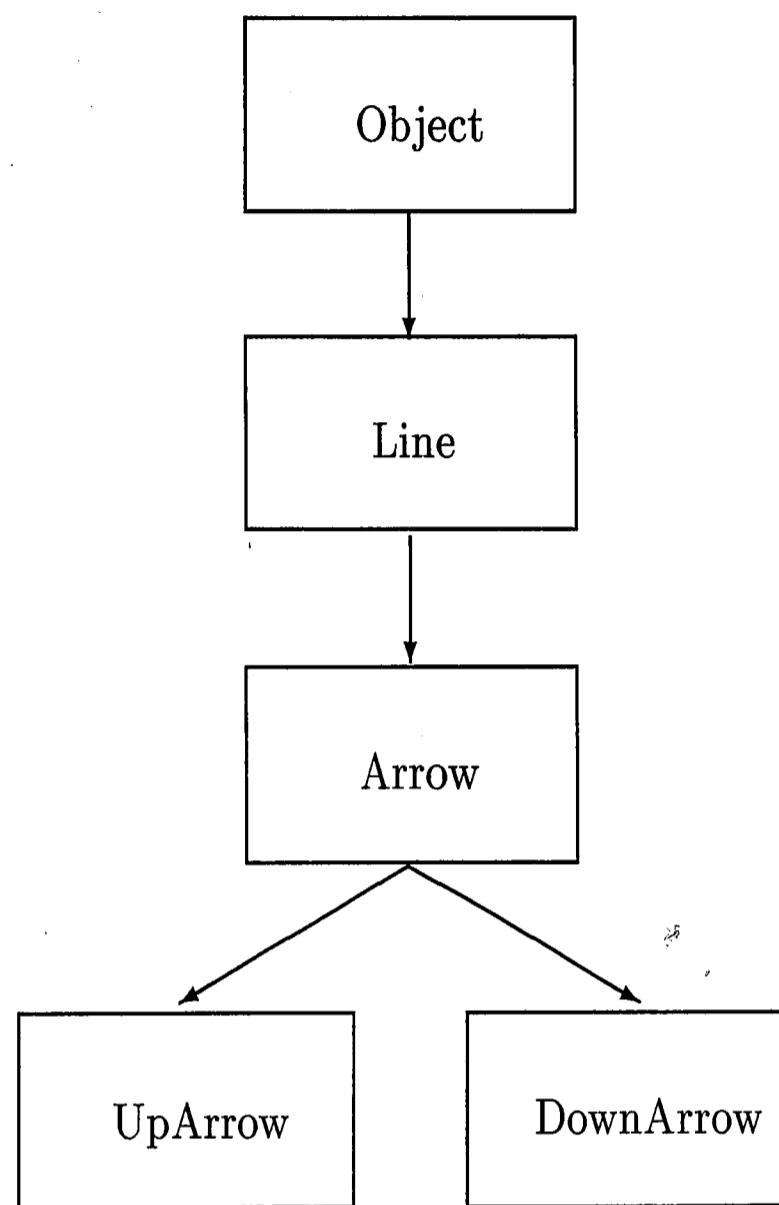


Figure 2.4: Example of Arrow and Related Class Hierarchy

than one type. Every function has a return type, which can be either a fundamental data type, or an abstract, or user-defined, data type. Also, this return type may require allocated space for the entire structure, or merely a pointer to the correct object. Following the NIH Class Library paradigm, the return types of most functions that return other library objects are pointers to the class **Object**. The burden of correctly transforming the pointer to the proper object class (in order for that object's methods to be used) is then left to the software engineer.

2.3.7 virtual functions

CHAPTER 2. THEORETICAL FOUNDATIONS

Object oriented languages allow derived classes to re-implement a function provided by a base class. If the function is not re-implemented, the base class implementation is invoked. **Virtual functions** allow derived classes to re-implement a function, such that when the base class' function is invoked, the derived functionality is called.

Pure virtual functions are virtual functions prototyped in a base class, but no implementation is given. An **abstract base class** is any class with at least one pure virtual function, since this class may not be instantiated.

When handling objects polymorphically, it is readily apparent that most functions in **Object** need to be pure virtual since the implementation would have no meaning in Object and one needs to call the correct derived class' implementation. For example, in a graphical package, an **Object** might have a function to draw itself on the screen, but the function in class Object would be a pure virtual function, meaning that it is available for all Objects, but no implementation exists for the abstract Object. The pure virtual functions must be implemented by any derived class that is not planned to be an abstract base class. For example, as shown in Figure 2.4, a derived class of Object called **Line** would re-implement draw() for a line. If **Arrow** were a derived class of Line, Arrow's draw() function could call the Line's draw() function, and then call its own routine to draw the arrowhead. If Arrow had two derived classes, **UpArrow** and **DownArrow**, and Arrow included directional information in its member variables and used this in its draw(). function, neither UpArrow nor DownArrow would need to re-implement draw(). Virtual functions are implemented only when the characteristics of the function will change in a derived class; otherwise, no modifications are necessary.

2.3.8 reuse of code

One of the major advantages of object oriented programming is that by using inheritance, client/supplier relationships, class hierarchies, virtual functions, and other object oriented techniques, the code duplication can be largely avoided. Routines to handle algorithms can be programmed once, thoroughly tested and debugged, and placed in libraries. A designer then has the task to choose which objects he will build

CHAPTER 2. THEORETICAL FOUNDATIONS

with, but no longer has the burden of recreating the basic data structures and algorithms. **Reuse** of code implies that less code needs to be compiled, and that existing code will have a much shorter debugging period. A software engineer can then spend time debugging only conceptual errors in his problem domain, and need not worry about the accuracy of memory management, data structures, searching and sorting algorithms, etc. In the NIH Class Library, sophisticated dynamic arrays are available which allow users to create instances of objects “on the fly”, place them in collections, and have the collection automatically grow larger if need be. In the same context, graphical object oriented libraries also are available. The reuse of code promotes the concept that libraries of objects will become available for all types of applications, and the designer of a system no longer needs to “reinvent the wheel.”

2.4 Creating the Object Oriented Process Model

The features of objected oriented languages allow one to readily model an entity into objects by decomposing the entity’s behavior into suitable classes. The next task for the object oriented designer is the decomposition of a simulation problem into an abstract process model. The process model is the environment in which the entities interact with each other. Some examples of this include the simulation of a parallel computer, in which the participating entities are Nodes, Links, and Messages, and the process model is the interconnection network of the Nodes and Links. In the simulation of a flight simulator, the entities include the aircrafts, the Earth, the Sun, the control tower, etc. and the process model is the atmosphere, wind, light, weather, etc. In a simulation of a structure, the entities are the beams and columns, and the process model is the space surrounding these members, along with the physical phenomena associated with the simulation, such as forces from wind, gravity, dead loads, etc.

In the object oriented paradigm, the process model is merely another object to be abstracted. The state variables of the process are the instances of the modeled entities, and the functions are the means by which these objects interact. The behavior of the environment then must be abstracted into this process object. One may now conceive

CHAPTER 2. THEORETICAL FOUNDATIONS

of the simulation process as merely another entity, which when initialized properly, and told to "GO!" will perform the simulation to completion.

2.5 Managing Collections of Objects

2.5.1 Overview of a Manager

The abstraction of the problem requires (1) modeling of the objects and specifying their interactions; and (2) describing the mechanisms through which the collections of such objects can be created, modified, queried, or deleted. The first of these two issues were discussed in the two preceding sections. The second issue is the topic of the current section.

In the simulation problem, collections of objects participate by interacting with each other. If one can create a *smart* way to manage these objects such that the interaction becomes simpler to implement, one has created the ideal framework for simulation.

The **Manager** concept may not appear to be object oriented in design. However, one can consider the Manager to be the state variable of the entire simulation, that is, as just another object in the simulation. Often, several Manager classes are needed. One Manager class should have the ability to interface with both the external simulation representation, and the internal simulation representations, namely, other Managers. In this sense, a Manager is an Object Oriented Database (OODB). If one decomposes the problem down further, each object of each class can be held in a specialized Manager. The sub-Managers are then equivalent to **tables** in a relational database. Managers must then follow extended rules of normal forms for relational databases. By following these rules, actions presented to the Manager will never corrupt the data.

CHAPTER 2. THEORETICAL FOUNDATIONS

2.5.2 Design of a Manager

Managers must be clever. When objects are entered into the Manager, consistency should be maintained with the previously held objects. If the new object is inconsistent, incomplete, or a duplicate, it must be discarded.

All modeled entities are required to own a unique **key**, such as a string, used to access the object from both internal and external objects. Other entities may use this key as a passcard to the manager for retrieving the object associated with that particular key.

Changes to objects held in Managers can be performed by passing the key and the updated version of the object. If the key has not changed, all objects holding the reference to that object still have a persistent method for access to it.

An object held in the Manager can be removed by handing its Manager the key for that selected object. Deletions, however, require that objects holding a reference to the soon-to-be-removed object must either be removed as well, or have their references re-attached to persisting objects. Because two way links do not exist between the keyed object and all others holding its key, maintaining persistent data is not trivial in this case. A brute force approach for propagating the deletion may be implemented by searching every entity in the representation of the problem and querying it as to whether or not this delete will require subsequent modifications. Other schemes such as **dependency lists** and **broadcasts** can also be used, but are not within the scope of this thesis.

One must design a Manager around two strategic areas, the internal structure of the Manager and the external accesses to the Manager. In order to select a base class for an efficient design of the Manager, the objects which it will hold must be examined. The following are the types of questions which must be answered:

- Are the objects homogeneous or heterogeneous?
- If they are heterogeneous, what is the most explicit class to which all the objects belong?
- Are the objects sortable? (i.e. can an object compare itself with another and

CHAPTER 2. THEORETICAL FOUNDATIONS

determine which would come first in a sort?)

- Do the objects need to be retrieved in any particular order?
- What will the most common access to an object look like?
- What kind of searches need to be performed to locate an object in the collection?
- Which existing collection do the objects most readily, and naturally fit into?
(e.g. Set, Bag, Stack, OrderedCollection, SortedCollection, Array, LinkedList, HashTable, Dictionary, KeySortedCollection, etc.)

By answering these questions, the determination of the Manager's base class will be an easy decision³. The chosen container class must be the inherent natural choice for holding the specified objects. If more explicit accesses are needed which cannot be done optimally on this type of collection, a second auxiliary collection, used as a lookup table for these special accesses, may be held as a member variable of the Manager.

The second focus, namely access to these objects through the Manager, is reviewed at this point. If a container holds sortable objects, and the Manager is chosen to be a sorted collection, one has a problem when one tries to access the collection with the key to an object (in the SortedCollection, the key is the object itself.) In this case, as the paragraph above notes, the Manager may hold an instance of a KeySortedCollection to manage associations of keys and references to the objects held by the inherent collection. This extra member collection is just another view to accessing the data (of objects) in the Manager. When messages are passed to this Manager, it has the choice of whether to use the base container class directly, or to use the auxiliary collection as a front-end for accesses to the base container class. In either case, all of the held collections must be updated whenever an object owned by this Manager is modified.

³See Appendix B for the NIH C++ Library class structure for Collections, as well as short descriptions of the available collections.

CHAPTER 2. THEORETICAL FOUNDATIONS

2.5.3 Functions of Managers

The Manager has various responsibilities, some of which include:

- maintaining persistent data objects
- adhering to insertion, deletion, and modification consistencies
- providing adequate access messages for searches
- printing, storing, and debugging collection onto an output stream or file
- retrieving contents from an input stream or file
- iterating through the collection to access each object held

The Manager's class description included in its header file should contain the messages needed to perform all these functions on the collection. By using **const** functions, the software engineer also knows which of these methods will modify the collection.

To the user of a collection, three main types of functions exist:

- construction (and destruction)
- access
- modification

The construction (destruction) functions handle the initialization (finalization) of an instance of this Collection class. The accessing methods allow for searches through the collection which do not modify the objects contained. The modification messages are of the **store**, **remove**, and **change** type. These modifications must perform the various function implied or return a token representing the error that might have occurred. If the return token is not **OK**, one should assume that the collection has rejected the operation asked of it, and has left the collection in the same state as before the message came along. Some other error messages, for example, might be: **InvalidLabel**, **DuplicateLabel**, **InvalidObject**, etc. The calling routine then may branch to resolve a specific error condition. At this stage, the user may be queried, or a routine may find the resolution to the error.

CHAPTER 2. THEORETICAL FOUNDATIONS

2.5.4 Heterogeneous Collections

A Manager will function optimally when its base collection holds homogeneous objects. Because of static type checking, if one knows that a collection holds instances of the **Widget** class, one can directly typecast an element in the container to be a widget and query the widget with public widget class member functions. By taking advantage of the compare function, the widget will know how to order itself with respect to other instances of the same class. Thus, a sorted collection needs to hold homogeneous objects. All entities are, in fact, homogeneous to the **Object** base class. But by referring to instances as **Objects**, the public member functions belonging to derived classes of this base class become inaccessible.

Object oriented programming does allow one to hold instances of different classes in the same collection. For example, a collection such as a **Stack** might hold an **Integer** in the first location, a **String** in the second, and an **Elephant** in the third. As long as the objects held are derived from the base class (**Object**, for example), specified by the definition of the collection class, polymorphism allows all these heterogeneous instances to be held as **Objects**.

2.5.5 Polymorphism of Objects

Polymorphism is the key concept to holding heterogeneous objects. If, for example, the **Widget** class is derived from a base class, say **Object**, a particular instance of the **Widget** class may be handled as though it were an instance of the base class **Object**. One drawback of this is that only the **Object** class functions may be accessed for an object once it has been handed to a collection for **Objects**. Other functions for the **Widget** class are not accessible until the **Widget** instance is typecast to its actual class.

The base class **Object** should have the necessary functionality available so that (at runtime) one can determine the class to which the instance belongs, and then cast it to that particular class. A member variable of the base class should hold the class name to which the instance belongs. An **Object** then has the ability to be cast to this class, or have functions from this class invoked. Using a pure virtual function in

CHAPTER 2. THEORETICAL FOUNDATIONS

the base class and requiring derived classes to implement an equivalence operator for that derived class, is one way to handle the testing of a polymorphic object when one does not know to which class that instance belongs.

2.5.6 Memory Allocation

Creation or modification of objects in a collection cannot be achieved without due consideration of its impact on the memory requirements. In classical programming techniques, the size of an array must be determined statically at compile time, forcing a known limit on the number of objects an array can hold. This limit coexists with the limits of free memory allocation to variables and data structures invoked by the program at runtime. A software engineer must place knowledgeable upper bounds on every array used in the scope of the program. However, these are limits to the overall performance of the program.

With object oriented memory allocations, collections are instantiated at an optimal size, but have the ability to **grow** or **shrink** as necessary to accommodate the objects held. The expansion and contraction of the container is handled invisibly to the user. The design engineer can dictate at compile time the behavior of the collection, for example, whether it will grow by doubling in size, or just adding sixteen more element positions. All the error checking and dynamic allocation techniques are written for this class, and thus, the engineer no longer has this issue to worry about or implement.

2.6 Problems with C++

2.6.1 Using Multiple Libraries

The stated advantage of object oriented systems, design, and programming is that one can work in the problem domain, borrowing existing objects from already complete and streamlined libraries. In practice, linking objects from separate libraries is extremely complicated, and further research must take place in this area. Specifically, compiling a program that needs objects from two distinct libraries has a chance of

CHAPTER 2. THEORETICAL FOUNDATIONS

running into conflicts between the libraries, such as definitions of what **NIL** is, or class definitions, for example, **Object**, **Point**, **String**, or **Set**.

Since every object in a particular program should be polymorphic to a single class, such as **Object** in the NIH Class Library, having different root objects for each library conflicts with the theoretical notion of one root object. This problem is evident when designing a front-end user interface for the internal objects of a simulation. Every entity needs an internal representation which models its **behavior** in the simulation, and an external representation which models its **physical** display properties for the screen.

Another technical problem encountered when using both the NIH Class Library [13] and InterViews Library [23] is that classes in one can be overwritten by a new class definition in the other, with no warning messages produced at compile time. For example, a common class called **Point** is utilized by both of these libraries. One dangerous solution to this type of problem would be for a software engineer to modify the source code to the library, making each class name distinct, such as changing **Point** to **nihclPoint**, etc. This is not a practical solution in cases where the library does not include modifiable source code, or the engineer does not have the capabilities of recreating the library. Modifying a library can also affect other packages previously written for the library.

Similarly, the concept of **NIL** is vastly different in available libraries. In NIH Class Library, **NIL** is an **Object**, whereas in InterViews, **NIL** is defined to be **0**. Globally renaming one of these literals is a solution to this problem, however, all the complications associated with the last problem are relevant here, too.

A non-object oriented solution is to maintain a distinction between objects from each library, and not to compile two objects from different libraries together in the same module of code. This solution is costly, though, in that it requires objects to interface between libraries, which pass data between objects derived from the different libraries. Once again, this solution has dangerous side-effects. A **header file** contains the prototypes of classes and the class functions. In the compilation of a module of code, one object may handle a second by merely knowing the prototype and not the implementation of that second class. The compiler merely leaves enough

CHAPTER 2. THEORETICAL FOUNDATIONS

memory space to include the second object when the modules are linked together. Inconsistency problems may occur due to the fact that the interface objects must have two separate header files, one for each side of the interface, and any differences in the size allocations of classes represented in these header files will have serious repercussions (via bad pointers) during execution. The interface module will have the responsibility of transferring objects to and from fundamental data types, such that each header file for each side of the application need not know about any objects from the other library. In the implementation of this module, however, objects can be passed between the two libraries. An example of this interface will be given in a later chapter.

2.6.2 encapsulation of library objects

A challenge arises for the library designer to create objects which are fully **encapsulated**, that is, the objects are modular and can be replaced easily with differing designs. In theory, a software engineer only needs to know the external functionality of an object, and all internal functions are both hidden and secure from him. The engineer need not know or understand the implementation of a library object, and is usually discouraged from even viewing it.

Although encapsulation is sought in libraries, an engineer has the ability to re-implement functions without realizing that the encapsulated object has been corrupted. Almost all public member functions in the NIH Class Library are implemented as **virtual**, encouraging the use of polymorphism. Virtual functions are necessary if we wish to have derived classes properly respond to a function call when the object is held as an instance of the base class. For example, the new implementation of the function may execute a few new lines of code, and then call the base class' function of the same name. The problem with re-implementing these functions is that an illegal action in a derived function may cause other base class functions which depend on the derived function to behave improperly, (e.g. in this manner, the library objects can never be completed and securely encapsulated.)

CHAPTER 2. THEORETICAL FOUNDATIONS

2.6.3 The need to know implementation of library objects

With encapsulation comes the concept that a software engineer need not know the implementation of a library object, or any object for that matter, since the interface should relay all the relevant information for accessing and modifying an object.

I disagree that there can ever be this external and limited view of objects in object oriented programming. In my experience with the NIH Class Library, one constantly needs to examine the implementation of library objects, checking for such things as whether or not a member function modifies its object, how memory management is attained, formatting for when the object is placed on the output stream, garbage collection, etc. These answers can be documented in a manual, but as one knows, the only reliable check is by tracing through the implementation.

As an example, if a software engineer uses a Container to hold objects, and the engineer places objects in the Container:

- Are the objects in the Container identical, or copies, or the placed objects?
- When an object is retrieved from the Container, will modifications to the object or Container affect each other?
- Does losing scope on the Container or an object lose scope of the objects held?
- When the destructor is called on the Container, does it destroy all the objects it holds?

These and other similar questions are applicable to most objects.

Another issue with library objects is implementational bugs which make function calls non-standard. For instance, one would believe that checking to see if a key is included in a keyed collection should always return a boolean. However, the NIH Collections need an `isEmpty()` function call first to test if any keys are, in fact, held, and then, if true, check for the inclusion of a specific key.

C++ allows the software engineer to specify functions that are `const` (constant). A const function has the property that none of the member variables of the object are modified when that particular function is called. Accessing methods are normally

CHAPTER 2. THEORETICAL FOUNDATIONS

constant. Constant functions are verified at compile time, and constant functions may only call other constant functions inside the object. This programming notation can give an engineer the false sense that a function is safe to execute, but many loopholes exist which allow the engineer the latitude to corrupt the object by accident. For instance, the NIH sortable objects have a constant function **compare** which returns a -1, 0, or +1 depending on if the second object passed to the first occurs to the left, at, or to the right of the current object. In one of my implementations, I mistakenly had the compare access the Collection in which the object sat, corrupting a collection pointer. When the collection was accessed a second time, the pointer no longer held the correct information. Although the constant function is an easy notation to represent functions meant only to access information, the concept gives the engineer a false sense of security over data integrity.

2.6.4 Type Casting

C++ is a strong typed language. The compiler must know the types of all literals at the compile time. Unfortunately, this is a major drawback to pure object oriented languages in which type resolutions occur dynamically at run time. C++ requires strong type checking because the C++ code is merely transformed into C code, such that the standard C compiler and object linker can be used. The C linker requires that all function calls are explicitly tied to the matching function at compile time. **Late binding**, as SmallTalk implements, binds the function invocations to their routines dynamically at runtime. The only late binding in C++ occurs when virtual functions are called, in order to allow an object to invoke the proper derived class implementation at runtime.

As a result of strong typing, the linker must know the class of each object calling a function, so that the tie can be established. Some variety exists on the way a function is called. For example, **Circle** is a derived class of **Shape**. If a Circle is held as a Shape, and the Shape is issued the function to compute its area, the Shape will call its own area function. However, if the Shape is coerced back into being a Circle, and the same function call is made, the object will respond accordingly.

When working with a library, one cannot insert virtual functions in the provided

CHAPTER 2. THEORETICAL FOUNDATIONS

base classes without recompiling the library. Without the virtual function provided for in the base class, the polymorphic object cannot call the intended function without first being typecast to the derived class which contains the desired function. The solution would be to make a new abstract class, derived directly from the existing root class, and in the new class, implement all the problem-related functionality. For example, if the root class is **Object**, create a derived class called **MyObject** from which all other entities can be modeled [25, p.393].

This solution only allows functionality added for Objects descended directly from **Object**. When an object class is derived from any other class, it no longer contains the **MyObject** class information. A creative solution would be to use **Multiple Inheritance**, such that the modeled entity would be derived from whichever existing library class, and also from **MyObject**. Objects constructed via multiple inheritance then can access functions from any of the superclasses, provided the compiler can resolve the typing scheme. Explicit typecasting will tell the compiler that an object with pointer “A” might really be of class “B”, but if the software engineer is incorrect, the program will not function in the desired manner.

2.6.5 Copying Objects

The assignment operator = is probably the most used operator, and if one does not understand its functionality, objects in memory can be corrupted. The assignment operator is used to copy an object, and there are two types of copies:

shallowCopy - a pointer copy

deepCopy - a contents copy

A **shallowCopy** merely copies a pointer to an area of memory holding the object, so that a second pointer has access to the same memory location and functionality. This access is useful when a knowledgeable software engineer wants to pass around objects, without any modifications, because the **shallowCopy** can be performed in constant time, with only one pointer of new memory space required. Also, the pointer can be passed to a function if one wishes to have the function modify the object. By passing

CHAPTER 2. THEORETICAL FOUNDATIONS

only a pointer, the entire object need not be put on the stack, i.e., on the memory space provided for passing arguments to functions.

A deepCopy allocates a block of memory identical in size to the original object, and proceeds to hierarchically decompose the object, and reconstruct a copy of it in the new location. A deep copy must be aware of circular pointers, so that the copy does not wind up in an endless loop. For example, if object **B** is held as a member variable of object **A**, and object **B** points back to object **A**, then a deep copy should not get caught in that circularity. Once a deep copy is made, modification to either copy will not affect each other. This copying technique is useful when an object needs to be passed to a function, and one has no knowledge of the function, but does not want the object modified.

Unless methods are provided, the assignment operator normally performs a shallow copy. In C++, by implementing following special constructor:

$$X :: X(const X&);$$

this function will be called when the assignment operator is used for class **X**. This constructor allows the software engineer to define a unique deep copy for class **X**.

2.6.6 Garbage Collection and Memory Management

Garbage collection and memory management are handled differently in objected oriented languages. Memory management refers to allocation and deallocation of memory space when objects are constructed and destructed, respectively. Garbage collection is the process by where objects are removed from memory when they are no longer needed.

Garbage collection may be implicit, as in SmallTalk, or explicit, as in C++. Theoretically, an object can be removed from memory space when no other object needs a reference to it. SmallTalk holds a list of pointers to objects, and periodically, or when space is needed, the system checks the list against objects in memory. Whenever an object exists in memory space without a pointer to it stored in the list, the memory space is relinquished to the free memory store. SmallTalk automatically provides this

CHAPTER 2. THEORETICAL FOUNDATIONS

service, however, a price must be paid in performance since the service needs to be processed without knowledge of the current state of objects in the system.

Explicit garbage collection is handled in C++ but having a **destructor** function called either directly in a function call by the software engineer, or inserted by the compiler when an object loses scope. The destructor function allows the engineer to explicitly free memory that was allocated by the constructor for that object. The engineer must maintain the memory management by remembering to implement the requisite destruction routine.

2.6.7 Documentation

Documentation is the key to the usability of object oriented libraries. Although a library might contain all the necessary objects and functionality, the library is worthless without adequate instructions on the use of these objects.

The following are means by which a library architect can document his library:

- User/Reference Manuals
- Example packages
- Header files
- Documented (Implementational) code

This list is given in increasing closeness to the actual code of the library. In theoretical terms, the software engineer should never need to read the library's code; the **Reference Manual** should include all the external functionality of the library's objects. For streamlining and debugging of applicational functions, it may become necessary to peek into the library for hints and answers to the problem at hand. This philosophy, in one sense, violates the notion of information hiding in libraries, but adheres to the idea that object oriented programming need not duplicate earlier programming efforts. Re-use of code can only be attained when the engineer has access to all the code implementation, from all the applicational routines and algorithms, to the lowest levels of the libraries.

CHAPTER 2. THEORETICAL FOUNDATIONS

A *good* reference manual should contain the following information on every object in the library:

- Object Class
- Base Class
- Derived Classes
- Related Classes
- Constructors and Destructors
- Public member variables and functions
- Behavioral properties of instances of the object class
- Any related information which is necessary when using instances of the object class
- Examples of instantiating and using the object from the designer's point of view.

Most C++ Libraries currently available do not have coherent documentation. Errors also occur in their examples. The **InterViews** package from Stanford [23] is an example of a very rich library of objects with poor documentation.

Some libraries have amazing reference manuals, examples, explanations, documented code, etc. As an example of this style, I refer to Keith Gorlen's release 3.10 of the NIH C++ Class Library reference manual (Draft Copy) [13], along with his book on Object Oriented Design [14].

Browsing is the key to learning object oriented libraries. Browsing consists of a presentation of classes in a fashion such that the engineer can search around for objects, messages, or concepts, copy ideas, learn the algorithms used, and gain a familiarity with the libraries. **SmallTalk** has a browser incorporated into its system [11], and the quicker one can master the Browser, the faster software development

CHAPTER 2. THEORETICAL FOUNDATIONS

becomes. The reference manual should be a hardcopy of the information one might want to know when browsing.

The information presented leads a software engineer to choose which objects one wants to design with. Questions as to inheritance, related structures, public messages provided, and behavior are quickly answered so that design can progress as fast as possible, with the right design decisions from the start. If one chooses the *correct* objects to begin with, one does not need to waste time redesigning objects created from inefficient structures.

At times, the software engineer wishes to know more about the library objects. Also, if the library has an ineffective reference manual, or even worse, does not include a reference manual, one must know how to browse the actual code. The reference manual should be a *readable* presentation of the collection of header files to the objects. By careful review of the header files, the same information can be extracted. These files contain the object class definitions, more technically providing the class hierarchies, member variables, public messages, etc. Commenting in the files can also help in this situation.

The ultimate source for browsing is the implementation of the object functions. Although reading someone else's source code might be difficult, depending on the library designer's programming styles and the reader's mastery of the language, this will provide examples that use the objects provided, and give insights into algorithms available.

If example programs using the libraries are available, one should refer to these to solve specific implementational problems, or even try to master the examples to gain expertise with the library. Rewriting an example will also allow a user to better understand the steps needed in the process of turning one's own code into an executable package.

As an example, reference pages for objects designed for use with Project: FIDS are provided in Chapter 3.

2.7 Interface between Simulation and External World

An object oriented simulation contains both the internal objects to perform the simulation and the external user interface. Every action taken by the user must affect the simulation as needed, and conversely, the simulation must be correctly portrayed in the user's view⁴. The interface between these two domains must have this functionality, namely, to process the internal-to-external and external-to-internal handshaking of messages, events, and objects.

The object oriented abstraction of the simulation holds the current state of the participating entities. The external user interface may be *any* modular package which allows one to view, or observe, this simulation. For example, the external portion may be a powerful X-Windows graphical user interface for a Sun Workstation or a simple ASCII text dialog with the user. The interface must provide a protocol for accessing and modifying the information held by the simulation.

One feature of this interface layer is that the designer can create a text-only debugging front-end for the simulation, before a more in-depth graphical representation is realized. Also, any graphical user interface is dependent on the computer platform currently being used, and although the code for the internal simulation is portable, the graphical routines usually are not. In this case, all the graphics are encapsulated in a section of the project solely responsible for display and disconnected from the abstracted entities. When the project is ported to a different platform, or when the graphical capabilities are upgraded on the current platform, the task of creating a new front-end will not affect the already-working internal objects.

The interface module links the internal and external representations. Because of this property, the internal simulation modules need not be compiled with any knowledge of the external world, and vice versa. Programming tricks may be necessary, however, to compile this interface. Modules in C++ include an implementation file as well as a file containing the class structures and prototype of the available functions. As long as no implementations are given in the header file, one may have several

⁴Please refer to Chapter 5 for a discussion of the user interface.

CHAPTER 2. THEORETICAL FOUNDATIONS

header files for the interface module. The complete header file with all functionality may be included when compiling the internal objects; whereas a stripped down version containing only the fundamental C++ data types and structures will be used when compiling the front-end portion of the project. This solution also eliminates any possible class name conflicts between the internal and external libraries by decoupling their compilations.

Chapter 3

Project: FIDS

3.1 Overview

The following chapter presents the reader with an example in object oriented design. The project chosen, simulating the Framework for Integrated Design System (FIDS), has the task of representing a two-dimensional structural frame, allowing user imposed constraints on locations for members, and satisfying all given load events. The decomposition of entities participating in this simulation will be discussed, focusing on their behavioral characteristics abstracted and encapsulated into object oriented models. The physical objects will be presented in the same order as designed, from the lowest level conceptual objects needed as base classes and dependent abstract member variables, to the objects representing the formal constraints. Next, the Container objects for holding these physical objects will be demonstrated. And finally, the Process Objects used as glue to mesh the various objects, internally and externally, along with the interface to the extern representation, will be presented.

3.2 Modeled Objects

The following sections contain the technical descriptions of the entities modeled for this project. These descriptions are presented as reference manual pages, for ease of use of this simulation library. The pages contain important design specifications

CHAPTER 3. PROJECT: FIDS

along with corresponding solutions to the problems encountered when abstracting the models. Each reference page contains the name of the object, base class, derived classes, related classes, descriptions, constructors (and destructors), public access functions, and constant objects of that type, such as NIL objects. The description section elaborates the design issues for the entity, along with the state variables needed to represent the behavioral state of the object.

CHAPTER 3. PROJECT: FIDS

3.2.1 GridLine

BASE CLASS

Object

DERIVED CLASS

None

RELATED CLASSES

GridHolder, GenericPoint

DESCRIPTION

Grid lines form a two dimensional mesh of reference lines where beams and columns are most likely to be placed.

Typically, **A** and **B** represent vertical grid lines, and **1, 2**, horizontal grid lines.

Note that the structure to be analyzed is defined to begin at grid lines A and 1, but extended grid lines below and to the left must exist as anchors for the perimeter of the structure. The same is true for the right-most vertical and top-most horizontal grid lines.

These grids lines are the most probable locations for beams and columns, but do not ensure that a beam or column will actually fall in these locations.

All objects in a structure are geometrically located via grid lines. Therefore, the grid line is the most primitive entity in this simulation model.

A GridLine has a label, orientation, and distance from the left- or bottom- most grid line (for vertical and horizontal, respectively). By changing the spacing between

CHAPTER 3. PROJECT: FIDS

grid lines, the geometry of the resulting structure will be modified accordingly. Areas in the grid are all scaled by these distances between grid lines.

A fixed absolute outside perimeter for a structure must be created. Then grid lines may be constructed inside the perimeter. A grid line may be moved, as long as it does not cross over another grid line. New grid lines may be added at any location in bounds. Grid lines may not be removed if any other objects are tied to the grid line by its label reference.

CONSTRUCTORS and DESTRUCTORS

```
GridLine(const String& label, const OrientType& orient,  
        const Float& dis)
```

```
GridLine(char* label, char* orient, double dis)
```

```
GridLine(char* label, OrientType orient, double dis)
```

PUBLIC ACCESS

```
String& label() const
```

```
OrientType& orient() const
```

```
Float& distance() const
```

```
bool isNIL() const
```

```
bool isBoundingGridLine() const
```

CONSTANT OBJECTS

```
GRIDLINENIL = GridLine("NIL", none, 0) /
```

```
BOTTOMGRIDLINE = GridLine("BOTTOM", horiz, 0)
```

```
LEFTGRIDLINE = GridLine("LEFT", vert, 0)
```

```
TOPGRIDLINE = GridLine("TOP", horiz, BIGNUM)
```

```
RIGHTGRIDLINE = GridLine("RIGHT", vert, BIGNUM)
```

CHAPTER 3. PROJECT: FIDS

3.2.2 GenericPoint

BASE CLASS

Object

DERIVED CLASS

ClearSpace

RELATED CLASSES

Load

DESCRIPTION

A GenericPoint specifies the location of an entity in space relative to bounding grid lines. By redundancy in the state variables of a generic point, four different types of points can be held: Intersection, Vertical Line, Horizontal Line, and Rectangular Area. The point also has the ability to be calculated in fixed measurements from the surrounding grid lines, or a ratio of the distance between the same grid lines. The following are definitions of the GenericPoint's member variables:

GenericPoint(gridA, gridB, grid1, grid2, x_start, y_start, x_stop, y_stop,
x_relative, y_relative)

gridA and gridB are the labels to the bounding vertical grid lines, and
grid1 and grid2 are the labels to the bounding horizontal grid lines

x_relative: TRUE \Rightarrow x_start and x_stop are relative measurements

$(0 \leq x_{start}, x_{stop} \leq 1)$ of the distance between grid lines A and B.

FALSE \Rightarrow x_start and x_stop are fixed distance offsets.

CHAPTER 3. PROJECT: FIDS

y_relative: TRUE \Rightarrow y_start and y_stop are relative measurements ($0 \leq y_{start}, y_{stop} \leq 1$) of the distance between grid lines 1 and 2.
FALSE \Rightarrow y_start and y_stop are fixed distance offsets.

x_start: a horizontal offset of a point from the given grid crossing in either a relative measurement (if x_relative \Rightarrow TRUE) or a fixed distance (if x_relative \Rightarrow FALSE).
This is a horizontal starting offset.

y_start: a vertical offset of a point from the given grid crossing in either a relative measurement (if y_relative \Rightarrow TRUE) or a fixed distance (if y_relative \Rightarrow FALSE).
This is a vertical starting offset.

x_stop: a horizontal offset of a point from the given grid crossing in either a relative measurement (if x_relative \Rightarrow TRUE) or a fixed distance (if x_relative \Rightarrow FALSE).
This is a horizontal stopping offset.

y_stop: a vertical offset of a point from the given grid crossing in either a relative measurement (if y_relative \Rightarrow TRUE) or a fixed distance (if y_relative \Rightarrow FALSE).
This is a vertical stopping offset.

The Four classes of points:

INTERSECTION POINT

- associated with a connection of two objects, or
a point in space for another object or force)
- Two grid lines given (A, 1) (grid lines 1 = 2, A = B)
- (x_{start}, y_{start}) is the offset from the grid crossing

CHAPTER 3. PROJECT: FIDS

- `x_start` must equal `x_stop`
- `y_start` must equal `y_stop`
- `getPoint` returns coordinate of intersection

HORIZONTAL LINE

- associated with a beam or girder object
- Three grid lines given (A, B, 1) (grid lines 1 = 2)
- (`x_start`, `y_start`) is the offset from the grid crossing
- `y_start` must equal `y_stop`
- For a complete horizontal grid line segment between vertical grid lines,
`x_start`, `y_start`, `y_stop` = 0
`x_stop` = 1, and
`x_relative` = TRUE
- `getOrigin` returns left coordinate
- `getCorner` returns right coordinate

VERTICAL LINE

- associated with a column object
- Three grid lines given (A, 1, 2) (grid lines A = B)
- (`x_start`, `y_start`) is the offset from the grid crossing
- `x_start` must equal `x_stop`
- For a complete vertical grid line segment between horizontal grid lines,
`x_start`, `x_stop`, `y_start` = 0
`y_stop` = 1, and
`y_relative` = TRUE

CHAPTER 3. PROJECT: FIDS

- `getOrigin` returns bottom coordinate
- `getCorner` returns top coordinate

RECTANGULAR AREA

- associated with a wall-type object
- Four distinct grid lines given (A, B, 1, 2)
- Inside rectangle given, `x_start`, `x_stop` represent left/right bounds
`y_start`, `y_stop` represent bottom/top bounds
- `getOrigin` and `getCorner` messages find absolute coordinates for an area

CONSTRUCTORS and DESTRUCTORS

```
GenericPoint(const String& gridA, const String& gridB,  
            const String& grid1, const String& grid2,  
            Float& x_start, Float& x_stop, Float& y_start, Float& y_stop,  
            bool x_relative, bool y_relative)
```

PUBLIC ACCESS

```
String& gridA() const  
String& gridB() const  
String& grid1() const  
String& grid2() const  
Float& x_start() const  
Float& x_stop() const  
Float& y_start() const  
Float& y_stop() const  
bool x_rel() const  
bool y_rel() const
```

CHAPTER 3. PROJECT: FIDS

```
bool actsOnGrid(const String& gridLabel) const  
void getPoint(Float& x, Float& y) const  
void getOrigin(Float& x0, Float& y0) const  
void getCorner(Float& x1, Float& y1) const  
  
PointType type() const  
bool good() const  
bool bad() const  
bool isNIL() const
```

CONSTANT OBJECTS

GENERICPOINTNIL = GenericPoint("NIL", "NIL", "NIL", "NIL")

CHAPTER 3. PROJECT: FIDS

3.2.3 Compass

BASE CLASS

Float

DERIVED CLASS

None

RELATED CLASSES

Load

DESCRIPTION

A Compass is a direction holder. A Compass may be constructed via an enumerated list of directions (for right_dir, up_dir, left_dir, down_dir), or with the enumerator “other_dir” and a double precision floating point value for the angle. Compass uses a right handed system with right_dir equal to 0° and up_dir equal to 90° .

CONSTRUCTORS and DESTRUCTORS

Compass(CompassType, double v)

PUBLIC ACCESS

CompassType dir() const

bool isNIL() const

CONSTANT OBJECTS

COMPASSNIL = Compass(other_dir, 0)

CHAPTER 3. PROJECT: FIDS

3.2.4 Load

BASE CLASS

Object

DERIVED CLASS

None

RELATED CLASSES

LoadList, LoadEvent, LoadManager

DESCRIPTION

All loads can be represented internally as:

Load(Name, NominalLoadCategory, Location, Mag1, Mag2, Direction,
SupportSurface)

Constant Loads

- Location \Leftarrow Intersection GenericPoint
 - load acting on a point
- Location \Leftarrow Line GenericPoint
 - constant load distributed on line
 - (must have Mag1 = Mag2)

Distributed Loads

- Location \Leftarrow Line GenericPoint
 - varying load distributed on line
 - (must have Mag1 \neq Mag2)

CHAPTER 3. PROJECT: FIDS

NominalLoadCategory: represents a type for the load, for example: loads due to members in the structure, wind, external loading, etc. A list of standard tags and user defined options exist.

Location: A GenericPoint which is either an Intersection or Line point

Intersection ↔ Point Load

Line Point ↔ Load along a surface

Mag1: Magnitude of force at left or bottom of LinePoint (or at Intersection)

Mag2: Magnitude of force at right or top of LinePoint

Direction: Indicates a Compass direction of force

SupportSurface: Indicates the surface of a member along a gridline to which the load acts on. This is an enumerated list of tags such as top_sur, bottom_sur, left_sur, right_sur, and centerline_sur

CONSTRUCTORS and DESTRUCTORS

Distributed Load:

```
Load(String& name, NominalLoadType& nominal_load_category,  
     GenericPoint& line_point, Float& magnitude1, Float& magnitude2,  
     Compass& dir, SurfaceType& support_surface)
```

Point Load:

```
Load(String& name, NominalLoadType& nominal_load_category,  
     GenericPoint& intersection_point, Float& magnitude,  
     Compass& dir, SurfaceType& support_surface)
```

PUBLIC ACCESS

String& name() **const**

NominalLoadType& nominalLoadCategory() **const**

CHAPTER 3. PROJECT: FIDS

GenericPoint& linePoint() const
Float& mag1() const
Float& mag2() const
Compass& direction() const
SurfaceType& supportSurface() const

loadType type() const
bool isNIL() const

CONSTANT OBJECTS

LOADNIL = GridLine("NIL", D, GENERICPOINTNIL, 0, 0,
COMPASSNIL, top.sur)

CHAPTER 3. PROJECT: FIDS

3.2.5 LoadList

BASE CLASS

Dictionary

DERIVED CLASS

None

RELATED CLASSES

LoadEvent

DESCRIPTION

A LoadList is a collection of related loads from a single entity in a structural design. The loads are accessible by the load name. A load list has a unique label used as a reference when placed in a LoadEvent.

LoadLists are constructed with only a load list label, but then may have loads added, removed, or changed inside the collection.

CONSTRUCTORS and DESTRUCTORS

LoadList(String& label)

PUBLIC ACCESS

String& label() **const**

bool holds(String& label) **const**

ManagerErrors addLoad(Load& aLoad)

ManagerErrors removeLoadWithLabel(String& label)

CHAPTER 3. PROJECT: FIDS

```
ManagerErrors changeLoad(String& oldLabel, Load& newLoad)
int numberOfLoads() const
Load& getFirstLoad()
Load& getLoadAfter(String& label)
bool isNIL() const
```

CONSTANT OBJECTS

```
LOADLISTNIL = LoadList("NIL")
```

CHAPTER 3. PROJECT: FIDS

3.2.6 LoadEvent

BASE CLASS

Dictionary

DERIVED CLASS

None

RELATED CLASSES

LoadEventManager

DESCRIPTION

A LoadEvent is a collection of LoadLists and other LoadEvents. The LoadEvent typically contains all the loads due to a certain physical phenomena. A LoadEvent may only be held by one other LoadEvent, and has a reference to this parent. If the LoadEvent is not held, the parent reference points to NIL.

The LoadEvent is comparable to a lookup table with keys and values. Its entries are in no particular order, so a Dictionary has been chosen as the base class. A LoadEvent is constructed on a key which names that LoadEvent. When adding a LoadList to the LoadEvent, an Association between the LoadList label and the actual LoadList object are entered into the Dictionary. When a second LoadEvent is added to a first LoadEvent, the second LoadEvent gets its **parent** label set to the first, and only a reference to the child is maintained in the parent LoadEvent.

A LoadEvent may add, remove, or change LoadLists and LoadEvents which it contains.

CONSTRUCTORS and DESTRUCTORS

CHAPTER 3. PROJECT: FIDS

LoadEvent(String& label)

PUBLIC ACCESS

String& label() **const**

String& parent() **const**

void setParent(String& parentLabel)

LoadManager& allLoads() **const**

ManagerErrors addLoadEvent(String& newLoadEventLabel)

ManagerErrors addLoadList(LoadList& newLoadList)

ManagerErrors removeLoadListWithLabel(String& label)

ManagerErrors removeLoadWithLabel(String& loadListLabel, String& loadLabel)

ManagerErrors changeLoad(String& loadLabel, Load& newLoad)

int numberOfLoads() **const**

int numberofLoadLists() **const**

int partialNumberofLoads() **const**

int partialNumberofLoadLists() **const**

bool holdsLoadWithLabel(String& loadLabel) **const**

bool holdsLoadListWithLabel(String& loadListLabel) **const**

bool holdsLoadEvent(String& loadEventLabel) **const**

LoadList& getLoadListWithLabel(String& loadListLabel)

LoadList& getFirstLoadList()

LoadList& getLoadListAfter(String& loadListLabel)

bool isNIL() **const**

CONSTANT OBJECTS

LOADEVENTNIL = LoadEvent("NIL")

CHAPTER 3. PROJECT: FIDS

3.2.7 ClearSpace

BASE CLASS

GenericPoint

DERIVED CLASS

None

RELATED CLASSES

ClearSpaceManager

DESCRIPTION

ClearSpace(Label, ClearSpaceTag, AreaPoint)

A ClearSpace is used to prevent walls and diagonal members from being placed at the given AreaPoint location. The Label is a unique string used to access the entity.

The ClearSpaceTag represents the cause of the ClearSpace, such as a lobby, modern office, elevator shaft, etc. These tags are kept in a global set from which a ClearSpace can select its type, or create a new type.

init() will clear all the clear space tags.

CONSTRUCTORS and DESTRUCTORS

```
ClearSpace(const String& label, const String& clearSpaceTag,  
          const String& gridA, const String& gridB,  
          const String& grid1, const String& grid2,  
          Float& x_start, Float& x_stop, Float& y_start, Float& y_stop,  
          bool x_relative, bool y_relative)
```

```
ClearSpace(const String& label, const String& clearSpaceTag,  
          const GenericPoint& anAreaPoint)
```

CHAPTER 3. PROJECT: FIDS

PUBLIC ACCESS

String& label() const

String& tag() const

Set& getTags() const

static void init()

void clear()

void addTag(const String&)

bool isNIL() const

CONSTANT OBJECTS

CLEARSPACENIL = ClearSpace(“NIL”, “NIL”, GENERICPOINTNIL)

CHAPTER 3. PROJECT: FIDS

3.2.8 LOG (List of Objects on a GridLine)

BASE CLASS

SortedCltn

DERIVED CLASS

None

RELATED CLASSES

LOGHolder

DESCRIPTION

LOGs are used to hold a List of Objects along a Grid line. The LOG has a unique label equivalent to that of the label of its grid line. The LOG holds sortable objects in its inherent sorted collection, but also maintains a lookup table, via a member variable which is a key sorted collection, for accesses by key.

The LOG is a sortable object itself, and its compare() function orders by the distance of the related grid line.

Objects can be stored or removed from the LOG by means of a unique key associated with the given object. This key, a String, is utilized as the lookup key.

CONSTRUCTORS and DESTRUCTORS

LOG(String& label)

PUBLIC ACCESS

CHAPTER 3. PROJECT: FIDS

String& label() const

ManagerErrors storeWithKey(String& key, Object& valueObject)

ManagerErrors removeObjectWithKey(String& key)

Object& getObjectWithKey(String& key)

Object& getObjectAfter(String& key)

bool holds(String& key) const

bool isNIL() const

CONSTANT OBJECTS

LOGNIL = LOG("NIL")

{~}

CHAPTER 3. PROJECT: FIDS

3.2.9 Offset

BASE CLASS

Object

DERIVED CLASS

None

RELATED CLASSES

OffsetManager

DESCRIPTION

Offset(key, gridLabel, bound_1, bound_2, Value, offset_surface)

An Offset contains a structure's offset along the grid line with label "gridLabel" bounded by the grid lines with labels "bound_1" and "bound_2". The Value of this offset is a Float.

The key is a unique string used to reference this offset.

The variable offset_surface holds a SurfaceType enumerated type.

top_sur, bottom_sur, and centerline_sur are used for horizontal members,
left_sur, right_sur, and centerline_sur are used for vertical members.

Offsets may check to see if they overlap with another, because overlapping offsets are generally disallowed.

CONSTRUCTORS and DESTRUCTORS

Offset(String& key, String& gridLabel, String& bound_1, String& bound_2,
Float& value, SurfaceType& offset_surface)

CHAPTER 3. PROJECT: FIDS

PUBLIC ACCESS

String& key() **const**
String& grid() **const**
String& bound1() **const**
String& bound2() **const**
Float& value() **const**
SurfaceType& offsetSurface() **const**

bool overlaps(Offset& anotherOffset) **const**
bool isNIL() **const**

CONSTANT OBJECTS

OFFSETNIL = Offset()

3.2.10 RelativePosition

BASE CLASS

Object

DERIVED CLASS

None

RELATED CLASSES

RelPosManager

DESCRIPTION

A relative position has the following structure:

```
RelPos(key, grid_1, grid_2, extent_Start, extent_Stop,  
       grid_1_surface, grid_2_surface, min_max_mode,  
       less_than, greater_than, equal_to)
```

A relative position is used to specify headrooms and widths in the structure. The key is a unique String used to access a particular RelPos. The RelPos specifically applies to the members which will be attached at locations along grid_1 and grid_2. The extent_Start and extent_Stop are the labels of the grid lines bounding the relative position constraint. The variables grid_1_surface and grid_2_surface reference the surfaces of the members along grid_1 and grid_2, respectively, for which the constraint applies.

The min_max_mode is used to toggle between using “<” and “>” constraint values or using an explicit “=” value. Both data variables are given, but the useful information is determined by the state of this mode indicator. In this scheme, one then has minimum and maximum position constraints, as well as fixed position requirements.

CHAPTER 3. PROJECT: FIDS

CONSTRUCTORS and DESTRUCTORS

```
RelPos(const String& key,  
        const String& grid_1, const String& grid_2,  
        const String& extentStart, const String& extentStop,  
        SurfaceType& grid_1_surface, SurfaceType& grid_2_surface,  
        bool min_max_mode,  
        Float& less_than, Float& greater_than,  
        Float& equal_to)
```

PUBLIC ACCESS

```
String& key() const  
String& grid1() const  
String& grid2() const  
String& extentStart() const  
String& extentStop() const  
SurfaceType& grid1Sur() const  
SurfaceType& grid2Sur() const  
bool minmax() const  
Float& lessThan() const  
Float& greaterThan() const  
Float& equalTo() const  
  
GenericPoint& rect() const  
bool good() const bool isNIL() const
```

CONSTANT OBJECTS

```
RELPOSNIL = RelPos("NIL", "NIL", "NIL",  
                    "NIL", "NIL", top_sur, top_sur)
```

3.3 Managing Objects

The following sections contain the reference pages for the manager, or container, objects in the FIDS. Although these objects seem independent, they are the variables needed for representing the state of a generic structural frame. The **Manager** class is the one object encompassing all the behavioral characteristic of a single frame problem. From the outside, this class will manage all accesses to the problem involving the entities such as GridLines, Loads, LoadLists, LoadEvents, Offsets, ClearSpaces, RelativePositions, etc. A diagram representing the theoretical structure of the manager is provided in Figure 3.1.

The simulation currently treats the sub-managers for the entity objects, such as the GridManager, LoadEventManager, and GeometricConstraintManager, as clients of the Manager, but a more precise model of this manager would be an object multiply inherited from these sub-managers. The Manager then would inherit the functionality of each sub-manager. By doing so, the Manager would implicitly accept calls to the sub-managers without having to explicitly pass each function call to the correct one. This method does give full public access from the sub-managers to the Manager's interface.

CHAPTER 3. PROJECT: FIDS

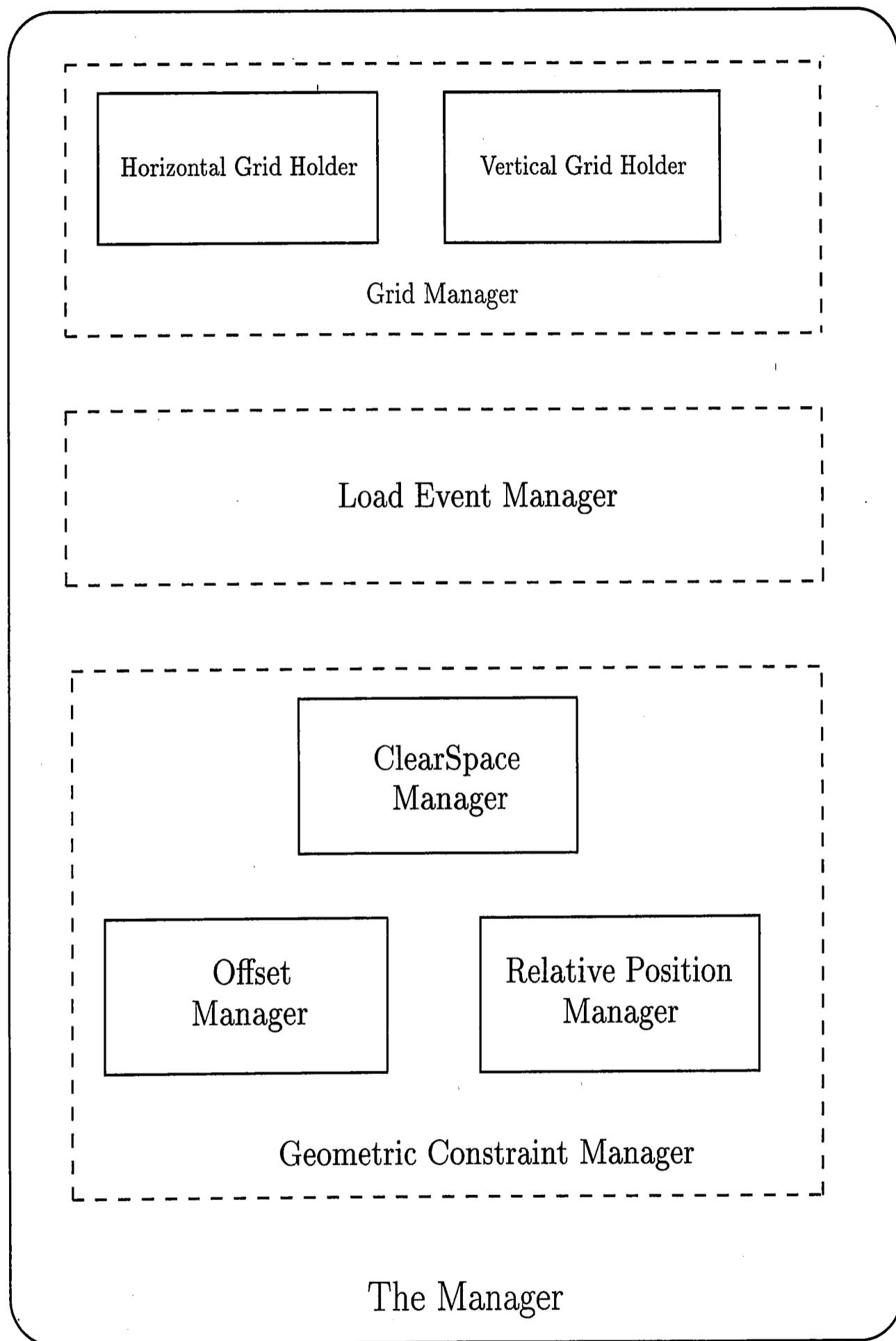


Figure 3.1: The Manager Structure

CHAPTER 3. PROJECT: FIDS

3.3.1 Grid Holder

BASE CLASS

SortedCltn

DERIVED CLASS

None

RELATED CLASSES

GridLine, GridManager

DESCRIPTION

A GridHolder has the task of holding homogeneous GridLines. A GridLine is a sortable object and uses its distance from either the bottom- or left- most grid line, (for horizontal and vertical grid lines, respectively), as the ordering criteria. All GridLines in a GridHolder, then, must have the same orientation, either *horiz* or *vert*.

The GridHolder accesses its GridLines by the GridLine **label**. GridLines may be stored, removed, or changed. Methods also exist for accessing the first, last, current, previous, and next GridLines. In order to accomplish these functions, the GridHolder maintains a pointer to the last accessed GridLine. Be aware that **side effects** from other objects might affect this position state.

Note that the `getFirst()`, `getLast()`, `getPrevious()`, and `getNext()` functions are not **const** functions because they modify this position pointer.

CONSTRUCTORS and DESTRUCTORS

`GridHolder()`

CHAPTER 3. PROJECT: FIDS

PUBLIC ACCESS

```
ManagerErrors removeAllGrids()  
ManagerErrors storeGridLine(GridLine& aGridLine)  
ManagerErrors removeGridLine(String& label)  
ManagerErrors changeGridLine(String& oldLabel, GridLine& newGridLine)  
  
bool holds(String& label) const  
int numberOfGridLines() const  
  
GridLine& getFirst()  
GridLine& getCurrent() const  
GridLine& getNext()  
GridLine& getLast()  
GridLine& getPrevious()  
GridLine& getGridLineWithLabel(String& label)  
GridLine& getGridLineWithDistance(const Float& distance)  
GridLine& getGridLineBefore(const Float& distance)  
GridLine& getGridLineAfter(const Float& distance)  
Float& findDistanceBetween(String& label_1, String& label_2)  
GridHolder& getGridLinesBetween(String& label_1, String& label_2) const
```

CONSTANT OBJECTS

None

CHAPTER 3. PROJECT: FIDS

3.3.2 Grid Manager

BASE CLASS

Object

DERIVED CLASS

None

RELATED CLASSES

GridHolder

DESCRIPTION

The GridManager holds two GridHolders, one for the horizontal grid lines and one for the vertical grid lines. When grid lines are handed to and from this GridManager, it must determine the orientation of the grid line and correctly place it in a GridHolder, or retrieve from the right collection.

Before any GridLines are placed in the GridManager, the message `init()` must be called to set up NIL objects and place in the GridHolders the four bounding grid lines: BOTTOMGRIDLINE, TOPGRIDLINE, LEFTGRIDLINE, and RIGHTGRIDLINE. The distance of the BOTTOMGRIDLINE and LEFTGRIDLINE are 0, and the TOPGRIDLINE and RIGHTGRIDLINE and a “large” distance, representing ∞ . The methods `setOrigin(x0, y0)` and `setCorner(x1, y1)` should be called before any GridLines are placed in the GridManager. These methods modify the the distances of the bounding GridLines to reflect the scale chosen by the user.

The GridManager also allows access to the public member functions for the Horizontal GridHolder and Vertical GridHolder.

CONSTRUCTORS and DESTRUCTORS

CHAPTER 3. PROJECT: FIDS

GridManager()

PUBLIC ACCESS

ManagerErrors init()

ManagerErrors clear()

ManagerErrors storeGridLine(GridLine& aGridLine)

ManagerErrors removeGridLineWithLabel(String& label)

ManagerErrors changeGridLine(String& oldLabel, GridLine& newGridLine)

bool bothHold(String& label) **const**

bool eitherHold(String& label) **const**

int numberOfHoriz() **const**

int numberOfVert() **const**

bool setOrigin(Float& x0, Float& y0)

bool setCorner(Float& x1, Float& y1)

GridLine& getFirstHoriz()

GridLine& getFirstVert()

GridLine& getLastHoriz()

GridLine& getLastVert()

GridLine& getCurrentHoriz() **const**

GridLine& getCurrentVert() **const**

GridLine& getNextHoriz()

GridLine& getNextVert()

GridLine& getPreviousHoriz()

GridLine& getPreviousVert()

GridLine& getGridLineWithLabel(String& label)

GridLine& getHorizGridLineWithDistance(**const** Float& y)

GridLine& getVertGridLineWithDistance(**const** Float& x)

GridLine& getHorizGridLineBefore(**const** Float& y)

GridLine& getVertGridLineBefore(**const** Float& x)

CHAPTER 3. PROJECT: FIDS

GridLine& getHorizGridLineAfter(**const** Float& y)
GridLine& getVertGridLineAfter(**const** Float& x)
Float& findDistanceBetween(String& label_1, String& label_2)
GridHolder& getGridLinesBetween(String& label_1, String& label_2) **const**

CONSTANT OBJECTS

None

CHAPTER 3. PROJECT: FIDS

3.3.3 LoadEvent Manager

BASE CLASS

Dictionary

DERIVED CLASS

None

RELATED CLASSES

Load, LoadList, LoadEvent, LoadManager

DESCRIPTION

The LoadEventManager holds a collection of LoadEvents. Since LoadEvents have no indigenous order, each is stored in the LoadEventManager as an Association, with the LoadEvent label used as the key and the actual object as the value. The LoadEvents are then accessible via their label. LoadLists and Loads may also be accessed and modified. LoadEvents may be added, removed, or changed inside this LoadEventManager.

Before any LoadEvents are placed in the LoadEventManager, the message `init()` must be called to set up NIL objects.

CONSTRUCTORS and DESTRUCTORS

`LoadEventManager()`

PUBLIC ACCESS

CHAPTER 3. PROJECT: FIDS

```
ManagerErrors init()
ManagerErrors clear()
ManagerErrors removeAllLoadEvents()
ManagerErrors addLoadEvent(LoadEvent& aLoadEvent)
ManagerErrors removeLoadEventWithLabel(String& label)
ManagerErrors changeLoadEvent(String& oldLabel, LoadEvent& newLoadEvent)
ManagerErrors changeLoad(String& loadLabel, Load& newLoad)

bool holds(String& label) const
bool holdsLoadWithLabel(String& label) const
bool holdsLoadListWithLabel(String& label) const
int numberOfLoadEvents() const
int numberOfLoads() const
int numberOfLoadLists() const

LoadEvent& getLoadEventWithLabel(String& label) const
LoadEventManager& getLoadEventsOnPoint(const GenericPoint& aGP) const
LoadEventManager& getLoadEventsOnGridLine(String& label) const
LoadManager& allLoads()
Load& getLoadWithLabel(String& loadLabel)
```

CONSTANT OBJECTS

None

CHAPTER 3. PROJECT: FIDS

3.3.4 Load Manager

BASE CLASS

Dictionary

DERIVED CLASS

None

RELATED CLASSES

Load

DESCRIPTION

The LoadManager holds a collection of Loads. Since Loads are have no specific ordering, the LoadManager can hold them in a Dictionary, with the Load label as the key, and the Load object as the value. Loads are accessed via their label and may be added, removed, or changed inside this collection.

A LoadManager has no relation to a LoadList or LoadEvent, since the LoadManager holds a *flat* collection of Loads, (unlike the LoadEventManager, which is comprised of a hierarchical tree-like description of loads in Loads in LoadLists and LoadEvents.)

Before any Loads are placed in the LoadManager, the message `init()` must be called to set up NIL objects.

CONSTRUCTORS and DESTRUCTORS

`LoadManager()`

PUBLIC ACCESS

CHAPTER 3. PROJECT: FIDS

```
ManagerErrors init()
ManagerErrors clear()
ManagerErrors removeAllLoads()
ManagerErrors storeLoad(Load& aLoad)
ManagerErrors removeLoadWithLabel(String& label)
ManagerErrors changeLoad(String& oldLabel, Load& newLoad)

bool holds(String& label) const
int numberOfLoads() const

Load& getLoadWithLabel(String& label) const
LoadManager& getLoadsOnPoint(const GenericPoint& aGP) const
LoadManager& getLoadsOnGridLine(String& gridLabel) const
LoadManager& getLoadsWithNominalCategory(NominalLoadType&
                                         nominalLoadCategory) const
virtual Collection& addContentsTo(Collection& cltn) const
```

CONSTANT OBJECTS

None

CHAPTER 3. PROJECT: FIDS

3.3.5 ClearSpace Manager

BASE CLASS

Dictionary

DERIVED CLASS

None

RELATED CLASSES

ClearSpace, GeometricManager

DESCRIPTION

The ClearSpaceManager holds a collection of ClearSpaces. ClearSpaces are accessed via their label and may be added, removed, or changed inside this collection.

Before any ClearSpaces are placed in the ClearSpaceManager, the message `init()` must be called to set up NIL objects.

CONSTRUCTORS and DESTRUCTORS

`ClearSpaceManager()`

PUBLIC ACCESS

`ManagerErrors init()`

`ManagerErrors clear()`

`ManagerErrors store(Object& aClearSpace)`

`ManagerErrors storeClearSpace(ClearSpace& aClearSpace)`

`ManagerErrors removeWithLabel(String& label)`

`ManagerErrors changeClearSpace(String& oldLabel, ClearSpace& newClearSpace)`

CHAPTER 3. PROJECT: FIDS

```
bool holds(String& label) const  
int numberOfClearSpaces() const  
  
ClearSpace& getClearSpaceWithLabel(String& label) const  
ClearSpace& getFirstClearSpace()  
ClearSpace& getClearSpaceAfter(String& lastLabel)
```

CONSTANT OBJECTS

None

CHAPTER 3. PROJECT: FIDS

3.3.6 LOG Holder

BASE CLASS

SortedCltn

DERIVED CLASS

OffsetManager

RELATED CLASSES

LOG

DESCRIPTION

The LOGHolder holds a collection of LOGs, (List of Objects along a Grid line). LOGs are accessed via their label (which is the label of the grid line that the LOG attaches to) and may be added, removed, or changed inside this collection.

Before any LOGs are placed in the LOGHolder, the message `init()` must be called to set up NIL objects.

The messages `holds()` and `numberOfLOGS()` refer to the LOG label and collection of LOGs, respectively, while `holdsObjectKey(...)` and `numberOfObjects()` refer to the objects held inside of all the LOGs which are held by this LOGHolder.

CONSTRUCTORS and DESTRUCTORS

`LOGHolder()`

PUBLIC ACCESS

CHAPTER 3. PROJECT: FIDS

```
ManagerErrors init()
ManagerErrors clear()
ManagerErrors storeLOG(LOG& aLOG)
ManagerErrors removeWithLabel(String& labelOfLOG)

bool holds(String& labelOfLOG) const
bool holdsObjectKey(String& objectKey) const
int numberOfLOGs() const
int numberOfObjects() const

LOG& getLOGWithLabel(String& labelOfLOG) const
LOG& getFirstLOG()
LOG& getLOGAfter(String& lastLOGLabel)
```

CONSTANT OBJECTS

None

CHAPTER 3. PROJECT: FIDS

3.3.7 Offset Manager

BASE CLASS

LOGHolder

DERIVED CLASS

None

RELATED CLASSES

GeometricManager

DESCRIPTION

The OffsetManager holds a collection of Offsets which are in turn held by LOGs. Note that the OffsetManager is a LOGHolder, and a lot of functionality for managing Offsets is provided for in the LOGHolder. Offsets may be added, removed, or changed inside this collection via their key.

Before any Offsets are placed in the OffsetManager, the message **init()** (found in LOGHolder) must be called to set up NIL objects.

The messages **holds()** and **numberOfLOGS()** refer to the LOG label and collection of LOGs, respectively, while **holdsObjectKey(...)** and **numberOfObjects()** refer to the offsets held inside of all the LOGs which are held by this OffsetManager.

Note that the functions given for the OffsetManager class supplement the functions already available in the LOGHolder class.

CONSTRUCTORS and DESTRUCTORS

OffsetManager();

CHAPTER 3. PROJECT: FIDS

PUBLIC ACCESS

ManagerErrors storeOffset(Offset& anOffset)

ManagerErrors removeOffsetWithKey(String& key)

ManagerErrors changeOffset(String& oldLabel, Offset& newOffset)

int numberOfOffsets() const

Offset& getFirstOffset()

Offset& getOffsetAfter(String& lastLabel)

Offset& getOffsetWithKey(String& key) const

CONSTANT OBJECTS

None

CHAPTER 3. PROJECT: FIDS

3.3.8 RelativePosition Manager

BASE CLASS

Dictionary

DERIVED CLASS

None

RELATED CLASSES

RelPos, GeometricManager

DESCRIPTION

The RelPosManager holds a collection of RelPos's. RelPos's are accessed via their key and may be added, removed, or changed inside this collection.

Before any RelPos's are placed in the RelPosManager, the message **init()** must be called to set up NIL objects.

CONSTRUCTORS and DESTRUCTORS

RelPosManager()

PUBLIC ACCESS

ManagerErrors init()

ManagerErrors clear()

ManagerErrors store(Object& aRelPos)

ManagerErrors storeRelPos(RelPos& aRelPos)

ManagerErrors removeWithKey(String& key)

ManagerErrors changeRelPos(String& oldKey, RelPos& newRelPos)

CHAPTER 3. PROJECT: FIDS

```
bool holds(String& key) const  
int numberofRelPos() const  
  
RelPos& getFirstRelPos()  
RelPos& getRelPosAfter(String& lastKey)  
RelPos& getRelPosWithKey(String& key) const
```

CONSTANT OBJECTS

None

CHAPTER 3. PROJECT: FIDS

3.3.9 Geometric Constraint Manager

BASE CLASS

Object

DERIVED CLASS

None

RELATED CLASSES

ClearSpaceManager, OffsetManager, RelPosManager, Manager

DESCRIPTION

The Geometric Constraint Manager is a conglomerate of a ClearSpace Manager, OffsetManager, and Relative Position Manager. ClearSpaces, Offsets, and RelPos's may be added, removed, or changed inside the GeometricManager via their label or key.

Before any Objects are placed in the GeometricManager, the message **init()** must be called to set up NIL objects.

CONSTRUCTORS and DESTRUCTORS

GeometricManager()

PUBLIC ACCESS

ClearSpaceManager& CSM() const

OffsetManager& OM() const

RelPosManager& RPM() const

CHAPTER 3. PROJECT: FIDS

ManagerErrors init()
ManagerErrors clear()
ManagerErrors store(Object& anObject)

bool holds(String& key) const
int numberOfGeometricConstraints() const

ManagerErrors storeClearSpace(ClearSpace& aClearSpace)
ManagerErrors removeClearSpaceWithLabel(String& label)
ManagerErrors changeClearSpace(String& oldLabel, ClearSpace& newClearSpace)
ClearSpace& getFirstClearSpace()
ClearSpace& getClearSpaceAfter(String& lastLabel)
ClearSpace& getClearSpaceWithLabel(String& label) const

ManagerErrors storeOffset(Offset& anOffset)
ManagerErrors removeOffsetWithKey(String& key)
ManagerErrors changeOffset(String& oldKey, Offset& newOffset)
Offset& getOffsetWithKey(String& key) const
int numberOfOffsets() const
LOG& getFirstOffsetLOG()
LOG& getOffsetLOGAfter(String& lastLOG_key)
Offset& getFirstOffset()
Offset& getOffsetAfter(String& lastOffset_key)

ManagerErrors storeRelPos(RelPos& aRelPos)
ManagerErrors removeRelPosWithKey(String& key)
ManagerErrors changeRelPos(String& oldKey, RelPos& newRelPos)
RelPos& getFirstRelPos()
RelPos& getRelPosAfter(String& lastKey)
RelPos& getRelPosWithKey(String& key) const

CONSTANT OBJECTS

None

CHAPTER 3. PROJECT: FIDS

3.3.10 Manager

BASE CLASS

Object

DERIVED CLASS

None

RELATED CLASSES

ClearSpaceManager, OffsetManager, RelPosManager, Manager

DESCRIPTION

The Manager is a conglomerate of a GridManager, LoadEventManager, and GeometricManager. GridLines (Horizontal and Vertical), LoadEvents (Loads and LoadLists), ClearSpaces, Offsets, and RelPos's may be added, removed, or changed inside the Manager via their label or key.

Generally, a `getFirstObject` returns the first object of that class in the Manager, and a `getNextObject(label)` returns the next object held, given that the last object returned had the label (or key) of `label`. Also, performing a `getNext("FIRSTTIME")` is the same as a `getFirst()`.

Before any Objects are placed in the Manager, the message `init()` must be called to set up NIL and bounding objects.

The Manager also may help with the consistency checks of GenericPoint, Load, and LoadEvent objects, via the functions `checkGP()`, `checkLoad()`, and `checkLoadEvent()`, respectively.

CONSTRUCTORS and DESTRUCTORS

CHAPTER 3. PROJECT: FIDS

Manager()

Manager(**const char*** filename) // create manager from filename

PUBLIC ACCESS

void setManagerPointer()

ManagerErrors init()

ManagerErrors store(Object& anObject)

ManagerErrors remove(Object& anObject)

ManagerErrors clear()

OrientType getGridLineOrient(String& label) **const**

GenericPoint& getIntersectionPoint(Float& x, bool x_rel,

Float& y, bool y_rel) **const**

GenericPoint& getAreaPoint(Float& x0, Float& y0, Float& x1, Float& y1,

bool x_rel, bool y_rel) **const**

GenericPoint& getLinePoint(Float& x0, Float& y0, Float& x1, Float& y1,

bool x_rel, bool y_rel) **const**

bool checkGP(GenericPoint& aGenericPoint) **const**

bool checkLoad(Load& aLoad) **const**

bool checkLoadEvent(LoadEvent& aLoadEvent) **const**

GridManager& GM() **const**

LoadEventManager& LEM() **const**

GeometricManager& GCM() **const**

void saveToFile(**const char*** filename) **const**

GridManager Messages:

GridLine& getGridLineFromUser() **const**

ManagerErrors GMclear()

ManagerErrors GMinit()

CHAPTER 3. PROJECT: FIDS

```
ManagerErrors storeGridLine(GridLine& aGridLine)
ManagerErrors removeGridLineWithLabel(String& label)
ManagerErrors changeGridLine(String& oldLabel, GridLine& newGridLine)
bool holdsGrid(String& label) const
int numberOfHoriz() const
int numberOfVert() const
bool setOrigin(Float& x0, Float& y0)
bool setCorner(Float& x1, Float& y1)
GridLine& getFirstHoriz()
GridLine& getFirstVert()
GridLine& getLastHoriz()
GridLine& getLastVert()
GridLine& getCurrentHoriz() const
GridLine& getCurrentVert() const
GridLine& getNextHoriz()
GridLine& getNextVert()
GridLine& getPreviousHoriz()
GridLine& getPreviousVert()
GridLine& getGridLineWithLabel(String& label)
GridLine& getHorizGridLineWithDistance(const Float& y)
GridLine& getVertGridLineWithDistance(const Float& x)
GridLine& getHorizGridLineBefore(const Float& y)
GridLine& getVertGridLineBefore(const Float& x)
GridLine& getHorizGridLineAfter(const Float& y)
GridLine& getVertGridLineAfter(const Float& x)
Float& findDistanceBetween(String& label_1, String& label_2)
GridHolder& getGridLinesBetween(String& label_1, String& label_2) const
```

LoadEventManager Messages:

```
ManagerErrors LEMinit()
ManagerErrors LEMclear()
```

CHAPTER 3. PROJECT: FIDS

```
bool holdsLoadEvent(String& label) const
bool holdsLoadWithLabel(String& label) const
bool holdsLoadListWithLabel(String& label) const
ManagerErrors addLoadEvent(LoadEvent& aLoadEvent)
ManagerErrors addLoadEvent(String& loadListLabel, String& loadEventLabel)
ManagerErrors addLoadList(String& loadEventLabel, String& loadListLabel)
ManagerErrors addLoad(String& loadEventLabel, String& loadListLabel,
                     Load& aLoad)
ManagerErrors removeAllLoadEvents()
ManagerErrors removeLoadEventWithLabel(String& label)
ManagerErrors removeLoadEventWithLabel(String& loadEventParentLabel,
                                      String& loadEventLabel)
ManagerErrors removeLoadListWithLabel(String& loadEventLabel,
                                      String& loadListLabel)
ManagerErrors removeLoadWithLabel(String& loadEventLabel,
                                  String& loadListLabel, String& loadLabel)
ManagerErrors changeLoadEvent(String& oldLabel, LoadEvent& newLoadEvent)
ManagerErrors changeLoad(String& oldLabel, Load& newLoad)
int numberOfLoadEvents() const
int numberOfLoads() const
int numberOfLoadLists() const
LoadEvent& getLoadEventWithLabel(String& label) const
LoadEventManager& getLoadEventsOnPoint(const GenericPoint&
                                         aGenericPoint) const
LoadEventManager& getLoadEventsOnGridLine(String& label) const
LoadManager& allLoads()
Load& getNextLoad(String& lastLoadEventLabel, String& lastLoadListLabel,
                   String& lastLoadLabel)
Load& getLoadWithLabel(String& label)
```

GeometricManager Messages:

CHAPTER 3. PROJECT: FIDS

```
ManagerErrors GCMinit()
ManagerErrors GCMclear()
int numberOfGeometricConstraints() const
bool holdsGeometricWithLabel(String& label) const

ManagerErrors storeClearSpace(ClearSpace& aClearSpace)
ManagerErrors removeClearSpaceWithLabel(String& label)
ManagerErrors changeClearSpace(String& oldLabel, ClearSpace& newClearSpace)
ClearSpace& getClearSpaceWithLabel(String& label) const
ClearSpace& getFirstClearSpace()
ClearSpace& getNextClearSpace(String& lastLabel)

ManagerErrors storeOffset(Offset& anOffset)
ManagerErrors removeOffsetWithKey(String& key)
ManagerErrors changeOffset(String& oldKey, Offset& newOffset)
Offset& getOffsetWithKey(String& key) const
int numberOfOffsets() const
LOG& getFirstOffsetLOG()
LOG& getOffsetLOGAfter(String& lastLOG_label)
Offset& getFirstOffset()
Offset& getNextOffset(String& lastKey)

ManagerErrors storeRelPos(RelPos& aRelPos)
ManagerErrors removeRelPosWithKey(String& key)
ManagerErrors changeRelPos(String& oldKey, RelPos& newRelPos)
RelPos& getRelPosWithKey(String& key) const
RelPos& getFirstRelPos()
RelPos& getNextRelPos(String& lastKey)
```

CONSTANT OBJECTS

None

CHAPTER 3. PROJECT: FIDS

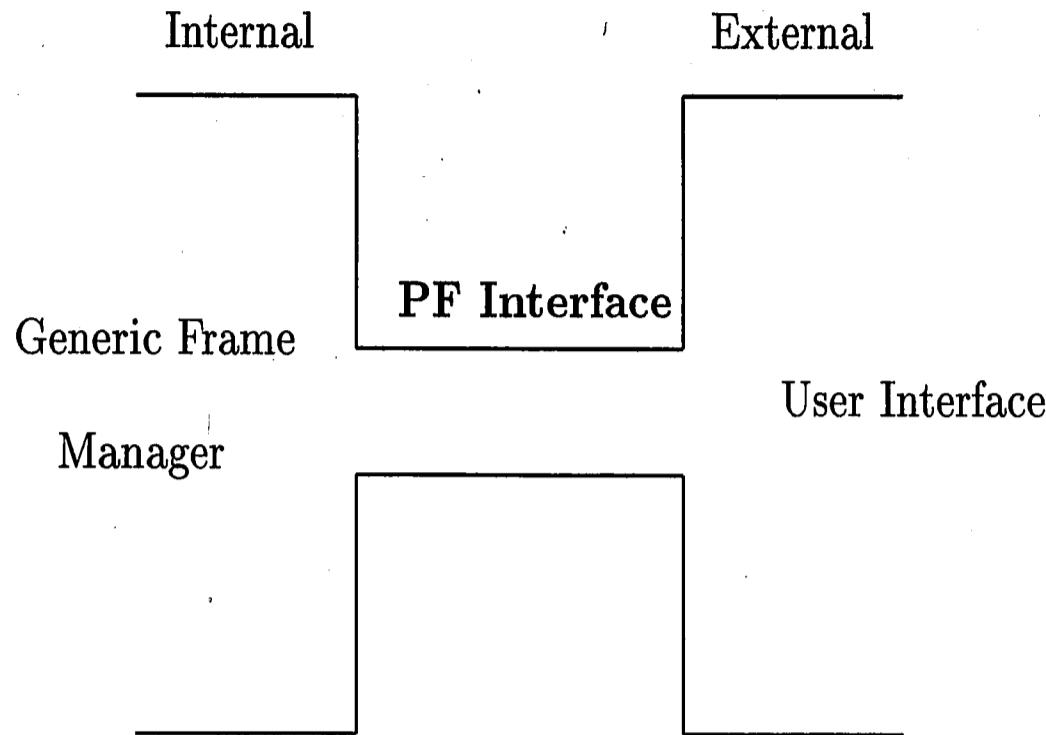


Figure 3.2: Problem Formulation Interface

3.4 Problem Formulation

Problem Formulation (PF) is the first stage of development in the FIDS simulation, whereby a user, or other computational front-end package, has the ability to modify the contents of the Generic Frame representation. As shown previously, we have the representation for a Generic Frame's state held in the **Manager** object which then can be handed off to a Specific Frame's Problem Development (PD) phase. In Problem Development, the explicit frame layout and materials are selected that satisfy the constraints of the problem formulation held in the Manager.

This section will discuss the explicit module called PF used to connect the simulation to a front-end.

3.4.1 Interface between Manager and User Interface

As shown in Figure 3.2, the PF interface links the Generic Frame Manager to the User Interface. This channel has a specific and narrow protocol for messages and objects to be relayed between the internal and external portions of the simulation.

CHAPTER 3. PROJECT: FIDS

3.4.2 Stripped Problem Formulation Interface

Since we wish to decouple the compilations of the simulation engine from the front-end, all messages and objects passed across the interface need to be fundamental classes, or classes rooted with a fundamental base class, i.e. no object traveling across PF is derived from objects in other libraries such as the NIH C++ Class Library or the InterViews Graphical Library. To accomplish this task, we have created a series of "fundamental" objects which only represent the current state of the corresponding abstract class, and contain no functionality, other than constructors. A fundamental object should be constructible from the following objects:

a **nil** - to allocate space for the fundamental object

fundamental data arguments - allowing the front-end to create an object from the user specified state

the corresponding abstract object - allowing the engine to pass the state of an object to the front-end.

For example, the GridLine class has the following fundamental structure:

```
class xGridLine {  
public:  
    char* label;  
    OrientType orient;  
    double distance;  
  
    xGridLine(); // nil constructor  
    xGridLine(char*, OrientType, double); // fundamental constructor  
    xGridLine(GridLine&); // abstract constructor  
};
```

In addition, the interface must have specific messages to convert the fundamental object into its related abstract object:

CHAPTER 3. PROJECT: FIDS

```
GridLine& PF::convertGridLine(xGridLine&);
```

Note that this function is included in the PF object, and could be a private member function, since this type of conversion is normally used internally to PF when converting the fundamental object passed in from the front-end to the abstract object handed off to the internal Manager.

As shown for GridLines, the interface will also handle the conversion of other simulation-based objects, including the lower-level building blocks of objects (e.g. such as GenericPoints used for geometric location). Also, all relevant access or modification queries on the Manager should be available from the interface.

As discussed in Section 2.7, the Problem Formulation object contains the two header files, (1) a complete prototype of all functions, and (2) the stripped down version with no library dependencies. Please refer to Appendix E for the complete header, and Appendix F for the stripped. Comparing these two modules will illustrate this concept.

Chapter 4

Parallel Simulator

4.1 Overview

ParSim [2], a simulator for designing parallel algorithms and architectures, is an interactive tool used for studying the performance of parallel algorithms on various parallel architectures. ParSim allows the user to loosely or strictly map the data flow graph of a static algorithm to an existing parallel topology or a user-created connectivity. The algorithm then can be simulated until conclusion, and the times of execution compared. The ParSim definition language allows for diverse specifications of the data flow graph and destination parallel architecture. This tool includes sophisticated algorithms for automatically generating Connectivity Matrices¹ for known parallel topologies, and for finding the optimal mapping from a data flow graph.

4.2 Introduction to Parallel Computation

In studying parallel computations, performance is a crucial issue. Deciding which parallel topology best suits a parallel algorithm class, the critical number of Processing Elements (PE's) needed in the parallel architecture, or how to optimize an algorithm for a specific topology, are critical design issues for parallel computations. ParSim was created as a modular program which allows a user to specify the Data Flow Graphs

¹Connectivity Matrices are discussed in Subsection 4.3.2.

CHAPTER 4. PARALLEL SIMULATOR

and realistic Parallel Topologies for static simulations of parallel algorithms.

A Connectivity Matrix holds the interconnection pattern of both the data flow graph and the parallel architecture. ParSim generates connectivity matrices for topologies given the number of Processing Elements in that architecture. ParSim then applies the data flow graph to a parallel architecture. This mapping can be loose (ParSim is free to make all placement decisions), or strict (the user specifies a set of nodes from the data flow graph that must be placed on specific PE's in the parallel architecture). Mappings are optimized by having the smallest total sum of dilations across the links of the data flow graph.

ParSim also incorporates tools for analyzing parallel architectures. ParSim finds the minimum distance between PE's and proves the correctness by showing the path. ParSim also finds the Mutual Partition Set of an architecture - namely, the set of all subsets of PE's of the architecture which are fully connected among themselves.

Users can run ParSim interactively via a menu system, or create a ParSim network file, to simulate parallel computations. A Parallel Processing Definition Language has been developed in this research to specify all the needed attributes of the data flow graph and parallel topology.

ParSim was originally designed in the Pascal programming language, but ran into restrictions with its data structures, for example, limiting the number of nodes or PE's addressable². In converting ParSim to an object oriented language such as C++, these constraints no longer exist.

Please note that ParSim models static parallel networks through their connectivity schemes only.

²A limit of 256 arises due to the maximum size of a Set in Borland's Turbo Pascal ver. 5.0.

CHAPTER 4. PARALLEL SIMULATOR

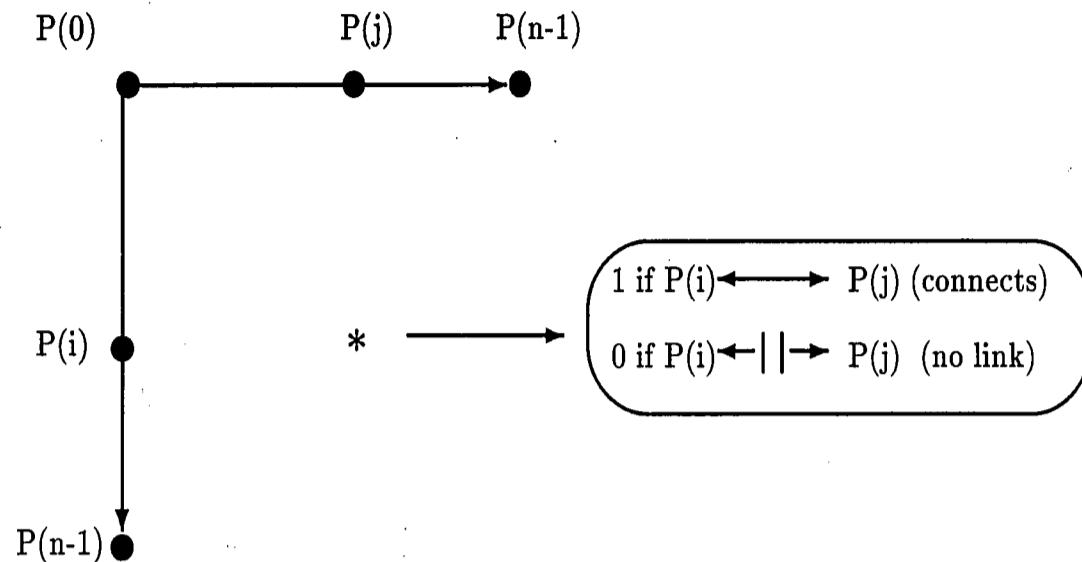


Figure 4.1: Connectivity Matrix

4.3 ParSim Engine Description

4.3.1 Activities Menu

ShowArch	View a Connectivity Matrix on the screen
Connect Arch	Create a C.M. from known connectivities
Read Arch from text file	Restore a C.M. from a C.M. File
Write Arch to text file	Save a Connectivity Matrix to a C.M. File
Enter unique Arch	Create a Connectivity Matrix interactively
Mappings	Map a source C.M. to a destination C.M.
Find minimum distances	Find the shortest paths between all or some PE's
Mutual Partitions	Find all sets of PE's totally connected

Table 4.1: ParSim Activity Menu

Given above are the valid simulation actions in ParSim.³

4.3.2 Connectivity Matrix

A Connectivity Matrix of dimension ' $n \times n$ ' is shown in Figure 4.1.

³Note that C.M. —> Connectivity Matrix.

CHAPTER 4. PARALLEL SIMULATOR

4.3.3 Connectivity Matrix File

A **Connectivity Matrix File** has the following structure:

Line 1 contains Connectivity Name

Line 2 contains an Integer number of processors

The remaining lines have the following structure:

I: A B C D

where **I** is a processor connected to P(**A**), P(**B**), P(**C**), and P(**D**).

Note that since a Connectivity Matrix is obviously symmetric, all symmetric connections are automatically included. Also, the diagonal entries are all '1' since $P(X) \iff P(X)$.

4.3.4 A Sample Connectivity Matrix File

Linear Array

4
0: 0 1
1: 0 1 2
2: 1 2 3
3: 2 3

yields the following connectivity matrix:

	0	1	2	3
0	1	1	0	0
1	1	1	1	0
2	0	1	1	1
3	0	0	1	1

The connectivity of a Linear Array with four processing elements is given in Figure 4.2.

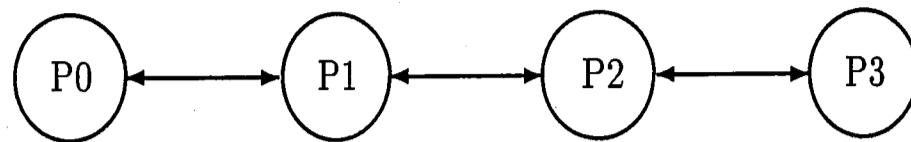


Figure 4.2: Linear Array with 4 PE's

4.3.5 Topologies of Connectivity Networks

Topologies of Connectivity Networks that can be algorithmically generated by ParSim, given the number of PE's, are provided in Table 4.2.

ArchNum	Connectivity	$P(i) \iff P(j)$ if:
1	Linear Array	$P(i)$ next to $P(j)$ numerically
2	Ring	$P(i)$ and $P(j)$ next to each other modulus n
3	Star	All PE's connected to $P(0)$
4	Binary Tree	Single Rooted Binary Tree
5	Illiac Mesh	Wrap-around mesh with dimension 4
6	Cylindrical	Systolic Array
7	Completely Connected	All PE's connected to All other PE's
8	Chordal Ring	Ring w/ added connections of chords = \sqrt{n}
9	HyperCube	$(i \oplus j) = 1$
10	Barrel Shifter	$j = i \pm 2^k \bmod n$
11	$\pm 1, \pm 3, \pm 5$ Ring	Chordal ring w/ odd near-nbr connections

Table 4.2: Algorithmically generatable topologies

4.4 Algorithms used in ParSim

The following section discusses the algorithms which have been written for and incorporated into ParSim. These routines include the creating a connectivity matrix, mapping a data flow specification to a given parallel architecture, finding the minimum distances and paths between processing elements, and finding the mutual partition set of an architecture.

4.4.1 Runtime Creation of a Connectivity Matrix

This program has two basic algorithms for defining the Connectivity Matrix:

CHAPTER 4. PARALLEL SIMULATOR

Brute Force - Use the criterion given in Table 4.2 for determining connectivity at *every* node. This method is useful when no regular pattern can be seen in the Connectivity Matrix.

Connect and Rotate - Because of symmetry in the connectivity, only the connections to $P(0)$ are computed in the first phase. For every remaining PE, the row above in the Connectivity Matrix is carried down and shifted once to the right. (i.e. $P(i) \rightarrow P(i+1)$) but no wrap around occurs; in fact, $P(0) \xleftrightarrow{?} P(j)$, i.e. checking to see if $P(0)$ connects to $P(j)$, is the only PE test computed for each row.

The following topologies take advantage of the **Connect and Rotate** algorithm in their generation:

- Linear Array,
- Ring,
- Illiac Mesh,
- Chordal Ring,
- Barrel Shifter, and
- $\pm 1, \pm 3, \pm 5$ Chordal Ring.

4.4.2 Mapping Algorithm

A greedy algorithm for mapping the given data flow structure into a connectivity architecture is implemented. All pre-decided mapping constraints are set first, then all remaining nodes try to get mapped to all permutations of remaining PE's making sure that any links between two of these such nodes has dilation 1 in the mapped structure. As soon as the algorithm finds one mapping that fits the criteria, the process is over. Otherwise, the mapping routine recursively calls itself with more PE's available to be mapped to.

CHAPTER 4. PARALLEL SIMULATOR

4.4.3 Finding the Minimum Distance/Paths between two PE's

The searching algorithm contains two passes:

- On the first pass, ALL unique connectivity paths between $P(i)$ and $P(j)$ are located, and the smallest path distance is stored.
- On the second pass, (if all the paths of minimum distance are required) a search identical to the first pass takes place, but this pass prints out the necessary distinct paths.

The recursive algorithm to find the distance and path from $P(i)$ to $P(j)$ starts with a set containing i and all the processors connected to i . If j is in the set, then it is finished with that path. Otherwise, a loop through every element in this first set is initiated, and with each PE pointed to, the union of its connection set with the first set is individually found. If the new set is equivalent to the first set, a dead end in the path has occurred. If the new set is larger, it gets sent into this algorithm recursively.

4.4.4 Mutual Partitions

Mutual Partitions are subsets of the architecture which are fully connected.

For example, in the following Connectivity Matrix:

	0	1	2	3
0	1	0	1	1
1	0	1	0	1
2	1	0	1	1
3	1	1	1	1

the mutual partitions are: $\{(0,2,3), (0,2), (0,3), (1,3), (2,3)\}$.

The algorithm used to find the mutual partitions takes the Connectivity Matrix row by row and creates a **possibility set** by putting each index of a '1' to the right of

CHAPTER 4. PARALLEL SIMULATOR

the diagonal in a set. (Since the Connectivity Matrix is symmetric, this will eliminate duplication by finding all partitions containing P(0), then all containing P(1) but not P(0), then all containing P(2) but not P(1) nor P(0), etc..) Now each row possibility set cannot contain PE's numbered less than the current row. Therefore, we will find all of P(0)'s subsets, then P(1)'s, etc. All subsets of the possibility set are created by counting in binary with a string the length of the cardinality of the set and masking that with the set. Each subset is then checked for the mutual property, namely, that all PE's present are directly connected.

4.5 General Execution Steps of Simulation

The following is the order of events in a ParSim simulation:

1. **Input** user's network
2. **Create** a model and define/determine parameters, modes, etc.
3. **Map** model into structure
4. **Run** simulation in time increments

4.5.1 Example of Data Flow Simulation

An example run of ParSim, consisting of data flow and topology, is shown in Figure 4.3.

CHAPTER 4. PARALLEL SIMULATOR

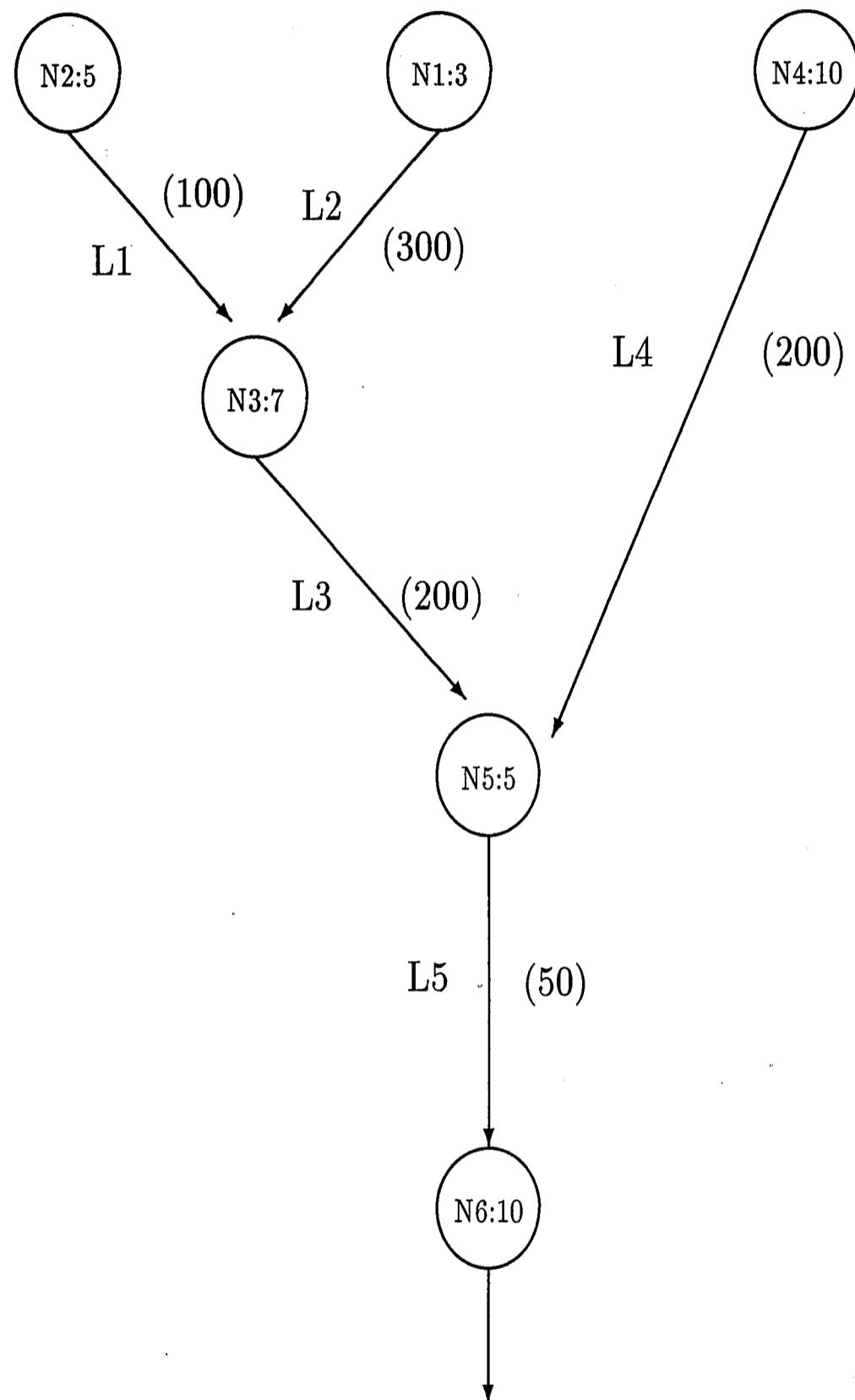


Figure 4.3: Example Data Flow Network

CHAPTER 4. PARALLEL SIMULATOR

* Example Network Description File

```
N1      3      C
N2      5      C
N3      7      S
N4      10     C
N5      5      S
N6      10     C
L1      N2      N3      100 ; 2 3 can be used instead of N2 N3 here
L2      N1      N3      300
L3      N3      N5      200
L4      N4      N5      200
L5      N5      N6      50
.MAP
N1      P3
N3      P7
N4      P5
N6      P1
.map end
.linkwts auto
.arch  5      8          ; Illiac Mesh with 8 PE's
.END
```

CHAPTER 4. PARALLEL SIMULATOR

NETSIM Log File for example.net

Run Date: 4-10-1990

Run Time: 21:02:28.68

```
1      * Example Network Description File
2
3      N1      3      C
4      N2      5      C
5      N3      7      S
6      N4      10     C
7      N5      5      S
8      N6      10     C
9      L1      N2      N3      100    ; 2 3 can be used...
10     L2      N1      N3      300
11     L3      N3      N5      200
12     L4      N4      N5      200
13     L5      N5      N6      50
14     .MAP
15     N1      P3
16     N3      P7
17     N4      P5
18     N6      P1
19     .map end
20     .linkwts auto
21     .arch   5      8          ; Illiac Mesh with 8 PE's
22     .END
```

CHAPTER 4. PARALLEL SIMULATOR

*** Mapping ***

Source Arch:

Example Network Description Network with 6 processors
connectivity map:

P 0	101000
P 1	011000
P 2	111010
P 3	000110
P 4	001111
P 5	000011

Dest Arch:

Illiac Mesh Network with 8 processors connectivity map:

P 0	11010101
P 1	11101010
P 2	01110101
P 3	10111010
P 4	01011101
P 5	10101110
P 6	01010111
P 7	10101011

N1 mapped to P3

N2 mapped to P0

N3 mapped to P7

N4 mapped to P5

N5 mapped to P2

N6 mapped to P1

CHAPTER 4. PARALLEL SIMULATOR

*** Link Weights ***

L1 from N2 to N3 [wt:1]

L2 from N1 to N3 [wt:2]

L3 from N3 to N5 [wt:1]

L4 from N4 to N5 [wt:1]

L5 from N5 to N6 [wt:1]

4.6 Controller Object

The **Controller** object is the representation of a manager in a parallel computation simulation. The controller is responsible for the promotion through the simulation steps outlined in Section 4.5.

The controller must first parse an input file, which might be from standard input (the keyboard), containing the user's representation of the Data Flow network representing the static algorithm to be simulated. Appendix G contains the reference list of directives available in the **ParSim Definition Language**. A **DataFlow** object is thus constructed and represents all aspects of the input. This **DataFlow** object contains a connectivity matrix with **Node** objects at each index, and **Link** objects created for '1' that would appear in the matrix.

The controller also holds an **Architecture** object, representing the physical connectivity network of PE's to which the **DataFlow** object has been applied.

Other responsibilities of the controller are:

- Hold statistics for the simulation, such as the goodness of fit of the mapping, simulation mode states specified by the user, a time counter, the current state of modeling and simulation, etc.
- Orchestrate the Mapping from the **DataFlow** object to the **Architecture** object
- Perform the analysis algorithms listed in Section 4.4
- Maintain a global clock and watch over the simulation, and report the results when the algorithm has run to completion.
- Report any errors encountered in the simulation

4.7 Nodes

A **Node** object represents a computational node in the data flow graph of the algorithm. A Node has label and holds a computation time for one block of data. The

CHAPTER 4. PARALLEL SIMULATOR

Node may either operate in the "Compute first, then transmit results" mode, or the "Simultaneous computation and transmission" mode.

A Node maintains a queue for the Message packets which are incoming or outgoing, and processes them as the time counter allows.

The Node also has a reference to the processing element it will be directly mapped to in the architecture, so that at simulation time, a quick cross-reference between algorithm and architecture may take place.

4.8 Links

The **Link** object represents the connections between nodes of the data flow graph. A Link has a unique label and holds a **Units** measurement, representing the capacity of this Link. A Link holds the Node labels for the two Node objects which are at its origination and destination. A multiplier, called **Weight**, for a Link represents the time delay factor a Message will have when traveling across this path.

Normally, the Weight is a measure of the dilation of a Link after the algorithm has been mapped to a real network topology. For example, a Weight of '1' implies that the nodes at the endpoints of a Link are directly connected in the associated architecture. A Weight of '2' implies that there is one intermediate processing element along the path of this link, and appropriate message passing techniques must be employed, with the overhead time delays calculated in the analysis as well.

4.9 Message

A **Message** object is a token for a block of data which occupies computational time in a Node, and then must be transmitted down the Link in the computational graph. Messages may be of any size and are queued at the input and output of Nodes. Messages are transferred between the Nodes by Links, but only when the resources are available. A Message knows how long it has been in the system, its Node of origin, and Node of destination.

4.10 Architecture

The **Architecture** object is another connectivity matrix, with processing elements along the diagonal, numbered according to the diagonal index, and connections in the matrix where PE's connect. The architecture may be constructed on an index from Table 4.2, and the number of PE's for that instance of a topology.

4.11 Mapper

The **Mapper** object participates with the Controller by accepting the DataFlow and Architecture objects and creating the optimal mapping, as explained in Subsection 4.4.2.

4.12 Conclusions

By taking advantage of object oriented techniques, a parallel processing simulation is easily decomposed into functional objects. A distinction is made between the original entities represented in the description of the static algorithm and the physical elements and connections of the architecture. By doing this, a controller may keep employing the mapper to overlay the algorithm on several architectures, or even a single architecture of various dimensionalities.

Analyzing these results will prove useful for finding the best architectures for certain classes of parallel processing algorithms, without having any parallel hardwares available. The computational times are normalized to unit blocks of data, and a unit time clock, which may be manipulated for the different simulation runs.

The front-end for ParSim bears no effect on the actual simulation engine described in this chapter. Refer to Chapter 5 for an outline of the techniques needed when designing a graphical user interface for the ParSim package.

Chapter 5

User Interface

As explained in Section 2.7, a user normally wishes to inspect the current state of an object oriented simulation through a concise and accurate graphical front-end. This **Graphical User Interface (GUI)** is the user's view to the internal representation. The GUI must dynamically change as the user progresses through the simulation; from formulating the problem to developing the solutions.

The GUI has the task of controlling the flow of events in the simulation. Every user action, such as a mouse click or key stroke, directly associates with either a function call to the internal manager or a modification of the view. This chapter will discuss the basic properties of this user interface.

In the FIDS simulation, the InterViews Graphical Library [23] was used to implement the GUI. However, any package or library of routines for displaying graphics on the screen and interacting with a user may be a sufficient starting point for the GUI.

5.1 Accessing the interface

When interacting with the manager, the front-end, or GUI, must follow the protocol, as outlined in Section 3.4, concerning the interface between the internal and external modules.

In the FIDS Project, the GUI maintains the top level control of the simulation. The reason that we place this responsibility on the GUI is to enable the user to control

CHAPTER 5. USER INTERFACE

the simulation. The simulation sits idle until the user triggers an action through an event, such as placing the mouse at a location, clicking a button, or stroking a key.

The GUI must have access to the manager via the interface. From a systems point of view, the GUI holds an instance of the interface, which in turn, holds an instance of the manager. All messages travel across this interface. Thus, the interface has full control over its instance of the manager, while the GUI calls the appropriate interface messages to accomplish its tasks. The interface should have a function available for every action required by the GUI. For example, if the user wishes to refresh the display, the GUI then asks the interface for the objects that the manager is holding, object by object, displaying the graphical representation for each object as it goes, until no more are left. When the manager has returned the last, it will pass a flag to the interface denoting that the end has been reached.

The GUI is responsible for telling the interface that a new simulation is about to be created, and that the manager should be initialized. Then when a user asks the GUI to construct a new entity for that simulation, the GUI will popup a form with the object's state variables, allowing the user to make any necessary changes in the description. From this information, the GUI will call the object constructor associated with the given entity, as described in Subsection 3.4.2. This new object is then handed over to the interface with as an argument in the function call to add or modify an object in the manager. The GUI normally will hold only the key, as described in Subsection 2.5.2, to refer back to that object once it has been entered into the manager. By using the key and the interface query functions, the GUI will have access to sufficient geometrical information for representing the object on the display.

5.2 Layout of workspace

The workspace needs to adapt to the current state of the simulation. The majority of the space will hold the graphical representation of the problem. Along the top border, pulldown menus for requesting a service are placed, and the right border will allow toggle buttons representing the state of various objects to be drawn, such

CHAPTER 5. USER INTERFACE

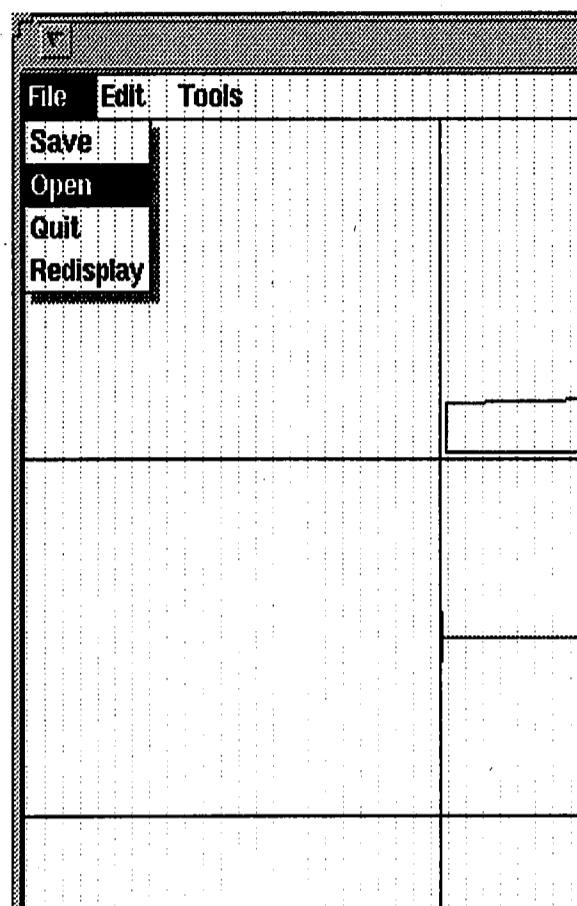


Figure 5.1: FIDS-GUI File Menu

as an orientation toggle for GridLines, Offsets, and Relative Positions in the FIDS simulation. Also, exact simulation coordinates of the mouse's current location should be available. The workspace should have slider bars, or a panner, and zoom factors for inspecting large projects.

The workspace for a simulation is initially clear. The user may pull down the various menus, such as a *File* menu, shown in Figure 5.1, to store, retrieve, and refresh the objects, or a *Tools* menu, shown in Figure 5.2, to create an entity of a given class, select an entity on the display (and possibly modify it), remove an entity, or move an entity to a new location.

When an entity is created or selected, a popup form for that class appears on the display, showing all the state variables which may be altered by the user. A GridLine object is shown in Figure 5.3, a RelativePosition in Figure 5.5, an Offset in Figure 5.6, a ClearSpace in Figure 5.7, a DistributedLoad in Figure 5.8, and a PointLoad in Figure 5.9. The user also has the choice of accepting the changes, canceling the form, or removing the object from the simulation. After this object has

CHAPTER 5. USER INTERFACE

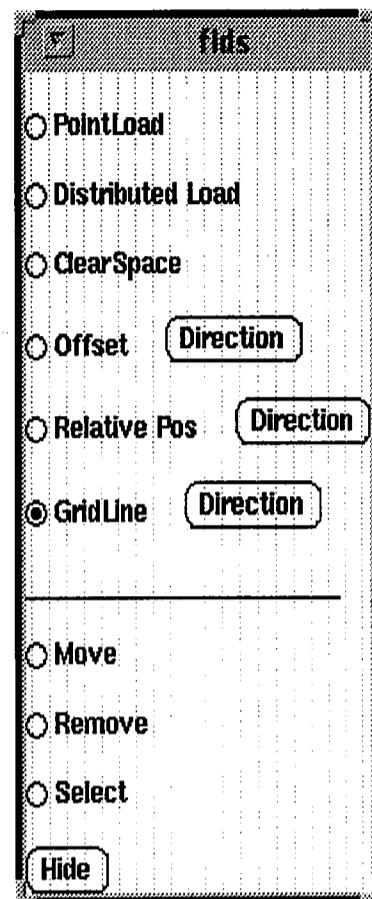


Figure 5.2: FIDS-GUI Activity/Tools Menu

been constructed and passed to the interface, the manager might find an inconsistency and return an error flag accordingly, alerting the GUI to pop up a window with the corresponding error message. An example of this is given in Figure 5.4, where the user has attempted to store a grid line with a label identical to the label of a grid line already held in the Manager. The user then must acknowledge the error response before continuing. The system then allows the user to fix any errors specified on the popup form.

CHAPTER 5. USER INTERFACE

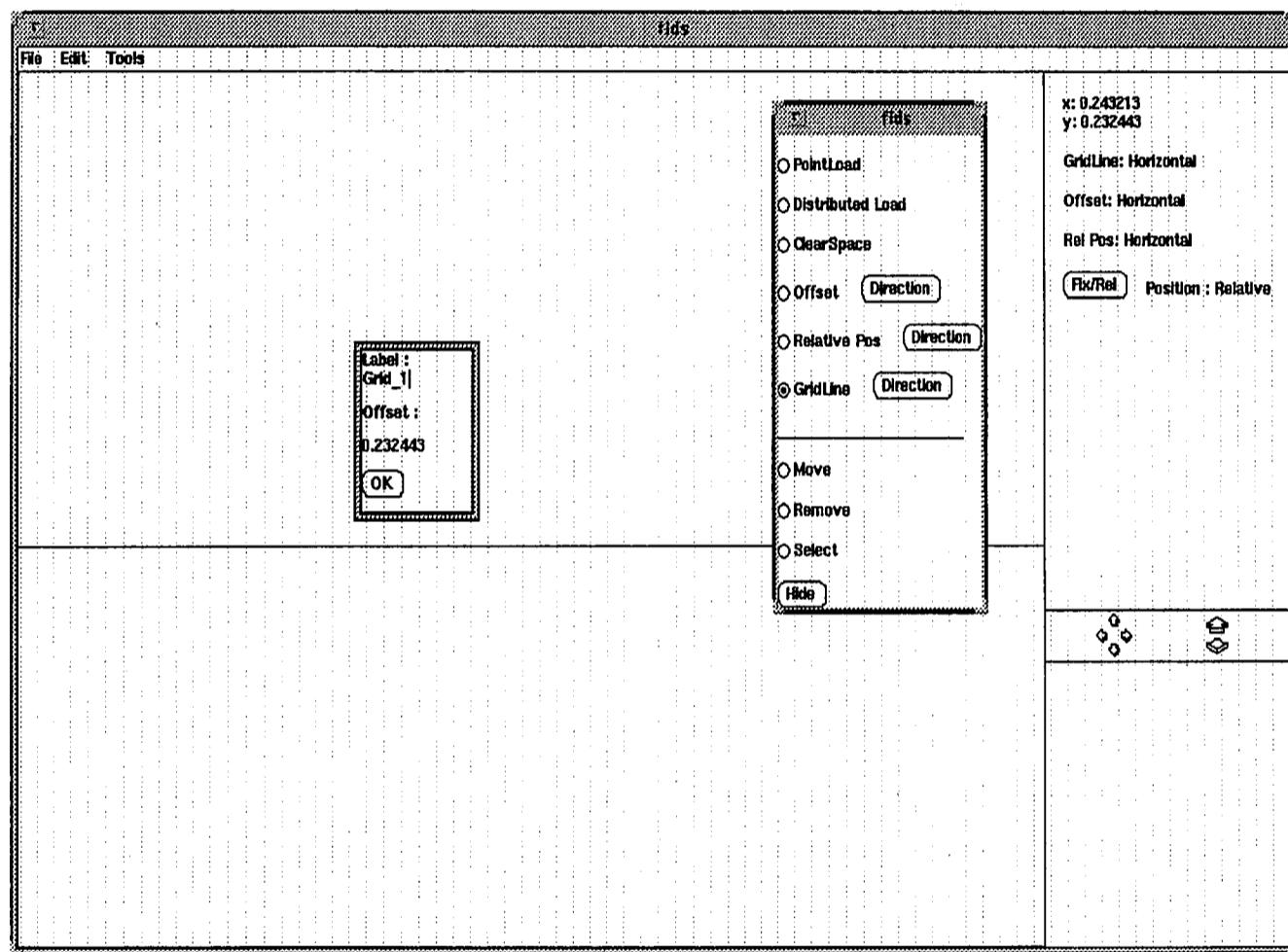


Figure 5.3: FIDS-GUI GridLine Form

CHAPTER 5. USER INTERFACE

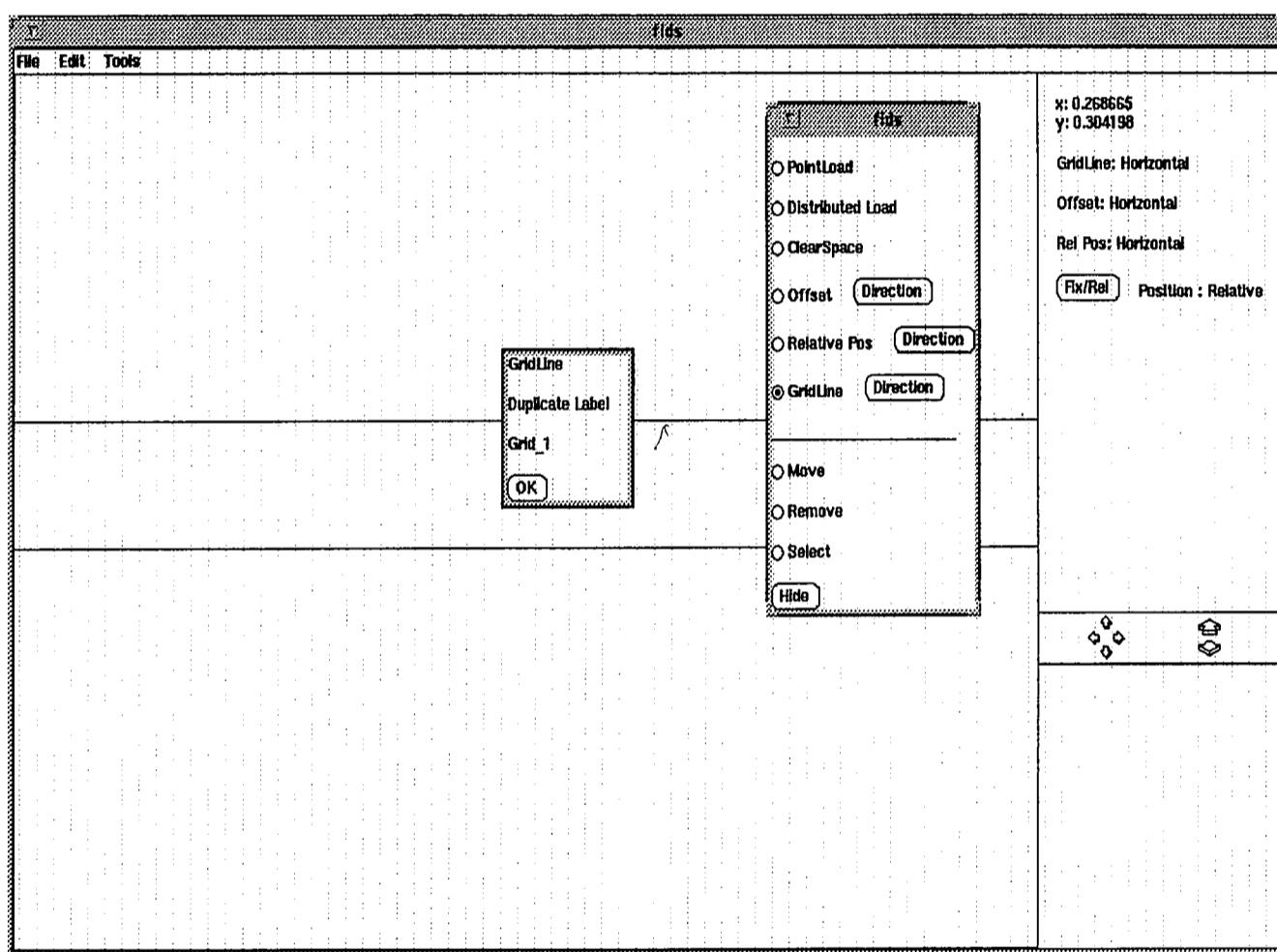


Figure 5.4: FIDS-GUI Duplicate Label Error Window

CHAPTER 5. USER INTERFACE

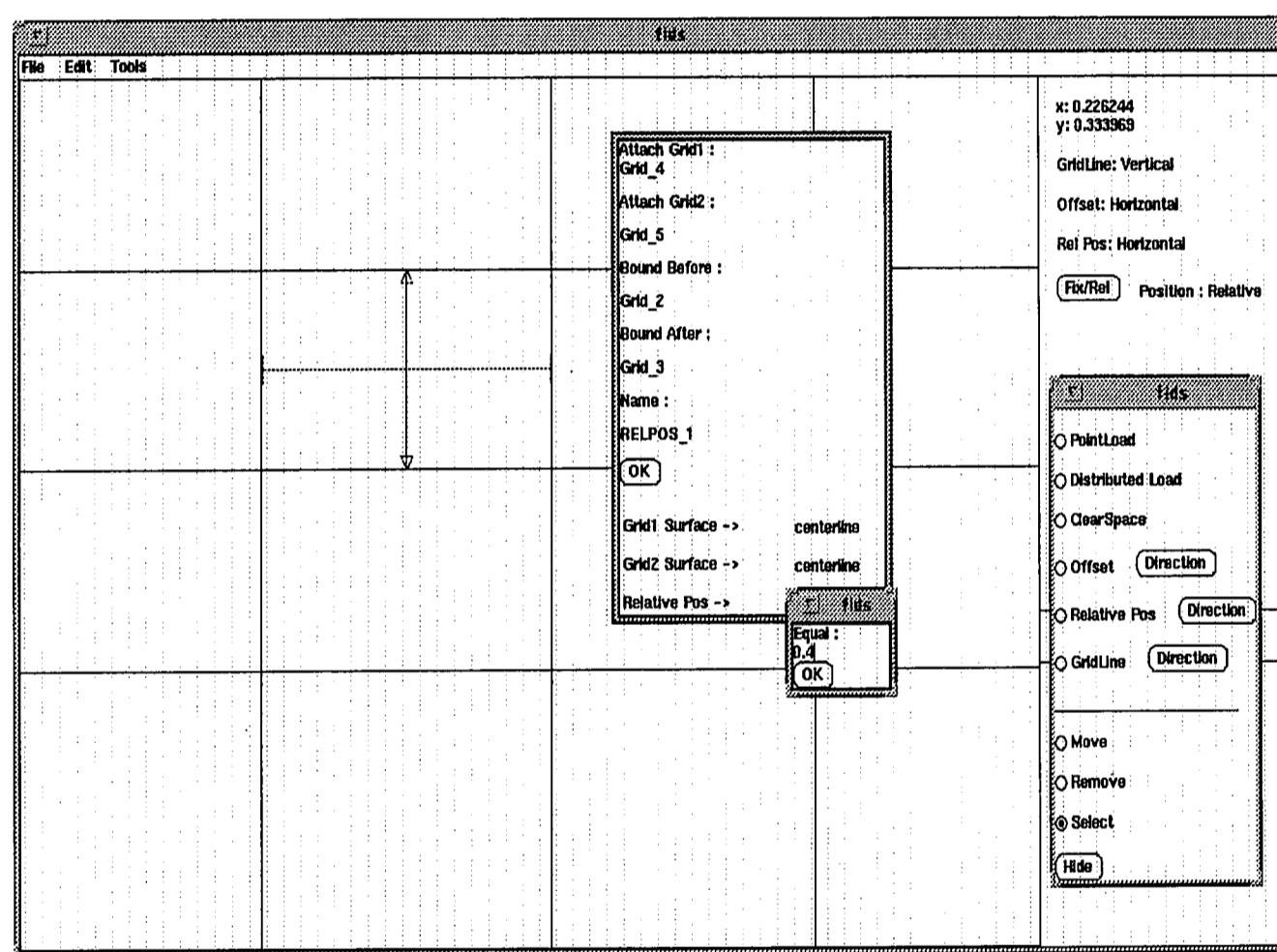


Figure 5.5: FIDS-GUI Relative Position Form

CHAPTER 5. USER INTERFACE

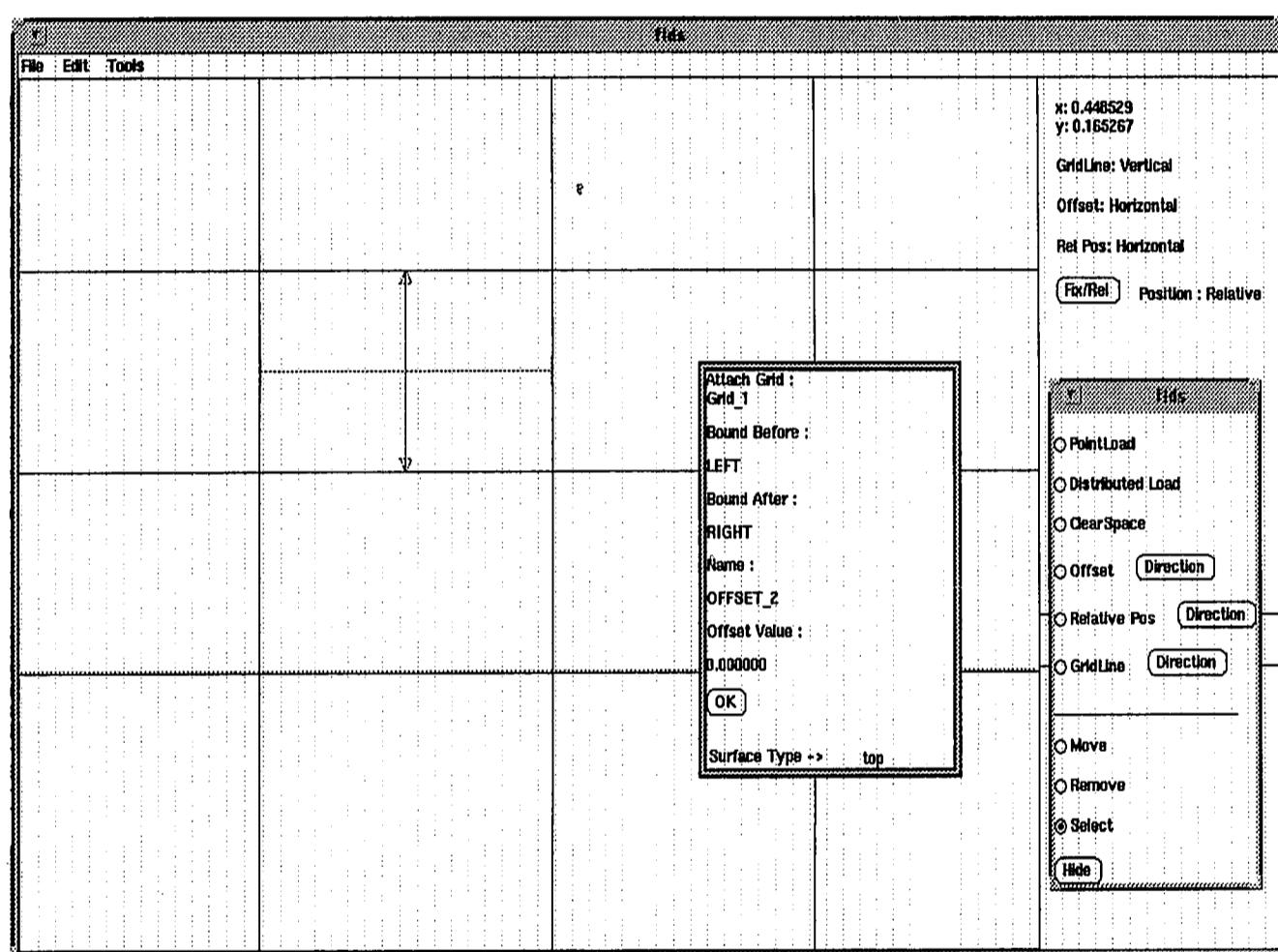


Figure 5.6: FIDS-GUI Offset Form

CHAPTER 5. USER INTERFACE

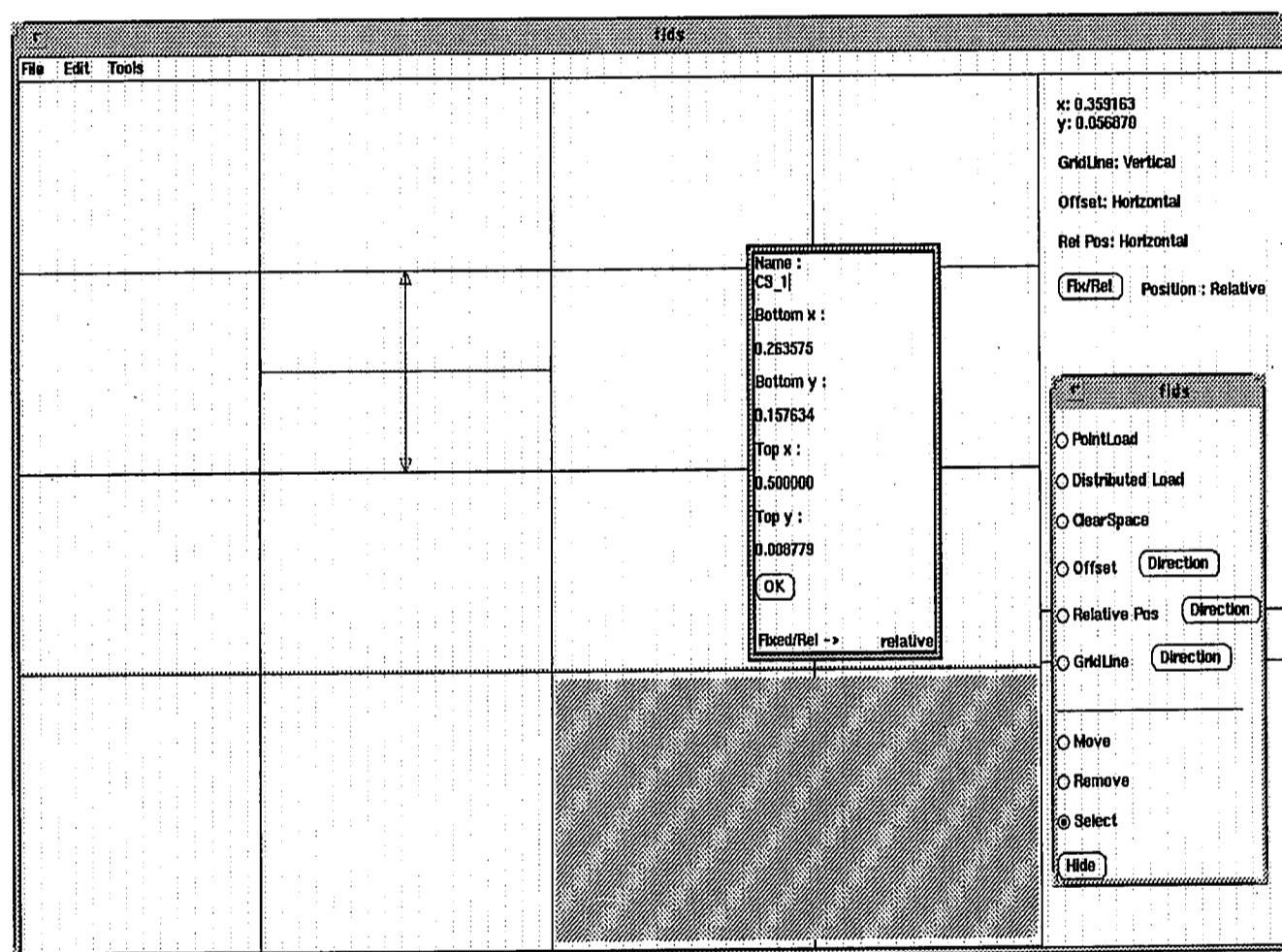


Figure 5.7: FIDS-GUI ClearSpace Form

CHAPTER 5. USER INTERFACE

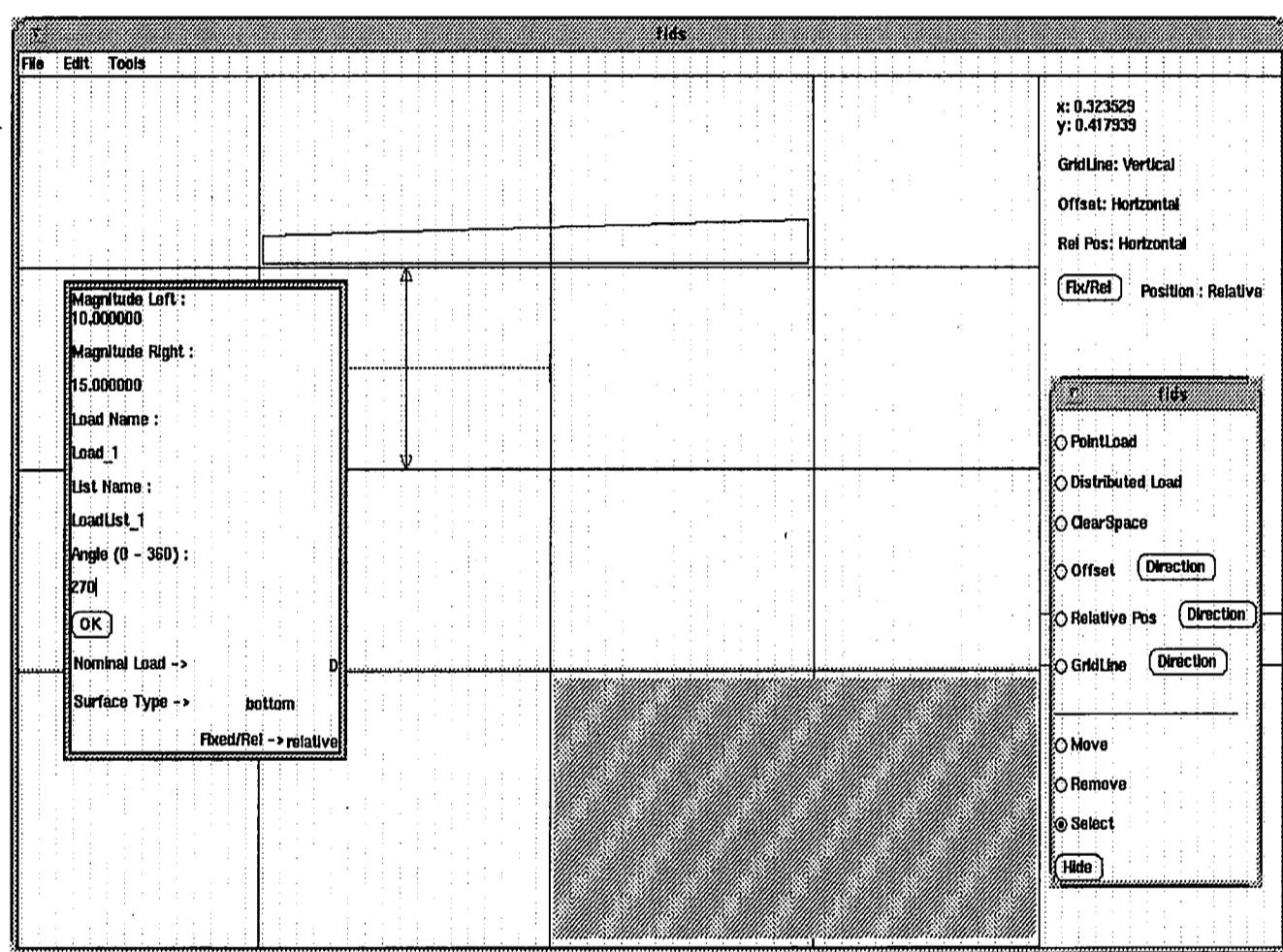


Figure 5.8: FIDS-GUI Distributed Load Form

CHAPTER 5. USER INTERFACE

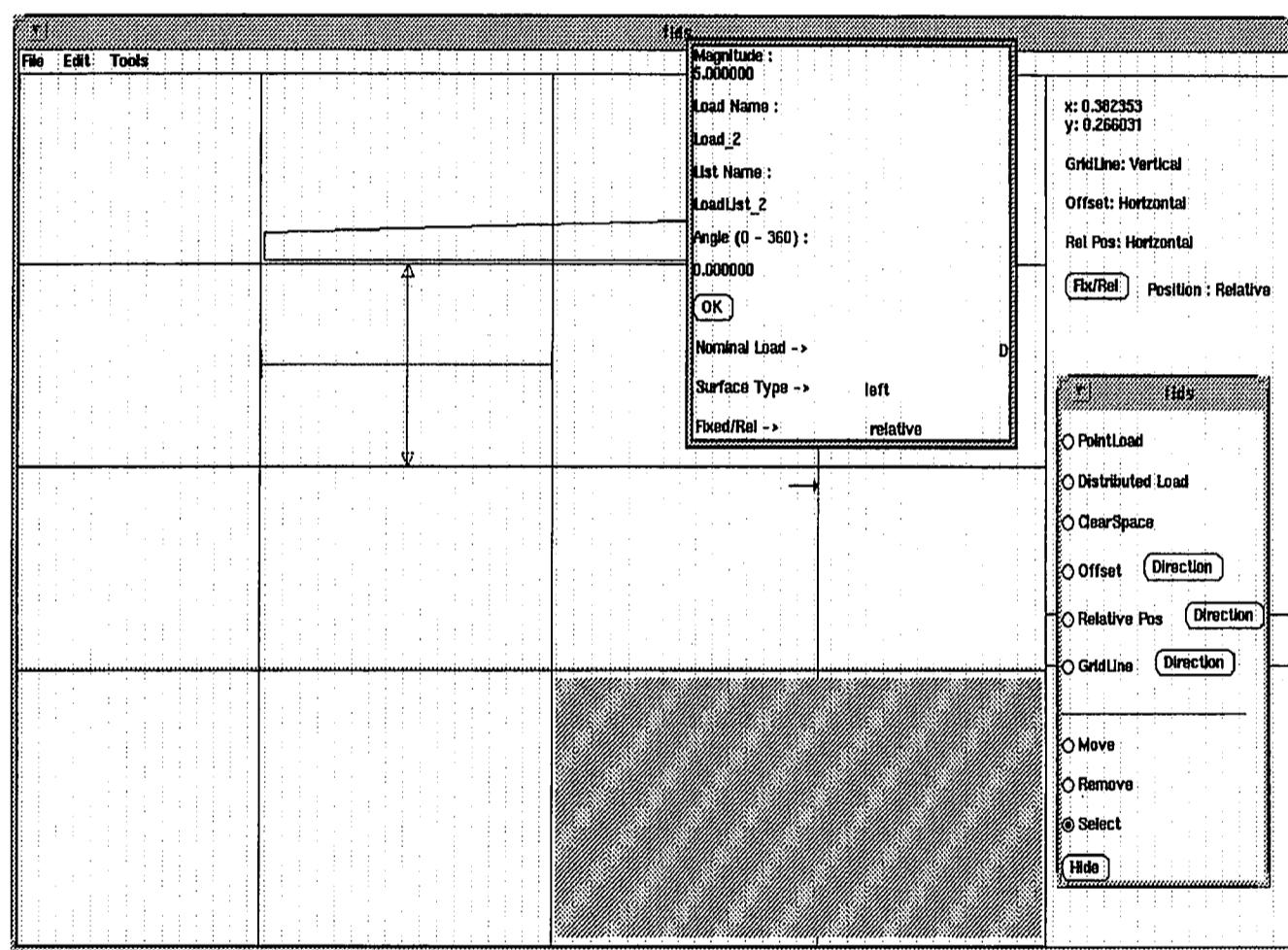


Figure 5.9: FIDS-GUI Point Load Form

5.3 Getting input from user

The user has control over the GUI through the mouse and keyboard. The mouse has the capability of selecting a location on the display, dragging a selection bar on a pulldown menu, highlighting text, etc. The keyboard should be able to mimic the same functions available to the mouse. Obviously the mouse will be a lot less tedious to use. In addition, the keyboard is also used for entering numbers and filling in labels for objects.

Chapter 6

Conclusions

In this thesis, I have discussed important aspects of designing an object oriented simulation; from the simulation engine to the front-end user interface. From the examples in two engineering disciplines, namely, structural design and parallel processing, one can readily see that the techniques outlined here apply to a variety of such systems. Object oriented programming allows modular libraries to be constructed, and then used by packages needing similar designs. The decomposition of a simulation into objects and managers also becomes fairly straightforward after learning the concepts of abstract object construction.

The Framework for Integrated Design System, which is a prototype structural design analysis system, shows the usefulness of the object oriented approach. With the aid of object libraries, the design of this simulation has been developed. In essence, one strives to create a simulation in the problem domain, theoretically distancing the engineer from the low level computer programming layer.

The FIDS package has been designed for future expansion. The front-end of an object oriented simulation system may be modular. Conceptually, and in actuality, one may easily unplug the current GUI of the FIDS, and replace it with an improved GUI, or some other input/output device not yet realized. For an example, **Virtual Reality** GUI's [21] are currently being researched and developed where a user is attached to a simulation via body sensors, and a helmet with three dimensional graphical perspectives.

CHAPTER 6. CONCLUSIONS

This thesis decomposes the ParSim parallel processing simulator into an object oriented simulation. Computer engineering researchers have found ParSim to be useful in determining parallel computation statistics for static algorithms. Because of object oriented design, the simulation is open for future enhancements, such as replacing the existing Node object with one that can simulate more detailed processing capabilities, or allowing dynamic algorithm simulation.

One feature of object oriented decomposition of a simulation is that as computer platforms are changed, minimal work is needed to re-create the simulation package. The language of C++ is in the process of being standardized, so that porting, or transferring, the code between platforms, or to a new release of the language and/or operating system, will only cause minor changes, e.g., merely changing the compiler options at compile time.

The C++ hierarchical class and graphical libraries are young, as is the language of C++. With time, more complete and coherent documentation will become available. Debugging software packages will move towards a more integrated environment, allowing more informative error detection schemes, and possibly automatic correction of these mistakes. Also, dedicated compilers and linkers for object oriented languages (rather than front-end translators) will be developed. For example, the standard C programming language preprocessor, compiler, and linker is currently used to create C++ executable code on most workstation platforms. The development of dedicated compilers and linkers will mean faster running applications, as well as quicker design time, since intelligent compilations with dependency information will be performed.

Future work needs to be done in using multiple libraries in one package. Although this thesis describes a method for creating an interface which decouples libraries at compile time, a better approach needs to be found which would allow objects in libraries to be used interchangeably.

Object oriented systems are shown to be beneficial and are quickly becoming the preferred method of designing large packages. The future task of software engineers will be to create the basic object libraries necessary for implementing any common application.

Appendix A

The NIH C++ Class Library Hierarchy

Version 3.0 of the NIH Class Library [14, Appendix A] contains the following classes:

- NIHCL---Library Static Member Variables and Functions
- Object---Root of the NIH Class Library Inheritance Tree
 - Bitset---Set of Small Integers (like Pascal's type Set)
 - Class---Class Descriptor
 - Collection---Abstract Class for Collections
 - Arraychar---Byte Array
 - ArrayOb---Array of Object Pointers
 - Bag---Unordered Collection of Objects
 - SeqCltn---Abstract Class for Ordered, Indexed Collections
 - Heap---Min-Max Heap of Object Pointers
 - LinkedList---Singly-Linked List
 - OrderedCltn---Ordered Collection of Object Pointers
 - SortedCltn---Sorted Collection
 - KeySortCltn---Keyed Sorted Collection
 - Stack---Stack of Object Pointers
 - Set---Unordered Collection of Non-Duplicate Objects
 - Dictionary---Set of Associations
 - IdentDict---Dictionary Keyed by Object Address
 - IdentSet---Set Keyed by Object Address
 - Date---Gregorian Calendar Date
 - FDSet---Set of File Descriptors for Use with select(2) System Call

APPENDIX A. THE NIH C++ CLASS LIBRARY HIERARCHY

Float---Floating Point Number
Fraction---Rational Arithmetic
Integer---Integer Number Object
Iterator---Collection Iterator
Link---Abstract Class for LinkedList Links
 LinkOb---Link Containing Object Pointer
Process---Co-routine Process Object
 HeapProc---Process with Stack in Free Store
 StackProc---Process with Stack on main() Stack
LookupKey---Abstract Class for Dictionary Associations
 Assoc---Association of Object Pointers
 AssocInt---Association of Object Pointer with Integer
Nil---The Nil Object
Point---X-Y Coordinate Pair
Random---Random Number Generator
Range---Range of Integers
Rectangle---Rectangle Object
Scheduler---Co-routine Process Scheduler
Semaphore---Process Synchronization
SharedQueue---Shared Queue of Objects
String---Character String
 Regex---Regular Expression
Time---Time of Day
Vector---Abstract Class for Vectors
 BitVec---Bit Vector
 ByteVec---Byte Vector
 ShortVec---Short Integer Vector
 IntVec---Integer Vector
 LongVec---Long Integer Vector
 FloatVec---Floating Point Vector
 DoubleVec---Double-Precision Floating Point Vector

APPENDIX A. THE NIH C++ CLASS LIBRARY HIERARCHY

OIOifd---File Descriptor Object I/O readFrom() Formatting
OIOin---Abstract Class for Object I/O readFrom() Formatting
 OIOistream---Abstract Class for Stream Object I/O readFrom()
 Formatting
 OIONihin---Stream Object I/O readFrom() Formatting
OIOofd---File Descriptor Object I/O storeOn() Formatting
OIOout---Abstract Class for Object I/O storeOn() Formatting
 OIOostream---Abstract Class for Stream Object I/O storeOn()
 Formatting
 OIONihout---Stream Object I/O storeOn() Formatting
ReadFromTbl---Tables used by Object I/O readFrom()
StoreOnTbl---Tables used by Object I/O storeOn()

Appendix B

The NIH C++ Collections

This Appendix is comprised of comments taken from the NIH Class Library Implementations [13].

Collection---Abstract Class for Collections

 Arraychar---Byte Array

 ArrayOb---Array of Object Pointers

 Bag---Unordered Collection of Objects

 SeqCltn---Abstract Class for Ordered, Indexed Collections

 Heap---Min-Max Heap of Object Pointers

 LinkedList---Singly-Linked List

 OrderedCltn---Ordered Collection of Object Pointers

 SortedCltn---Sorted Collection

 KeySortCltn---Keyed Sorted Collection

 Stack---Stack of Object Pointers

 Set---Unordered Collection of Non-Duplicate Objects

 Dictionary---Set of Associations

 IdentDict---Dictionary Keyed by Object Address

 IdentSet---Set Keyed by Object Address

APPENDIX B. THE NIH C++ COLLECTIONS

Collection.c – implementation of abstract Collection class

Collection is an abstract class that actually implements only the addAll, removeAll, includes, isEmpty, and Collection conversion functions. Note that the functions Collection::asBag, asOrderedCltn, asSet, and asSortedCltn are defined in the file that implements the respective target Collection so that all of these classes are not loaded whenever any one Collection is used.

ArrayOb.c – member functions of class ArrayOb

Member function definitions for class ArrayOb (Array of Object*). Objects of class ArrayOb are used in the implementations of several other Collection classes such as: Bag, Dictionary, Set, and OrderedCltn. Note that the ArrayOb constructor initializes the array with pointers to the nil object.

Bag.c – implementation of a Set of Objects with possible duplicates

A Bag is like a Set, except Bags can contain multiple occurrences of equal objects. Bags are implemented by using a Dictionary to associate each object in the Bag with its number of occurrences.

SeqCltn.c – implementation of abstract sequential collections

SeqCltn is an abstract class representing collections whose elements are ordered and are externally named by integer indices.

Heap.c – implementation of abstract Heap class

The Min-Max Heap is implemented as described by Atkinson, Sack, Santoro, and Strothotte (1986). Objects may be added; the min or max may be accessed with first() or last(), respectively, or removed with removeFirst() or removeLast(), respectively.

APPENDIX B. THE NIH C++ COLLECTIONS

LinkedList.c – implementation of singly-linked list

LinkedLists are ordered by the sequence in which objects are added and removed from them. Object elements are accessible by index.

OrderedCltn.c – implementation of abstract ordered collections

OrderedCltns are ordered by the sequence in which objects are added and removed from them. Object elements are accessible by index.

SortedCltn.c – implementation of sorted collection

A SortedCltn is a Collection of objects ordered as defined by the virtual function "compare", which the objects must implement. The "add" function locates the position at which to insert the object by performing a binary search, then invokes the private function "OrderedCltn::addAtIndex" to insert the object after shifting up all the objects after it in the array; therefore, a SortedCltn is not efficient for a large number of objects.

Stack.c – implementation of class Stack

Member function definitions for class Stack.

Set.c – implemenation of hash tables

A Set is an unordered collection of objects in which no object is duplicated. Duplicate objects are defined by the function isEqual. Sets are implemented using a hash table. The capacity() function returns the 1/2 the capacity of the hash table and the size() function returns the number of objects currently in the Set. For efficiency, the capacity is always a power of two and is doubled whenever the table becomes half full.

Dictionary.c – implementation of Set of Associations

A Dictionary is a Set of Associations. A Dictionary returns the value of an association given its key.

APPENDIX B. THE NIH C++ COLLECTIONS

IdentDict.c – implementation of Identifier Dictionary

An IdentDict is like a Dictionary, except keys are compared using `isSame()` rather than `isEqual()`.

IdentSet.c – implementation of Identity Set

An IdentSet is like a Set, except keys are compared using `isSame()` rather than `isEqual()`.

LookupKey.c – implementation of Dictionary LookupKey

LookupKey is an abstract class for managing the key object of an Assoc. It is used to implement class Dictionary.

Assoc.c – implementation of key-value association

Objects of class Assoc associate a key object with a value object. They are used to implement Dictionaries, which are Sets of Associations.

AssocInt.c – implementation of key-Integer association

Objects of class AssocInt associate a key object with an Integer value object. They are used to implement Bags, which use a Dictionary to associate objects with their occurrence counts.

Appendix C

NIH Template header file

```
#ifndef THIS_CLASS_H
#define THIS_CLASS_H

/*$Header: ... */

/* Template.h -- example header file for an NIH Library class
```

THIS SOFTWARE FITS THE DESCRIPTION IN THE U.S. COPYRIGHT ACT OF A "UNITED STATES GOVERNMENT WORK". IT WAS WRITTEN AS A PART OF THE AUTHOR'S OFFICIAL DUTIES AS A GOVERNMENT EMPLOYEE. THIS MEANS IT CANNOT BE COPYRIGHTED. THIS SOFTWARE IS FREELY AVAILABLE TO THE PUBLIC FOR USE WITHOUT A COPYRIGHT NOTICE, AND THERE ARE NO RESTRICTIONS ON ITS USE, NOW OR SUBSEQUENTLY.

Author:
K. E. Gorlen
Computer Systems Laboratory, DCRT
National Institutes of Health
Bethesda, MD 20892

Modification History:

```
$Log: Template_h,v $
# Revision 3.0 90/05/20 00:21:43 kgorlen
# Release for 1st edition.
#
```

APPENDIX C. NIH TEMPLATE HEADER FILE

```
*/  
  
// Define "MI" if this class uses multiple inheritance:  
//#ifndef MI  
//#define MI  
//#endif  
  
#include "BASE_CLASS.h"  
// #include .h files for other classes used  
// Insert only class declarations for classes accessed  
// by pointer and reference ONLY  
  
// If BASE_CLASS is Object:  
// class THIS_CLASS: public VIRTUAL Object {  
  
    class THIS_CLASS: public BASE_CLASS {  
        DECLARE_MEMBERS(THIS_CLASS);  
        // member variables here  
        protected: // storer() functions for object I/O  
        virtual void storer(OIOofd&) const;  
        virtual void storer(OIOut&) const;  
        public:  
        bool operator==(const THIS_CLASS&) const;  
        bool operator!=(const THIS_CLASS& a) const  
        { return !(*this==a); }  
        virtual int compare(const Object&) const;  
        virtual Object* copy() const; // shallowCopy() default  
                                    // if not defined  
        virtual void deepenShallowCopy();  
        virtual unsigned hash() const;  
        virtual bool isEqual(const Object&) const;  
        virtual void printOn(ostream& strm = cout) const;  
        virtual const Class* species() const;  
    };  
  
    #endif
```

Appendix D

NIH Template implementation file

/* Template.c -- example implementation of an NIH Library class

THIS SOFTWARE FITS THE DESCRIPTION IN THE U.S. COPYRIGHT ACT OF A "UNITED STATES GOVERNMENT WORK". IT WAS WRITTEN AS A PART OF THE AUTHOR'S OFFICIAL DUTIES AS A GOVERNMENT EMPLOYEE. THIS MEANS IT CANNOT BE COPYRIGHTED. THIS SOFTWARE IS FREELY AVAILABLE TO THE PUBLIC FOR USE WITHOUT A COPYRIGHT NOTICE, AND THERE ARE NO RESTRICTIONS ON ITS USE, NOW OR SUBSEQUENTLY.

Author:

K. E. Gorlen
Bg. 12A, Rm. 2033
Computer Systems Laboratory
Division of Computer Research and Technology
National Institutes of Health
Bethesda, Maryland 20892
Phone: (301) 496-1111
uucp: uunet!nih-csl!kgorlen
Internet: kgorlen@alw.nih.gov
February, 1987

Function:

Modification History:

\$Log: Template_c,v \$

APPENDIX D. NIH TEMPLATE IMPLEMENTATION FILE

```
# Revision 3.0 90/05/20 00:21:40 kgorlen
# Release for 1st edition.
#
*/

#include "THIS_CLASS.h"
#include "nihclIO.h"
// #include .h files for other classes used

#define THIS THIS_CLASS
// Define BASE only for classes with one base class
#define BASE BASE_CLASS
// Define list of addresses of descriptors of all base classes:
#define BASE_CLASSES BASE::desc()
// Define list of addresses of descriptors of all member classes:
#define MEMBER_CLASSES
// Define list of addresses of descriptors of all virtual base
// classes:
#define VIRTUAL_BASE_CLASSES

DEFINE_CLASS(THIS_CLASS,1,"$Header: ...",NULL,NULL);
// For abstract classes:
//DEFINE_ABSTRACT_CLASS(THIS_CLASS,1,"...",NULL,NULL);
// For non-abstract classes with multiple base classes:
//DEFINE_CLASS_MI(THIS_CLASS,1,"...",NULL,NULL);
// For abstract classes with multiple base classes:
//DEFINE_ABSTRACT_CLASS_MI(THIS_CLASS,1,"...",NULL,NULL);

extern const int // error codes

/* _castdown() for classes with multiple base classes:

void* THIS_CLASS::_castdown(const Class& target) const
// (Probably a good candidate for memorization.)
{
    if (&target == desc()) return (void*)this;
    void* p = BASE1::_castdown(target);
    void* q = p;
    if (p = BASE2::_castdown(target)) ambigCheck(p,q,target);
    // ...
    if (p = BASEn::_castdown(target)) ambigCheck(p,q,target);
}
```

APPENDIX D. NIH TEMPLATE IMPLEMENTATION FILE

```
return q;
}

*/
bool THIS_CLASS::operator==(const THIS_CLASS& a) const
// Test two instances of THIS_CLASS for equality
{
}

const Class* THIS_CLASS::species() const
// Return a pointer to the descriptor of the species of this class
{
return &classDesc;
}

bool THIS_CLASS::isEqual(const Object& p) const
// Test two objects for equality
{
return p.isSpecies(classDesc) && *this==castdown(p);
}

unsigned THIS_CLASS::hash() const
// If two objects are equal (i.e., isEqual) they must have
// the same hash
{
}

int THIS_CLASS::compare(const Object& p) const
// Compare two objects.  If *this > p return >0,
// *this == p return 0, and if *this < p return <0.
{
assertArgSpecies(p, classDesc, "compare");
}

void THIS_CLASS::deepenShallowCopy()
// Called by deepCopy() to convert a shallow copy to a deep copy.
// deepCopy() makes the shallow copy by calling the copy
// constructor.
{
/*

```

APPENDIX D. NIH TEMPLATE IMPLEMENTATION FILE

Deepen base classes in order specified in class declaration.

Deepen virtual base classes (VBase):

```
VBase::deepenVBase(); // do not do this for class Object
```

Deepen non-virtual base classes (BASE):

```
BASE::deepenShallowCopy(); // do not do this for class Object
```

Nothing need be done for member variables that are fundamental types. Copy a member variable o that is an NIHCL object:
o.deepenShallowCopy();

Copy a member variable p that is a pointer to an NIHCL object of class CLASS:

```
p = (CLASS*)p->deepCopy();
*/
}
```

```
void THIS_CLASS::printOn(ostream& strm) const
// Print this object on an ostream
{
}
```

// Object I/O

```
/*
Member class instances are constructed in the order they are
declared in the class declaration, regardless of the order they
appear in the constructor initialization list, so they must be
stored in this order. Note that member class instances are
constructed before body of constructor is executed.
*/
```

// Construct an object from OIOin "strm".

```
THIS_CLASS::THIS_CLASS(OIOin& strm)
```

```
:
```

```
#ifdef MI
Object(strm),
#endif
/*
```

Call readFrom() constructors of all ancestor virtual base classes:

APPENDIX D. NIH TEMPLATE IMPLEMENTATION FILE

```
VBase(strm),
*/
BASE(strm)
/*
Read a member variable o that is an instance of an NIHCL class:
o(strm)
{
Read a member variable f that is a fundamental type using ">>":
strm >> f;

Read a member variable p that is a pointer to an instance of
the NIHCL class CLASS:
p = CLASS::readFrom(strm);

Read member variables in the same order that they are stored.
*/
}

void THIS_CLASS::storer(OI0out& strm) const
// Store the member variables of this object on OI0out "strm".
{
/*
Store virtual base classes (VBase) in inheritance DAG order:
VBase::storeVBaseOn(strm);

Store non-virtual base classes in order specified in class
declaration:
BASE::storer(strm);

Store a member variable f that is a fundamental type using "<<":
strm << f;

Store a member variable o that is an instance of the NIHCL class
CLASS:
o.storeMemberOn(strm);

Store a member variable p that is a pointer to an instance of an
NIHCL class:
p->storeOn(strm);

Store member variables in the same order that they are read.
```

APPENDIX D. NIH TEMPLATE IMPLEMENTATION FILE

```
*/  
}  
  
// Construct an object from file descriptor "fd".  
THIS_CLASS::THIS_CLASS(OIOfd& fd)  
:  
#ifdef MI  
Object(fd),  
#endif  
/*  
Call readFrom() constructors of all ancestor virtual base classes:  
VBase(fd),  
*/  
BASE(fd)  
/*  
Read a member variable o that is an instance of an NIHCL class:  
o(fd)  
{  
Read a member variable f that is a fundamental type:  
fd >> f;  
  
Read a member variable a that is a pointer to an array of length l:  
fd.get(a,l);  
  
Read a member variable p that is a pointer to an instance of the  
NIHCL class CLASS:  
p = CLASS::readFrom(fd);  
  
Read member variables in the same order that they are stored.  
*/  
}  
  
void THIS_CLASS::storer(OIOfd& fd) const  
// Store an object on file descriptor "fd".  
{  
/*  
Store virtual base classes (VBase) in inheritance DAG order:  
VBase::storeVBaseOn(fd);  
  
Store non-virtual base classes in order specified in class  
declaration:
```

APPENDIX D. NIH TEMPLATE IMPLEMENTATION FILE

BASE::storer(fd);

Store a member variable f that is a fundamental type:
fd << f;

Store a member variable a that is a pointer to an array
of length l:
fd.put(a,l);

Store a member variable o that is an instance of the NIHCL class
CLASS:

o.storeMemberOn(fd);

Store a member variable p that is a pointer to an instance of an
NIHCL class:
p->storeOn(fd);

Store member variables in the same order that they are read.

*/

}

Appendix E

Complete PF Header File (PF.h)

```
#ifndef PF_H
#define PF_H

#include "GridLine.h" // break the include nesting tree length

#include "Manager.h"

class xGridLine {
public:
    char* label;
    OrientType orient;
    double distance;

    xGridLine(char*, OrientType, double);
    xGridLine(GridLine&);

};

class xGenericPoint {
public:
    char* gA;
    char* gB;
    char* g1;
    char* g2;
    double x_st, x_sp, y_st, y_sp;
    int x_rl, y_rl;
```

APPENDIX E. COMPLETE PF HEADER FILE (PF.H)

```
xGenericPoint();
xGenericPoint(char*, char*, char*, char*,
              double xst =0, double xsp =0, double yst =0, double ysp =0,
              int xr =1, int yr =1);
xGenericPoint(GenericPoint&);
xGenericPoint(xGenericPoint* );
};

class xCompass {
public:
    CompassType dir;
    double val;

    xCompass();
    xCompass(CompassType, double);
    xCompass(xCompass* );
    xCompass(Compass&);
};

class xLoad {
public:
    char* name;
    NominalLoadType nominalLoadCategory;
    xGenericPoint* linePoint;
    double mag1, mag2;
    xCompass* direction;
    SurfaceType supportSurface;

    xLoad();
    xLoad(char*, NominalLoadType, xGenericPoint* ,
          double, double, xCompass*, SurfaceType);
    xLoad(Load&);
};

class xClearSpace : public xGenericPoint {
public:
    char* label;
    char* tag;

    xClearSpace();
    xClearSpace(char*, char*, xGenericPoint* );
};
```

APPENDIX E. COMPLETE PF HEADER FILE (PF.H)

```
xClearSpace(ClearSpace&);  
};  
  
class xOffset {  
public:  
    char* key;  
    char* grid;  
    char* bound1;  
    char* bound2;  
    double val;  
    SurfaceType offsetSurface;  
  
    xOffset();  
    xOffset(char*, char*, char*, char*, double, SurfaceType);  
    xOffset(Offset&);  
};  
  
class xRelPos {  
public:  
    char* key;  
    char* grid1;  
    char* grid2;  
    char* extentStart;  
    char* extentStop;  
    SurfaceType grid1Sur, grid2Sur;  
    int minmax;  
    double lessThan, greaterThan;  
    double equalTo;  
  
    xRelPos();  
    xRelPos(char*, char*, char*, char*, char*, SurfaceType, SurfaceType, int,  
           double, double, double);  
    xRelPos(RelPos&);  
};  
  
extern double toDouble(Float&);  
extern char* toChar(String&);  
  
class PF {  
    int dirtyHorizGrid;  
    int dirtyVertGrid;
```

APPENDIX E. COMPLETE PF HEADER FILE (PF.H)

Manager aManager;

public:

```
    GridLine& convertGridLine(xGridLine&);  
    GenericPoint& convertGenericPoint(xGenericPoint*);  
    Compass& convertCompass(xCompass*);  
    Load& convertLoad(xLoad*);  
    ClearSpace& convertClearSpace(xClearSpace*);  
    Offset& convertOffset(xOffset*);  
    RelPos& convertRelPos(xRelPos*);  
  
    xGenericPoint* getIntersectionPoint(int, double, double);  
    xGenericPoint* getLinePoint(int, double, double, double, double);  
    xGenericPoint* getAreaPoint(int, double, double, double, double);  
  
    xGenericPoint* getIntersectionPoint_fixed(double, double);  
    xGenericPoint* getLinePoint_fixed(double, double, double, double);  
    xGenericPoint* getAreaPoint_fixed(double, double, double, double);  
  
    xGenericPoint* getIntersectionPoint_rel(double, double);  
    xGenericPoint* getLinePoint_rel(double, double, double, double);  
    xGenericPoint* getAreaPoint_rel(double, double, double, double);  
  
    loadType xLoadType(xLoad*);  
  
    void init();  
    void clear();  
    int openFile(const char*);  
    int saveFile(const char*);  
  
    int holdsGridLabel(const char*);  
    void initGrid();  
    void initHorizGrid();  
    void initVertGrid();  
    int numberOfHoriz();  
    int numberOfVert();  
    int number_of_Grids();  
    int getNextHorizGrid(xGridLine&);  
    int getNextVertGrid(xGridLine&);  
    int getNextGrid(xGridLine&);  
    xGridLine* getGridWithLabel(const char*);
```

APPENDIX E. COMPLETE PF HEADER FILE (PF.H)

```
char* getHorizGridLineBefore(double);
char* getVertGridLineBefore(double);
char* getHorizGridLineAfter(double);
char* getVertGridLineAfter(double);
ManagerErrors storeGrid(xGridLine&);
ManagerErrors removeGrid(const char* );
ManagerErrors changeGrid(const char*, xGridLine&);

void getOrigin(double&,double&);
int setOrigin(double,double);
void getCorner(double&,double&);
int setCorner(double,double);

void getPoint(xGenericPoint*, double&, double&);
void getLinePoint(xGenericPoint*, double&, double&, double&, double&);
void getOrigin(xGenericPoint*, double&, double&);
void getCorner(xGenericPoint*, double&, double&);

xLoad* getLoadWithLabel(const char* );
ManagerErrors addLoadEvent(const char* ); // (LE_label)
ManagerErrors addLoadEvent(const char*, const char* );
//(LE_parent,LE_child)
ManagerErrors addLoadList(const char*, const char* ); // (LE_label, LL_label)
ManagerErrors addLoad(const char*, const char*, xLoad* ); // (LE, LL, L)
ManagerErrors removeLoadEvent(const char* ); // (LE_label)
ManagerErrors removeLoadEvent(const char*, const char* );
//(LE_par,LE_child)
ManagerErrors removeLoadList(const char*, const char* ); // (LE_lab, LL_lab)
ManagerErrors removeLoad(const char*, const char*, const char* );
ManagerErrors changeLoad(const char*, xLoad* );
int getFirstLoad(char*, char*&, xLoad&); // (LE, XX, X) -> (LE, LL, L)
int getNextLoad(char*, char*&, xLoad&); // last (LE,LL,L) -> next (LE,LL,L)

int numberOfLoads(); // in all LEs
int numberOfLoadLists(); // in all LEs
int holdsLoadWithLabel(const char* ); // in all LEs
int holdsLoadListWithLabel(const char* ); // in all LEs

int holdsGeometricWithLabel(const char* );
int numberOfGeometrics();
```

APPENDIX E. COMPLETE PF HEADER FILE (PF.H)

```
xClearSpace* getClearSpaceWithLabel(const char*);  
ManagerErrors storeClearSpace(xClearSpace*);  
ManagerErrors removeClearSpace(const char*);  
ManagerErrors changeClearSpace(const char*, xClearSpace*);  
int getFirstClearSpace(xClearSpace&);  
int getNextClearSpace(xClearSpace&);  
  
ManagerErrors storeOffset(xOffset*);  
ManagerErrors removeOffsetWithKey(const char*);  
ManagerErrors changeOffset(const char*, xOffset*);  
xOffset* getOffsetWithKey(const char*);  
int getFirstOffset(xOffset&);  
int getNextOffset(xOffset&);  
  
ManagerErrors storeRelPos(xRelPos*);  
ManagerErrors removeRelPosWithKey(const char*);  
ManagerErrors changeRelPos(const char*, xRelPos*);  
xRelPos* getRelPosWithKey(const char*);  
int getFirstRelPos(xRelPos&);  
int getNextRelPos(xRelPos&);  
};  
  
#endif
```

Appendix F

Stripped PF Header File (PF.g)

```
#ifndef PF_G
#define PF_G

#include "enum_types.h"

class xGridLine {
public:
    char* label;
    OrientType orient;
    double distance;

    xGridLine(char*, OrientType, double);
};

class xGenericPoint {
public:
    char* gA;
    char* gB;
    char* g1;
    char* g2;
    double x_st, x_sp, y_st, y_sp;
    int x_rl, y_rl;

    xGenericPoint();
    xGenericPoint(char*, char*, char*, char*,
                  double xst =0, double xsp =0, double yst =0, double ysp =0,
```

APPENDIX F. STRIPPED PF HEADER FILE (PF.G)

```
        int xr =1, int yr =1);
xGenericPoint(xGenericPoint* );
};

class xCompass {
public:
    CompassType dir;
    double val;

    xCompass();
    xCompass(CompassType, double);
    xCompass(xCompass* );
};

class xLoad {
public:
    char* name;
    NominalLoadType nominalLoadCategory;
    xGenericPoint* linePoint;
    double mag1, mag2;
    xCompass* direction;
    SurfaceType supportSurface;

    xLoad();
    xLoad(char*, NominalLoadType, xGenericPoint* ,
          double, double, xCompass*, SurfaceType);
};

class xClearSpace : public xGenericPoint {
public:
    char* label;
    char* tag;

    xClearSpace();
    xClearSpace(char*, char*, xGenericPoint* );
};

class xOffset {
public:
    char* key;
    char* grid;
};
```

APPENDIX F. STRIPPED PF HEADER FILE (PF.G)

```
char* bound1;
char* bound2;
double val;
SurfaceType offsetSurface;

xOffset();
xOffset(char*, char*, char*, char*, double, SurfaceType);
};

class xRelPos {
public:
    char* key;
    char* grid1;
    char* grid2;
    char* extentStart;
    char* extentStop;
    SurfaceType grid1Sur, grid2Sur;
    int minmax;
    double lessThan, greaterThan;
    double equalTo;

    xRelPos();
    xRelPos(char*, char*, char*, char*, char*, SurfaceType, SurfaceType, int,
             double, double, double);
};

class PF {
public:
    void init();
    void clear();

    xGenericPoint* getIntersectionPoint(int, double, double);
    xGenericPoint* getLinePoint(int, double, double, double, double);
    xGenericPoint* getAreaPoint(int, double, double, double, double);

    xGenericPoint* getIntersectionPoint_fixed(double, double);
    xGenericPoint* getLinePoint_fixed(double, double, double, double);
    xGenericPoint* getAreaPoint_fixed(double, double, double, double);

    xGenericPoint* getIntersectionPoint_rel(double, double);
    xGenericPoint* getLinePoint_rel(double, double, double, double);
```

APPENDIX F. STRIPPED PF HEADER FILE (PF.G)

```
xGenericPoint* getAreaPoint_rel(double, double, double, double);

loadType xLoadType(xLoad*);
int holdsGridLabel(const char*);
int openFile(const char*);
int saveFile(const char*);
void initGrid();
void initHorizGrid();
void initVertGrid();
int numberOfHoriz();
int numberOfVert();
int numberOfGrids();
int getNextHorizGrid(xGridLine&);
int getNextVertGrid(xGridLine&);
int getNextGrid(xGridLine&);
xGridLine* getGridWithLabel(const char*);
char* getHorizGridLineBefore(double);
char* getVertGridLineBefore(double);
char* getHorizGridLineAfter(double);
char* getVertGridLineAfter(double);
ManagerErrors storeGrid(xGridLine&);
ManagerErrors removeGrid(const char*);
ManagerErrors changeGrid(const char*, xGridLine&);

void getOrigin(double&,double&);
int setOrigin(double,double);
void getCorner(double&,double&);
int setCorner(double,double);

void getPoint(xGenericPoint*, double&, double&,
             double&, double&,
             double&, double&,
             double&,
             double&);

xLoad* getLoadWithLabel(const char*);
ManagerErrors addLoadEvent(const char*); // (LE_label)
ManagerErrors addLoadEvent(const char*, const char*);  
// (LE_parent,LE_child)
ManagerErrors addLoadList(const char*, const char*); // (LE_label, LL_label)
ManagerErrors addLoad(const char*, const char*, xLoad*); // (LE, LL, L)
ManagerErrors removeLoadEvent(const char*); // (LE_label)
```

APPENDIX F. STRIPPED PF HEADER FILE (PF.G)

```
ManagerErrors removeLoadEvent(const char*, const char*);  
//(LE_par,LE_child)  
ManagerErrors removeLoadList(const char*, const char*); // (LE_lab, LL_lab)  
ManagerErrors removeLoad(const char*, const char*, const char*);  
ManagerErrors changeLoad(const char*, xLoad*);  
int getFirstLoad(char*, char*&, xLoad&); // (LE, XX, X) -> (LE, LL, L)  
int getNextLoad(char*, char*&, xLoad&); // last (LE,LL,L) -> next (LE,LL,L)  
  
int numberOfLoads(); // in all LEs  
int numberOfLoadLists(); // in all LEs  
int holdsLoadWithLabel(const char*); // in all LEs  
int holdsLoadListWithLabel(const char*); // in all LEs  
  
int holdsGeometricWithLabel(const char*);  
int numberOfGeometrics();  
  
xClearSpace* getClearSpaceWithLabel(const char*);  
ManagerErrors storeClearSpace(xClearSpace*);  
ManagerErrors removeClearSpace(const char*);  
ManagerErrors changeClearSpace(const char*, xClearSpace*);  
int getFirstClearSpace(xClearSpace&);  
int getNextClearSpace(xClearSpace&);  
  
ManagerErrors storeOffset(xOffset*);  
ManagerErrors removeOffsetWithKey(const char*);  
ManagerErrors changeOffset(const char*, xOffset*);  
xOffset* getOffsetWithKey(const char*);  
int getFirstOffset(xOffset&);  
int getNextOffset(xOffset&);  
  
ManagerErrors storeRelPos(xRelPos*);  
ManagerErrors removeRelPosWithKey(const char*);  
ManagerErrors changeRelPos(const char*, xRelPos*);  
xRelPos* getRelPosWithKey(const char*);  
int getFirstRelPos(xRelPos&);  
int getNextRelPos(xRelPos&);  
};  
#endif
```

Appendix G

ParSim Definition Language Overview

Processor Description

N_LABEL **Proc_Time** {**C|S**}

Each node in the algorithmic representation has a distinguishing label, namely **N_LABEL**. A node has a real time, **Proc_Time**, associated with computation for a single data block. Two modes of operation exist for a node, either sequential computation followed by a communications phase, or concurrent computation and communication. A flag can be set where

C = Compute first, then transmit,

S = Simultaneous Computations and Communications.

Link Description

L_LABEL **N_FROM** **N_TO** **LENGTH** {**LINK_WEIGHT**}

describes a directed link from the node **N_FROM** to node **N_TO**. The **LENGTH** of a link describes the capacity of a link. The relative delay, **LINK_WEIGHT**, in communicating a message along the link can be manually assigned. (See **.LINKWTS** in Compiler Directives)

APPENDIX G. PARSIM DEFINITION LANGUAGE OVERVIEW

Compiler Directives

.MAP {ON|AUTO|END}

The **.MAP** directive begins the mapping mode until map mode end, **.MAP END**, is reached.

The directive **.MAP AUTO** causes the simulator to create the mapping.

The default of **.MAP** is **.MAP ON** and if no **.MAP** directive is given, **.MAP AUTO** is assumed.

Example:

```
.MAP
    N1      P3
    N3      P7
    N4      P5
    N6      P1
.MAP END
```

.LINKWTS {AUTO}

This directive will weight each link by a constant proportional to the mapped PE's shortest path length. Default of **.LINKWTS** is **.LINKWTS AUTO**. If no directive is given, the simulator will prompt the user for Link Weights.

APPENDIX G. PARSIM DEFINITION LANGUAGE OVERVIEW

.ARCH ArchNum Number_of_PE's

Create a destination architecture with connectivity given in the following table for a given Number_of_PE's:

ArchNum	Connectivity	P(i) \iff P(j) if:
1	Linear Array	P(i) next to P(j) numerically
2	Ring	P(i) and P(j) next to each other modulus n
3	Star	All PE's connected to P(0)
4	Binary Tree	Single Rooted Binary Tree
5	Illiad Mesh	Wrap-around mesh with dimension 4
6	Cylindrical	Systolic Array
7	Completely Connected	All PE's connected to All other PE's
8	Chordal Ring	Ring w/ added connections of chords = \sqrt{n}
9	HyperCube	$(i \oplus j) = 1$
10	Barrel Shifter	$j = i \pm 2^k \text{ mod } n$
11	$\pm 1, \pm 3, \pm 5$ Ring	Chordal ring w/ odd near-nbr connections

.END

Label to mark the end of an input file.

{Statement} ;COMMENTS

Commenting on a statement line.

* COMMENTS

Commenting on a separate line.

Bibliography

- [1] Selim G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1989.
- [2] David A. Bader. PARSIM: A simulator for designing parallel algorithms and architectures. Technical Report CSEE-TR-90-07, Lehigh University, Bethlehem, Pa., 1990.
- [3] Alan A. Bertocci and Maurizio A. Bonuccelli. Some parallel algorithms on interval graphs. *Discrete Applied Mathematics for combinatorial operations research*, 16(2):101–111, February 1987.
- [4] Gilles Brassard and Paul Bratley. *Algorithms: Theory and Practice*. Prentice-Hall, Englewood Cliffs, NJ, 1988.
- [5] Thomas Braunl. A specification language for parallel architectures and algorithms. In *Fifth International Workshop on Software Specifications and Design*, pages 49–51, Pittsburgh, Pennsylvania, May 1989.
- [6] Marina C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, 3(4):461–491, December 1986.
- [7] Brad J. Cox. *Object Oriented Programming: An Evolutionary Approach*. Addison-Wesley, Reading, Massachusetts, 1986.
- [8] Kshitij A. Doshi and Peter J. Varman. Optimal graph algorithms on a fixed-size linear array. *IEEE Transactions on Computers*, C-36(4):460–470, April 1987.

BIBLIOGRAPHY

- [9] Tse-yun Feng. A survey of interconnection networks. *Computer*, 14(12):12–27, December 1981.
- [10] Joydeep Ghosh and Kai Hwang. Mapping neural networks onto message-passing multicomputers. *Journal of Parallel and Distributed Computing*, 6(2):291–330, April 1989.
- [11] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, Massachusetts, 1984.
- [12] Keith E. Gorlen. An object-oriented class library for C++ programs. *Software - Practice and Experience*, 17(12):899–922, December 1987.
- [13] Keith E. Gorlen. *NIH Class Library Reference Manual*. National Institutes of Health, Bethesda, Maryland, revision 3.10 DRAFT edition, April 1990.
- [14] Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley & Sons Ltd., West Sussex, England, October 1990.
- [15] Andrew Grimshaw. Mentat: A computation model for parallel software problems. Technical report, University of Virginia, 1991.
- [16] Kai Hwang and Faye A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill Book Company, New York, 1984.
- [17] Kai Hwang, Ping-Sheng Tseng, and Dongseung Kim. An orthogonal multiprocessor for parallel and scientific computations. *IEEE Transactions on Computers*, 38(1):47–61, January 1989.
- [18] Borland International. *Turbo C++ Reference Manual*. Scotts Valley, California, 1990.
- [19] Yehuda E. Kalay. Worldview: An integrated geometric-modeling/drafting system. *IEEE Computer Graphics and Applications*, 7(2):36–46, February 1987.

BIBLIOGRAPHY

- [20] Brian M. Kennedy. The features of the object-oriented abstract type hierarchy (OATH). Technical report, Computer Science Center, Texas Instruments, August 1991.
- [21] Myron W. Krueger. *Artificial Reality II*. Addison-Wesley, Reading, Massachusetts, 1991.
- [22] Wilf R. LaLonde and John R. Pugh. *Inside Smalltalk*, volume 1. Prentice Hall, Englewood Cliffs, NJ, 1990.
- [23] Mark A. Linton, Paul R. Calder, and John M. Vlissides. *InterViews Reference Manual*. Stanford University, Ver. 3.0 draft edition, March 1991.
- [24] G. Jack Lipovski and Miroslaw Malek, editors. *Parallel Computing: Theory and Comparisons*. John Wiley & Sons Ltd., New York, 1987.
- [25] Stanley B. Lippman. *C++ Primer 2nd Edition*. Addison-Wesley, Reading, Massachusetts, 1991.
- [26] Kirk Martini and Graham H. Powell. Geometric modeling requirements for structured design. *Engineering with Computers*, 6(2):93–102, April 1990.
- [27] R. D. McLeod and J. J. Schnellenberg. Percolation and anomalous transport as tools in analyzing parallel processing interconnection networks. *Journal of Parallel and Distributed Computing*, 8(4):376–387, April 1990.
- [28] Thanasis Mitsolides and Malcolm Harrison. Generators and the replicator control structure in the parallel environment of ALLOY. In *ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 189–196, White Plains, NY, June 1990.
- [29] Michael J. Quinn and Narsingh Deo. Parallel graph algorithms. *Computing Surveys*, 16(3):319–348, September 1984.
- [30] Richard Sause. *A Model of the Design Process for Computer Integrated Structural Engineering*. Ph.D. Dissertation, University of California, Berkeley, 1989.

BIBLIOGRAPHY

- [31] Richard Sause. Towards management of design alternatives in object oriented databases. Lehigh University, 1991.
- [32] Richard Sause and Graham H. Powell. A design process model for computer integrated structural engineering. *Engineering with Computers*, 6(3):129–143, July 1990.
- [33] Richard Sause and Graham H. Powell. A design process model for computer integrated structural engineering: Design phases and tasks. *Engineering with Computers*, 1990. (In Press).
- [34] David B. Skillcorn and William Kocay. A global measure of network connectivity. *Journal of Parallel and Distributed Computing*, 7(1):165–177, August 1989.
- [35] Dan Stenger. C++ object-oriented library (COOL). Technical report, Information Technology Group, Texas Instruments, September 1991.
- [36] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 1986.
- [37] John M. Vlissides and Mark A. Linton. Applying object-oriented design to structured graphics. In *Proceedings of the USENIX C++ Conference*, Denver, Colorado, October 1988.
- [38] Chuan-lin Wu and Tse-yun Feng, editors. *Tutorial: Interconnection Networks for parallel and distributed processing*. IEEE Computer Society Press, Silver Spring, MD, 1984.
- [39] Zhiwei Xu and Kai Hwang. Molecule: A language construct for layered development of parallel programs. *IEEE Transactions on Software Engineering*, 15(5):587–599, May 1989.

Vita

David A. Bader was born a 4 pound, 4 ounce baby boy at 4:44 AM on May 4, 1969, in Bethlehem, Pennsylvania, to his proud parents, Dr. Morris Bader and Sophie Karen Bader, of Bethlehem, Pa. In June 1987, David graduated from Liberty High School, Bethlehem, Pa., and in June 1990, graduated Magna Cum Laude from Lehigh University, Bethlehem, Pa., with his Bachelor of Science in Computer Engineering.

David was awarded an Honorable Mention in the 1991 National Science Foundation (NSF) Graduate Fellowship search. In June 1990, David received the Elizabeth Major Nevius Award from Lehigh University for outstanding leadership, scholarship, and citizenship. He has received an ATLSS Center Graduate Research Fellowship from Lehigh University for 1990 - 91. David is a member of Tau Beta Pi, the National Engineering Honor Society, and Eta Kappa Nu, the Electrical Engineering Honor Society. He has earned the rank of Eagle Scout in the Boy Scouts of America, and has attained the Vigil Honor in the Order of the Arrow, B.S.A. David has also received a Morning Call Newspapers Scholarship, is listed in Who's Who Among American Universities for 1990, and is a National Merit Commended Student. Currently, he is a student member of IEEE and the IEEE Computer Society.

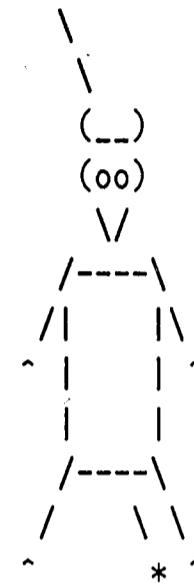
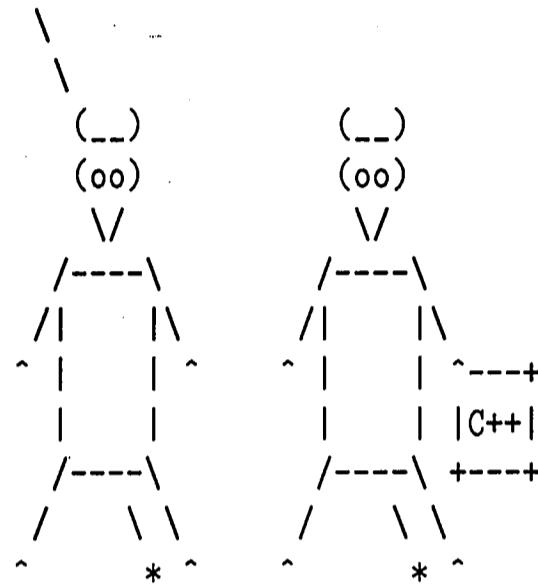
As a graduate student at Lehigh University, David has been a Research Assistant for a National Science Foundation (NSF) funded project investigating object oriented simulations in C++, and also in algorithms to enhance and detect edges with reliability in video scans of barcodes. As an undergraduate, he participated in an NSF Research Experience for Undergraduates in error - correcting codes and data security. David has had several computer oriented jobs, such as Student Technician for the University

Microcomputer Store and Computer Specialist for Lehigh University's Vice-President of Research.

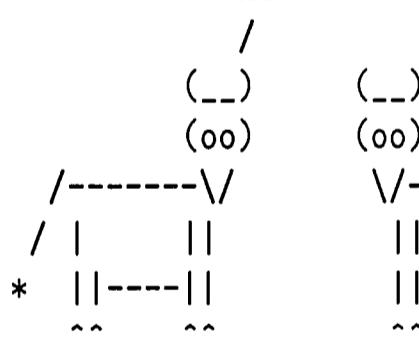
David has published the following technical reports at Lehigh University: "PAR-SIM: A Simulator for Designing Parallel Algorithms and Architectures," and "Table of Lower Bounds on the Minimum Distance of Cyclic and BCH Codes." The latter report has been referenced in at least four papers published in the *IEEE Transactions on Information Theory* or presented at the *1990 IEEE International Symposium on Information Theory*.

So, Bessy, C++ *is* a great tool
for designing object-oriented
simulations.

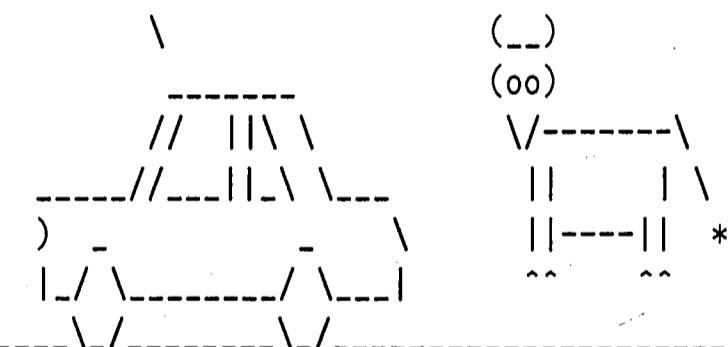
QUICK, ONE'S COMING !!!



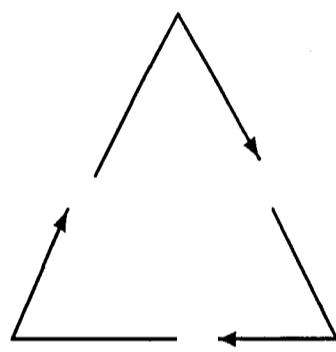
MOO



Oh, how boring it must
be to be a cow.



Please Recycle.



Peace.