A Hypertext System for Integrating Heterogeneous, Autonomous Software Repositories[*]

John Noll and Walt Scacchi
Information and Operations Management Dept.
University of Southern California
Los Angeles, CA 90089-1421

Abstract

Hypertext is a simple concept for organizing information into a graph structure of linked container objects. This paper examines issues involved in applying hypertext concepts to the integration of heterogeneous, autonomous software repositories, and presents a solution called the Distributed Hypertext System (DHT). Based on a hypertext data model and client-server architecture, DHT features powerful modeling capabilities, integration of heterogeneous, preexisting repositories, update with concurrency control, and full local autonomy.

1 Introduction

Considerable research has been devoted to software object management. Table 1 summarizes some of the results. For the most part, these efforts assume that the objects in the repository are stored in a central database or storage manager, which may be accessed by distributed clients. In reality, however, software artifacts may be stored in many diverse locations under independent control. This may happen due to multi-contractor development projects, corporate mergers, cooperative relationships among otherwise autonomous organizations, or introduction of new repository technology into existing environments. It may also happen when seeking to support the distributed development of software systems and related documents over the Internet.

Consider the following two examples.

Multiplatform project. A software development group is developing a database application to run in a client-server environment. The database server runs on a Unix host; the clients will run on PCs with PC operating systems.

Developers of the database software will use the traditional Unix programming tools (cc, make, RCS) to develop and test their code. PC programmers will use native integrated development environments. In addition, there will be a shared network file system so all platforms can share files.

The group desires tools for doing analysis on the project artifacts taken as a whole, to do requirements traceability, impact of change analysis, and metric collection.

Despite the logical centralization afforded by a shared filesystem, there is still considerable heterogeneity present. Each programming environment has its own mechanisms for storing dependency relationships, function cross references ("tags"), and versions. Thus, the global analysis tools will

[*]Appears in Proc. 3rd. Irvine Software Symposium, ACM SIGSOFT and UC Irvine, Irvine, CA, 49-60, April 1994.

System	Concurrency	Collections	Granularity	Query	Versioning
SoftBench[7]	optimistic	directories[a]	file	no	tool-level[b]
AtFS[19]	locks[c]	directories	file	attr.[d]	yes
NSE[1]	copy-edit-merge	directories	file	no	yes
SLCSE[29]	via dbms[e]	no	record	dml[f]	yes
PMDB[25]	locks	no	record	attr.	no
Aspect[5]	locks	no	record	dml	yes
AD/Cycle[21]	locks	trees	record, file	dml	tool-level
NuMil[22]	via dbms	directories	file	dml	yes
Atherton[24]	versions	directories	object	attr.	yes
Triton[15]	optimistic	complex objects	object	no	no
PCTE/OMS[3]	locks	trees	file	no	yes
CAIS-A[5]	nested trans.	directories	file	no	?
PGRAPHITE[30]	optimistic	collection	primitive types	no	no

[a] "Directories" refers to hierarchical containers of objects or other directories.
[b] Tool-level versioning implements versioning through tools built on top of the repository layer.
[c] "Locks" refers to the "check-out" style of long duration locking used by RCS.
[d] "Attr." refers to searching for nodes with matching attribute values.
[e] SLCSE and NuMil use the underlying relational dbms to provide concurrency control.
[f] Dml queries use a full-featured data manipulation language that would be found in a DBMS.

Table 1: Software engineering repositories.

have to retreive data from several different "repositories".

Collaborative development across the Internet. Several teams of researchers throughout North America have been assembled to collaborate on research into sharing resuable software process models. Each team is expected to provide technical reports that document their individual research results, as well as to cooperate in the production of project-wide reports on their collective findings. Each of these documents will evolve in both structure and content over the course of the project. Additionally, the teams intend to develop a shared collection of reusable process assets and experience reports that can be accessed over the Internet, although each team will use their pre-exisiting repositories.

Due to geographic separation and the loose cooperative structure of this project, each team will be highly autonomous, managing its own computing environment and tools. Thus, there will be no shared file system or database to serve as an integrating resource. Similarly, we cannot assume that all teams will agree to adopt and use any one team's object storage manager or data model. Nonetheless, it will be necessary for each team to be able to modify jointly developed documents or process assets.

Each of these two scenarios indicate that there is a need to organize and manage dispersed collections of related software objects across autonomous computing environments, as well as heterogeneous repositories and data models. To date, little research has been conducted on how to manage software engineering artifacts, whether as hypertext nodes, files, or database entities, that are stored across many heterogeneous, autonomous repositories.

The Distributed Hypertext system (DHT) is a research project at USC exploring the application of Hypertext concepts to the integration of heterogeneous, autonomous software repositories. The goal is to devise a scheme for software object management that provides the following:

1. Services to support software engineering activities that require concurrency control and versioning;
2. Integration of pre-existing, heterogeneous repositories that preserve autonomous control over local repositories and existing applications;
3. Transparent object access; and
4. A simple implementation strategy.

Note that in a heterogeneous environment composed of autonomous, pre-existing repositories, some of these requirements are difficult to meet. For example, concurrency control based on data-processing style transactions violates execution autonomy by requiring participants to defer to a global transaction manager. Likewise, atomic operations applied to collections of objects may violate association or execution autonomy if they require participants to coordinate the commit or abort of such operations. Finally, some repositories (such as relational database systems) do not have built-in support for versioning or full-text searching. Solutions to these issues must take repository autonomy and the variation in the capabilities of heterogeneous storage managers into account.

The problem has three aspects:

Heterogeneity McLeod and colleagues define the "spectrum of heterogeneity" for databases, comprised of several facets: data model, schema, object comparability, data format, and storage manager [13]. This is equally applicable to software repositories. We find a variety of data models, from file-oriented to relational to object-oriented. Likewise, schema and object comparability are problems: one person's module is another's subsystem; dependency relationships may be called by different names, such as "derived-from", "depends-on", etc.

Autonomy Our view of autonomy addresses three concepts: *design* autonomy reflects the degree to which a repository can specify its own data model, schema, storage manager, applications, etc; *execution* autonomy is the ability of a repository to dictate who has access to what objects, in what fashion; *association* autonomy refers to the freedom of a repository to choose with what other repositories, if any, it will cooperate [28].

Transparency Schatz [27] defines three types of transparency: *type* transparency, the ability to apply the same operations to any type object; *location* transparency, the ability to access remote objects in the same manner as local objects; and *scale* transparency, the ability of a system to perform as well with one million as with 1000 objects. To this we add *source* transparency, the ability to apply the same operations to an object regardless of where it is stored.

This property has to do with differences among storage managers and the access techniques they provide for storing and retrieving objects. Source transparency requires that one should be able to retrieve an object using the same command or access function regardless of how the object is stored.

Note that this is not the same as location transparency. Location transparency requires only that the same commands or access functions apply to both local and remote repositories. Thus, the same SQL commands would apply to a local or remote relational database; the open() function should open a remote or a local file.

In contrast, source transparency requires that there be a single function (say, GetObject()) that would retrieve an object from a relational database or a file system or any other repository type.

In the following sections, we present the DHT system and show how it addresses the issues above to meet our stated requirements.

2 DHT Architecture and Data Model

We now present a brief overview of DHT, a typical example of which is depicted in figure 1. A more detailed description of an earlier version of DHT can be found in [23]. However, this report highlights refinements and enhancements incorporated into DHT in its current version.

Architecture. The DHT architecture is based on a client-server model. Clients implement all application functionality that is not directly involved in storage management; a client is typically an individual tool, but may be a complete environment.

Several servers manage the objects (nodes or links) in the hypertext, where different servers can manage nodes and links. Each server consist of two components: the repository that owns and manages local objects, and a gateway process that transforms local objects into DHT nodes and links, and DHT messages into local operations (see figure 2). From the repository viewpoint, the gateway process can be just another application.

Note that the gateway process is equivalent to a combination of transforming and accessing processor components of the federated database architecture reference model described by Sheth and Larson [28]. The style of access implemented, however, is navigational rather than query-oriented, as discussed later in section 4.

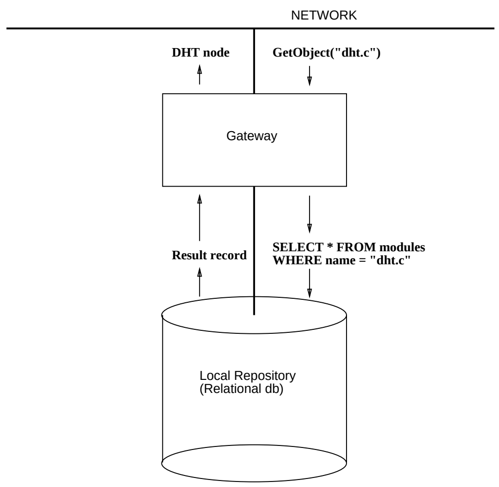
NETWORK
DHT node
GetObject("dht.c")
Gateway
Result record
SELECT * FROM modules WHERE name = "dht.c"
Local Repository (Relational db)

Figure 2: A server for a software module database.

A request-response style communication protocol implements the operations specified in the DHT data model (see section 2), and includes provisions for locating servers and authenticating and encrypting messages.

Data Model. The DHT Data Model defines hypertext primitives that provide sufficient modeling power to represent software engineering concepts, without compromising the autonomy of local repositories. It consists of four basic objects: *nodes*, the content objects; *links* that model relationships among nodes; *anchors*, that specify the points within node contents that anchor the endpoints of a link; and *contexts*, that contain links to allow specification of object compositions as sub-graphs. Nodes, links, and contexts have types, attributes, and unique object identifiers (oids).

A fixed set of operations can be applied to DHT objects: *create*, *delete*, *read*, and *update* an object. Additionally, *retrieve links* can be applied to a context to obtain the links within a context associated with a specified node. The important feature of these operations is that any one can be performed by a single repository on its own objects. Cooperation among repositories is not required. In addition, a given repository can

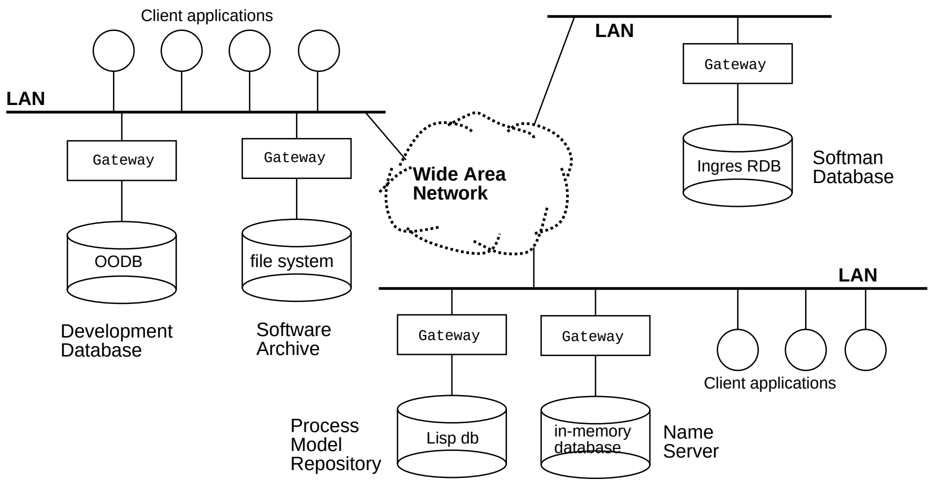

Figure 1: A typical DHT environment.

elect to provide any subset of these operations, as appropriate for the level of access it intends to provide.

3 Issues

Some additional issues related to support of software engineering, such as versioning, aggregations, and concurrency control, must be addressed. We discuss these next.

Versioning. Since participating repositories may not have native support for versioning, it is a difficult issue to address. There are three possible alternatives for versioning in hypertext (figure 3): An object identifier (oid) can refer to all of the versions together, as in HAM [8]. In this case, a link to the object can refer to any version of the object. Alternately, an oid can refer to the most recent version. In this case a link always points to the current version of an object. Finally, an oid can refer to a specific version in the history of an object's evolution, in which case a link always points to the specified version. Subsequently, links that point to the most recent version are not possible, because each new version gets a distinct new oid.

Each of these alternatives can be represented using DHT primitives. The first requires a repository to manage a collection of versions as a context. The second is simpler, requiring the repository to assign a new oid to the last version of an object before updating the current version. The last alternative simply requires creation of a new entity. In addition, note that all three require relationships to be maintained as links among the versions.

The choice of versioning model depends on the application. For example, to build a specific configuration of a system, specific versions of each module must be selected. A configuration can be modeled as a dependency graph where the nodes are specific versions of a module, hence the third versioning alternative is appropriate.

On the other hand, a programmer's workspace should include the most recent versions of each module that the programmer is working on, indicating that the second versioning alternative (where links point to the most recent version of a node) would be most appropriate.

Finally, a tool tracing requirements will likely refer to modules implementing particular requirements. In this case, it is important to know which module implements a requirement. This case would likely consider all versions as part of the same whole, thus the first

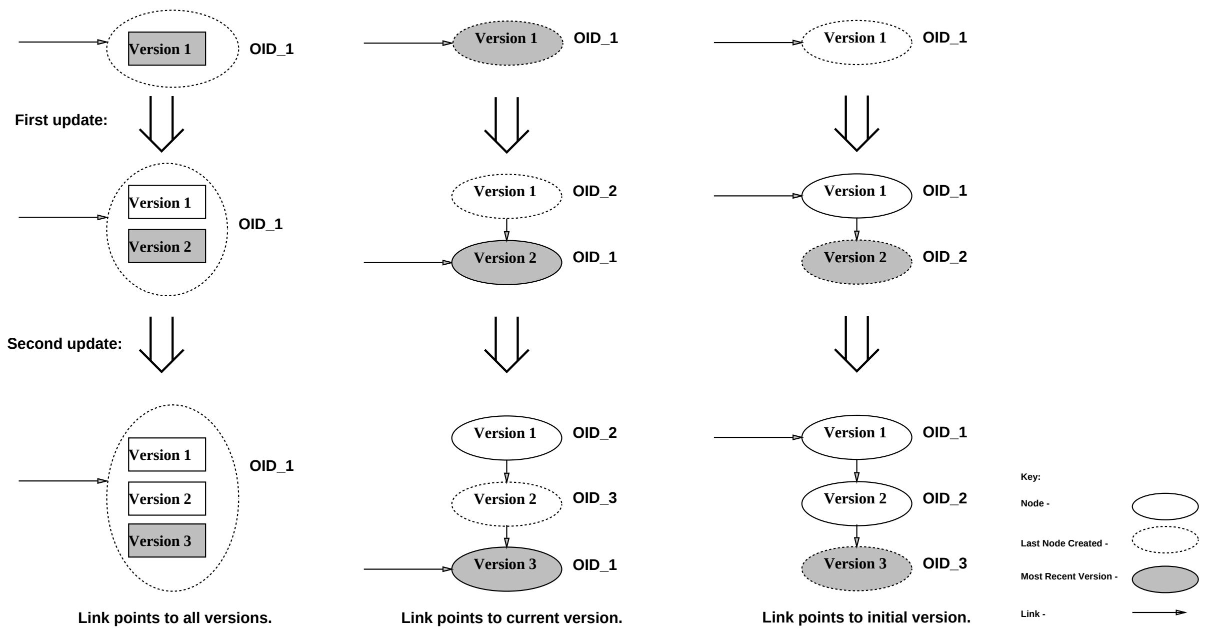

Figure 3: Versioning models.

alternative is appropriate.

Aggregation. Aggregate object are common in software engineering environments: a project is a collection of lifecycle documents; a system is compiled from a set of software modules. It is therefore desirable to have a modeling construct that allows sets of objects to be referred to as a single entity. Aggregation, however, introduces conflicts between the goals of transparency and autonomy. Transparency requires that the same operations (create, get, update, delete) apply to aggregations as to unit objects; this may conflict with autonomy constraints depending on the semantics of delete operations applied to aggregates. For example, if delete implies that the components of the composite object are deleted as well, and the composite contains objects from more than one repository, the operation will violate execution and association autonomy because the affected repositories will need to coordinate to ensure that the components are deleted.

To overcome this difficulty, aggregation is modeled by contexts in DHT. A context specifies a set of links, which in turn can describe a variety of graph structures from simple sets of nodes to trees, DAGs, and general graphs. However, deleting a context only deletes its links. The nodes connected by the links remain unaffected. Since a context's links are created and managed by the same repository that owns the context, autonomy is preserved.

Concurrency Control. Providing concurrency control presents problems similar to those of aggregation, because each repository must be free to implement its own concurrency mechanism (or none at all). As such, we cannot assume that there is a common notion of concurrency control. The challenge is to provide a concurrency mechanism that does not violate a repository's autonomy.

As discussed in Section 2, DHT operations apply to single objects, and can be performed by one server without coordination with other servers. The DHT architecture exploits this feature to provide a form of timestamp concurrency control [18, pp. 380-383] designed to prevent "lost updates".

When a client reads an object, the server returns a *timestamp* along with the object data; this timestamp is used to determine whether subsequent updates conflict. The client includes the timestamp with any subsequent update requests on the object. The server, upon receiving an update request, examines the timestamp to determine when the last update occurred in relation to the timestamp. If the update precedes the timestamp, the update succeeds; if an update has occurred since the timestamp, the server refuses the update and returns an error message. The client then takes whatever action it deems appropriate.

4 Related Work

We now compare the DHT approach to possible alternatives, as represented by (i) heterogeneous database management systems, (ii) hypertext systems for software development, and (iii) proposed "standard" object models.

Heterogeneous Database Management Systems. HDBMSs provide query-oriented access to data stored in heterogeneous component databases. There are three broad classes of such systems[4]: global schema multidatabases, in which applications access data through a single unified schema [9, 12, 26]. federated databases, wherein *import schemas* are used to provide access to external data through the existing local database schema [16, 13]; and multidatabase language systems, that retrieve data by posing queries directly to participating databases using a multidatabase query language [20, 6].

They main difference between heterogeneous databases and DHT is the style of access. Heterogeneous databases provide query oriented access, with the accompanying requirement for query processors, schedulers, and local translators for the global query language. As a result, most heterogeneous database systems assume that components are also databases. In contrast, DHT objects are accessed primarily by navigation, which requires a much simpler local interface, and less capability from the component repositories.

Hypertext. A number of research projects have applied hypertext to software object management, including the Hypertext Abstract Machine (HAM) [8], the Documents Integration Facility (DIF) [14], and HyperCASE [11]. For the most part these are based on a single, centralized repository architecture. In contrast, systems such as PROXHY [17] and Chimera [2] seek to move away from this position. These two systems enable linking among objects from diverse sources. Subsequently, links are managed by the hypertext system, while nodes are managed by individual applications. This contrasts sharply with the DHT approach, which attempts to insulate applications from the details of object storage management through a uniform access interface.

Standardized Object Models Program integration mechanisms such as the Common Object Request Broker (CORBA), Object Linking and Embedding (OLE), and the IBM System Object Model (SOM) specify interfaces to potentially heterogeneous objects. However, these mechanisms focus on sharing application behavior, such as the rendering capability of a drawing tool or calculation functions of a spreadsheet. In these cases, the data objects used by the application (graphic file or spreadsheet) remain hidden behind the object interface. Thus, they are more appropriate for application integration and interoperability rather than data integration.

5 Discussion and Observations

Conklin describes the "essence" of hypertext as a combination of three components: a *data access method* based on navigation; a *representation scheme* similar to

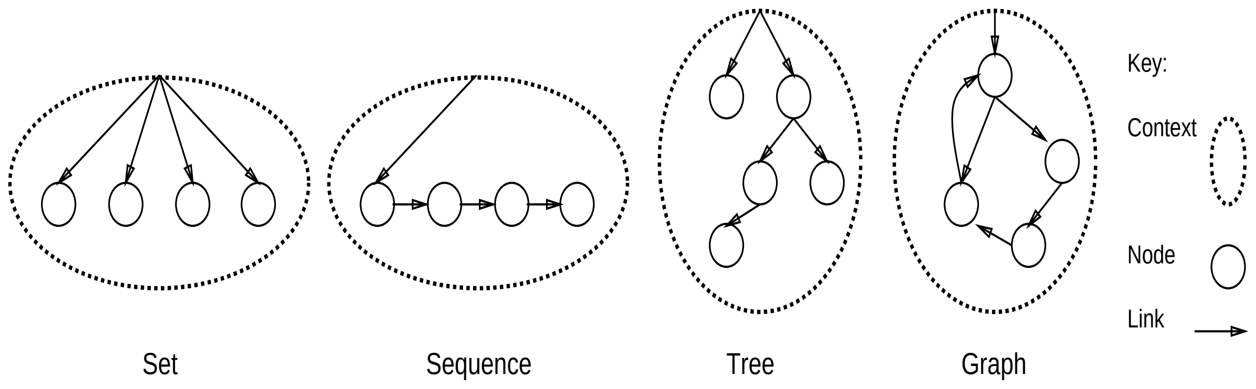

Figure 4: Modeling aggregation in DHT.

a semantic net; and an *interface modality* incorporating browsing by direct manipulation of link anchors [10]. By exploiting this multifaceted nature of hypertext, the DHT solution embodies a comprehensive approach to software object management across heterogeneous environments, with the following benefits:

(1.) The navigational style of access is an intuitive, natural mode for interacting with the structured and semi-structured (textual, graphical) data prevalent in software environments.

(2). DHT lends itself to very simple gateway implementations, since only get and put operations are required. Most of the implementation of a gateway consists of code to manage connections and process requests and replies. This code is repository independent and can therefore be reused. The current DHT prototype provides a library of basic server functions for this purpose. The repository-dependent part of a gateway comprises an interface to the local storage manager, and must be hand crafted for each repository. This task, however, is straightforward requiring the implementation of eight basic storage management interface functions that are called by the server library (see Table 2). (Note that a simple read-only gateway need only implement OpenRepository(), CloseRepository(), and GetObject().)

OpenRepository()
CloseRepository()
CreateObject()
DeleteObject ()
GetObject()
UpdateObject()
GetAttributes()
GetLinks()

Table 2: Storage manager interface functions.

Repository type	Gateway Size
Unix File system	1400
Relational database	900
Object-oriented database	600

Table 3: Gateway implementations (lines of code).

Our experience shows these are simple to implement. For instance, we have been able to get gateways running within a single day of programming effort. Examples of actual effort to implement several gateways, measured in lines of C language source code, are shown in Table 3.

(3.) The semantic net representation scheme of attributed nodes and links enables multiple, flexible structures to be overlayed on the same core set of objects. This means that users can organize objects in different

ways to suit their specific needs. For example, users can access shared nodes using the same or independent contexts. Additionally, links can represent relationships between *parts* of objects (via anchors), in addition to linking objects as a whole.

(4.) The user interaction technique complements navigational access with a common interaction style that applies to all software object types, yielding application- and type-transparent interfaces.

6 Conclusions

We believe the DHT concept and architecture propose an interesting solution to a research problem and a practical problem. The research problem entails how to provide object management services to a dispersed group of autonomous, heterogeneous software object repositories. The practical problem is how to provide the geographically dispersed software development teams with a logically centralized repository of sharable and reusable software assets, where each has its own locally developed artifact repository. An evolving prototype implementation of DHT which demonstrates a solution to these problems, in support of distributed software development environments, is operational. Further, experience with DHT access performance shows that its retrieval ("get") and storage ("put") operations across Internet sites rivals that of local-area network file systems for many different types of repositories. Thus, we believe these results begin to demonstrate the viability of the DHT concept, architecture, and implementation.

References

[1] Evan W. Adams, Masahiro Honda, and Terrence C. Miller. Object management in a CASE environment. In *Proceedings of the 11th International Conference on Software Engineering*. The Association for Computing Machinery, 1989.

[2] Kenneth M. Anderson, Richard N. Taylor, and E. James Whitehead, Jr,. Hypertext for heterogeneous software environments. Technical report, University of California, Irvine, 1993.

[3] Gerard Boudier, Ferdinando Gallo, Regis Minot, and Ian Thomas. An overview of PCTE and PCTE+. In *SIGSOFT '88: Proceedings of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston*. The Association for Computing Machinery, 1988.

[4] M. W. Bright, A. R. Hurson, and Simin H. Pakzad. A taxonomy and current issues in multidatabase systems. *IEEE Computer*, March 1992.

[5] Alan W. Brown. *Database Support for Software Engineering*. Halstead Press (John Wiley and Sons), 1989.

[6] Omran A. Bukhres, Jiansan Chen, Weimin Du, Ahmed K. Elmagarmid, and Robert Pezzoli. Interbase: An execution environment for heterogeneous software systems. *Computer*, 26(8):57–68, August 1993.

[7] M. H. Cagan. The HP SoftBench environment: An architecture for a new generation of software tools. *The Hewlett-Packard Journal*, 41(3), June 1990.

[8] Brad Campbell and Joseph M. Goodman. HAM: A general purpose hypertext abstract machine. *Communications of the ACM*, 31(7), July 1988.

[9] Chin-Wan Chung. DATAPLEX: An access to heterogeneous distributed databases. *Communications of the ACM*, 33(1):70–79, January 1990.

[10] Jeff Conklin. Hypertext: An introduction and survey. *Computer*, 20(9):17–41, September 1987.

[11] Jacob L. Cybulski and Karl Reed. A hypertext based software-engineering environment. *IEEE Software*, March 1992.

[12] Umeshwar Dayal and Hai-Yann Hwang. View definition and generalization for database integration in a multidatabase system. *IEEE Transactions on Software Engineering*, SE-10(6):628–644, November 1984.

[13] D. Fang, J. Hammer, D. McLeod, and A. Si. Remote-exchange: An approach to controlled sharing among autonomous, heterogenous database systems. In *Proceedings of the IEEE Spring Compcon, San Francisco.* IEEE, February 1991.

[14] Pankaj K. Garg and Walt Scacchi. A hypertext system for software life cycle documents. *IEEE Software*, 7(3):90–99, May 1990.

[15] Dennis Heimbigner. Experiences with an object manager for a process-centered environment. Technical Report CU-CS-484-92, University of Colorado at Boulder, 1992.

[16] Dennis Heimbigner and Dennis McLeod. A federated architecture for information management. *ACM Transactions on Office Information Systems*, 3(3), July 1985.

[17] Charles J. Kacmar and John J. Leggett. PROXHY: A process-oriented extensible hypertext architecture. *ACM Transactions on Information Systems*, 9(4):399–420, October 1991.

[18] Henry F. Korth and Abraham Silbershatz. *Database System Concepts.* McGraw-Hill, 1986.

[19] Andreas Lampen and Axel Mahler. An object base for attributed software objects. In *Proceedings of the EUUG Autumn '88 Conference, London.* European UNIX Users Group, October 1988.

[20] Witold Litwin and Abdelaziz Abdellatif. An overview of the multi-database manipulation language MDSL. *Proceedings of the IEEE*, 75(5):621–631, May 1987.

[21] V. J. Mercurio, B. F. Meyers, A. M. Nisbet, and G. Radin. AD/Cycle strategy and architecture. *IBM Systems Journal*, 29(2):170–187, 1990.

[22] K. Naryanaswamy and Walt Scacchi. A database foundation to support software systems evolution. *The Journal of Systems and Software*, 7(1):37–49, March 1987.

[23] John Noll and Walt Scacchi. Integrating diverse information repositories: A distributed hypertext approach. *IEEE Computer*, 24(12):38–45, December 1991.

[24] William Paseman. Architecture of the Atherton software BackPlane. In *Proceedings of 1989 ACM SIGMOD Workshop on Software CAD Databases.* The Association for Computing Machinery, 1989.

[25] Maria H. Penedo and E. Don Stuckle. PMDB–a project master database for software engineering environments. In *Proceedings of the 8th International Conference on Software Engineering.* IEEE, August 1985.

[26] Ming-Chien Shan. Unified access in a heterogeneous information environment. *IEEE Office Knowledge Engineering*, 3(2), August 1989.

[27] B.R. Shatz. Telesophy: A system for manipulating the knowledge in a community. In *Proc. Globecom 87*, pages 1181–1186, New York, Sept. 1987. The Association for Computing Machinery.

[28] Amit P. Sheth and James A. Larson. Federated database systems for managing distributed, heterogeneous, and au-

tonomous databases. *ACM Computing Surveys*, 22(3), September 1990.

[29] Tom Strelich. The software life cycle support environment (SLCSE) a computer based framework for developing software systems. In *SIGSOFT '88: Proceedings of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston.* The Association for Computing Machinery, 1988.

[30] Jack C. Wileden, Alexander L. Wolf, Charles D. Fisher, and Peri L.Tarr. PGRAPHITE: an experiment in persistent typed object management. In *SIGSOFT '88: Proceedings of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston.* The Association for Computing Machinery, 1988.