Neptune: a Hypertext System for CAD Applications

Norman Delisle
Mayer Schwartz

Computer Research Laboratory
Tektronix Laboratories
Tektronix, Inc
P O Box 500
Beaverton, Oregon 97077

ABSTRACT

Even though many of the essential notions of hypertext were first contained in the description of a "memex," written by Vannevar Bush in 1945 [Bus45], there are today only a few scattered implementations of hypertext, let alone any serious use of it in a CAD environment In what follows, we describe what hypertext is all about We describe a prototype hypertext system, named Neptune, that we have built We show how it is useful, especially its broad applicability to CAD

1 INTRODUCTION

Traditional databases have certain weaknesses when it comes to their use in Computer Aided Design (CAD) systems for electrical engineering, software engineering, and other design disciplines The most glaring weakness is the relative lack of support they give to version control and configuration management, though Katz and Lehman [KaL84] describe an experimental system that attacks one aspect of the version control problem Another weakness is that the traditional models (hierarchical, CODASYL, and relational) do not map well to the kinds of data which need to be stored in a CAD system However, the entity-relationship model, and other semantic models, seem to provide a better fit [BaK85] At the very lowest levels a relational model can be useful, possibly at the expense of performance [Lin84]

To support a large CAD application, many different kinds of data, both textual and graphical, need to be kept This data includes the design data itself, such as IC layouts, logic descriptions, timing descriptions and so forth in a VLSI CAD system It also includes source and object code in a CASE (Computer-Aided Software Engineering) system, as well as many forms of supporting documentation

Hypertext has typically been applied to documentation in all of its various guises Yankelovich *et al* give an excellent introduction to this view of hypertext systems [YMD85] We believe that hypertext can provide an excellent storage model for CAD systems In particular, hypertext can provide for complete version histories, for making arbitrary connections between pieces of data, and for interactively viewing and traversing the hypertext storage system

In Section 2 we give a brief description, history, and representative sampling of hypertext systems Section 3 gives an overview of Neptune, a hypertext system we have built at Tektronix Laboratories An Appendix gives a detailed description of the Hypertext Abstract Machine upon which CAD applications can be built This machine has been implemented and is currently running on top of Unix* 4 2 BSD In Section 4 we briefly describe how the hypertext abstract machine is being used to support a generic documentation application and briefly show how it can be applied to a CASE application In the last section of this paper we identify the major shortcomings of this approach and indicate future research directions

The original contributions reported in this paper are the recognition that hypertext can provide an appropriate storage model for CAD systems, the description of an abstract hypertext machine suitable for use in a CAD environment, as well as extensions of the hypertext notion itself These extensions include complete versions of "everything" and a query facility on

* Unix is a trademark of AT&T

attributes

2 HYPERTEXT

Hypertext in its essence is non-linear or non-sequential text In a hypertext system, documents consist of a collection of *nodes* connected by directed *links* A node by itself is similar to a piece of normal text — the links between nodes give hypertext its non-linear aspects The nodes of a hyperdocument are not restricted to be text They can represent graphical images, combined text and graphics, digitally encoded voice, or even an animation Either end of a link may be attached to a specific place within a node (e g a character position), a span (of text) within a node, or simply attached to the entire node Links can be made between a node in one document and another node in the same or a different document The complete collection of documents in a hypertext system can be thought of as one big *hyperdocument* If the nodes and links of a hyperdocument are mapped in the obvious way to nodes and edges of an abstract graph, then a hyperdocument can map into an arbitrary graph (with the possibility of cycles) called a *hypergraph*

2.1 Existing Hypertext Systems

Vannevar Bush was clearly being a futurist when he described his "memex" in 1945 [Bus45] Bush described the memex as a supplement to a person's own memory in which all of the person's books, records and communications are stored, the memex has an indexing scheme providing the functionality of hypertext links Nothing practical was done with Bush's ideas until the 1960's when Douglas Engelbart developed a hypertext-like system, originally named NLS but now called Augment, at the Stanford Research Institute [Eng63,EnE68] In many ways the system was ahead of its time, it introduced such notions as structured editing, using a mouse for cursor manipulation, and multi-person distributed editing [EnE68,Eng84] The term "hypertext" was coined by Ted Nelson more than a decade ago to describe Xanadu [Nel81], an electronic publishing system. A number of hypertext systems have been developed at Brown University [YMD85] including an early documentation system called FRESS, a graphically oriented hypertext system called the Electronic Document System [FND82], and a new system being developed called Intermedia Other hypertext systems include Xerox PARC's Notecards [HaT85], and the Electronic Encyclopedia [WeB85] CMU's ZOG system [RMN81] has some features of hypertext, but it is limited to hierarchically structured text Finally there is Neptune — the hypertext system described here

2.2 Properties of Hypertext Systems

This section outlines some of the important properties that have been demonstrated in existing hypertext systems

Editing Hyperdocuments

The most basic capability of hypertext systems is the ability to create (and delete) nodes and links, to modify the information contained within nodes, and to modify the structure of the hyperdocument. A complete version history of nodes and links may be maintained so that it is possible to see *any* version of the hyperdocument back to its beginning Neptune and Xanadu provide the capability for complete version histories at the granularity of "writes" from a text editor Both systems allow side-by-side comparison of different versions of the same node Most hypertext systems include a facility that ensures that a link attached to an old version retains an attachment in a corresponding place in a new version

Traversing the Hyperdocument

The directed-graph structure supported by hypertext systems can be used by authors as the means for structuring documents Because a document is frequently structured as a tree, several systems, including Zog, Augment and Neptune, provide special support for constructing and viewing hierarchical documents Links are also used as a cross-referencing device allowing a part of a document to reference or actually access a part of another document.

A hypertext document is browsed by traversing links Readers may restrict their attention to a single document by following only the links that serve to structure that particular document or readers may choose to follow diversions such as footnotes, references or annotations that are linked to the document. As a hypertext reader follows link after link in reading portions of hyperdocuments, he or she may want to keep a trail of which links were followed This trail allows other readers to follow the same path and makes it easier to resume reading a document after a diversion has been followed. A capability for saving a traversal history was a key component of Bush's memex and is supported both in Zog and in the Electronic Document System

Multimedia content

The name "hypertext" is actually a misnomer for many of the implementations. Several systems, including Augment, Xanadu, Notecards, Neptune and the Electronic Document System, do not restrict the con-

tents of a node to text In general the contents of a node in a hypertext system can be arbitrary digital data whose interpretation may include graphics, animations or digitized speech

Multi-person, distributed access

The concept of multi-person, distributed editing, allowing joint authorship, was pioneered by Augment. Several persons can access a hyperdocument simultaneously and the hyperdocument itself can be distributed over multiple, networked machines A hypertext implementation therefore must deal with concurrency control and recovery issues (e g , in case a site crashes in the middle of a hypertext transaction) Neptune has a central server which is accessible over a local area network from a variety of workstations, it is transaction-oriented and provides for complete recovery from any aborted transaction

Interactive User Interface

A hypertext document is meant to be viewed interactively As the reader views a node, visible links may be followed or not at the discretion of the reader If a link is followed, then the node at the end of the link is made visible so that it may be read in turn In a multi-window display with some sort of pointing device, the operation of following a link and viewing what it points to is straightforward to implement. The mapping of a hyperdocument to an abstract graph can also be made viewable, providing an alternate way of selecting a node for reading Both Neptune and Notecards include a pictorial view of a hyperdocument, and both provide a windowed user-interface

2.3 Applications of Hypertext

The most obvious application of hypertext is to documentation Ted Nelson in his book, *Literary Machines* [Nel81], describes an all-encompassing electronic publishing system, where all books and articles are in one gigantic hypertext system including even provision for royalties to be collected. The vision is awesome * An interesting documentation example is a simple dynamic history book developed at Xerox PARC by Stephen Weyer [Wey82] Weyer and Alan Borning have also done some related work in developing browsers for an animated encyclopedia [WeB85] — these browsers give a hypertext flavor to the encyclopedia.

* Imagine if all computer science, electrical engineering, and mathematics books, journals, technical reports, and conference proceedings were in one hypertext distributed worldwide and accessible via some network linked by satellite Any reference could immediately be tracked down Reader comments could be read by all Corrections could easily be made while providing easy access to previous versions.

3 AN OVERVIEW OF NEPTUNE

Neptune is designed as a layered architecture The bottom level is a transaction-based server named the Hypertext Abstract Machine (HAM) The HAM presents a generic hypertext model which provides storage and access mechanisms for nodes and links The HAM provides distributed access over a computer network, synchronization for multi-user access and transaction-based crash recovery.

Additional layers of functionality are built on top of the HAM Typically, one or more application layers are built on top of the HAM and a user interface layer is built on top of the application layers. The application layers consist of progams that automatically manipulate or transform hypertext data In a CAD application this layer could include VLSI design tools, high level language compilers or document processors The user interface layer can provide a windowed interface for browsing and editing hypertext data and for controlling application layer programs

Section 4 outlines some requirements for an application layer and describes a typical user interface layer The remainder of this section gives a summary of the HAM in enough detail to understand the rest of the paper The Appendix provides a more detailed description of the HAM

When we speak of Neptune, we are generally referring to the functionality provided by the HAM The HAM defines operations for creating, modifying and accessing nodes and links It maintains a complete version history of the hypergraph and provides rapid access to any version of a hypergraph The HAM makes no restrictions about the contents of nodes There is no interpretation at the HAM level — it is just binary data

Each end of a link can be attached to an offset within the contents of a node. If the node contains text, the offset can be interpreted as a character position If the node contains graphics, the offset could be interpreted as a cartesian or polar coordinate Additionally, Neptune supports two mechanisms for associating the link attachment with versions of a node, the link attachment may refer to a particular version of a node or it may always refer to the 'current' version of the node The former mechanism is a useful primitive for building a configuration manager. The latter form of attachment is best thought of as an automatic update mechanism, a history of link attachment offsets is saved, allowing the link to be attached to different offsets for each version of the node

The HAM provides two mechanisms that are particularly useful for building application layers First, an unlimited number of attribute/value pairs can be attached to a node or link Second, a demon mechanism is provided that invokes application or user code when a specific HAM event occurs, such as an update to a particular node

Two basic query mechanisms are supported by the HAM *traversal* and *query* The traversal mechanism, **linearizeGraph**, starts at a designated node and follows a depth-first traversal of out-links ordered by the links' offsets within the node The associative query mechanism, **getGraphQuery**, directly accesses a set of nodes and their interconnecting links Both of these mechanisms use predicates based on attribute/value pairs to determine which nodes and links satisfy the query As an example, suppose a user (or an application program) adopts the convention of attaching an attribute called *document* to each node This attribute is used to indicate which document the node contains, in a CASE system its values could include *requirements*, *design*, *sourceCode* and *objectCode* The node visibility predicate '*document = requirements*' could then be used in a **getGraphQuery** operation to access only those nodes that are part of the specification document.

Our goal was to put as little semantics as possible into the HAM, but still maintain performance and storage efficiency The range of applications that we considered (from documentation to CASE) places a heavy, though certainly not exclusive, emphasis on text and other interpretations of large chunks of binary data (such as executable binaries and bitmaps) The interpretation of node data is entirely up to the application which uses it. For example, an application could consider the data in a node to represent a set of constant length records or fields Because version control is a central theme of Neptune, we wanted effective storage of many versions of such data without copying each individual item, for nodes this is provided by backward deltas similar to RCS [Tic82] Attributes, on the other hand, as we envision them provide the semantics for these chunks (nodes) and the relationships (links) between them Attribute names and values tend therefore to be short strings of characters In view of the above considerations, we made a separation between the data in a node and the value of an attribute This separation led us to choose the particular HAM operations that we did

4 HYPERTEXT-BASED CAD SYSTEMS

For a CASE application, all documentation, source and object code, project management information and any other data associated with a design project are stored in hyperdocuments A hypertext system provides complete version histories, allows simultaneous access by project members and provides the capability for explicitly building links between related portions of the project information Additionally, hypertext provides the primitives needed to structure and organize the project data

This section outlines how we are using Neptune to build a CAD system for software engineering First, we give a brief overview of Neptune's generic documentation user interface Then, we outline how to take advantage of Neptune's capabilities to build CASE application layer programs

4.1 Neptune's Documentation User Interface

Neptune's user interface is implemented in Smalltalk-80 [Gol84]. The user interface process communicates with the HAM using a remote procedure call mechanism, the HAM runs as a separate process, typically on a machine accessed over a network The hyperdocuments and the contents of nodes are viewed and edited in display windows called *browsers* There are three primary kinds of browsers a *graph browser* provides a pictorial view of a sub-graph of nodes and links, a *document browser* supports the browsing of hierarchical structures of nodes and links, and a *node browser* views an individual node in a hyperdocument Several other browsers are provided by Neptune including attribute browsers, version browsers, node differences browsers and demon browsers

Graph Browsers

The graph browser shows a pictorial view of a hyperdocument or a portion of a hyperdocument. A graph browser that views this paper is shown in Figure 1 Each node is represented by an icon that consists of a name enclosed in a rectangle The user specifies the name associated with a node by attaching the attribute *icon* to the node and defining the desired character string as the attribute's value The graph browser itself has four panes, the upper pane contains the view of the graph, the lower left pane is a scroll area for zoom and pan operations, the two panes on the lower right contain text editors used to define the visibility predicates on nodes and links

Document Browsers

The document browser is designed to simplify the manipulation of hierarchically structured hyperdocuments Figure 2 shows a document browser viewing this paper It consists of five panes· the four upper panes contain lists of names of nodes, the lower pane is

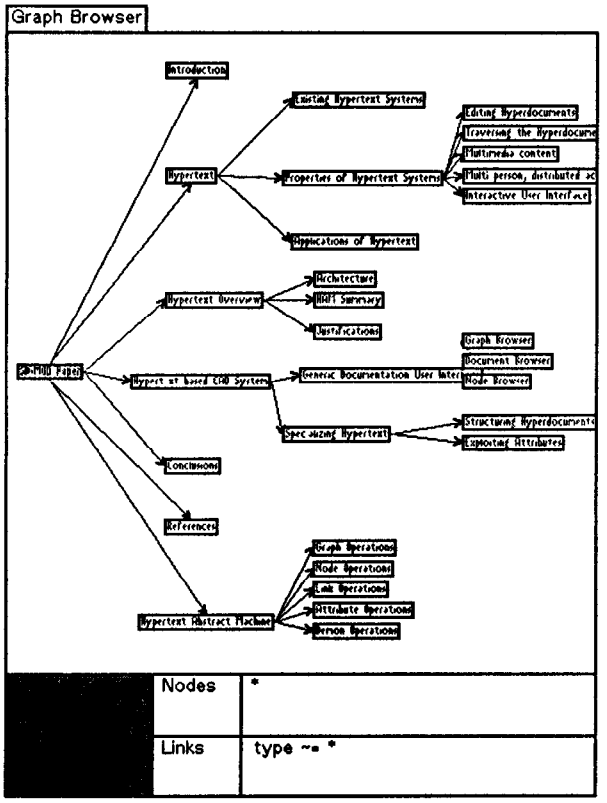

Figure 1 A Graph Browser

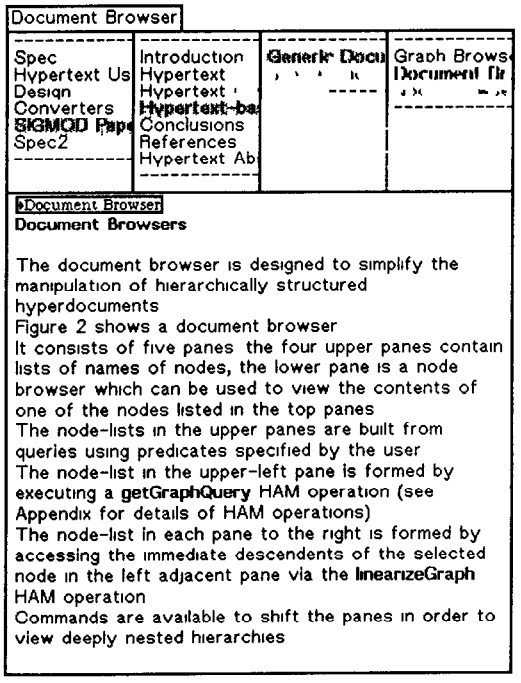

Figure 2 A Document Browser

a node browser which can be used to view the contents of one of the nodes listed in the top panes The node-lists in the upper panes are built from queries using predicates specified by the user The node-list in the upper-left pane is formed by executing a **getGraphQuery** HAM operation (see Appendix for details of HAM operations) The node-list in each pane to the right is formed by accessing the immediate descendents of the selected node in the left adjacent pane via the **linearizeGraph** HAM operation Commands are available to shift the panes in order to view deeply nested hierarchies

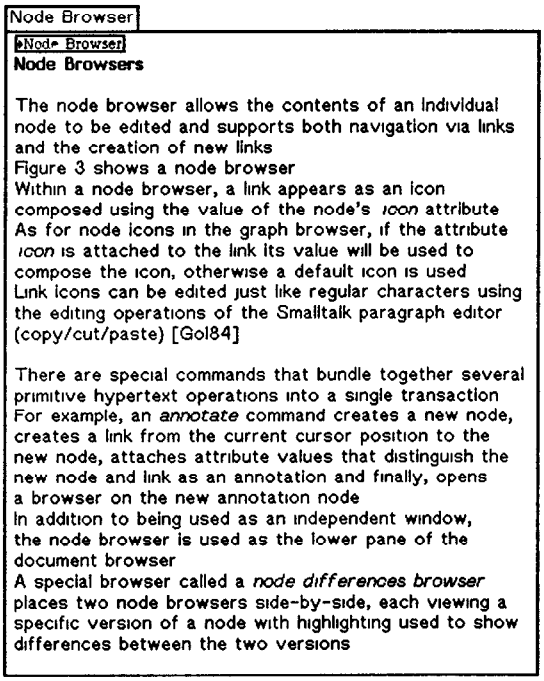

Figure 3 A Node Browser

Node Browsers

The node browser allows the contents of an individual node to be edited and supports both navigation via links and the creation of new links Figure 3 shows a node browser Within a node browser, a link appears as an icon composed using the value of the node's *icon* attribute As for node icons in the graph browser, if the attribute *icon* is attached to the link its value will be used to compose the icon, otherwise a default icon is used. Link icons can be edited just like regular characters using the editing operations of the Smalltalk paragraph editor (copy/cut/paste) [Gol84]

There are special commands that bundle together several primitive hypertext operations into a single transaction For example, an *annotate* command creates a new node, creates a link from the current cursor position

to the new node, attaches attribute values that distinguish the new node and link as an annotation and finally, opens a browser on the new annotation node In addition to being used as an independent window, the node browser is used as the lower pane of the document browser A special browser called a *node differences browser* places two node browsers side-by-side, each viewing a specific version of a node with highlighting used to show differences between the two versions

4.2 Specializing Hypertext for a CASE Application

Our primary motivation for building a hypertext system was to provide database support for software engineering environments Recent proposals describing project data base support for software engineering environments [Hun81, PeS85] repeatedly state the need to logically link together documentation and source code, the need for making annotations for recording explanations and assumptions, and the need for good version management. PSL/PSA, a popular software specification tool, can be thought of as a very specialized hypertext system [TeH77] PIE was an experimental system which allowed for multiple views of documentation as well as providing for design alternatives [GoB81] PIE was based on a network of nodes — a hypertext-like structure

Two questions must be addressed to determine how Neptune's primitives should be used for a particular CAD application First, how will nodes and links be used to represent each item of project data, second, what attributes need to be attached to each node and link

Structuring Hyperdocuments

The first question is probably the easier to answer Documents are typically organized as a hierarchy of sections and sub-sections This structure can be directly expressed in hypertext by using a node to represent each section or sub-section with links connecting each node to its immediate descendent sections or sub-sections If a section is lengthy, portions of the section can be broken out into separate nodes Additionally, if a section contains illustrations or tables, separate nodes can be used for these portions so that specialized editors can be used to view the node The HAM's **linearizeGraph** operation can be used to extract a document from the hypertext graph so that hardcopies can be produced

The static structure of program source code can also be directly represented using hypertext For example, a Pascal program is a simple hierarchy of nested procedures and functions and can be represented directly as a tree with a node for each procedure or function In a language like Modula-2 a program requires a directed graph to represent its static structure Each module can be represented by a simple tree similar to the Pascal program, the need for a directed graph is due to links that are used to specify imported modules Associated with each import list in a module is a link that points to the node representing the module being imported A compiler integrated with hypertext can use nodes for object code and symbol tables, links can be used to associate these objects with their source code

Within the framework outlined above, the question of how much or how little should go into a node is still not answered completely In our hypertext system the node is the atomic data unit. The **getGraphQuery** and **linearizeGraph** HAM operations return nodes, not portions of nodes, our link attachments point to positions within a node, not to spans of the node's contents Therefore, if a piece of information needs to be viewed in several distinct sub-graphs, then this information must be in a separate node For example, if a paragraph in a design document is also being used as a comment in the source code, then that paragraph should be represented by its own node An additional consideration involves the unit of incrementality that will be used to process the information For example, a compiler may be able to recompile a changed procedure individually, that is without recompiling the entire module that contains the procedure [SDB84, MeF81] In this case, the unit of incrementality of the compiler should be used to determine what syntactic code fragment the source code nodes represent

Exploiting Attributes/Value Pairs

Having determined how to represent CAD information as a directed graph, the next step is to decide how to use attributes to organize sub-graphs The primary objective is to make it easy to access all the information that is needed and to restrict the access to only what is needed

The general guideline for using attributes is that attributes attached to nodes describe what object the node represents and attributes attached to links describe the relationship that the link forms between two nodes In Neptune there are no limits to the number of attributes that can be attached to a node or link Additionally, both users of Neptune and application programs built on top of Neptune can define attributes Neptune's attribute/value pairs are very dynamic — at any time the user or an application program can attach an additional attribute to a node or link, delete an attribute attachment, or modify the value of an attribute The next two paragraphs outline how attributes could be used in a

Modula-2 CASE environment built on top of Neptune For a more complete treatment on how to use attributes in a CASE environment see [PeS85]

In a Modula-2 CASE environment every node has an attached attribute, named *contentType*, that identifies what the node contains (if this attribute is not attached, a default value is assumed) Values of contentType could include *text*, *graphics*, *Modula-2 source code*, *Modula-2 object code* or *Modula-2 symbol table* Additional attributes could be used to further describe the type of the node's contents For example, nodes that contain portions of a Modula-2 source program could have an attribute *codeType* with values that describe what kind of syntactic code fragment the node represents, such as *definitionModule*, *implementationModule*, or *procedure* Every link has an attached attribute, named *relation*, that names the relationship that the link denotes Values of 'relation' could include *isPartOf*, *annotates*, *references*, or *compilesInto*

Additional attributes can be assigned to nodes or links by the user or by the application level programs to provide semantic information that is useful for composing queries Examples include attributes to describe which document the node is contained in, what function of the software system the node describes, or management information such as which project team member is responsible for the node

5 CONCLUSIONS

The major shortcomings in the current definition and implementation of Neptune are two-fold In a multi-person design effort, there is frequently the need for an individual to try out tentative designs in that individual's own "private world" and then eventually to merge the chosen design back with the main design database There are currently no provisions for multiple version threads in any existing hypertext systems The second major shortcoming is that as defined, the functionality provided by demons is very weak There needs to be a set of parameters associated with each demon, such as the demon invoking event, an invocation time-stamp, or an identification of the invoking node or graph Examples of demon use could be sending mail to the person responsible for a node when someone other than that person modifies the node, performing special checking code when a node is modified, or invoking an incremental compiler when a node which contains code is modified.

The two current shortcomings of Neptune, identified above, are being addressed We have designed, and are currentlty implementing, a scheme for multiple version threads that allows multiple simultaneous *contexts* to exist in a given Neptune database These contexts can also be used for clustering related nodes and links as well as for configuration management We will be providing an environment that will allow parameterized demons to be written in Smalltalk, Modula-2, or C

There is a possible synergy, which is not currently being addressed, between the use of a relational database in conjunction with hypertext Hypertext can adequately capture the relationship between all the major pieces of information that are created as part of an engineering project Hypertext might not be as suitable for finer grained relationships such as definition-use links in an incremental compiler's symbol tables It could be very beneficial to combine the advantages that hypertext provides with those provided by a relational data base For example, given such fine grained information as a symbol table, one might want to find all references to a variable, not only in the code, but in all the documentation as well A relationally complete query language makes possible a wide range of interesting questions which can be asked

We have shown how hypertext can provide an appropriate storage model for CAD, and in particular CASE We have briefly decribed Neptune, a hypertext system currently being used in software development environments research Hypertext is particularly good as a storage system for all the information associated with a software (or hardware) project because it allows arbitrary structuring of the information, and it keeps a complete version history of the information and the structure In Neptune, we have provided a hypertext machine that is particularly suited for building applications, especially CASE systems

REFERENCES

[BaK85] Batory, D S and Kim, W Modeling concepts for VLSI CAD objects *ACM Transactions on Database Systems* **10**, 3 (Sep 85), 322-346

[Bus45] Bush, V As we may think *Atlantic Monthly* **176**, 1 (July 1945), 101-108

[EnE68] Engelbart, D C and English, W K A research center for augmenting human intellect. *AFIPS Proceedings, Fall Joint Computer Conference*, 33, 395-410

[Eng63] Engelbart, D C A conceptual framework for the augmentation of man's intellect In *Vistas in Information Handling, Vol 1*, P D Howerton and D C Weeks, eds Spartan Books, Washington, D C 1963, 1-29

[Eng84] Engelbart, D.C. Authorship provisions in Augment. *IEEE 1984 COMPCOM Proceedings*, Spring 1984, 465-472.

[FND82] Feiner, S., Nagey, S., and van Dam, A. An experimental system for creating and preserving graphical documents. *ACM Transactions on Graphics* 1, 1 (Jan. 1982), 59-77.

[Gol84] Goldberg, A. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley Publishing Company, Reading, Mass. 1984.

[GoB81] Goldstein, I. and Bobrow, D. An Experimental Description-Based Programming Environment: Four Reports. CSL-81-3 Xerox Palo Alto Research Center (Mar. 1981).

[HaT85] Halasz, F. and Trigg, R. personal communication.

[Hun81] Hünke, H., editor. *Software Engineering Environments* (Proceedings of the Symposium held in Lahnstein, Federal Republic of Germany, June 16-20, 1980). North-Holland Publishing Company, Amsterdam, the Netherlands, 1981.

[KaL84] Katz, R.H. and Lehman, T.J. Database support for versions and alternatives of large design files. *IEEE Transactions on Software Engineering* SE-10, 2 (Mar. 1984), 191-200.

[Lin84] Linton, M.A. Implementing relational views of programs. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburg, PA, April 1984, published as *SIGPLAN Notices* 19, 5 (May 1984), 132-140.

[MeF81] Medina-Mora, R. and Feiler, P.H. An incremental programming environment. *IEEE Transactions on Software Engineering* SE-7, 5 (Sep. 1981), 472-482.

[MeD82] Meyrowitz, N and van Dam, A. Interactive editing systems. *ACM Computing Surveys*, 14, 3 (Sep. 1982) especially pp. 339-340, 380-383.

[Nel81] Nelson, T.H. *Literary Machines*. T.H. Nelson, Swarthmore, PA., 1981.

[PeS85] Penedo, M.H., and Stuckle, E.D. PMDB — a project master database for software engineering environments. *Proceedings of the 8th International Conference on Software Engineering*, Aug. 1985, 150-157.

[RMN81] Robertson, G., McCracken, D. and Newell, A. The ZOG approach to man-machine communication. *International Journal of Man-Machine Studies*, 14, 461-488, 1981.

[SDB84] Schwartz, M.D., Delisle, N.M., and Begwani, V.S. Incremental compilation in Magpie. *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada, June 17-22, 1984, published as *SIGPLAN Notices* 19, 6 (June 1984), 122-131.

[TeH77] Teichrow, D. and Hershey, E. PSL/PSA: A computer aided technique for structured documentation and analysis of information processing systems. *IEEE Trans. on Soft. Eng.* SE-3, 1 (Jan. 1977), 41-48.

[Wey82] Weyer, S.A. The design of a dynamic book for information search. *International Journal of Man-Machine Studies*, 17, 87-107, 1982.

[WeB85] Weyer, S.A. and Borning, A.H. A prototype electronic encyclopedia. *ACM Transactions on Office Information Systems*, 3, 1 (Jan. 1985), 66-88.

[YMD85] Yankelovich, N., Meyrowitz, N., and van Dam, A. Reading and writing the electronic book. *Computer* 18, 10 (Oct. 1985), 15-30.

APPENDIX: Hypertext Abstract Machine Specification

The notation used for describing the hyperdata abstract machine operations is of the form:

operation: $operand_1 \times operand_2 \times \ldots \times operand_n \rightarrow result_0 \times result_1 \times result_2 \times \ldots \times result_m$

where $n \geq 1$ and $m \geq 0$. Each $operand_i$ and $result_j$ has a *domain* of values. Additionally, x^n means $x \times x \times \ldots \times x$ with n x's, x^* means x^n for some $n \geq 0$, and x^+ means x^n for some $n \geq 1$. The atomic domains used are the following (in alphabetical order):

Attribute:	an attribute name.
AttributeIndex:	unique identification for an attribute name.
Boolean:	either **true** or **false**.
Contents:	data which can reside in a node.
Context:	unique identification for the "current graph."
Demon:	a demon value.
Difference:	a deletion, insertion or replacement.
Directory:	a valid file directory name.
Event:	a demon event.
Explanation:	explanatory text.
LinkIndex:	unique identification for a hyperdata link.
Machine:	a valid computer name in a networking environment.
NodeIndex:	unique identification for a hyperdata node.
Position:	an ordinal number.
Predicate:	a Boolean formula in terms of attributes and their values.
ProjectId:	unique identification for a hyperdata graph.
Protections:	one of the possible file protection modes.
Time:	a non-negative integer representation for a given date and time.
Value:	an attribute value.

There are some additional domains which can described as combinations of the above domains:

LinkPt =	*NodeIndex* × *Position* × *Time* × *Boolean*
Version =	*Time* × *Explanation*

For all of the hyperdata abstract machine operations, $result_0$ has domain *Boolean*; $result_0$ is implicit. If the operation is successful then **true** is returned otherwise **false** is returned.

A.1 Graph Operations

createGraph: *Directory* × *Protections* → *ProjectId* × *Time*

Creates a new empty hyperdata graph, in *Directory* using *Protections* to set up the files representing the new hyperdata directory. If successful, returns *ProjectId*, a unique identification for the graph, and *Time*, the creation time of the new graph.

destroyGraph: *ProjectId* × *Directory* →

Destroys the existing graph, located in *Directory*. *ProjectId* must have the same value as returned by the createGraph operation that created the graph in the first place.

openGraph: *ProjectId* × *Machine* × *Directory* → *Context*

Opens an existing graph, located in *Directory* on *Machine*. *ProjectId* must have the same value as returned by the createGraph operation that created the graph in the first place. *Context* is the unique identification for the graph. This operation can trigger a demon.

addNode: *Context* × *Boolean* → *NodeIndex* × *Time*

Creates a new empty node in the graph given by *Context*. If *Boolean* has value **true** then a complete version history is maintained for the node. If successful, returns *NodeIndex*, the unique identification for the new node, and *Time*, the creation time of the node. This operation can trigger a demon.

deleteNode *Context* × *NodeIndex* →
Removes node *NodeIndex* from the graph given by *Context* All links into or out of the node are deleted This operation can trigger a demon

addLink *Context* × $LinkPt_1$ × $LinkPt_2$ → *LinkIndex* × *Time*
Creates a new link between two nodes $LinkPt_1$ represents the "from node" and $LinkPt_2$ represents the "to node " The from and to nodes must exist at their respective times If a *Time* is zero then the link always refers to the current version of the corresponding node If successful, returns *LinkIndex*, a unique identification for the new link and *Time*, the creation time of the link This operation can trigger a demon

copyLink *Context* × *LinkIndex* × $Time_1$ × *Boolean* × *LinkPt* → *LinkIndex* × *Time*
Creates a new link between two nodes where one end of the link is identical to that of link *LinkIndex* at $Time_1$ and the other end of the link is specified by *LinkPt* If *Boolean* has value **true** then the source of the new link is identical to that of *LinkIndex*, otherwise the destination of the new link is identical to that of *linkIndex* The link defined by *LinkIndex* must exist at $Time_1$ If a *Time* is zero then the link always refers to the current version of the corresponding node If successful, returns *LinkIndex*, a unique identification for the new link and *Time*, the creation time of the link This operation can trigger a demon

deleteLink *Context* × *LinkIndex* →
Removes link *LinkIndex* from the graph given by *Context* This operation can trigger a demon

linearizeGraph *Context* × *NodeIndex* × *Time* × $Predicate_1$ × $Predicate_2$ × $AttributeIndex_1^m$ × $AttributeIndex_2^n$ →
(*NodeIndex* × $Value^m$)* × (*LinkIndex* × $Value^n$)*
Returns a sub-graph of the graph given by *Context* at *Time*, composed by a depth first search via links starting at node *NodeIndex* Each of the nodes in *NodeIndex** satisfies $Predicate_1$, each link traversed satisfies $Predicate_2$ and each link in *LinkIndex** connects two nodes in *NodeIndex** For each node also returns $Value^m$ for the *m* requested attributes $AttributeIndex_1^m$ and for each link returns $Value^n$ for the *n* requested attributes $AttributeIndex_2^n$

getGraphQuery *Context* × *Time* × $Predicate_1$ × $Predicate_2$ × $AttributeIndex_1^m$ × $AttributeIndex_2^n$ →
(*NodeIndex* × $Value^m$)* × (*LinkIndex* × $Value^n$)*
Returns a sub-graph of the graph given by *Context* at *Time*, composed by all nodes and links such that each of the nodes in *NodeIndex** satisfies $Predicate_1$, each link traversed satisfies $Predicate_2$ and each link in *LinkIndex** connects two nodes in *NodeIndex** For each node also returns $Value^m$ for the *m* requested attributes $AttributeIndex_1^m$ and for each link returns $Value^n$ for the *n* requested attributes $AttributeIndex_2^n$

A 2 Node Operations

Each node is either an archive or a file Complete version histories are maintained for archives, only the current version is available for files

openNode *NodeIndex* × $Time_1$ × $AttributeIndex^m$ → *Contents* × *LinkPt** × $Value^m$ × $Time_2$
If successful returns the *Contents* for node *NodeIndex* If node *NodeIndex* is an archive then the *Contents* are at time $Time_1$; if $Time_1$ is zero then the *Contents* are current Also returned are the *LinkPt** attached to the desired version of the node as well as $Value^m$ for the *m* requested attributes $AttributeIndex^m$ The $Time_2$ returned is the version time of the current version of the node This operation can trigger a demon

modifyNode *NodeIndex* × *Time* × *Contents* × *LinkPt** →
Check in a node with modified *Contents*. *Time* must be equal to the version time of the current version of the node There must be a *LinkPt* for each link associated with the current version of the node If the node is an archive then creates a new version of node and a new version of each of its link attachments whose *Position* has changed This operation can trigger a demon

getNodeTimeStamp *NodeIndex* → *Time*
Returns the current version *Time* for node *NodeIndex*

changeNodeProtection *NodeIndex* × *Protections* →
Sets the protections for the file storing the contents of node *NodeIndex* to *protections*

getNodeVersions $NodeIndex \rightarrow Version_1^+ \times Version_2^*$
Returns the version history for node *NodeIndex* Major versions are updates to the contents of the node and are returned in $Version_1^+$ Minor versions are updates that relate to the node but do not change its contents, for example adding a link or defining an attribute value, they are returned in $Version_2^*$

getNodeDifferences $NodeIndex \times Time_1 \times Time_2 \rightarrow Difference^*$
For node *NodeIndex* returns in *Difference** the differences between the version at time $Time_1$ and at time $Time_2$

A 3 Link Operations

getToNode $LinkIndex \times Time_1 \rightarrow NodeIndex \times Time_2$
Returns the node *NodeIndex* and its version $Time_1$ corresponding to the destination of the link *LinkIndex* at time $Time_2$

getFromNode $LinkIndex \times Time_1 \rightarrow NodeIndex \times Time_2$
Returns the node *NodeIndex* and its version $Time_1$ corresponding to the source of the link *LinkIndex* at time $Time_2$

A 4 Attribute Operations

getAttributes $Context \times Time \rightarrow (Attribute \times AttributeIndex)^*$
For the graph given by *Context* returns in *(Attribute × AttributeIndex)** all the attributes, and their respective unique identifiers, that existed at time *Time*

getAttributeValues $Context \times AttributeIndex \times Time \rightarrow Value^*$
For the graph given by *Context* returns in *Value** the set of all values defined for attribute *AttributeIndex* at time *Time*

getAttributeIndex $Context \times Attribute \rightarrow AttributeIndex$
Returns the unique identification for *Attribute* in *AttributeIndex* If no attribute *Attribute* exists, then creates one

setNodeAttributeValue $NodeIndex \times AttributeIndex \times Value \rightarrow$
Sets the value for attribute *AttributeIndex* for the node *NodeIndex* to *Value* If the node *nodeIndex* is an archive then creates a new version of the attribute value

deleteNodeAttribute $NodeIndex \times AttributeIndex \rightarrow$
Deletes the attribute *AttributeIndex* for the node *nodeIndex*

getNodeAttributeValue $NodeIndex \times AttributeIndex \times Time \rightarrow Value$
Returns *Value* for attribute *AttributeIndex* at time *Time* for node *NodeIndex*

getNodeAttributes $NodeIndex \times Time \rightarrow (Attribute \times AttributeIndex \times Value)^*$
Returns in *(Attribute × AttributeIndex × Value)** all the attributes, their respective unique identifiers, and their values that existed at time *Time* for node *NodeIndex*

setLinkAttributeValue $LinkIndex \times AttributeIndex \times Value \rightarrow$
Sets the value *Value* for attribute *AttributeIndex* for the link *LinkIndex* If the link *LinkIndex* is attached to an archive then creates a new version of the attribute value

deleteLinkAttribute $LinkIndex \times AttributeIndex \rightarrow$
Deletes the attribute *AttributeIndex* for the link *LinkIndex*

getLinkAttributeValue $LinkIndex \times AttributeIndex \times Time \rightarrow Value$
Returns *Value* for attribute *AttributeIndex* at time *Time* for link *LinkIndex*

getLinkAttributes $LinkIndex \times Time \rightarrow (Attribute \times AttributeIndex \times Value)^*$
Returns in *(Attribute × AttributeIndex × Value)** all the attributes, their respective unique identifiers, and their values that existed at time *Time* for link *LinkIndex*

A 5 Demon Operations

setGraphDemonValue *Context* × *Event* × *Demon* →
For the graph given by *Context* sets the demon corresponding to event *Event* to *Demon* Creates a new version of the graph demon If *Demon* is null then demon is disabled

getGraphDemons *Context* × *Time* → *(Event* × *Demon)**
For the graph given by *Context* returns the demons for the graph at time *Time*

setNodeDemon *NodeIndex* × *Event* × *Demon* →
For node *NodeIndex* sets the demon corresponding to event *Event* to *Demon* Creates a new version of the node demon If *Demon* is null then demon is disabled

getNodeDemons *NodeIndex* × *Time* → *(Event* × *Demon)**
For node *NodeIndex* returns the demons for the node at time *Time*