Implementation of a Hypertext System in an Object-Oriented Database

Sadhana Shenoy
B.S. University of Phoenix, Phoenix, 1985

A thesis submitted to the faculty of the

Oregon Graduate Institute
of
Science and Technology

in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science

October, 1989

The thesis "Implementation of a Hypertext System in an Object-Oriented Database" by Sadhana Shenoy has been examined and approved by the following Examination Committee.

Dr. T. Lougenia Anderson
Servio Logic Corporation
Thesis Research Advisor

Norm DeLisle
Textronix Corporation

Dr. David Maier
Professor

Dr. Richard Hamlet
Professor, Portland State University.

Acknowledgements

I am very grateful to my thesis advisor Dr. T. Lougenia Anderson for her guidance and encouragement during the course of this work. A special thanks to Norm DeLisle for his help with Neptune. I also want to thank Dr. Maier and Dr. Hamlet for being on my thesis committee.

Jianhua Zhu and Belinda Flynn worked very patiently with me on the GemStone configuration, and I would like to express my gratitude. Arne Berre's discussions and comments about the thesis were very valuable.

And last, but very certainly not the least, I would like to thank Sunil, whose love and patience saw me through this Master's degree.

Table of Contents

ABSTRACT

Implemention of a Hypertext System in an
Object-Oriented Database

Sadhana Shenoy
Oregon Graduate Center, 1989

Supervising Professor: Dr. T. Lougenia Anderson

Hypertext is non-sequential or non-linear text. It is an electronic medium for information processing, where data is stored as a network of nodes interconnected by links. The fundamental concept of hypertext, namely, the imposition of structure on a collection of related data to facilitate its easy access, is not new. However, it is the "electronification" of documents and especially their interconnections that has lead to a renewed interest in Hypertext.

Conventional record-oriented database systems pose limitations of a fixed and finite set of data types, and the need to normalize data. Object-oriented databases overcome these limitations with their flexible data-typing facility and ability to model data that is not suited to normalized relations. GemStone is an object-oriented database server that supports a model of objects similar to that of Smalltalk-80. GemStone

provides an object-oriented database language, called OPAL. GemStone merges object-oriented language technology with database technology to solve data management and information modeling problems that are not easily solved by relational or hierarchical systems.

The purpose of this thesis is to implement the functionality of a prototype hypertext system (Tektronix's HAM - Hypertext Abstract Machine) upon an object-oriented database (GemStone) and report on the ease in using the object-oriented paradigm for a given application.

CHAPTER 1

INTRODUCTION

The purpose of this thesis is to implement a software system using an object-oriented language, and to report on the ease in using this paradigm. The software system chosen for this project is a hypertext system, (Tektronix's HAM - Hypertext Abstract Machine), and is implemented upon GemStone, which is an object-oriented database.

The first section in this chapter introduces the concepts of hypertext. The second section describes the object-oriented programming technique, and the third discusses the need for object-oriented database servers.

1.1. Hypertext

The term "hypertext" was coined by Ted Nelson, and means non-sequential or non-linear reading and writing. Hypertext is an electronic medium for communication and thought-processing. In a hypertext system, documents are represented by *graphs*. Graphs consist of ***nodes*** that contain data, and of ***links*** that connect the nodes.

The basic capabilities of hypertext are the linking together of discrete blocks of material to form networks of related information,

following paths through these networks, and attaching annotations. Nodes represent objects in the database, machine-supported references (links) between these objects allow non-sequential access to these nodes.

Nodes and links provide the representation system in hypertext. Nodes provide a data structure and links a representation scheme to hypertext applications. A node is a module of text that has some data in it. The node size is entirely up to the user and his application. A node can be considered a file, with semantic connotations ascribed by the user. Thus, a single character, a single idea, or all information about a specific subject can be put in a node. Hypertext nodes are not restricted to contain only text, they can contain any representational medium, such as structured graphics, digitized sound, bit-maps, audio recordings, etc. Nodes have names, and usually, properties ascribed by the user.

Nodes are connected by links. Links are used for connecting related portions of separate documents, for referencing, for comments, footnotes, annotations, margin notes and documentation. Links also have names and attributes. They are directed, ie., they have a source and destination. Sources and destinations of links are described differently by hypertext systems. They can be single points in the text, portions of text or even entire nodes. A link can usually be traversed in

both directions. Links reference by name (of destination node) or by value. Linking can be done explicitly through referential and organizational links. Implicit linking occurs through keywords.

All hypertext systems have an underlying database. The database can be thought of as a web or interconnection of nodes. Each node can be displayed by a window on the screen. Nodes can be created or removed, data can be entered, read, manipulated and stored back into the database through these windows. Links are indicated through icons, these links can be followed, and windows at the other end point of the link opened and viewed.

Hypertext is a tool for readers as well as writers. To writers, it provides an ideal environment in which varying material from numerous sources can be brought together and processed harmoniously. New ideas are not built all at once, they are developed along various different planes and orientations. These parallel lines of thought contribute much to each other, and grow together into a communicable and coherent idea. Traditional text processing mechanisms make it difficult to represent this process, mainly because there is no efficient way to represent the connections (between independent modules of information) that exist in the mind of the writer, no way to integrate independent ideas.

Hypertext provides readers with a medium where voluminous information from a multitude of sources is made easily accessible. The implicit structure in the material is made explicit. High speed windows allow readers to browse through referenced and documented material effortlessly and links allow easy non-sequential access to physically disconnected data, enriching the understanding of the subject matter.

The most striking and unique feature of hypertext is that it enables non-sequential reading and writing. Links impart this nonlinieaty to hypertext systems by providing a machine-supported mechanism to reference and integrate various chunks of information within and between documents. This feature makes it easy and unobtrusive to express branches in the flow of thought, and allows the viewing of material from varying angles by providing alternative successors to a portion of text. Conventional flat and sequential text is thus extended beyond the "single dimension of linear flow" [Conklin 86].

Thus, it is this "machine-supported links" feature that is the distinguishing feature of hypertext systems. Jeff Conklin's paper "A Survey Of Hypertext" [Conklin 86] seeks to establish the criterion of machine-supported arbitrary cross-linking between data items or interdocument links as being the fundamental feature of hypertext systems. Vannevar Bush, who is credited with first describing hypertext in his "memex",

considered this linking capability to be of primary importance,

" ... associative indexing, the basic idea of which is a provision where by any item may be caused at will to select immediately and automatically another. This is the essential feature of the memex. The process of tying two items together is the important thing. " [Bush 45].

Most hypertext systems provide the ability to "filter" information. Filtering allows users to choose and view pertinent information only. "Attributes" or other selection criteria define characteristics of and relationships between chunks of information by defining "values", and can be used to select the desired information [Yankelovich et al. 85]. The hypertext database can be browsed by search using filters or keywords, and also by following trails established by links.

Hypertext systems can aid in creating better audiovisualization, as the nodes in hypertext systems are not limited to contain only text, but can contain a combination of many media. "The inclusion of animation, computer-generated sound, and audio and video recordings adds a richness to electronic document systems that is impossible to recreate with paper media." [Yankelovich et al. 85].

In conclusion, hypertext systems are particularly useful for idea-processing, browsing and problem solving. They provide appropriate tools for structuring ideas, and organizing material in various ways to

suit different viewpoints. Hypertext systems are also well suited for exploring representational problems. They allow the user to view not only the contents of documents, but the structure as well. "The ability to work with unstructured information in conjunction with formalized, systematically organized information is the chief advantage in using a hypertext system rather than a database description language." [Marshall 87].

1.2. The Object-Oriented Programming Technique

Object-oriented programming was introduced in the 1960's by a language called SIMULA, where the notion of **object** was presented. However, it was Smalltalk, an interactive, display-based implementation, that gave the object-oriented paradigm wide recognition. This programming methodology has found use in a variety of applications, particularly in software systems development, artificial intelligence programming, and simulation.

The following paragraphs define the primary features of object-oriented programming (as supported by SmallTalk-80).

- **Objects & Methods**

Objects, not surprisingly, are the basic units in an object-oriented programming language. They perform computations and save state, and

hence combine the functionality of procedures and data in a conventional programming language. An object has a data structure as well as a group of operations that define how to access and manipulate the data contained in the data structure. An object has **instance variables** (fields). These variables can contain any values, and can even be pointers to other structured objects.

Objects communicate with each other by sending **messages.** Each message contains a **selector,** which can be described as a procedure name, and possibly message **arguments** corresponding to parameters. A message expression is like a subroutine call. The set of messages that an object understands are called its **protocol.**

When a message is sent to an object (the **receiver**) it responds to the message by executing a piece of code called a **method,** which is an operation that implements the named message. A method consists of a set of executable statements. Each method corresponds to a method selector.

- **Classes**

A class is a description of a group of similar objects, a template that is defined to describe the properties, characteristics and behavior of a set of common objects [Yankelovich et al, 88].

Every object is an **instance** of some class. When more than one object is an instance of the same class, each object has the same number of variables as defined by the class, but the data in those variables can be different for each of the instances.

A class can have many types of variables. The two most commonly used are instance variables and class variables. The value of a class variable is shared by all the instances of a class, whereas the value of an instance variable is specific to the instance of the class. All instances of a class share the same methods.

- **Subclasses and Inheritance**

Inheritance is a technique that enables specialization in object-oriented languages. It allows new classes to be built upon, or derived from other, pre-existing classes. The new class is more specialized as it contains more information or is more constrained than the pre-existing class. The more specialized class is called the **subclass,** and the more general one is the **superclass.** An object in a subclass inherits all the instance variables and methods of the old class. New instance variables and methods can be added by the subclass. The subclass can override a method in the superclass by defining a method with the same selector as the superclass.

- **Data Abstraction & Encapsulation**

Abstract data objects are fully characterized through the operations defined on them. Thus, there is no need to understand the internal representations and implementation of these objects. Objects in object-oriented languages are abstract data objects with an external interface, consisting of the set of operations defined on them. Object-oriented languages support data abstraction since an object can be manipulated only through its external interface. Changes can be made to the structure or behavior of an object, and so long as visible behavior of the object remains undisturbed, this change is not seen by the user.

Encapsulation is a technique for minimizing interdependencies among separately written modules by defining strict external interfaces. Data abstraction and encapsulation thus describe the same concept. These techniques make it possible to change the implementation of a module without affecting other interacting modules. This improves the understandability of programs and makes it easy to modify them.

- **Polymorphism**

Polymorphism is a concept where different classes of objects can be used in the same context. Each class responds to a set of messages expected in the context. Different objects can respond to the same message in their unique way, i.e., the methods that are executed in response

to the message are directly associated with the receiver. This is an important feature of object-oriented languages since it allows the definition of flexible software elements amenable to extension and reuse.

- **Binding**

In many object-oriented languages, variables are not typed and may contain objects of any class. It is not possible to determine statically exactly what method in what class will respond to a message. When a message is sent to an object, the corresponding method is looked up at run time in the object's class. This is late or dynamic binding, where the binding is based on the object whose operation is invoked. Dynamic binding allows a programmer to create very flexible code. Any object can be used as an argument as long as it can respond to the messages sent to it.

This is in contrast to statically-typed languages, where operation calls could be statically bound to the respective operations. Thus, the check for whether values of the appropriate type are being assigned to the variables that are the arguments of functions or operations is done at compile time rather than at run time.

1.3. Object-Oriented Database Servers

Why object-oriented servers?

Database systems technology has grown up in an environment of commercial data processing, where data-intensive business applications have requirements for persistence and sharing of data. Traditional database systems based on relational or hierarchical data models have evolved to meet the needs of these business applications. However, these database systems are characterized by a predefined and limited set of data types. In these systems, data is modeled by collections of rigidly typed records. Many modern-day application domains consist of complex data structures, and the traditional data models just do not have the framework or power to represent them. Current database systems "are primarily an effort to implement abstract data types over the memory of a machine" [Copeland, Maier 87]. They do not support easy and natural modeling of data since designers are constrained to a fixed set of predefined data types. Data has to be normalized, and contorted to fit into the available form, thus artificial structure is imposed upon it.

In contrast, object-oriented databases are built on the concept of an object rather than a record. Objects are a uniform programming unit for computation and saving state, and as such are ideal for representing collections of things that interact. The basic concepts of

object-oriented languages (object identity, abstract data-typing, message sending, data abstraction, inheritance and encapsulation) provide users rich data modeling facilities. Object-oriented programming is a powerful technique where a system can be structured as a collection of interacting components, and provides a robust framework for creating, extending, modifying and maintaining systems. Since a database is intended to stand as a model of the application environment, there is a need to incorporate the semantics into the data model so that the database can closely reflect the application. Object-oriented databases enable the user to define data semantics through type extensibility.

The purpose of this thesis is to implement a software system using an object-oriented language, and to report on the ease in using this paradigm. The software system chosen for this project is a hypertext system, (Tektronix's HAM - Hypertext Abstract Machine) , and it was implemented upon GemStone, which is an object-oriented database.

The remainder of this thesis is organized as follows: Chapter 2 provides an overview of some existing hypertext systems, Chapter 3 describes two commercially available object-oriented systems, Chapter 4 outlines the implementation of this thesis and Chapter 5 reports the results of the implementation.

CHAPTER 2

Hypertext Research Survey

2.1. Introduction

Jeff Conklin has categorized existing hypertext systems according to the nature of their applications: macro-literary systems, problem exploration tools, browsing systems and general hypertext technology [Conklin 86]. Macro-literary systems were the pioneer hypertext systems. They were conceived to function as electronic libraries, providing tremendous volumes of information on-line. Machine-supported interdocument links, which constitute a fundamental feature of hypertext, were introduced in these systems. The memex [Bush 45], NLS/Augment [Engelbart, English 68] and the Xanadu project [Nelson 80] are examples of systems in this category. Problem exploration tools provide a platform for developing ideas and solving problems. For example, Xerox PARC's PIE [Goldstein, Bobrow 80] is a tool for software design, development and documentation. Browsing systems include ZOG [Robertson et al. 81] and Hyperties, which allow easy access to, and manipulation of, stored information. These find use in academic environments. The more recently developed hypertext systems explore

specific issues in the area of hypertext in addition to providing all the facilities of the older systems. For example, Neptune [DeLisle, Schwartz 87a], NoteCards [Halasz et al. 87] and Intermedia [Meyrowitz 86a] have found a variety of applications, and serve as tools for further research and study of hypertext.

The rest of this chapter briefly describes some of the hypertext systems mentioned above. The hypertext system used as a model in this implementation is Tektronix's Neptune. Neptune is discussed in detail in Chapter 4.

2.2. An Overview of some Hypertext Systems

2.2.1. Bush's Memex

Vannevar Bush introduced the concept of hypertext in "As We May Think" [Bush 45]. He describes "memex", a mechanized private library, as

"Consider a future device for individual use, which is a sort of mechanized private file and library .... A memex is a device in which an individual stores his books, records and communications, and which is mechanized so that it may be consulted with exceeding speed and flexibility. It is an enlarged intimate supplement to his memory." [Bush 45]

His envisioned system consisted of "slanting translucent screens" on which data could be projected, and a keyboard. Material or data is stored on "improved microfilm". Books, pictures, periodicals, newspapers, pictures and business correspondences would all be stored in the library. An indexing scheme would be provided for reading the material, and facilities for non-sequential access would also be provided.

The memex sought to mechanically reproduce the mental process of retrieving and characterizing information in that "selection by association rather than by indexing may yet be mechanized." He described the essential feature of memex to be the ability to join two items and create "trails". Subsequently, recalling any one item meant that the other(s) in the trail could also be instantly recalled.

The memex was certainly a visionary idea but the required hardware was far beyond the (then) current technology. However the two primary features of the Memex, mechanized information storage and the fast and easy retrieval of this information define hypertext as it is known today.

2.2.2. Engelbart's NLS/ Augment

NLS (oN Line System) was developed in 1968 by Douglas Engelbart at the Augmented Human Intellect (AHI) Research Center at Stanford Research Institute. The research objective of the "interactive, multi-

console computer-display system" [Engelbart, English 68] was to develop a tool that would amplify the intellectual abilities of the user. The research group's aim was to do all of its work on-line, designing, planning, debugging and documenting, as well as intercommunications and scratch work.

The research group had access to very sophisticated CRT workstations. These work stations had, in addition to a television display and typewriter keyboard, two of Engelbart's inventions, the mouse and a 5 key handset. All of the "working information" was organized into files. Files were organized as hierarchical structures, while specifying structure was left to the user. Text was broken into arbitrary length segments called statements, which were numbered according to their serial location and positional hierarchy within the text. The first word of a statement was its name. Reference links could be established between statements within or across files.

Special mechanisms were provided for viewing and studying files, allowing users to view specified portions of the file. View-specification conditions allowed the user to specify the level or depth of the hierarchy at which he wished to view the file (level clipping), displaying only a certain number of lines from each statement depending on the truncation parameter (line truncation), and viewing only statements containing

some desired content. This specification was made in a "high-level content analysis language."

The concept of structured, multi-person distributed editing was introduced in this system. Standard editing operations were available for file modification, and hard copies could be produced with a help of different output devices. Compiling, debugging, calculating and conferencing facilities were also provided.

The NLS was unique, as it introduced many of the capabilities mentioned above that are standard in all systems today. Known today as NLS/Augment, it is marketed as a commercial network system.

2.2.3. Xanadu

Xanadu was developed by Ted Nelson, and is a system for storing, accessing and manipulating linked and windowing text. Ted Nelson wanted to replicate and extend the structure of the literary system online, creating a unified literary environment — it was he who coined the terms "hypertext" and "hypermedia". The goal of the system was to form a basic "linkage structure" that was a computerized version of the implicit linkage and referral processes existing in the literature system. A second goal was to build a universal, simple and fair system of electronic literature that builds on the structure and relationships already present in literature.

The creators of Xanadu "have endeavored to create a system for text editing and retrieval that will receive, handle and present documents with links between them" [Nelson 80]. Documents and links are the primary elements of the system. A document can contain both text and links, thus everything in the system is a document.

A link is a connection between pieces of text. Many types of links are allowed in the system (basically, the user can define any type of link he desires), and links may have multiple end points. "Literary" links include jump links (footnotes), quote links (quotations), correlinks (marginal notes) and equilinks (between the same item in two separate versions of the same document). Links may be filtered by type and origin.

Ted Nelson considers Xanadu to be a full blown implementation of Vannevar Bush's "memex", designed for "indefinite expansion". Xanadu is also intended for use as a publishing system since allocation of credit and royalty is provided for. Royalty is paid automatically by a user on a "byte-by-byte" basis.

The Zanadu system has been implemented and is marketed as an on-line service. There is a version that runs on Sun workstations.

2.2.4. PIE

A "Personal Information Environment" (PIE) was proposed in 1980 by Ira Goldstien and Daniel Bobrow [Goldstein, Bobrow 80]. The PIE

environment is a network-based approach to developing and documenting software. The PIE description language is based on the concepts in KRL and FRL, and is written in Smalltalk. This description language enables interactive program development.

PIE contains a network of nodes, each having several **perspectives.** Each perspective presents a unique view of the entity represented by the node and provides for specialized actions for the specific view. Perspectives help a user document the various aspects of his application. The authors offer the example of multiple views of a Smalltalk class: one providing its definition and structure, another the hierarchical organization of its methods, a third stating external methods called from the class, and yet another contains documentation.

Nodes have **contexts,** which provide a way of storing different values for the same node. During software development, alternate designs can be stored and evolved in different contexts of the same node. Contexts can be extended by creating **layers.** These layers store modifications to contexts, thus back-up to a previous state can be easily handled. **Contracts** exist between nodes and describe dependencies between elements.

Traditional programming environments do not allow "alternative definitions of procedures and data structures to exist simultaneously in

the programming environment" [Goldstein, Bobrow 80]. The PIE system overcomes this difficulty by providing support for comparing and storing alternative designs, and highlighting their differences. It provides the capabilities to present the various aspects of an application program as a single integrated structure. PIE's networked database facilitates cooperative design efforts. PIE can also be used as a prototype office information system.

2.2.5. CMU 's ZOG

ZOG is a networked, menu-based system developed in 1975 at Carnegie-Melon University [Robertson et al. 81]. ZOG was initially implemented on PDP 10 and VAX 11/780 machines.

In the ZOG system, each data segment, called a **frame,** is displayed on the screen along with a set of menu items called **selections.** Standard ZOG commands such as Edit, Help, Back, Next, Return, etc. are represented by global **pads** on the bottom of the screen. A sequence of frames can be viewed by making the desired selections. Data is represented as a hierarchical network of frames divided into **subnets.** Subnets operate essentially as subroutines. ZED, the ZOG editor, allows new frames to be added and modified, and the network can be augmented by a user to suit his specific needs.

ZOG has been designed to serve a large user community, and as "a particular interface for man-computer interaction." It has found use as a guidance system, a computer-aided instruction system, a database and browsing system. ZOG has been installed as a computer-based information management system on the USS CARL VINSON.

A major problem with the system is that it is very difficult to grow the really large nets that are an essential ingredient of ZOG.

2.2.6. Brown University's Intermedia

Intermedia is a large-scale, object-oriented hypertext/hypermedia system. The Intermedia project was developed in 1984 at the Institute for Research in Information and Scholarship (IRIS) at Brown University. It was a part of Brown University's campus-wide "Scholar's WorkStation Project" . The system provides the ability to create sophisticated linkages between documents from a variety of applications. It also provides a development framework for creating additional applications with that capability [Meyrowitz 86a].

Intermedia was developed as a tool to facilitate computer-aided, computer-supported teaching and research in a scholastic environment. The educational goals of the project are audio-visualization, encouraging exploration of an information-rich environment, making software tools available to professors to "create webs of information," and to students

to "follow trails of linked information, annotate text... and communicate with other students and professors " [Meyrowitz 86a].

Intermedia runs on a network of Unix-based workstations. The system is composed of the MacApp layer, the Intermedia layer, and the Graphics and Text Building Blocks. Additionally, the six major application units are Intertext, InterDraw, Interval, Interpix, and Interspect. These are the text processor, graphics editor, timeline editor, scanned-image viewer and a three-dimensional object viewer.

The system was developed as a framework of software tools with integrated hypertext capabilities so that it could be used in routine application tasks such as the creation and update of documents.

Intermedia integrates all the functionality of a hypertext system into each application, so that users can take advantage of the system's capabilities within the framework of their routine work rather than use Intermedia as a distinct application. The system creates a **document** that can contain texts, graphics, spreadsheets etc. A document can be viewed through a **window. Links** are created between **blocks** in a document. Keywords and explainers attach attributes to blocks and links. These facilities allow the creation of **webs.** Webs are contexts or databases containing sets of documents and interconnecting links to these documents. **Maps** are visual representations of webs and facilitate

the understanding of webs by providing easy and efficient access to the webs.

To evaluate the success and utility of Intermedia, it is being used in a variety of educational and work settings. Intermedia material already used in courses has proved very useful, demonstrating that hypermedia systems have tremendous potential.

2.2.7. Xerox PARC's NoteCards

The NoteCards system was developed at Xerox PARC by Frank Halasz, Thomas Moran and Randall Trigg [Halasz et al 87]. NoteCards is an extensible computer environment developed to aid the collection, analysis and processing of information. It is designed as a "general purpose idea processing environment" [Halasz et al 87]. The NoteCards system is implemented on Xerox D series Lisp machines. The interface consists of Lisp functions that give a programmer tremendous flexibility to create new applications using NoteCards. NoteCards, Links, Browser and FileBoxes are the basic objects defined in the system. A notecard is a computerized version of the 3X5 paper index card. Each notecard contains editable data, such as text, graphics etc. A notecard can be characterized by its type. A user can define new types to suit his application. Links are typed objects that connect notecards. A browser depicts a network of notecards, and FileBoxes file or order collections of

related notecards. With these basic objects, NoteCards creates a "semantic network of electronic notecards interconnected by typed links" [Halasz et al. 87].

NoteCards has about 70 users within Xerox, and externally the system is in use at universities, government and industrial sites. NoteCards has met its goal of creating an environment for information management. However, it lacks support for multi-person or collaborative work, and it does not have sophisticated tools for displaying large graphs and other structures.

CHAPTER 3

Object-Oriented Databases

There are two object-oriented database servers that are commercially available today, Servio Logic's GemStone and and Ontologic's VBASE. The GemStone database server has been used in this implementation. The next sections will summarize VBASE, and describe GemStone at some depth.

3.1. VBASE: an object-oriented development environment

Ontologic's VBASE Integrated Object System is an object-oriented development environment that combines a procedural object language and persistent objects into one integrated system [Andrews, Harris 87]. VBASE was inspired by the need to bring together language and database functionality to form a single, object-oriented system, to develop software systems and production applications.

The VBASE architecture comprises of four layers: the language, abstraction, representation and storage layers. Each layer has a VBASE specification and implementation. VBASE is implemented on top of Sun OS 3.2 Unix. The TDL (Type Definition Language) specifies the data model. The COP (C Object Processor) is used to write application pro-

grams, and implement operations.

Probably the most distinctive feature of VBASE is that it is an object system that also provides strong typing. In TDL, all object definitions and properties are associated with data types. It is a block structured language, and allows the definition of constants, variables, enumerations, unions and variants. Parameterization (the ability to specify the types of objects contained within other objects), a capability often not found even in procedural languages, is another significant feature of VBASE. VBASE also has a special exception handling mechanism. Exceptions are types, thus a hierarchy of exceptions can be defined, and properties and operations can be defined for them. 'Except' and 'raise' statements transfer control to a exception handler rather than return control to the caller.

In addition to providing most of the expected database functionalities, VBASE also supports persistence of objects, and clustering objects on disks. Support for inverse relationships automatically implies that one-to-one, one-to-many, and many-to-one relationships between objects can be maintained.

VBASE had set two goals for itself, integrating a procedural language with support for persistent objects, and providing strong typing. VBASE has emerged as a relatively complete development system,

it is object-oriented, strongly typed, and supports persistent objects [Andrews, Harris 87].

3.2. GemStone

GemStone is an object-oriented database server developed at Servio Logic Development Corporation that merges object-oriented language concepts with database systems. GemStone provides an object-oriented programming language called OPAL, which serves as the data definition and data manipulation language for the GemStone system. OPAL provides standard database storage and retrieval functions as well as programming facilities for general computation that are comparable to C or Pascal. The GemStone system was built on the premise that a "combination of object-oriented language capabilities with the storage management functions of a traditional data management system will result in a system that offers further reductions in application development efforts." [Maier et al. 86]

The goals and requirements of the GemStone system, as specified by Maier et al. fall under three categories, that it provide an extensible data model, general database amenities, and a programming environment. These are briefly outlined in the following paragraphs.

An extensible data model means that a user should be able to define new data types and operations on them so that the structure as well as the behavior of objects can be modeled. Its data model must support arbitrary levels of data structuring, and separate type definition from type instantiation. No artificial limitations should be imposed on data items. Additionally, variations in structured objects should be allowed, and arbitrary data items as values should be allowed.

General database functions such as a multiuser environment, concurrent access and serializability of transactions, private ownership of data objects, stable storage for data objects, atomic commit and abort of transactions and authorization and system management functions need to be provided.

The programming environment must include an interactive interface for the OPAL language (for defining new database objects, executing queries, etc.) and a procedural interface to conventional languages.

(a) The GemStone Architecture

The GemStone configuration used in this research consists of the OPAL language and the storage management software running in the DEC VAX/VMS environment. IBM-PC's or Smalltalk machines networked to a VAX run a set of interactive interface programs called

the OPE (Opal Programming Environment) to enable the user to build, execute and debug OPAL programs. The OPE consists of an OPAL browser, source code workspace, and a bulk loader and dumper. Figure 3.1 illustrates the architecture.

Conceptually, the Stone process provides secondary storage management, concurrency control, authorization, transactions, recovery, and support for associative access. It manages workspaces for active sessions [Maier et al. 86]. In Figure 3.1, the Stone process is represented by the rectangle named "DATA MANAGEMENT KERNEL". A separate Gem process is maintained for each session. The Gem process is above the Stone process; it compiles and executes OPAL code, provides session control. In Figure 3.1, the rectangles named "OPAL COMPLIER/INTERPRETER" represent two Gem processes.

(b) **OPAL: GemStone's unified database language**

The object-oriented programming language called OPAL serves as the data definition and data manipulation language for the GemStone system. OPAL is a powerful, general purpose programming language based on ideas from knowledge representation, abstract data types, semantic data languages, set-theoretic data models and non-procedural query languages. It provides rich data modeling facilities with an interface to a high level programming language (C) [Copeland, Maier 87].

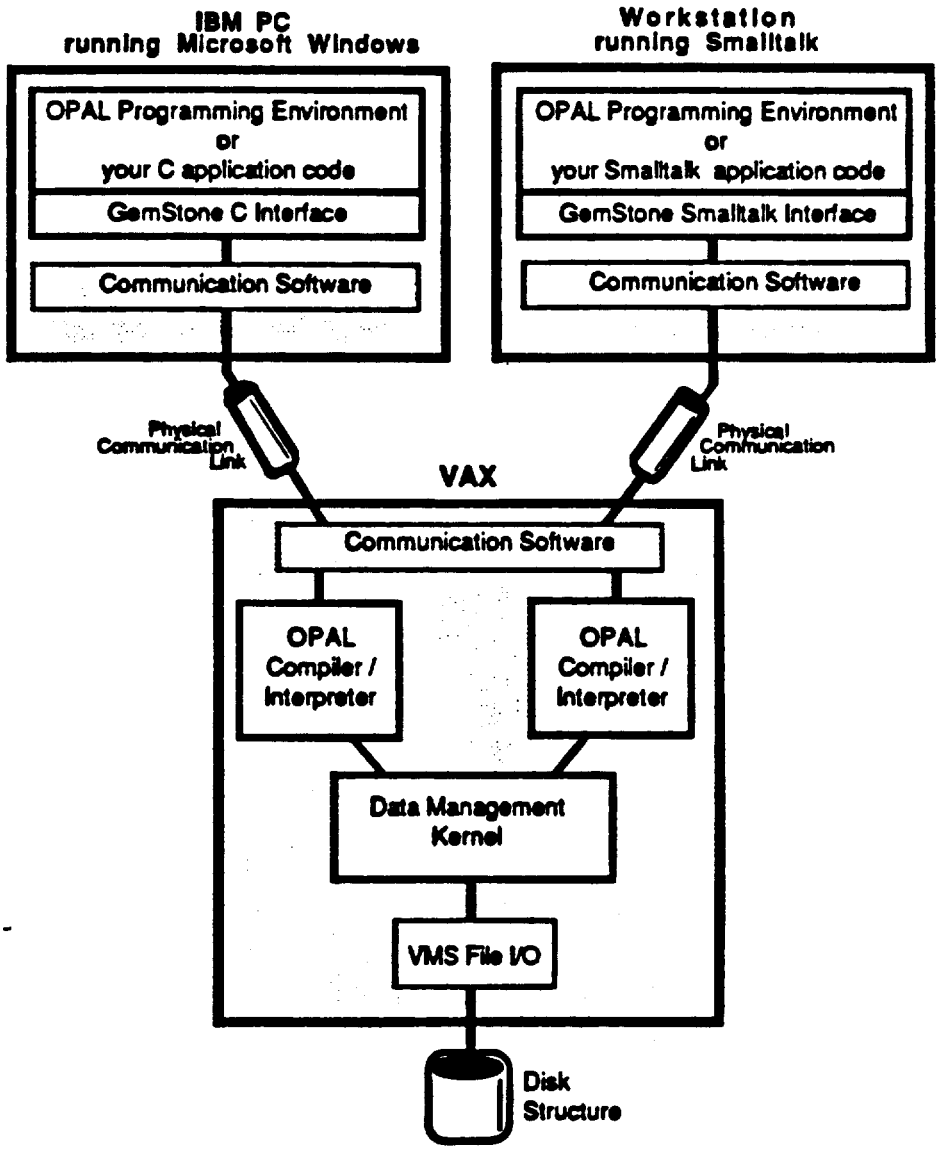


Fig 3.1 The Gemstone Architecture

(Servio-Logic Corp.'s
GemStone manual)

OPAL is derived from Smalltalk-80, and its syntax and semantics are almost identical to it.

OPAL's object model is identical to that of Smalltalk-80. The three primary concepts are object, message and class, which are equivalent to record, procedure call and record type respectively, in a conventional system [Purdy et al. 87]. An object is a well designed data structure with a set of operations that provide access to and enable the manipulation of the data contained the data structure. Communication between objects is achieved through messages. Message sending is the only way through which the data contained in an object can be accessed. A class is a description of one or more similar objects. The structure, methods and messages of a class's instances are factored and stored once in a single object describing the class, i.e. the class defining object. Classes are organized into a class hierarchy through which the structures and methods are inherited.

The basic syntactic unit of an OPAL program is the statement. A statement contains one or more expressions. An OPAL expression can be a literal, variable name, message expression, assignment, etc. OPAL's major language constructs are message expressions and method definitions [Purdy et al. 87].

The form of all message expressions is <receiver> <message>. The receiver is a variable or expression representing the object for whom the message is intended, and which will receive and interpret the message. The second part of a message expression, the message, is made up of a selector, and possibly some arguments. On execution of the message expression, a result is returned to the sender.

There are three kinds of messages, unary, binary and keyword. Unary messages have no arguments, and the selector is a single identifier. For example,

zoo closingTime

is an unary message expression. *'closingTime'* is the selector, and this expression has no arguments.

(zoo1 numberOfAnimals) <= (zoo2 numberOfAnimals) and

*8 * 9*

are examples of binary messages, where *'<='* and *'*'* are selectors, and *'(zoo2 numberOfAnimals)'* and *'9'* are arguments of the message.

Keyword messages have multipart selectors, the selector in the message expression below is *at:put:*, while the arguments are *'1'* and *'sheba the lioness''*,

zooAnimals at: 1 put: "Sheba the lioness".

Methods describe all the action and execution in GemStone. Each class defines the methods its instances will understand. A method definition names the method's selector, and possibly some formal arguments. A method can also contain temporary variables. The body of a method consists of OPAL statements, and possibly a return statement. Methods are defined within the scope of the object (a class instance) that is the receiver of the message, and can thus access the named instance variables of the receiver. An example of a method is,

name: aName

This method sets the instance variable 'name'.

^ name := aName.

GemStone is a computationally complete and extensible programming language. OPAL extends Smalltalk in the area of associative access support for queries. This extension is supplied by providing an indexed associative access mechanism, which is a system for maintaining indexes to large collections and for using these indexes to retrieve collections elements quickly.

OPAL was implemented by writing the object storage manager, the OPAL compiler, and interpreter. This system provides a multi-user,

disk-based system as opposed to the single-user, memory resident Smalltalk system [Maier et al. 85].

(d) **GemStone's database features**

GemStone combines the powerful data type definition and code inheritance properties of Smalltalk-80 with permanent data storage, multiple concurrent users, transactions and secondary indexes [Maier et al. 85]. The main database features of GemStone are the provision for sharing of objects (through a private list of dictionaries called the SymbolList for each user), resilience to common failure modes, security (through user authentication), a centralized server and primary and secondary storage management facilities [Purdy et al. 87]. GemStone provides a Smalllalk-like execution model for execution of its methods, and support for multiple concurrent users.

GemStone supports multiple concurrent users. GemStone's transaction control uses an optimistic concurrency control policy. GemStone provides for fast associative access to members of collections by allowing users to dynamically add or remove associative access structures to aid such searches.

In summary, GemStone is a pioneer object-oriented database system developed to merge object-oriented programming language technol-

ogy with database technology. It solves data management and information modeling problems that are not easily solved by relational or hierarchical systems.

CHAPTER 4

Implementation

The purpose of this thesis is to implement a large software system in an object-oriented language, and report on the ease of using the object-oriented paradigm for an application. An existing software system, Tektronix's Neptune, (or specifically, its engine, the HAM) was selected as the candidate application and a re-implementation was done using the object-oriented database management system GemStone. This implementation, which we have called GemDesign, will be discussed in this chapter. The first section describes the Neptune hypertext system. The second section provides an overview of the design of the system; the third details the implementation. The fourth section provides some general notes on the implementation, while the fifth describes the interface between GemDesign and Neptune's Smalltalk interface. The final section discusses the results of implementing GemDesign.

Figure 4.1 (a) gives an overview of Neptune's architecture. The Hypertext Abstract Machine (HAM) is implemented in C, its user interface is implemented in Smalltalk-80. GemDesign uses the GemStone database. It is written in OPAL, which is GemStone's data definition and data manipulation language. (Fig 4.1 (b)). No separate interface

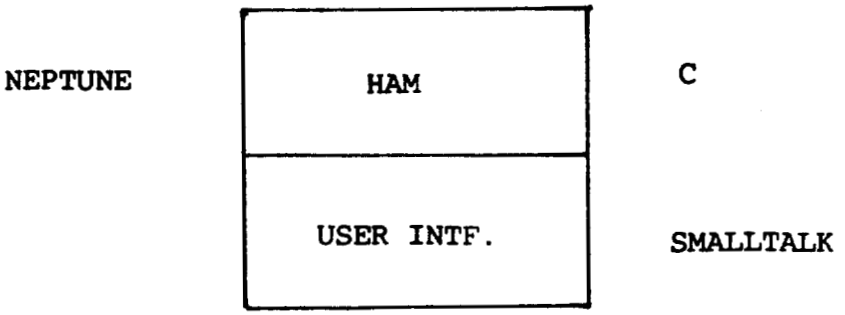


Figure 4.1 (a) NEPTUNE

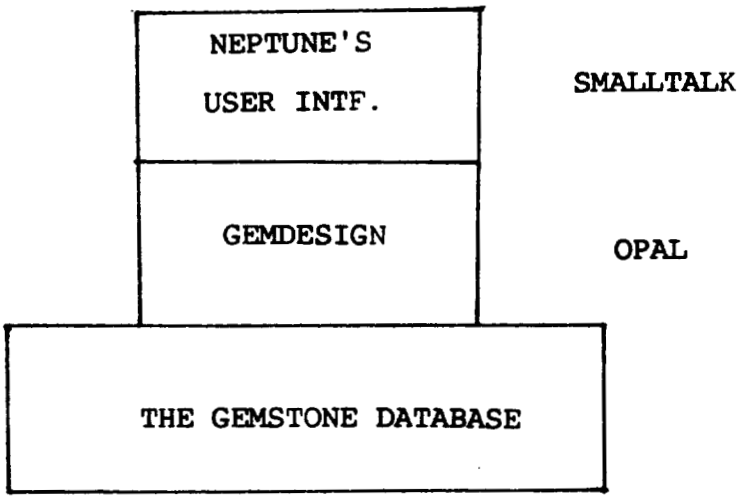


Figure 4.1 (b) GEMDESIGN

was written for GemDesign (for reasons detailed later in the chapter). Instead, Neptune's SmallTalk-80 interface was ported and used in GemDesign.

4.1. Tektronix's NEPTUNE

Tektronix's Neptune was designed to support software development. Neptune has a layered architecture consisting of two major layers: the Hypertext Abstract Machine (HAM) and the user interface. The HAM is a bottom-level transaction-based server. It is implemented in C, and runs on Unix servers such as Magnolia's (a workstation developed for internal use at Tektronix) , Tektronix's 4400 series workstations, VAX machines, etc. The user interface is the topmost graphical layer and provides browsing and editing facilities. Application layers consisting of programs that use hypertext data can be built on top of the HAM.

The HAM is a generic hypertext model based on five entities: node, link, attribute, graph and context. Nodes contain arbitrary, editable data. A node is an **archive** or a **file.** Complete version histories are maintained for archives, while a file contains only the current version. Efficient storage and access facilities are provided to all versions of a node.

Nodes can be connected by links. Links can be traversed bi-directionally. The endpoints of a link have offsets within the node; a link is anchored at an iconic point in the data of a node. Links relate two nodes either within the same context, or in two different contexts. (Contexts allow users to keep related information together, and will be defined more fully below.) The latter type of link is known as a cross-context link, and facilitates data sharing between contexts by making a single node accessible in both the contexts. Links can have attribute/value pairs attached to them, and version histories can be maintained for a link, depending on the types of the nodes to which the link is attached.

Attributes are defined for nodes, links and contexts. The HAM provides a set of predefined attributes for which values can be assigned. Attributes characterize the objects for which they are defined and are used primarily for querying and filtering the graph. Changes to attributes can be archived. Nodes and links can have any number of attribute-value pairs attached to them. These serve as an efficient accessing mechanism, allowing desired objects to be 'filtered' from large graphs.

A graph is a top-level HAM object. It is made up of nodes and links, attributes and contexts. The version history (past states) of a

graph is maintained through the versions of all its components. A graph usually contains all the information available for a particular project.

Contexts partition the objects or data within a graph, i.e., a context contains nodes, links and attributes. Contexts are organized in a tree structure. Version histories are maintained for contexts. When a graph is created, its root context is created as well. All contexts except the root context have parent contexts.

The user interface provides browsers through which documents and nodes can be viewed and edited. The three primary browsers are the graph browser, which displays a pictorial view of the graph, the document browser, for browsing the hierarchical structures of nodes and links, and the node browser, which displays an individual node in a graph. Other browsers provided by Neptune include the attribute browser, node differences browser, and the versions browser. Figure 4.2 [DeLisle, Schwartz 87b] shows a graph as displayed by Neptune's graph browser. Nodes are represented by rectangles and the arrows between nodes are links. This graph shows a paper, titled **TOOIS** paper, with sections *Introduction, Hypertext systems, Contexts,* etc. Each section is represented by a node in the graph. Some sections have further subsections which are also represented by nodes. Sections are 'linked' to their subsections.

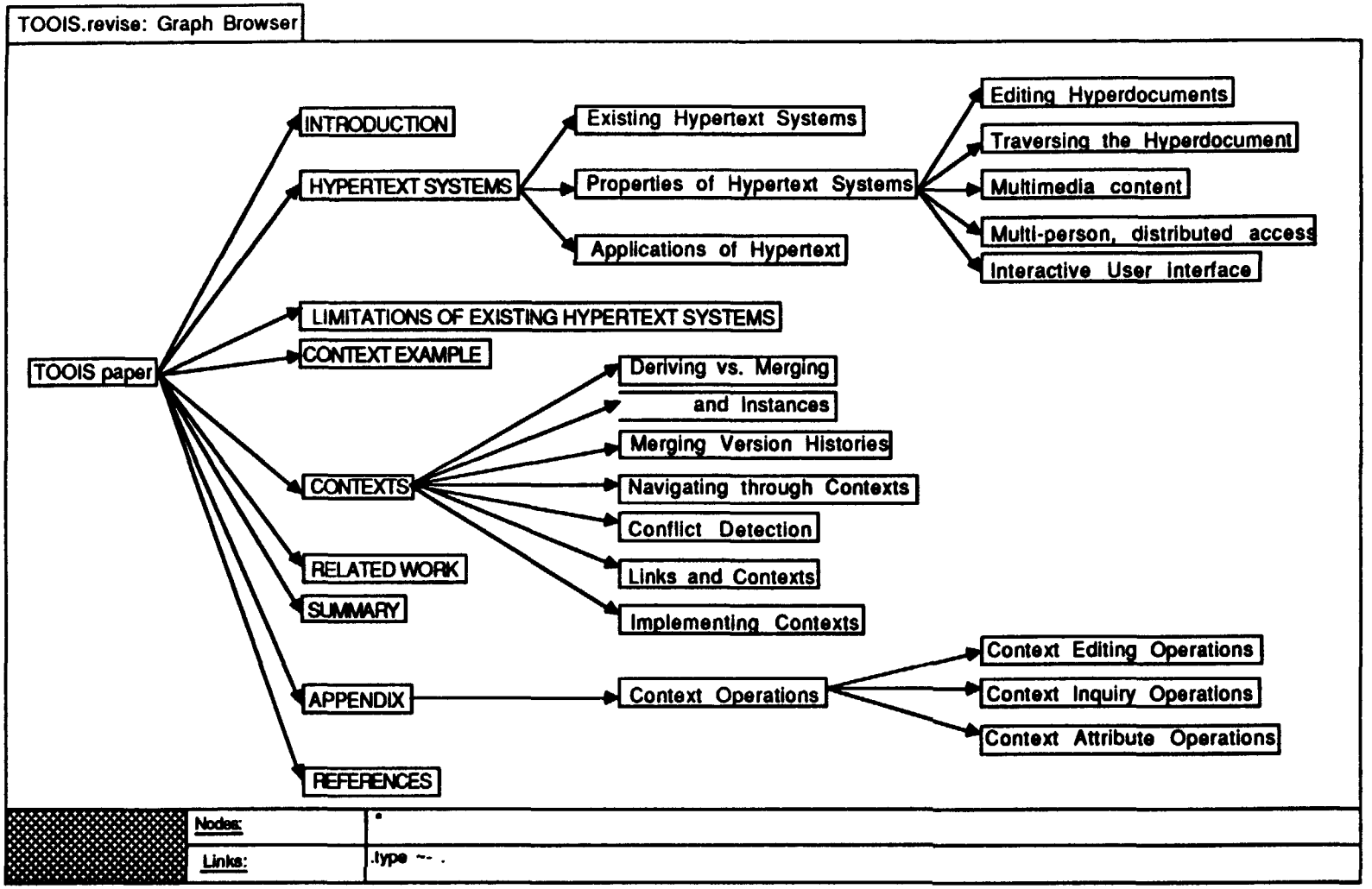


Fig. 4.2. Neptunes Graph Browser

The HAM maintains the integrity of the data that it manages by providing a transaction recovery mechanism. All the operations provided by the HAM are atomic transactions; if a transaction does not complete the completed portions of the transaction are undone. A multi-person, exclusive-writer protocol is enforced for the data contained in a graph.

The primary applications of Neptune have been for supporting document preparation and managing software code. Ongoing applications include use as a CAEE database, a CAD database and a field service support database.

4.2. Design

The last section summarized the functionality and the architecture of the original Neptune system as implemented at Tektronix. The remainder of this chapter discusses the re-implementation of Neptune using GemStone.

Graph

A graph in GemDesign is created with a unique name. A graph contains all the information pertaining to a document. All objects in a graph are uniquely identified by their names. A graph contains nodes and links, and attributes defined for these objects. Figure 4.2 displays a

graph named **TOOIS.revise** with a number of nodes, and links between them.

Node

Nodes are be created by specifying a name and some contents. Example nodes in Figure 4.2 are *Hypertext systems, Properties of Hypertext*systems, *Interactive User Interface,* etc. Contents of a node can be modified, i.e., edited. The hypertext system allows attributes to be defined for a node. Attributes have names and values. These allow the user to characterize and attach special meaning to a node. Nodes can be deleted from a graph.

Link

Links are also given unique names. Links have source and destination nodes, and are attached to specific positions in the contents of a node. For example, in Figure 4.2, source node *Appendix* and destination node *Context Operations* are joined by a link. Links can also have attributes. Links can be explicitly deleted, and if either the source or the destination node is deleted, so is the link.

Attributes

Attributes are defined for nodes and links. An attribute has a value, which is either a string or an integer. Attributes contain the oid (object identifier) of their owners. Attributes are used mainly to

traverse or filter a graph, i.e., retrieve a subset of all the nodes and links in a graph based on the values of their attributes.

Versions

Past states of nodes, links and a graph are maintained by a versioning scheme in GemDesign. Nodes and links are maintained as linked lists of past versions. When a versioned node is modified, a new node is created with the new information. All unchanged data is passed on to the new node.

A node is versioned if its contents or name change as a result of changing the contents, or if a link is added to or deleted from a node. A link is versioned if its attachment in either the source or the destination node changes, or if its name changes. Attributes are also versioned when their values change.

Context

Contexts are a partitioning scheme for hypertext graphs. When a hypertext system is used for a project that requires cooperative work by its developers, contexts allow individuals to work independently with subsections of the graph, and then 'merge' this work with the group effort. Contexts support multi-person, co-operative software development efforts, providing a computer environment to support software design and development efforts. Figure 4.2 shows one context of the

graph named **TOOIS.revise.** Versions of the graph could exist in other contexts of the graph.

The contexts scheme implemented in GemDesign is a linear, non-branching versioning scheme. Nodes can be copied into subsidiary contexts, and then merged back into the primary context. Links can be cross-context, ie, connect two nodes existing in different contexts.

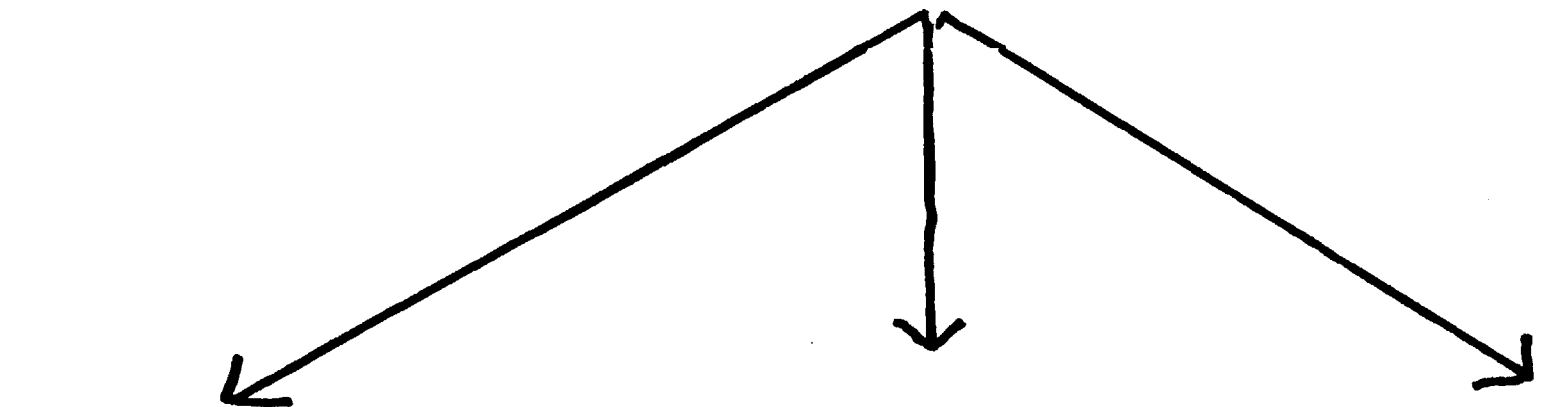
4.3. Implementation

GemDesign has been implemented in GemStone's database language, OPAL. The following sections give the functional specifications and descriptions of the implementation. The major classes with their instance and class variables are presented, and their operations are discussed. For a description of the other classes that support these major classes, refer to the Appendix of the thesis, which contains the code of the implementation.

4.3.1. HyperGraph

This class is the superclass of three primary classes, graph, node and link. Figure 4.3 presents the definition of this class, along with that of its subclasses. This class has four instance variables, *name, creationTime, rootContext* and *contextSet.* The instance variable *name* is not used by instances of HyperGraph but by instances of its subclasses. It

```
Object subclass: 'HyperGraph'
   instVarNames: #( 'name' 'rootContext'
            'creationTime' 'contextSet' )
   classVars: #( 'graphDirectory' )
   poolDictionaries: #[]
   inDictionary: HyperModel
   constraints: #[]
   isInvariant: false
```



```
HyperGraph subclass: 'Graph'
  instVarNames: #( 'nodeSet'
     'linkSet'  'allNodeAttrs'
     'allLinkAttrs' 'allNames'
     'root')
  classVars: #()
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: #[#[#name, String]]
  isInvariant: false
```

```
HyperGraph subclass: 'Node'
  instVarNames: #( 'ofGraph'
     'attributeSet' 'sourceLinks'
     'destLinks' 'contents'
     'flink' 'blink' )
  classVars: #( )
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: #[#[#name, String]]
  isInvariant: false
```

```
HyperGraph subclass: 'Link'
  instVarNames: #( 'ofGraph'
     'attributeSet' 'fromNode'
     'toNode' 'fromPos' 'toPos'
     'flink' 'blink' )
  classVars: #( )
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: #[#[#name, String]]
```

Fig 4.3 The Class HyperGraph and SubClasses

represents the user-given name, and uniquely identifies objects in the graph. The instance variable *creationTime* is computed from the local computer's system clock and calendar, and is assigned to each object when created. *RootContext* contains the oid of the root context of a graph. *ContextSet* is a set containing all the contexts of a graph.

GraphDirectory is a class variable, containing a list of all the graphs created. Each of its elements is an Association, with names of the graphs as key and the oid of the graph as value.

4.3.2. Graph

Graph is a subclass of class hypergraph. It contains all the information about a particular graph. It inherits variables from its superclass. In addition, *nodeSet, linkSet, allNodeAttributes, allLinkAttributes,* and *allNames* are instance variables declared by this class. Instance variable *nodeSet* contains all the nodes that are created in the graph. *LinkSet* contains the links that connect nodes in this graph. The instance variables *allNodeAttrs* and *allLinkAttrs* are sets that contain all the attributes defined for all the nodes and links, respectively, in a graph. The variable *allNames* is a set containing all the names assigned to the nodes and links in the graph. Through this variable, the uniqueness of names declared for nodes and links is ascertained.

The following methods define graph operations in GemDesign:

- *createGraph:* **aName**
- *openGraph:* **aName**
- *deleteGraph:* **aName**
- *getGraphViaAttributes:* **nAttrName** *value:* **nAttrValue** *linkAttr:* **lAttrName** *withValue:* **nAttrValue** *versionTime:* **aTime**
- *linearizeGraph:* **rootNode** *versionTime:* **aTime** *nodeAttrName:* **aName** *value:* **aValue** *linkAttrName:* **sLAttrName** *attrValue:* **aLAttrValue**

The method *createGraph:* **aName** creates a new hypertext graph with name **aName** uniquely identified by its name. The method *openGraph:* **aName** opens a graph named **aName** , i.e., returns a pointer to the graph. The method *deleteGraph:* **aName** deletes a graph named **aName** by removing its entry from the graphDirectory .

The method *getGraphGraphViaAttributes:* **nAttrName** *value:* **nAttrValue** *linkAttr:* **lAttrName** *withValue:* **lAttrValue** returns a sub graph of the existing hypertext graph, formed by accessing all the nodes and links of the graph, and filtering these nodes and links based on certain attributes. First, all nodes that have an attribute **nAttrName** with value **nAttrValue** are returned. Then all links that have the attribute **lAttrName** with value **lAttrValue** and connect two filtered

nodes (above), are returned. Thus, a partially connected sub-graph of the original graph is the result.

The method *linearizeGraph:* **rootNode** *versionTime:* **aTime** *nodeAttrName:* **aName** *value:* **aValue** *linkAttrName:* **aLAttrName** *attrValue:* **aLAttrValue** returns a sub-graph of the hypertext graph at **aTime** , formed by doing a depth first search via the links starting at node **rootNode.** An array of two elements is returned, the first element is the node list comprising of the nodes encountered during this depth first search, having the attribute named **aName** with value **aValue.** The second element is a link list containing links, each of which connect two nodes in the node list, and have an attribute named **aLAttrName** with value **aLAttrValue.**

In Neptune, users can filter and browse graphs by supplying a *nodePredicate* and a *linkPredicate.* These predicates are expressions consisting of attribute names, relational operators and values, and are based on the predicate grammar described in [DeLisle, Schwartz 87a]. In GemDesign, no extensive grammar has been defined, the filtering process occurs as described above.

4.3.3. Nodes

Node is a subclass of class HyperGraph. The instance variables of this class are *ofGraph, sourceLinks, destLinks, attributeSet, contents, fLink* and *bLink*. Instance variable *ofGraph* is the name of the graph to which the node belongs. *SourceLinks* and *destLinks* are sets of oid's of all the links of which a node is the source or destination respectively. *AttributeSet* is a set of all the attributes of a node. The variable contents contains the text, or the data of a node. The instance variables *fLink* and *bLink* are pointers to the node which is ahead (behind) in the version list of a node.

The following methods define node operations in GemDesign:

- *createNode:* **nodeName** *withContents:* **someContents**
- *deleteNode:* **aName**
- *openNode:* **aNode** *versionTime:* **aTime**
- *modifyContents:* **someContents** *ofNode:* **aNode** *withLinks:* **aSet**
- *changeName:* **aName** *ofNode:* **aNode**

The method *createNode:* **nodeName** *withContents:* **someContents** creates a new node in the hypertext graph, with name **nodeName** and contents **someContents.** The name of a node has to be unique, so a check is done to verify that. The node is added to the instance variable

nodeSet of the graph. The method *deleteNode:* **aName** deletes node **aName** from the current version of the graph. All the links attached to this node, and the attributes of this node are deleted. The method *openNode:* **aNode** *versionTime:* **aTime** returns a pointer to the appropriate version (specified by **aTime**) of node **aNode.** If the **versionTime** is 0, then the current node is opened. Otherwise, the version list of the node is traversed, and the appropriate version returned. The method *modifyContents:* **someContents** *ofNode:* **aNode** *withLinks:* **aSet** creates a new version of the node **aNode** with contents **someContents** is created. Each link whose position in the contents of the node has been changed (links in **aSet**) is also versioned. The method *changeName:* **aName** *ofNode:* **aNode** changes the name of node **aNode** to **aName.** A new version is created for the node.

4.3.4. Link

Link is a subclass of class HyperGraph. The instance variables of this class are *toNode, fromNode, fromPos, toPos, attributeSet, fLink* and *bLink.* The instance variables *toNode* and *fromNode* are the source and destination nodes respectively, of a link. The instance variables *fromPos* and *toPos* are the character positions of the link in the contents of a node. These represent an integer value that refers to the position numerically. Variable *attributeSet* is a set of all the attributes of a link.

Finally, *fLink* and *bLink* are the oid's of the link which is ahead (behind) in the version list of a link.

The following methods define link operations in GemDesign:

- *createLink:* **linkName** *from:* **sourceNode** *spos:* **sInteger** *to:* **destNode** *dpos:* **dInteger**
- *deleteLink:* **aLink**
- *openLink:* **aLink** *versionTime:* **aTime**
- *newVersion:* **ofLink** *sourceNode:* **aNode** *newPos:* **anInteger**
- *newVersion:* **ofLink** *destNode:* **aNode** *newPos:* **anInteger**
- *newVersion:* **ofLink** *newName:* **aName**

The method *createLink:* **linkName** *from:* **sourceNode** *spos:* **sInteger** *to:* **destNode** *dpos:* **dInteger** creates a new link **linkName** between **sourceNode** and **destNode,** at character positions **sInteger** and **dInteger.** Links are identified by unique names, so a check is done to verify that. The new link is added to the instance variable **linkSet** of the graph, and also to the instance variables **sourceLinks** and **destLinks** of the source and destination nodes respectively.

The method *deleteLink:* **aLink** deletes link **aLink** from the current version of the graph. Both the source and destination nodes of the link are versioned, and the link's attributes are deleted. The method *open-*

Link: **aLink** *versionTime:* **aTime** returns the oid of the appropriate version (specified by **aTime**) of link **aLink versionTime** is 0, then the current link is opened. Otherwise, the version list of the link is traversed, and the appropriate version returned.

The next three methods describe link versioning. Links are versioned when the position at which the link is attached to a node changes as a result of the contents of the either the source or destination node changing. Links are also versioned when their name changes. The method *newVersion:* **ofLink** *sourceNode:* **aNode** *newPos:* **anInteger** versions link **ofLink** when the contents of its source node change. A new version is created for the link, the value of its instance variable **fromPos** is set to **anInteger** (the new position of the link in the source node). Similarly, the method *newVersion:* **ofLink** *destNode:* **aNode** *newPos:* **anInteger** versions link **ofLink** when the contents of its destination node change. A new version is created for the link, the value of its instance variable **toPos** is set to **anInteger** (the new position of the link in the destination node). The method *newVersion:* **ofLink** *newName:* **aName** versions link **ofLink** when its name changes. The new name is **aName.**

4.3.5. Attribute

Attributes are defined for nodes, links and contexts. Figure 4.4 shows the class definition. The instance variables declared in this class are *name, creationTime, value, owner, fLink* and *bLink*. The instance variable *name* is the name of an attribute. The variable *creationTime* is computed from the the local computer's system clock and calendar. *Value* contains the contents assigned to an attribute, and is a string or an integer. The variable *owner* is the oid of the object for which the attribute has been defined. Instance variables *flink* and *bLink* are the oid's of the attribute which is ahead (behind) in the version list of an attribute.

The method *newForOwner:* **ownerItself** *name:* **attrName** *withValue:* **someContents** creates a new attribute for the object **ownerItself.** If **someContents** is a string, then an object of class StringAttribute (subclass of Attribute) is created, else an object of class IntegerAttribute (subclass of Attribute) is created.

The method defined in the above paragraph creates a new attribute. The methods that handle the definition, deletion and versioning of attributes for nodes and links are defined as follows.

- *addAttribute:* **attrName** *withValue:* **someValue** *toNode:* **aNode**

```
Object subclass: 'Attribute'
 instVarNames: #( 'name' 'creationTime' 'value'
                  'owner' 'flink' 'blink')
 classVars: #()
 poolDictionaries: #[]
 inDictionary: HyperModel
 constraints: #[#[#name, String]]
 isInvariant: false
```

Fig 4-4. Attribute Class Definition

```
Object subclass: 'Context'
 instVarNames: #( 'name' 'ownGraph' 'attributes'
                  'creationTime')
 classVars: #()
 poolDictionaries: #[]
 inDictionary: HyperModel
 constraints: #[]
 isInvariant: false
```

Fig 4-5. Context Class Definition

- *deleteAttribute:* **anAttribute** *ofNode:* **aNode**
- *newAVersion:* **ofAttribute** *with:* **aValue** *for:* **anElement**

The method *addAttribute:* **attrName** *withValue:* **someValue** *toNode:* **aNode** creates a new attribute with the name **attrName** and value **someValue.** This attribute is added to the variable *attributeSet* of the node, and to the instance variable *allNodeAttrs* of the graph. The method *deleteAttribute:* **anAttribute** *ofNode:* **aNode** deletes the attribute **anAttribute** from the current version of the node **aNode.** The method *newAVersion:* **ofAttribute** *with:* **aValue** *for:* **anElement** versions the attribute **ofAttribute** , and gives it the new value **aValue**

The methods pertaining to links are very similar:

- *addAttribute:* **attrName** *withValue:* **someValue** *toLink:* **aLink**
- *deleteAttribute:* **anAttribute** *ofLink:* **aLink**

The first method creates a new attribute with the name **attrName** and value **someValue** for link **aLink** and to the instance variable 'allLinkAttrs' of the graph. The second method deletes the attribute **anAttribute** from the current version of the link **aLink.**

4.3.6. Context

Contexts are defined for graphs. Figure 4.5 shows the class definition. The instance variables of this class are *name, creationTime, attributes* and *ownGraph.* The instance variable *name* is the name of the context, *creationTime* is computed from the the local computer's system clock and calendar. Instance variable *attributes* is a set of attributes defined for a context, and *ownGraph* is the oid of the graph that the context represents.

The following methods define context operations in GemDesign:

- *createNew:* **aName** *ofGraph:* **aGraph**
- *copyNode:* **aNode** *into:* **aContext**
- *mergeNode:* **aNode** *fromContext:* **aContext**

The method *createNew:* **aName** *ofGraph:* **aGraph** creates a new context, with a new empty graph. The method *copyNode:* **aNode** *into:* **aContext** copies node **aNode** from the current context into **aContext. aNode** could be a current node, or a version. The attributes of aNode are also copied. The links are copied, and maintained as cross context links in the graph of **aContext.** The method *mergeNode:* **aNode** *fromContext:* **aContext** merges node **aNode** from context **aContext** into the primary context. If this node already exists in the graph, then it is versioned, and a copy of **aNode** is the new current node. The links are

similarly dealt with, i.e., if the link already exits, then it is versioned, else a new link is created. Attributes are copied in the same manner.

The methods for creating and versioning attributes for contexts are

- *createStrAttribute:* **aString** *withValue:* **aValue**
- *cAtrUpdate:* **anAttr** *withVal:* **aVal**

The first method creates an attribute for the context, and adds it to the variable 'attributes' of the context. The second method versions an existing attribute **anAttr,** and assigns it the new value **aVal.**

4.4. General notes on the implementation

4.4.1. Access to database objects

The HAM maintains directories to represent the structure of the graph and its objects. These directories provide access to the basic components of the graph. GemDesign makes use of OPAL's indexed associative access mechanism, which is a way to efficiently access elements of large collections. This mechanism is implemented (in GemStone) using index structures such as B-trees on the object's instance variables. Using indexes it is possible to find values without having to do a full sequential search. In order to perform comparisons without message passing and to be able to build and maintain indexes on an

object's instance variables, OPAL needs some assurances about what kinds of objects those variables represent. As OPAL builds indexes that use as keys the values of instance variables within the elements of a collection, there is a requirement that these instance variables be constrained to contain only specified kinds of objects. OPAL can create two kind of indexes, identity and equality indexes.

In this implementation, objects are identified by unique names, and retrieved on the basis of their names, i.e., indexes are built (by OPAL) on this variable. To make this possible several constraints were needed. When a graph is created, an Equality index is created on its variables *nodeSet, linkSet, allNodeAttrs,* and *allLinkAttrs.* All these variables are sets, constrained to contain elements of only one class, class *Header.* The *name* instance variable of this class is constrained to be of class *String.*

It is pertinent to mention here that the instance variable *name* of objects in GemDesign provide a unique entry point into a graph, in the absence of an user interface with a pointing device. Navigation between links, nodes and attributes in the graph is done using objects identifiers.

4.4.2. Physical vs Logical Design

- Attributes

An attribute defined for a node (link) can be accessed from the node's (link's) instance variable *attributeSet,* as well as from the instance variable *allNodeAttributes* (*allLinkAttributes*) of the graph to which the node (link) belongs. This does not mean that there is more than one copy of a particular attribute, rather, both the instance variables mentioned above contain the object identifier of the particular attribute. This promotes the efficiency of the program by involving less search while accessing attributes.

- Versioning the contents of a node

In GemDesign, changing the contents of a node results in the versioning of that node with the new version containing the updated contents. Contents are represented by objects of class **Attribute,** and are kept as a linked list of its versions. Any change in the contents means that a new version will be created to contain the new, updated contents. This approach is in contrast to the system used in Neptune, where storage overhead is kept to a minimum by using a reverse delta to capture differences in the updated contents.

- Database Aspects

In Neptune, transactions and concurrency control mechanisms were explicitly managed. GemDesign did not have to implement this mechanism, since GemStone provides these facilities automatically. GemStone manages concurrent object access, and prevents concurrent operations from fouling shared objects by requiring users to encapsulate object accesses in sets of OPAL instructions called transactions.

4.5. The Interface

The user interface was implemented by porting Neptune's graphical interface layer, written in Smalltalk-80 to GemDesign. The GemStone-Smalltalk Interface (GSI) provides a set of SmallTalk classes that allow a Smalltalk application to communicate with the GemStone database. Figure 4.6 shows the structure of the GemStone-Smalltalk Interface. Class *GemStone* represents the GemStone database, while instances of class *GemStoneObject* are "proxies", within the Smalltalk object space, for corresponding objects in the GemStone database. *GemStoneMessage* implements the protocol for communicating with the GemStone Object Server. *Vserver* implements the communications layer to the GemStone Object Server.

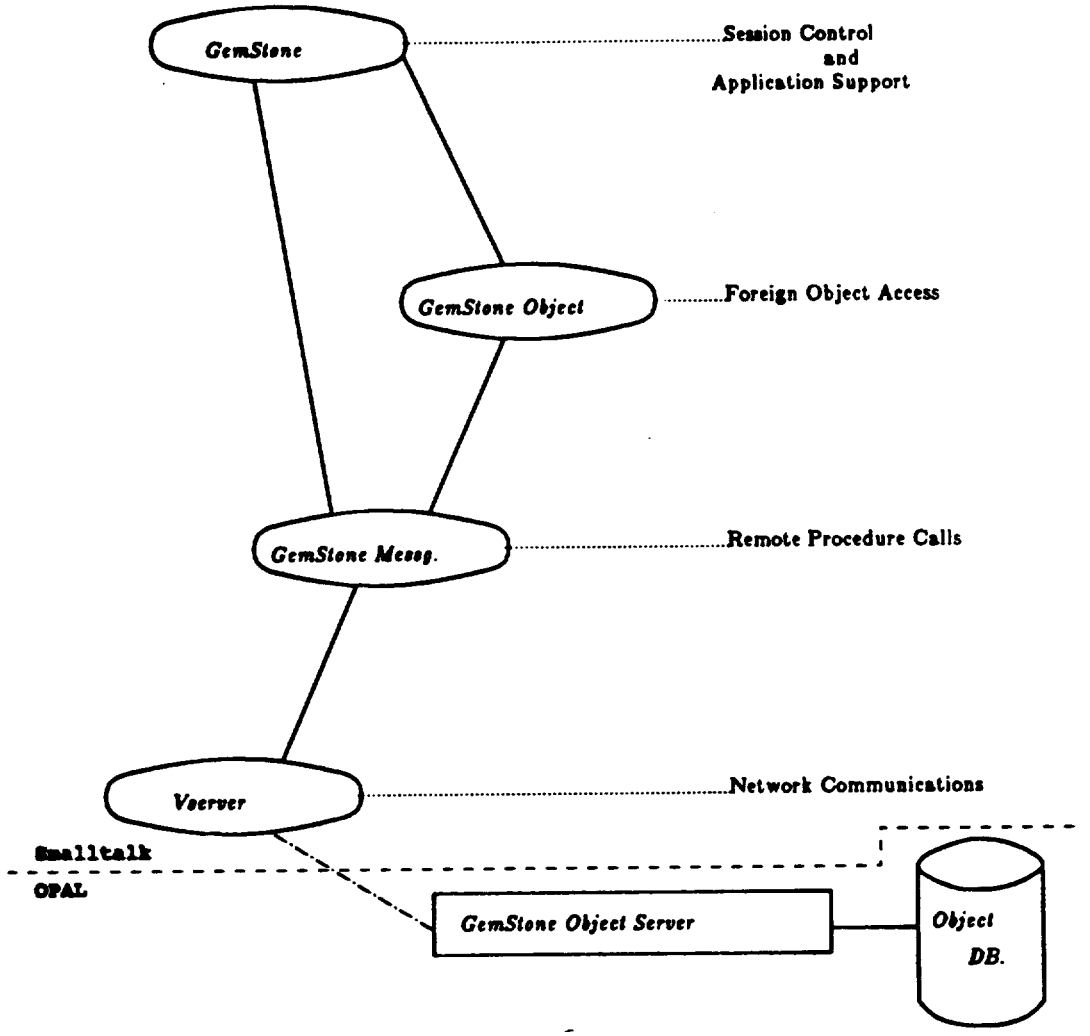


Fig 4.6 The GemStone SmallTalk Interface

(Servio-Logic Corp.'s
GemStone manual)

The following section discusses the classes in GemDesign that serve as a layer between the OPAL implementation of the hypertext model, and the Smalltalk-80 interface layer of Neptune.

4.5.1. The Interface Classes

The two primary classes used in GemDesign for the interface are *InterfaceElement* and *GSSTInterface*. These are detailed below:

- *InterfaceElement*

Every object created in the database is represented to Neptune through instances of this class. Fig 4.7 shows the class definition. The instance variables are *name, id* and *objectItself*. The instance variable *name* is the name of the database object, *id* is the identification number created for each object by GemDesign, and *objectItself* is a pointer to the object in the database. In GemDesign all objects are uniquely identified by their names. On the other hand, Neptune identifies an object with a unique identification number. The class *InterfaceElement* establishes a correspondence between objects in Neptune and GemDesign. When a new object is created, an instance of this class is also created to represent it. The instance variable *name* records the name of the object, (for future retrieval from the database), a new identification number is created for the object, recorded in instance variable *id* (for

```
Object subclass: 'InterfaceElement'
  instVarNames: #( 'name' 'id' 'element')
  classVars: #()
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: #[#[#name, String],
                 #[#id, Integer]]
  isInvariant: false
```

Fig 4-7 InterfaceElement Class Definition

```
Object subclass: 'GSSTInterface'
  instVarNames: #( 'allObjects' 'currentIndex' 'currentGraph'
                   'currentContext')
  classVars: #()
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: #[]
  isInvariant: false
```

Fig 4-8 InterfaceElement Class Definition

identification in Neptune) and this id is sent to Neptune.

Thus, class *InterfaceElement* provides a mapping between objects in Neptune and GemDesign. This level of indirection is essential to the port, since the alternative was recoding major portions of Neptune.

- *GSSTInterface*

This class communicates all the information from the hypertext graphs created in the database to Neptune's Smalltalk-80 interface. The messages that were sent to Neptune's HAM unit (HAM is Neptune's transaction-based server) from its Smalltalk interface are now sent to this class. This class then directs them to the appropriate method in GemDesign, and returns the results of executing these messages. Figure 4.8 shows the class definition. The instance variables are *currentGraph, currentContext, allObjects* and *currentIndex.* The instance variable *currentGraph* is a pointer to the graph that is currently active, and *currentContext* is a pointer to the context in which the graph is operating. The variable *allObjects* is a set, the elements of which are of class *InterfaceElement.* Each time the interface sends a message to this class, the elements of this set are searched for the appropriate object in the graph. This search is conducted by using OPAL's indexed associative access mechanism. The variable *currentIndex* is an integer that is used to assign numerical identification to new objects created in the graph.

```
Object subclass: #HyperRPC
 instanceVariableNames: 'toUnix fromUnix hyperRPCErrorCode'
                        'hyperRPCErrorMessage hyperRPCErrorParams'
 classVariableNames: 'CurrentInstance ToGS'
 poolDictionaries: ''
 category: 'Hypertext-KernelIO'
```

Fig 4.9 HyperRPC Class Definition

An example is provided to illustrate how this interface works. Figure 4.9 shows class *HyperRPC*. *HyperRPC* is the class in Neptune's Interface that allows the hypertext user interface subsystem (implemented in Smalltalk) to communicate with the hypertext abstract machine, the 'HAM' (implemented in C and running as a separate process under Unix). Instance variables *toUnix* and *fromUnix* are used to communicate with the HAM. These are implemented using 'pipes' (Unix), and messages are sent to the HAM one byte at a time. While this is an efficient and fast way of communication in Unix, the code required is quite complex.

Figure 4.10 shows a method from class *HyperRPC* (in Neptune's Small-Talk-80 interface), under the category 'node Operations'. This is an example of how Neptune's interface communicates with HAM, sending requests for and then receiving and storing information. This method gets the attribute identified by *attributeIndex* from node *nodeIndex* at *versionTime*. The variable *toUnix* sends information (*nodeIndex, attributeIndex, versionTime*) to HAM through pipes, and receives back some information from HAM (the value of the attribute).

In GemDesign, this method talks to GemStone instead of Unix. *HyperRPC* is the class in Neptune's Interface that handles all communication to Unix. The user interface of GemDesign has used Neptune's

```
getNodeAttributeValue: nodeIndex for: attributeIndex at: versionTime
    "gets the value of attributeIndex for node nodeIndex at time versionTime"
    | r |
toUnix nextPut: 54 . getNodeAttributeValue command
toUnix nextNumber: 4 put: nodeIndex.
  "nodeIndex"
toUnix nextNumber: 4 put: attributeIndex.
  "attribute index"
toUnix nextNumber: 4 put: versionTime.
  "versionTime"
self waitForResponse isNil ifTrue: [^nil].

r _ Array new: 2.  r at: 1 put: ((fromUnix next) = 1).
  "isString"
(r at: 1) ifTrue: [r at: 2 put: self getStringArgument ] .
  "string value"
            ifFalse: [r at: 2 put: (fromUnix nextNumber: 4) ] .
  "integer value"
```

Fig 4.10 Neptune method *getNodeAttributeValue: for: at:*

interface, with some modifications. Instead of communicating with Unix (on which the HAM is based), the interface needs to communicate with the GemStone database. To achieve this a class variable is created for class *HyperRPC, ToGS,* which is an instance of the Smalltalk class GemStoneObject. As mentioned in a previous paragraph, instances of class *GemStoneObject* are 'proxies' within the Smalltalk database for corresponding objects in the GemStone database. *ToGS* is initialized to contain an instance of GemStone class *GSSTInterface* (described above). All information from the interface will be relayed to GemDesign (on GemStone) through this class. For GemDesign, the method described in Figure 4.10 is replaced by that in Figure 4.11.

In Smalltalk, when an instance of class *GemStoneObject* (*ToGS*) finds the prefix 'gs' to a message, it removes the prefix and passes the message to its corresponding object in the database, which in this case is an instance of GemStone class *GSSTInterface.* Instead of reading values from Unix and creating the data structure represented by 'r' (Fig 4.9), the GemStone application gets all the information it needs, (through *nodeIndex, attributeIndex* and *versionTime*) and just passes the data structure *r.* The message *asLocalObject* sent to *r* replicates the data structure represented by *r* (which is a GemStone data structure) into a Smalltalk object.

```
getNodeAttributeValue: nodeIndex for: attributeIndex at: versionTime
    "gets the value of attributeIndex for node nodeIndex at time versionTime"
    | r |
r _ ToGS gsgetValue: nodeIndex ofAttribute: attributeIndex at: versionTime.
^ r asLocalObject
```

Fig 4.11 The method *getNodeAttributeValue: for: at:* modified for GemDesign.

```
getValue: nodeIndex ofAttribute: attributeIndex at: versionTime
    " gets value of attribute 'attributeIndex' for nodeIndex at versionTime"

| n theAttr attrName x  r |
n := self getNode: nodeIndex.
x := ( nodeIndex + attributeIndex - 1 ).
"Index of the system attribute. "
(versionTime = 0 ) ifTrue: [ theAttr := ( self getNodeAttribute: x ) ]
                ifFalse:
        [ n := currentGraph findVersion: versionTime ofNode: n .
attrName  := ( ( allObjects detect: [ :a | a.id = x])  name ) .
theAttr :=  n attributeSet detect: [ :a | a name = attrName ] ] .
r := Array new: 2.
   ( theAttr isMemberOf: StringAttribute )
     ifTrue: [ r at: 1 put: true]
     ifFalse: [ r at: 1 put: false] .
r at: 2 put: theAttr value.
^ r
```

Fig 4.12 The method *getValue: ofAttribute: at:* in GemDesign.

Replacing the code of the method *getNodeAttribute:for:at:* by a single message to the variable *ToGS* is not the whole story. As mentioned earlier in this chapter, an additional level of indirection has been added to the interface of GemDesign. Objects are identified by integers in Neptune, and strings in GemStone. This mapping is done in the method *getValue:ofAttribute:at:* (Figure 4.12) in class *GSSTInterface*.

This method first gets a pointer to the node specified by *nodeIndex*. (*n := self getNode: nodeIndex*). The method *getNode:* takes the numerical identification of the node as passed by the interface, maps this to the nodes name in GemDesign, and then gets a pointer to it. It then gets the current version of the node from the graph, (*currentGraph findVersion: ofNode*), and then gets the attribute. It creates a data structure similar to that created by the original HyperRPC method, and returns this structure to Neptune's interface.

4.6. Implementation Results

The object-oriented paradigm provided a very elegant vehicle for the implementation of a hypertext system. The components of hypertext, namely, graphs, nodes, links etc., were very naturally modeled as objects in the OPAL language. Modifications in design could be kept localized to the specific objects in which they arose, without concern

about any other parts of the code. Similarly, proposed extensions could also be handled easily. It took the researcher much less time than anticipated to finish and test the code for GemDesign.

The problems that arose were concerning the environment at the Oregen Graduate Center in which GemStone was running, and specifically, the serial connector between the Smalltalk workstation and the database. The Tektronix 4400-series workstation used for this thesis is connected to the uVax via an RS-232-C cable between the workstations serial port and one of the Vax's terminal ports.

Under this configuration, GemStone's response time is very poor. Actions such as bringing up the browser, the method categories and methods in the browser, performing a 'commit' to the database are extremely slow. For example, it takes on an average 20-30 seconds to bring up the GemStone browser, approximately 20 seconds to bring up a class category, another 15 seconds to bring up a class, 15-20 seconds to bring up method categories, and approximately 15 seconds to bring up a method. A deterioration of performance as compared to Smalltalk is naturally to be expected, due to the fact that the database is not housed on the same machine. But GemStone's response time over the RS-232 was very poor.

There were three reasons for this poor performance. The uVax, on which the GemStone database was running, is the slowest host for GemStone. Secondly, this was the first release of GemStone, and hence not as efficient as the subsequent releases. Thirdly, the RS-232-C cable that connected the uVax to the workstation was very slow. In addition, this network malfunctioned on an average 2 to 3 times a week during the period that it was used for the implementation of GemDesign. When the bridge to Neptune was written, the situation deteriorated and the network started crashing twice a day (5 hours of use). This was a serious problem, as the server would go down without warning, thereby returning the database to the previously committed state. As mentioned above, the response time of the network was very slow, making frequent commits to the database impractical.

The researcher has not worked with any other object-oriented database server, and has no other means for comparing the performance of this system with its contemporaries. Even so, the current GemStone installation is unsatisfactory as far as speed and reliability goes.

GemStone itself presented problems of its own. OPAL, GemStone's database language, does not provide any graphics classes at all. There is no support for user interfaces of any sort. This proved to be a real handicap, because it took the researcher almost as long to design and

implement the bridge to Neptune's Interface, as it took to design the actual hypertext system. GemDesign identifies objects by their name, while Neptune needs an integer identification. Due to this difference in naming systems, the mapping between these two systems took as much effort as GemDesign, and the code created was as complex as the underlying hypermodel. If OPAL had some provision for graphics, we would have written a much simpler interface to GemDesign in OPAL, thereby saving the effort of understanding Neptune's interface, and mapping it to GemDesign.

Amenities such as a 'file out' mechanism for code in the GemStone browser are not provided by GemStone. This means that the user has to individually copy methods out into files. (Subsequent releases of GemStone have corrected this). Error debugging facilities were not provided by GemStone in OPE release 1.3. An error during implementation is brought to the users notice through an error notifier, which simply mentions the kind of error that has occurred (e.g., "GemStone error: An attempt was made to store an object of type blah into an instance variable constrained to be of type blah1"). This does not give the user a clue as to the whereabouts of the error (Assuming that there are plenty of instances of the class whose instance variable is constrained to be of type blah1). Smalltalk, on the other hand, puts up a debugger that

takes the user right up to the method where the error was caused, and the user can also view the values of all variables in that method. This makes finding the reason for the error quite simple.

No break-down analysis was done on the amount of time taken to perform an operation in GemDesign. The important fact in this regard is that the time taken by the GemStone database to perform hypertext functions such as creating a node is orders of magnitude less compared to the total time taken by the current GemStone configuration at the Oregon Graduate Center to perform and display such a function. As mentioned in one of the preceding paragraphs, the performance of the network was very poor. The network's response time dominated GemStone's performance completely, and to the extent that measuring and analyzing running times of the two implementations (GemDesign and Neptune) seemed pointless.

CHAPTER 5

Conclusions

The goal of this thesis was to implement a prototype hypertext system (Tektronix's Neptune) in an object-oriented database. The implementation was completed using GemStone's database language, OPAL. OPAL is derived from Smalltalk-80, its syntax and semantics are almost identical to it.

The purpose of the thesis was to test the ease of using the object-oriented paradigm for a given application. This paradigm lent itself very well for the design of a prototype hypertext system. Nodes and links, which are the basic objects in a hypertext systems, were objects in OPAL as well. The behavior of objects was also very well captured. For example, operations on nodes such as creating, opening and modifying could be elegantly coded as methods for the corresponding class.

Many software systems can be naturally de-composed into objects. The functionality of such a system can be distributed amongst its object components, so that the system consists of indiviual objects, each describing what it does. The object-oriented paradigm with its object-message model exactly fits the requirements of many system software tasks today.

Thus the main results of implementing this thesis can be summarized as follows:

1. The prototype hypertext system was easily modeled by the object-oriented paradigm.

2. No comparative analysis could be performed to comment on the performance of the GemStone database. The primary reason was that GemStone's current network configuration at the Oregon Graduate Center performed very, very poorly, and completely dominated the performance of the GemStone database. Thus, this suggests that the current GemStone configuration at the Oregon Graduate Center is not appropriate for any practical application.

REFERENCES

[Andrews, Harris 87]
Timothy Andrews, Craig Harris
Combining Language and Database Advances in an Object-Oriented Development Environment
OOPSLA 1987 Conference Proceedings.

[Bloom, Zdonik 87]
Toby Bloom, Stanley B. Zdonik
Issues in the Design of Object-Oriented Programming Languages
OOPSLA 1987 Proceedings.

[Booch 86]
Grady Booch
Object Oriented Development
IEEE Transaction On Software Engineering, Vol SE12, No 2 February 1986.

[Bush 45]
Vannevar Bush
As We May Think
Atlantic Monthly, no.176, pp 101-108, July 1945.

[Campbell, Goodman 87]
Brad Campbell, Joseph Goodman
HAM: A General-Purpose Hypertext Abstract Machine
Hypertext 1987 Papers.

[Conklin 86]
Jeff Conklin
A Survey of Hypertext
1986 Microelectronics and Computer Tecnology Corporation (MCC).

[Copeland, Maier 87]
George Copeland, David Maier
Making Smalltalk a Database System
Proc. ACM/SIGMOD International Conference on the Management of Data, 1984.

[Cox 84]

Brad J Cox

Message/Object Programming: An evolutionary Change in Programming Tecnology

IEEE Software 1:1, January 1984.

[Cox 86]

Brad J Cox

Object-Oriented Programming An Evolutionary Approach

Addison Wesley, 1986.

[DeLisle, Schwartz 87a]

Norm DeLisle, Mayer Schwartz

A Hypertext System for Design Information management: Functional Specification

Technical Report No. CR-84-15a, CRL Tektronix Inc.

[DeLisle, Schwartz 87b]

Norm DeLisle, Mayer Schwartz

Contexts- A Partioning Concept for Hypertext

ACM Transactions on Office Information Systems, Vol.5 No.2, April 1987, Pages 168-186.

[DeLisle, Schwartz 87c]

Norm DeLisle, Mayer Schwartz

A Hypertext System for Design Information management: Design and Implemention

Technical Report No. CR-86-59, CRL Tektronix Inc.

[DeLisle, Schwartz 87d]

Norm DeLisle, Mayer Schwartz

Neptune: a Hypertext System for CAD Applications

Technical Report No. CR-85-50, 1986, CRL Tektronix Inc.

[Engelbart, English 68]

D C Engelbart, W K English.

A Research Center for Augmenting Human Intellect

AFIPS Conference Proceedings, Volume 33, Part 1 ,The Thompson Book Company, Washington DC, 1968.

[Goldberg 84]

Adele Goldberg

SMALLTALK-80 The Interactive Programming Environment

Addison-Wesley, 1983

[Goldstein, Bobrow 80]
Ira GoldStein, Daneil Bobrow
Description for a Programming Environment
Proceedings of the First Annual Conference of the National Association for Artificial Intelligence, Stanford, Ca, August 1980.

[Halasz et al. 87]
Frank Halasz, Thomas Moran, Randall Trigg
NoteCards in a Nutshell
CHI + GI 1987 , Toronto, Canada, April 5 -9 , 1987.

[Jacobson 87]
Ivar Jacobson
Object Oriented Development in an Industrial Environment
OOPSLA 1987 Proceedings.

[Kerr, Percival 87]
R K Kerr, D B Percival
Use of Object-Oriented Programming in a Time Series Analysis System
OOPSLA 1987 Proceedings.

[Maier, Price 84]
David Maier, D. Price
Data Model requirements for engineering applications
Proc. IEEE 1st Int. Workshop on Expert Database Systems, 1984, pp. 759 - 765 .

[Maier et al. 85]
David Maier, Allen Otis, Alan Purdy
Object-Oriented Database Development at Servio Logic
Database Engineering 8:4, Deccember 85.

[Maier et al. 86]
David Maier, Jacob Stein, Allen Otis, Allan Purdy
Development of an Object-Oriented DBMS
Proceedings of OOPSLA '86 , Portland Oregon.

[Maier, Stein 86]
David Maier, Jacob Stein
Indexing in an Object-Oriented DBMS
Tecnical Report CS/E-86-006, Oregon Graduate Center, Beaverton, Oregon, May 86.

[MacLennan 83]
Bruce J MacLennan
A View Of Object Oriented Programming
Naval PostGraduate School, Monterey California, February 1983.

[Meyer 87]
Bertrand Meyer
Reusability: The Case for Object-Oriented Design
IEEE Software, March 1987.

[Meyrowitz 86a]
Norman Meyrowitz
InterMedia: The Architecture and Construction of an Object-OrientedHyperMedia System and Applications FrameWork
Proceedings of OOPSLA '86, Portland Oregon.

[Meyrowitz 86b]
Norman Meyrowitz
Inheritence C Report
IRIS Tecnical Report 85-7, Institute for Research in Information and Scholarship, Providence, RI , July 1985.

[Micallef 88]
Josephine Micallef
Encapsulation, Reusibility and Extendibility in Object-Oriented Programming Languages
Journal Of Object-Oriented Programming, Vol.1, No.1, April/May 88.

[Nelson 80]
Theodore H Nelson
Replacing the Printed Word: A Complete Literary System,
1013-1023, IFIP Proceedings, October 1980.

[Nygaard 86]

Kristen Nygaard

Basic Concepts in Object Oriented Programming

SIGPLAN Notices, V 21 #10 October 1986.

[Penny, Stein 87]

D. Jason Penney, Jacob Stein

Class Modification in the GemStone Object-Oriented DBMS

Proceedings of OOPSLA '86, Portland Oregon.

[Purdy et al. 87]

Alan Purdy, David Maier, Bruce Schuchardt

Integrating an Object Server With Other Worlds

ACM Transactions on Office Automation Systems, April 87.

[Rentsch 82]

Tim Rentsch

Object Oriented Programming

SIGPLAN Notices, vol.17. no.9. p.51, September 1982.

[Robertson et al 81]

G.Robertson, D. McCracken, A.Newell

The ZOG Approach to Man-Machine Communication

International Journal of Man-MAchine studies, Vol.14 , 1981 .

[Seidwitz 87]

Ed Seidwitz

Object-Oriented Programming in SmallTalk and Ada

1987 Conference on Object-Oriented Systems, languages and Applications, October 1987.

[Smith, Zdonik 87]

Karen Smith, Stanley B Zdonik

Intermedia : A Case Study of the Differences Between Relational and Object-Oriented DAtabase Systems

Proceedings of OOPSLA '87.

[Snyder 86]

Alan Snyder

Encapsulation and Inheritance in Object-Oriented Programming Languages

Proceedings of OOPSLA '86.

[Stroustrup 88]
Bjarne Stroustrup
What is Object-Oriented Programming ?
IEEE Software, May 1988.

[Weyer, Borning 85]
Stephen H Weyer, Alan A. Borning
A Prototype Electronic Encyclopedia
ACM Transactions on Office Information Systems, Vol 3 No.1 January 85.

[Yankelovich et al. 85]
Nicole Yankelovich, Norman Meyrowitz, Andries van Dam
Reading and Writing the Electronic Book
IEEE Software October 1985.

[Yankelovich et al. 88]
Nicole Yankelovich, Bernard Haan, Norman Meyrowitz, Steven Drucker
Intermedia: The Concept and the Construction of a Seamless Information Environment
Computer January 1988.

APPENDIX

GEMDESIGN: The Code

Class Definitions: HyperModel

```
Object subclass: 'Attribute'
  instVarNames: #( 'name' 'creationTime' 'value'
                   'owner' 'flink' 'blink')
  classVars: #()
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: #[#[#name, String]]
  isInvariant: false


Object subclass: 'Context'
  instVarNames: #( 'name' 'ownGraph' 'attributes' 'creationTime')
  classVars: #( )
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: #[]
  isInvariant: false



Attribute subclass: 'DeletedAttribute'
  instVarNames: #()
  classVars: #()
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: #[]
  isInvariant: false


Object subclass: 'DeletedElement'
  instVarNames: #( 'name' 'creationTime' 'blink'
                   'flink')
  classVars: #()
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: #[]
  isInvariant: false



DeletedElement subclass: 'DeletedLink'
  instVarNames: #()
  classVars: #()
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: #[]
  isInvariant: false
```

```
DeletedElement subclass: 'DeletedNode'
  instVarNames: #()
  classVars: #()
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: #[]
  isInvariant: false


HyperGraph subclass: 'Graph'
  instVarNames: #( 'nodeSet' 'linkSet' 'allNodeAttrs'
                   'allLinkAttrs' 'allNames' )
  classVars: #()
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: #[]
  isInvariant: false

Object subclass: 'GSSTInterface'
  instVarNames: #( 'allObjects' 'currentIndex' 'currentGraph'
                   'currentContext')
  classVars: #()
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: #[]
  isInvariant: false


Object subclass: 'Header'
  instVarNames: #( 'name' 'element')
  classVars: #()
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: #[#[#name, String]]
  isInvariant: false


Object subclass: 'HyperGraph'
  instVarNames: #( 'name' 'rootContext' 'creationTime'
                   'contextSet')
  classVars: #( 'graphDirectory')
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: #[]
  isInvariant: false


Attribute subclass: 'IntegerAttribute'
  instVarNames: #()
  classVars: #()
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: #[]
```

```
  isInvariant: false


Set subclass: 'InterfaceSet'
  instVarNames: #()
  classVars: #()
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: InterfaceType
  isInvariant: false


Object subclass: 'InterfaceElement'
  instVarNames: #( 'name' 'id' 'element')
  classVars: #()
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: #[#[#name, String],
        .          #[#id, Integer]]
  isInvariant: false

HyperGraph subclass: 'Link'
  instVarNames: #( 'ofGraph' 'attributeset' 'fromNode'
                'toNode' 'flink' 'blink' 'fromPos'
                'toPos')
  classVars: #()
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: #[#[#name, String]]
  isInvariant: false


HyperGraph subclass: 'Node'
  instVarNames: #( 'ofGraph' 'attributeSet' 'sourceLinks'
                'destLinks' 'contents' 'flink' 'blink')
  classVars: #()
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: #[#[#name, String]]
  isInvariant: false


Set subclass: 'SetOfAttributes'
  instVarNames: #()
  classVars: #()
  poolDictionaries: #[]
  inDictionary: HyperModel
  constraints: Attribute
  isInvariant: false


Set subclass: 'SetOfHeaders'
  instVarNames: #()
```

classVars: #()
poolDictionaries: #[]
inDictionary: HyperModel
constraints: Header
isInvariant: false

Set subclass: 'SetOfLinks'
instVarNames: #()
classVars: #()
poolDictionaries: #[]
inDictionary: HyperModel
constraints: Link
isInvariant: false

Set subclass: 'SetOfNodes'
instVarNames: #()
classVars: #()
poolDictionaries: #[]
inDictionary: HyperModel
constraints: Node
isInvariant: false

Attribute subclass: 'StringAttribute'
instVarNames: #()
classVars: #()
poolDictionaries: #[]
inDictionary: HyperModel
constraints: #[]
isInvariant: false

Class ATTRIBUTE

Instance Protocol for class Attribute

- **Category: Instance Variables**

  name

  name: aName

  creationTime

  creationTime: aTime

  value

  value: aValue

  owner

  owner: anOwner

  flink

  flink: aLink

  blink

  blink: aLink

Class protocol for class Attribute

- **Category: Instance Creation**

  newForOwner: ownerItself *name:* attrName *withValue:* someContents

Class CONTEXT

Instance Protocol for class Context

- **Category: Instance Variables**

  name

  name: aName

  creationTime

  creationTime: aTime

  attributes

  attributes: aSet

  ownGraph

  ownGraph: aGraph

- **Category: Attribute Operations**

  createStrAttribute: aString *withValue:* anotherString

  createIntAttribute: aString *withValue:* anInteger

  cStrUpdate: anAttr *with:* aString

  cIntUpdate: anAttr *with:* anInteger

Class protocol for class Context

- **Category: Instance Creation**

  createNew: aName *ofGraph:* aGraph

Class GSSTINTERFACE

Instance Protocol for class GSSTInterface

- **Category: Instance Variables**

  currentIndex

  currentIndex: anIndex

  currentGraph

  currentGraph: aGraph

  currentContext

  currentContext: aContext

  allObjects

  allObjects: aSet

- **Category: Context Operations**

  getContextAttrValue: contextIndex *for:* attributeIndex *at:* versionTime

  setContextAttribute: contextIndex *withIndex:* attributeIndex *as:* boolVal *with:* aString *with:* anInteger

- **Category: Graph Operations**

  addNode

  createGraph

  openGraphPath: graphId *atTime:* crTime

  getGraph: versionTime *nodePredicate:* aString1 *linkPredicate:* aString2 *nodeAttrs:* na *linkAttrs:* la

  addLinkFrom: fromNodeIndex *to:* toNodeIndex *spos:* fromCurPos *dpos:* toCurPos *fromContextId:* fcontextId *toContextId:* tcontextId

  destroyGraph

- **Category: Node Operations**

  deleteNode: aNodeIndex

  deleteAttribute: attributeIndex *forNode:* nodeIndex

  getNodeAttributes: aNodeIndex *versionTime:* aTime

  setNodeAttrValue: nodeIndex *for:* attributeIndex *as:* boolVal *with:* aString *with:* anInteger

  getValue: nodeIndex *ofAttribute:* attributeIndex *at:* versionTime

  getNodeVersions: nodeIndex

  openNode: nodeIndex *nodeVersionTime:* aTime *nodeAttrs:* anInt1 *attrInd:* anAr1 *linkAttrs:* anInt2 *iAttrInd:* anArr2

  modifyContents: s *ofNode:* aNode *withLinks:* linkArr

- **Category: Link Operations**

  deleteLink: aLinkIndex

  deleteAttribute: attributeIndex *forLink:* linkIndex

  setLinkAttrValue: linkIndex *for:* attributeIndex *as:* boolval *with:* aString *with:* anInteger

  getValue: linkIndex *ofLinkAttribute:* atributeIndex *at:* versionTime

  getLinkAttributes: aLinkIndex *versionTime:* aTime

- **Category: Private**

  getNode: objectIndex

  getAttribute: objectIndex

  getLink: objectIndex

  getLinkAttribute: attrIndex

  getNodeAttribute: attrIndex

  getAttributeIndex: aString

  createIntElement: fromIndex

  findId: anObj

createCIntElement: fromIndex

makeName

extractAttributeValues: numAttrs *forIndices:* attrValues *forNode:* nodeIndex

extractAttributeValues: numAttrs *forIndices:* attrValues *forLink:* linkIndex

extractLinkAttributes: aLink

extractNodeAttributes: aNode

findTimeStamp: objId

findElement: anId

getTimeStamp: aNode *id:* anInteger

Class protocol for class GSSTInterface

- **Category: Instance Creation**

createNew

Class GRAPH

Instance Protocol for class GRAPH

- **Category: Instance Variables**

  allNames

  allNames: aSet

  allNodeAttrs

  allNodeAttrs: aSet

  allLinkAttrs

  allLinkAttrs: aSet

  linkSet

  linkSet: aSet

  nodeSet

  nodeSet: aSet

- **Category: Graph Operations**

  getGraphViaAttributes: nAttrName *value:* nAttrValue *linkAttr:* lAttrName *withValue:* nAttrValue *versionTime:* aTime

  linearizeGraph: rootNode *versionTime:* aTime *nodeAttrName:* aName *value:* aValue *linkAttrName:* sLAttrName *attrValue:* aLAttrValue

- **Category: Node Operations**

  createNode: nodeName *withContents:* someContents

  deleteNode: aNode

  openNode: aNode *versionTime:* aTime

  findVersion: aTime *ofNode:* aNode

  modifyContents: someContents *ofNode:* aNode *withLinks:* linkArr

changeName: aName *ofNode:* aNode

deleteAttribute: anAttribute *fromNode:* aNode

newAVersion: ofAttribute *with:* aValue *for:* anElement

addAttribute: attrName *withValue:* someValue *toNode:* aNode

updateLinks: newNode *with:* linkArr

- **Category: Link Operations**

createLink: linkName *from:* sourceNode *spos:* sInteger *to:* destNode *dpos:* dInteger *otherContext:* aContext1

.i deleteLink:
aLink

openLink: aLink *versionTime:* aTime

findVersion: aTime *ofLink:* aLink

newVersion: ofLink *sourceNode:* aNode *newPos:* anInteger

newVersion: ofLink *destNode:* aNode *newPos:* anInteger

newVersion: ofLink *newName:* aName

addAttribute: attrName *withValue* aValue *toLink:* aLink

- **Category: Private**

versionFromNode: aLink

versionToNode: aNode *ofLinks:* aLinkSet

deleteAllLinks: aNode

deleteDestLinks: aNode

deleteSourceLinks: aNode

deleteAInSet: anAttribute

deleteAttributes: anElement

createNewVersion: aNode

newCVersion: aNode *with:* someContents

createDelLink: aLink

getHeader: anElement

parsePred: string1

findVersion: versionTime *ofAttr:* anAttr

getGraphViaAttributes: string1 *value:* string2 *linkAttr:* string3 *withValue:* string4

Class Protocol for class Graph

- **Category: Instance Creation**

createGraph: aName

Class HYPERGRAPH

Instance Protocol for class HyperGraph

- **Category: Instance Variables**

  name

  name: aName

  rootContext

  rootContext: aContext

  creationTime

  creationTime: aTime

  contextSet

  contextSet: aSet

- **Category: Graph Access**

  addGraph: aGraph

  openGraph: aName

  deleteGraph: aName

Class protocol for class HyperGraph

- **Category: Instance Creation**

  createNew

Class InterfaceElement

Instance Protocol for class InterfaceElement

- **Category: Instance Variables**

  name

  name: aName

  id

  id: anInteger

  element

  element: anObject

Class protocol for class InterfaceElement

- **Category: Instance Creation**

  createNew: aName *andId:* anInteger *andElement:* theElement

VITA

The author was born on October 7, 1960 in Dhulia, India. She received a Bachelor of Commerce in Accounting and Computer Science from the University of Bombay, India in 1981. She also completed requirements of the Indian Institute Of Chartered Accountants for certification as a Chartered Accountant.

The author came to the US in 1983, and received a Bachelor of Science degree from the University of Phoenix, Arizona in 1985. She began studies in Computer Science at the Arizona State University. In 1986, the author moved to Beaverton, Oregon and enrolled in the Oregon Graduate Center as a part-time student in the Computer Science Department.

The author is married to Sunil Shenoy, and has a daughter, Anushka, age 3.

The author is working for Servio Logic Develpoment Corporation in Beaverton.