ABC: A Hypermedia System for Artifact-Based Collaboration

John B. Smith and F. Donelson Smith

Department of Computer Science
CB#3175, Sitterson Hall
University of North
Chapel Hill, NC 27599-3175
jbs@cs.unc.edu or smithfd@cs.unc.edu
919-962-1792 or 919-962-1884

ABSTRACT

Our project is studying the process by which groups of individuals work together to build large, complex structures of ideas, and we are developing a distributed hypermedia system to support that process. This description includes a brief overview of the system, but emphasizes three components: a hypermedia data management system or *graph server*, a set of *browsers* for working with graph objects, and a set of *applications* for working with data contents of graph nodes. A number of research issues are raised and discussed in context, including: composite objects; anchored links; scaling up for large applications; partitioning the hypermedia graph; consistency and completeness across subgraphs; and an open, extensible architecture for applications.

INTRODUCTION

Our research focuses on the process of collaboration and on technology to support that process. We are concerned with the intellectual collaboration that is required for software development and other similar tasks in which groups of people work together to build large, complex structures of ideas.

The work of such groups -- either directly or indirectly -- is concerned with producing some form of *artifact*. For software development, that artifact consists of concept papers, architecture, requirements, specifications, programs, diagrams, reference and user manuals, as well as administrative documents; for other tasks, the artifact may contain these and/or other kinds of information. Our project is studying how groups merge their ideas and their efforts to build the artifact, and we are developing a computer system to support that process.

Our work builds on hypertext in two ways. First, hypertext provides a conceptual model for the artifact. It gives us a way to think about its organization and structure so that we can identify individual documents or components and also define and use relations among them. Second, hypertext technology is particularly well-suited for building systems with which to develop and use the artifact.

We elected to build, rather than buy, a hypertext system for several reasons. Most important, we have not been able to identify an existing system that meets all of the needs of our project. Second, there are a number of important and interesting problems in hypertext research that can be addressed by the experience and prior work of our group. In the remainder of this paper, we first explain in more detail the concept of the artifact, then provide a brief overview of the system we are building; after that we discuss several

issues raised by this work that are important if hypertext is to become a widely used technology for large projects.

ARTIFACT

The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions. This essence is abstract in that such a conceptual construct is the same under many different representations. It is nonetheless highly precise and richly detailed.

I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation. [Brooks, 1987]

According to Brooks, the *fundamental* problem in software development is building the large "conceptual construct" that we call the *artifact*. What a software development team *does* is produce the various types of information -- documents, diagrams, programs, etc. -- that constitute the system and its accompanying description. These materials are usually thought of and handled as separate objects, each likely to be stored in a separate file. Yet, they have strong semantic and symbolic relations: an idea included in a concept paper becomes a requirement that is then specified and implemented before being tested, maintained, and explained to users. A key problem is maintaining consistency across these various representations of the concept -- e.g., as a system is built, programmers alter the initial design but don't update the design documents. Other problems include the coherence, completeness, and correctness of these materials.

Hypermedia* does not automatically solve these problems but it does provide a technology and a way of thinking that are helpful. The various materials that constitute a software project can be viewed as a large graph structure. Individual documents can be treated as trees whose nodes contain components such as chapters, sections, paragraphs, diagrams, etc.; the source code for the system can be thought of as a second graph whose nodes contain programming language statements. Different graph components that are semantically related through cross references, function calls, inclusion, etc., can be explicitly related, through links, in a hypertext data model. Figure 1 illustrates this concept: the four horizontal planes denote documentation, source code, object code, and test data. The shaded nodes represent different manifestations of the same module, as it is described, implemented, compiled into object code, and tested. The vertical plane, on the right, shows a projection of this module across the four horizontal contexts.

Figure 2 shows a high-level view of a hypertext data model we built for a sample of materials for IBM's Systems Network Architecture (SNA). Indicated are the basic types of information and generic link relationships among them. These diagrams server both as a visual representation of the information organization and as a user interface for access. At the center of "The Big Picture" are four basic types of system information: data type definitions; programs, referred to as FAPL code and Finite State Machines; prose documentation; and interface descriptions (called protocol boundaries). The "Little Picture" shows additional link relationships among these four central components, omitted from the "Big Picture" for purposes of clarity. At the bottom of the "Big Picture" are two forms of information that provide additional means of access to the core

* For the remainder of this discussion, we use the term *hypermedia*, instead of *hypertext*, to indicate the inclusion of different types of information, or media, in the system, including human and programming language text, 2-D graphics, recorded audio and video, etc.

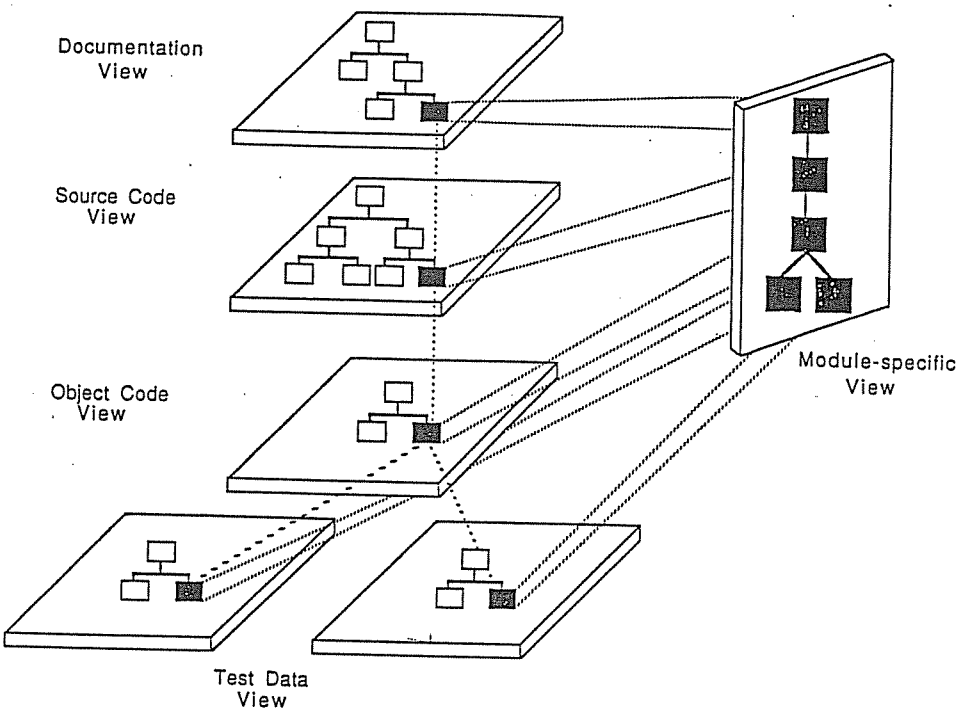

Figure 1. Central Artifact: Schematic View.

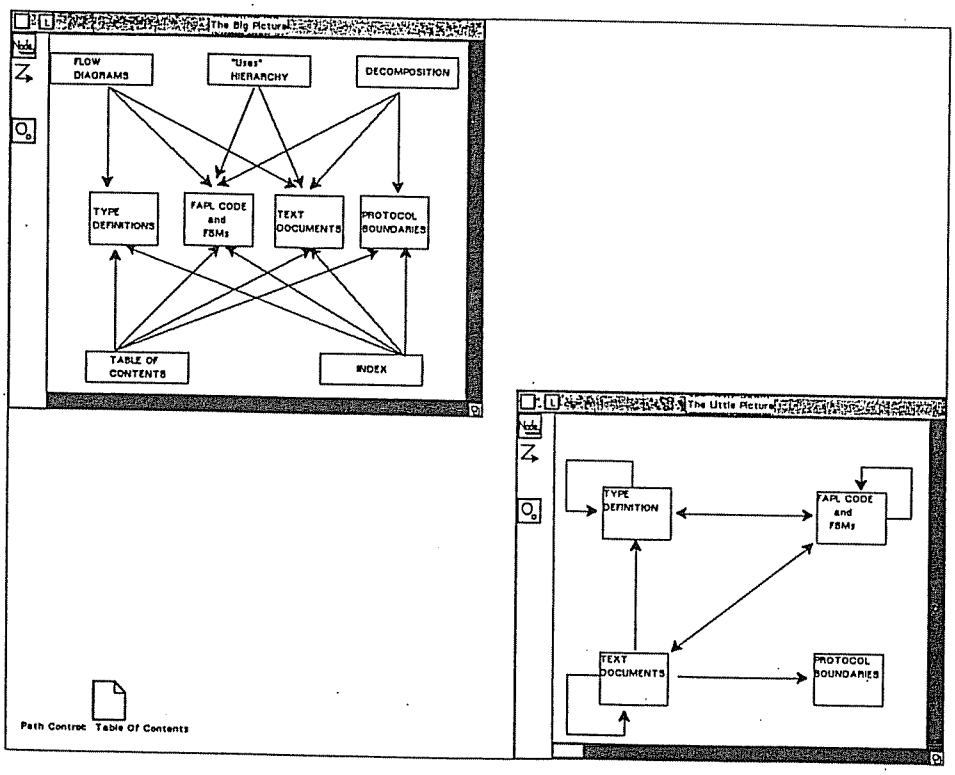

Figure 2. Central Artifact: SNA Materials.

materials: a table of contents and an index. At the top are three types of diagrams; they also provide access, but they provide substantive information in their own right, as well. *Decomposition diagrams*, illustrated in Figure 7, below, consist of box diagrams and arrows that show successively more detailed views of system components and the messages that flow among them. *Flow diagrams*, shown in Figure 8, provide a different view of the sequence, or "flow," of messages exchanged among functions and modules. Finally, a *uses hierarchy* shows the set of functions called by each individual function in the system. Users of the prototype system move among these materials by following links both within and between documents.

This total construct of information is an example of what we mean by the *artifact*. We emphasize that it is *one* information structure, not a collection of independent components. Parts of it can be viewed separately, but changes to content and structure in one place affect content and structure in other places. For example, when a function is added to the code, changes should also be made to the decomposition diagrams, the flow diagrams, the uses hierarchy, the prose documentation, as well as to the table of contents and the index. We are developing automated tools to monitor these relationships and to help developers maintain the integrity of the artifact. At present, these tasks must be done manually; but the hypertext data model simplifies the task conceptually, and a hypermedia support system can reduce the effort

SYSTEM OVERVIEW

We call our hypermedia-based system the Artifact-Based Collaboration system (ABC)* . It has two goals: first, to provide a distributed environment in which a group of individuals can work both synchronously and asynchronously over an extended period of time to develop a large artifact such as the one described above; and, second, to provide a testbed for addressing basic issues in hypertext research. ABC has six key components (see Figure 3) that include a *graph server*, a set of *graph browsers*, a set of *data application* programs, a *shared window* conferencing facility, and real-time *video and audio*. The sixth component, a set of *protocol tools* for studying group behaviors and strategies, is not illustrated in the figure.

The *graph server* is the (logically central) data management system in which all of the data objects associated with a project are stored. Individual documents (or other collections of data) are represented as separate graph structures in which nodes in one subgraph may be linked to nodes in another subgraph. The working environment is distributed across multiple workstations connected to one another and to the graph server through a data communications network. One set of tools available on the workstations is a set of *graph browsers*, built by our project, that support development and use of different types of subgraphs -- e.g., trees, general directed graphs, lists, etc. A second kind of tool is a set of conventional *data applications* used to work with data stored as the contents of nodes within graph structures. We are taking an open-architecture approach so that people may use familiar text editors, drawing packages, etc., as applications.

The *shared windows* facility permits any browser or application program to be shared by two or more users in a conference. Users can, thus, set-up conferences, add and delete members, pass control from one member to another, launch a browser or application, etc.. A *video/audio system* enables users to see and talk with one-another as they work on the same drawing or document using the shared windows facility,. At present, video is supported by a closed-circuit analog network, but we are also developing a real-time

---

* ABC is a contraction of a longer acronym -- ABCDE -- that refers to Artifact-Based Collaboration in a Distributed Environment; we use the longer form when emphasizing the system's distributed or synchronous collaboration features.

digital video system, currently based on DVI technology, that will allow voice and video to be sent over the data network and integrated directly into workstations.

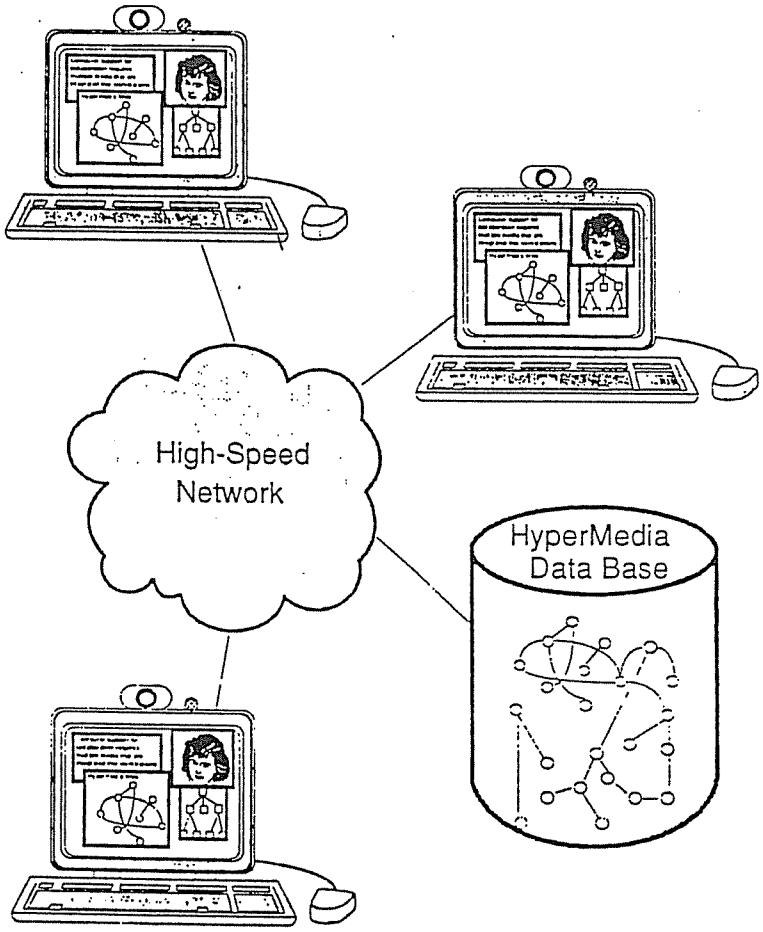

Figure 3. Overview of Artifact-Based Collaboration (ABC) System.

A sixth component, included in the system but not illustrated in the figure, is a set of *protocol tools* for recording users' actions in machine-readable form, for analyzing them, and for displaying collaborative strategies. These data will be analyzed to identify patterns of behavior so that we can see who works with whom, on what portions of the artifact, at what time during a project, etc.

Thus, ABC is a composite of separate subsystems that, together, provide a comprehensive environment for both synchronous and asynchronous collaboration. It shares function and characteristics with other hypermedia systems, but we believe it is unique in its configuration of components. Other hypertext or hypermedia systems that have been designed for software development include NLS [Engelbart, et.al., 1973], PIE [Goldstein & Bobrow, 1984], Neptune [Delisle & Schwartz, 1986], IPSEN [Lewerentz, 1988], and the USC Systems Factory Project [Garg & Scacchi, 1988]. gIbis [Conklin & Begeman, 1987] has been used to capture the decisions and rationals for software design decisions, and Trellis [Furuta & Stotts, 1990] provides powerful browsing semantics in its Petri net model applicable to both hardware and software designs. Hypermedia systems can be divided into those that view the graph as fundamental with content contained within nodes and, alternatively, those that view content (e..g., text) as fundamental with the graph of relationships superimposed over it. ABC is among the first group. Other, general-purpose hypermedia systems of this type include NoteCards

[Halasz, 1987], Storyspace [Bolter, & Joyce, 1987], and an earlier system from our group -- WE [Smith, et.al., 1987].

In the discussion that follows, we will focus on the portions of the system most directly related to hypermedia: the browsers, applications, and the graph server. We will relate specific features of ABC to general hypermedia research issues, particularly those that relate to scale and to large, multiuser applications. Since the server is described in more detail in a companion paper [Shackelford, et.al, 1991], our discussion of that component will be brief.

GRAPH SERVER

The graph server is the central data management system in which the artifact is built and maintained. When we began work on the server, we had hoped to buy a storage platform on which to implement our data model, but we could not find a system that met our needs. We still hope to find a suitable commercial platform in the future; in the meantime we have proceeded on our own.

Both the data model and the architecture of a hypermedia data management system are active topics of research, and the problems they raise are deep and complex. Two key issues are scalability and an open architecture policy for application programs. With respect to scale, most hypermedia systems have been single-user systems running on individual microcomputers or workstations supporting structures containing a few hundred to a few thousand nodes. A multiuser system that can support industrial software development will require at least two orders of magnitude greater capacity, for small projects, and considerably more for defense, aerospace, and other large systems efforts. In addressing this issue, we have made two key assumptions: to be scalable, the architecture must permit distribution across multiple hardware platforms; to be distributable, the data model must be partitionable into objects that tend to be accessed separately and that have relatively few links and/or dependencies with other objects. With respect to an open architecture, most hypermedia systems have included integral data storage components. To support an extensible set of application programs as well as multiple users, the graph server must function as a utility on the network that can be accessed by clients through a well-defined data model and set of communications protocols. We are addressing both issues in our work on the server.

The server supports a multi-level data model. Primitive objects include *nodes* and *links*, with *attributes* on both nodes and links. Links as well as nodes may have associated *contents*, which may be "raw" data that are meaningful to a particular application and, in the current implementation, are stored as a file under the control of the server.

Subgraphs play a dual role in the data model. From one perspective, they form a second layer since they consist of sets of nodes and links. However, they are handled by the server as primary objects, comparable to nodes and links The most important function of this dual property is with respect to *contents*. Subgraphs, as well as files, may serve as the contents for a node. The implications of this are subtle, but far-reaching. Most important, this design provides a natural way to partition the overall graph structure. For example, the end user may construct a tree down to three or four levels, perhaps representing the overall organization structure of a project or a document. An object of this scale can be displayed on the workstation and understood by the user. The user can then construct a separate tree that is the contents of a leaf node, perhaps representing further elaboration of the project or document. Details can be hidden or revealed, and the two graph objects can be handled and stored separately by the server. Thus, this data model addresses the issue of partitioning as well as the issue of composite objects.

The third level of the data model is a typing scheme for subgraphs. The server currently supports five graph types: *general directed graphs, connected graphs, acyclic connected graphs, trees*, and *lists*. As transactions are processed, no link or node operation is

permitted that would violate the integrity of the structure types for the affected subgraphs. This aspect of the model is a necessary first step in addressing issues concerned with the integrity, consistency, and completeness of the artifact.

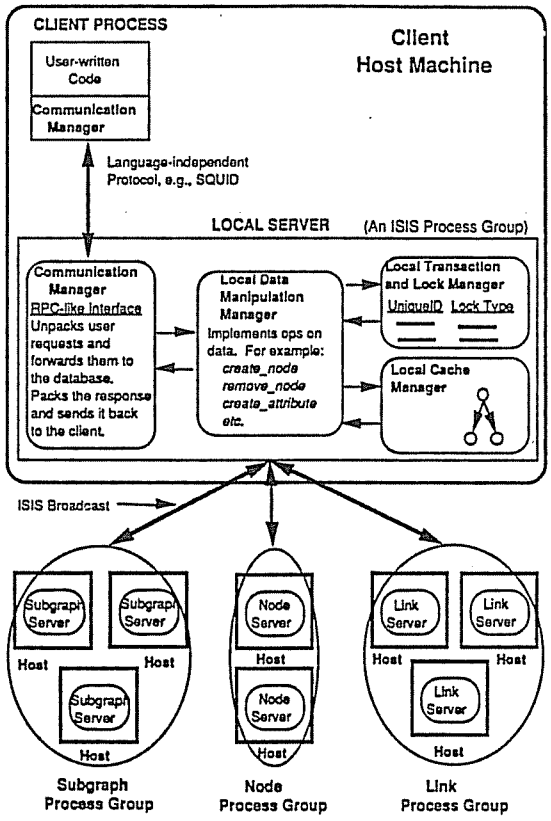

Figure 4. Architecture of Distributed Graph Server.

To maintain integrity of subgraph type while allowing flexibility in defining relationships/links, we differentiate between two types of links and two, more basic types of subgraphs. Referring, once again, to level 1 of the data model: the server differentiates between *structural links* and *hyperlinks*. Constraints on structural links determine the type of the subgraph; -- e.g., no node in a tree subgraph may have more than one incoming link. Hyperlinks, however, may join any two nodes in the same subgraph or in different subgraphs. The set of hyperlinks coming into or going out from nodes in a given subgraph determine a *hypergraph* associated with that subgraph. This hypergraph can then be stored separately but maintained and retrieved easily when the associated subgraph is accessed. Thus, the first level of the data model actually consists of *nodes*, *structural links*, *hyperlinks*, *subgraphs*, and *hypergraphs*, all with associated *contents* and *attributes*.

The architecture for the server is shown in Figure 4. We will not comment further on it except to note, first, that primitive objects in the data model -- such as nodes, links, and subgraphs -- are layered in their definitions and, second, that the architecture is designed to allow flexible, dynamic partitioning across multiple hardware platforms.

Thus, the server functions as a support system for hypermedia environments. Our work in developing it is addressing five basic issues in hypermedia research: an open, extensible architecture with respect to application programs; scale and partitioning; composite nodes and data objects; information hiding and user comprehension; integrity and related issues of consistency and completeness for the artifact as a whole.

BROWSERS AND APPLICATIONS

The user's view of the ABC system is provided by a set of *browsers* and *applications*. We differentiate between them in two respects. First, browsers are programs used to view and manipulate graph structures, while applications are generally programs, such as editors and drawing programs, used to view and manipulate "raw" data. Second, our project is developing the set of browsers, but applications programs are generally developed by others. In this section, we describe these tools from the user's perspective and discuss several research issues they raise.

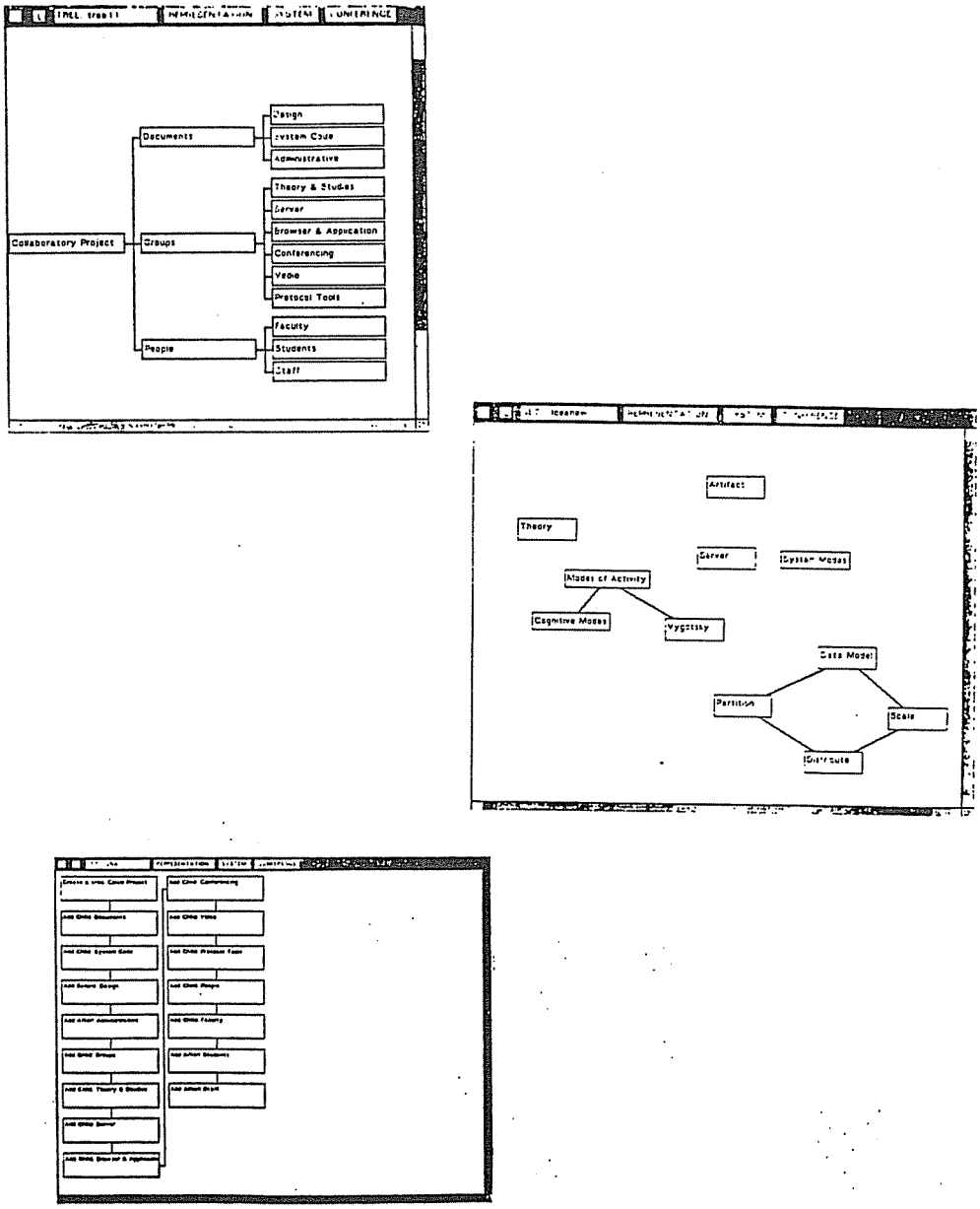

Figure 5. Browser Tree, Network, and List Modes.

Browsers

Browsers are constrained by the graph types supported by the server. To date we have developed browsers for *general graphs* (which we call *networks*), *trees*, and *lists*. These are illustrated in Figures 5. Each browser maintains the integrity of its associated data type; thus, each supports only operations consistent with that type. For example, the tree browser supports operations expressed in terms of parents, children, etc., rather than in terms of nodes and links, as is the case for the network browser. Users cannot construct structural links in a tree browser that would violate the integrity of a tree. The user can, however, construct *hyperlinks* that freely link nodes within the tree or link nodes in the tree with nodes in another subgraph. Thus, like the graph server, browsers make a fundamental distinction between *structural* links that determine the type of a subgraph and *hyperlinks* that lie within and between subgraphs but do not affect subgraph type.

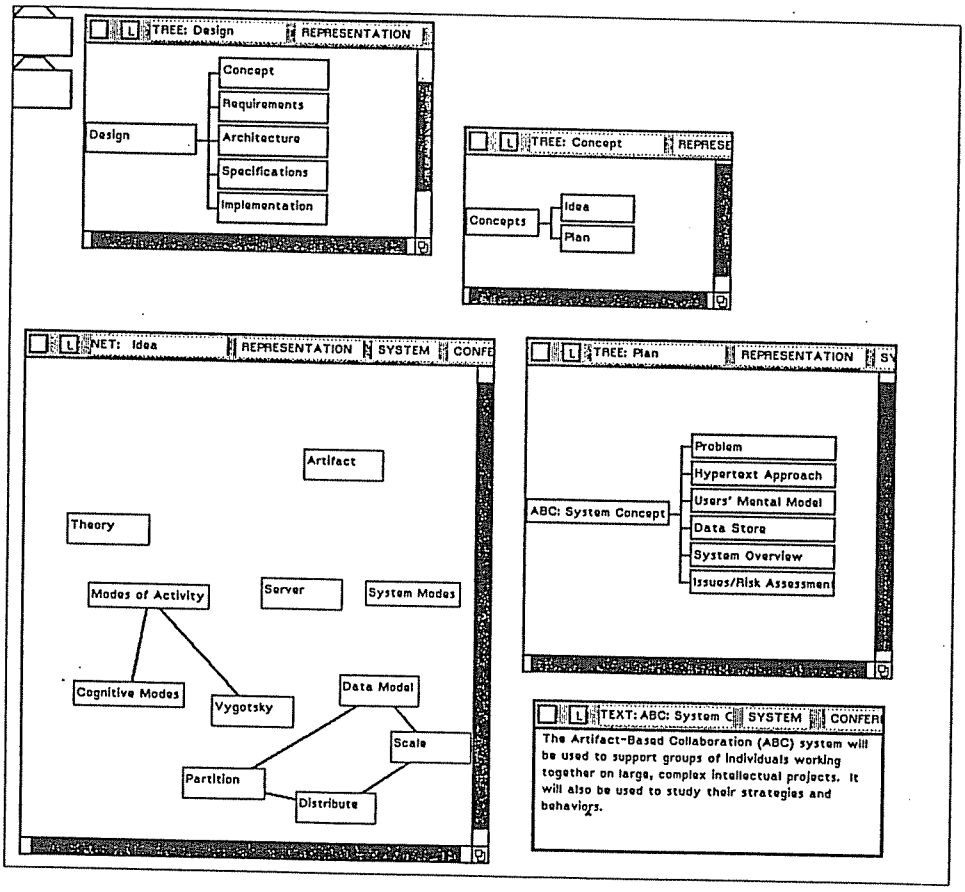

Figure 6. Combination of Browser and Application Modes.

To see how these browsers can used together, consider Figures 5 and 6. In the tree included in Figure 5, a hypothetical project organization is shown that divides the artifact into *documents*, *groups*, and *people*; one type of document is *design*. In figure 6, *design* is further specified -- as a separate subgraph that is the contents of the node shown in Figure 5 -- to include *concept* papers, *requirements*, etc. In the upper right of Figure 6, a user has begun to plan and write an actual concept document, by creating a third subgraph. This user has indicated that at this stage of the writing process, he/she is treating *ideas* separately from *plan*. For brainstorming and exploratory thinking, the user has opened a network browser (lower left) to produce the subgraph that is the contents of *ideas*; for organizing, a tree browser is opened next to it. Below that tree browser is a text editor application where the user has begun writing a section of text that is the

content for one of the nodes in tree for the document. Thus, in this example, four subgraphs are included as contents for a succession of nodes; each subgraph represents one or two levels of further detail for a concept represented by a node in a higher-level subgraph.

We have not yet built browsers for *connected* and *acyclic* graphs. When we do, they will be modified versions of the network browser since they differ not in terms of user operations but in terms of context sensitive constraints on operations relative to the type of graph object being constructed.

We are currently developing a specialized combination of tree and network browsers for building *decomposition diagrams*. Decomposition diagrams are box and line drawings that show successively finer levels of detail for a system architecture in a series of drawings in which a simple object in one diagram can be expanded in another to show its internal structure. Figure 7 shows such a sequence.

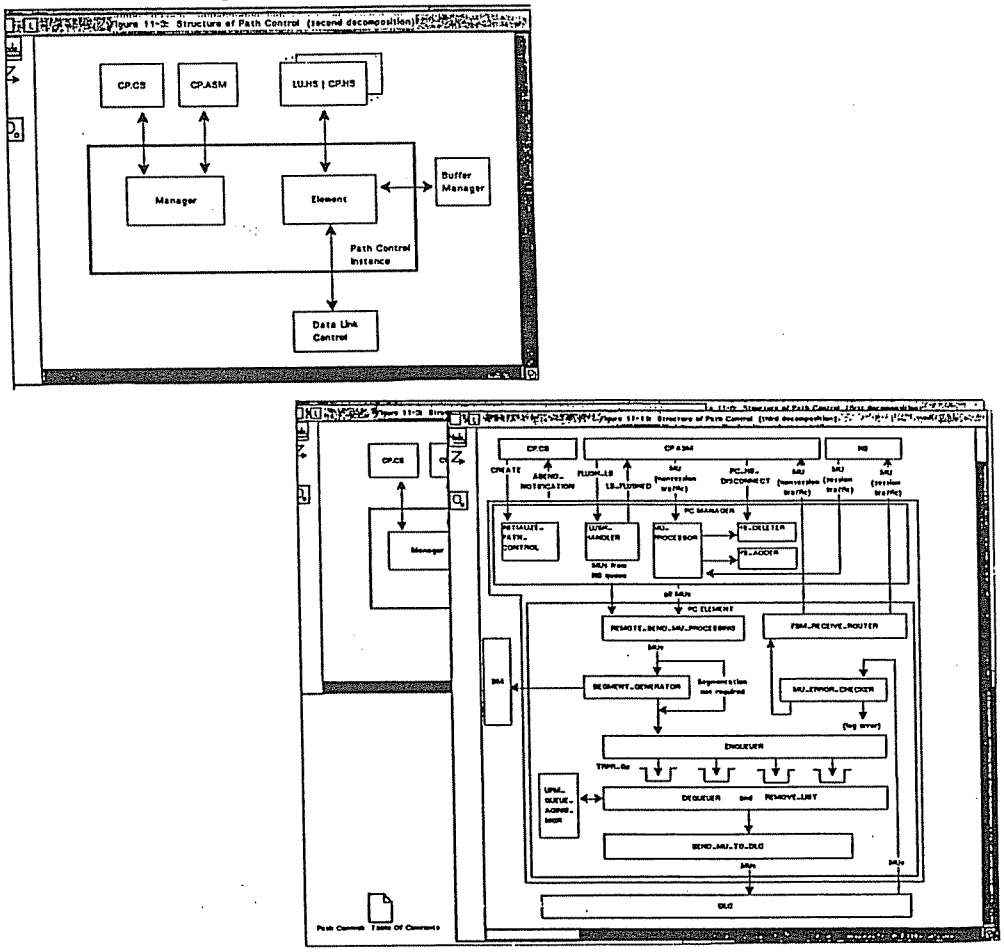

Figure 7. Browser Decomposition Diagram Modes.

What is unusual about our approach is that we are treating these diagrams as graph structures -- consisting of nodes and links -- rather than as drawings -- consisting of boxes and lines. The distinction carries several implications. Since each "diagram" is a well-defined graph structure, a syntax and semantics can be defined for them, either in terms of or in the spirit of graph grammars [Nagl, 1986]. Thus, we have the basic tools with which to consider whether or not the architecture of a system, as represented by its decomposition structure, is well-formed with respect to a specified set of rules. Second, algorithms can be developed for comparing decomposition structures to structures inherent in the system code and/or the hypertext graph structure of the artifact. This could lead to

algorithms for monitoring and maintaining consistency between the *de facto* architecture of a system as it exists in the code and its structure as described in design documents and diagrams. This would be difficult, if not impossible, were the architecture represented as conventional line drawings. Finally, for existing systems, we should be able to infer decomposition relationships from internal cues within the code -- such as procedure calls, inclusion relationships, etc. -- and generate decomposition graphs automatically. This capability would be useful for continued long-term maintenance of existing systems.

A fifth browser, which is now being designed, will support message *flow diagrams*, as shown in Figure 8. These diagrams, frequently used by communications architects, show the sequence, or "flow," of messages between system modules. Flow diagrams will also

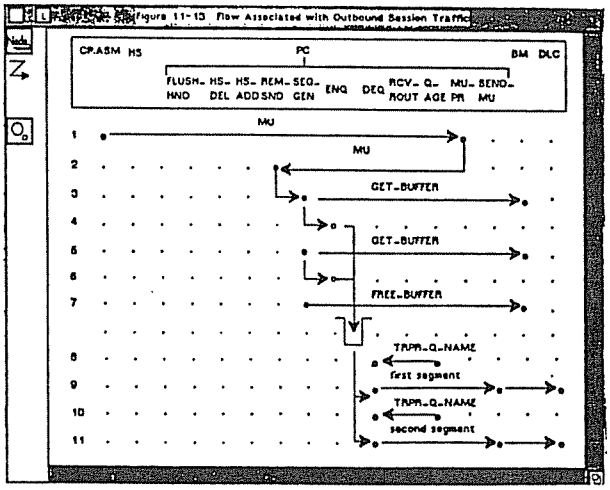

Figure 8. Browser Message Flow Diagram Mode.

be treated as a form of decomposition diagram; thus, a module that appears in one diagram as a node that sends and receives messages can be expanded in another diagram to show the detailed flow of messages internal to that module. Again, this editor will be developed as a specialized form of network browser, in which nodes correspond with functions or modules and links indicate message paths among them. As with decomposition diagrams, interesting possibilities are raised by treating flow diagrams as graph objects rather than as drawings: for animated displays, consistency checks between documented and implemented forms, inferring diagrams from existing code, etc.

As the last two browsers illustrate, basic graph browsers can be reused to develop specialized tools. At present, we are reusing browser components in an *ad hoc* way; however, we will explore, as a research issue, developing a set of enriched browsers specifically designed for reuse. When general functions for comparing and manipulating attribute values and for specifying and maintaining the appearance of the graph are included, these browsers could become a form of high-level, composable programming "language." We can envision using them to develop applications that have very different purposes and appearances at the user interface level but are actually composed or modified graph browsers underneath.

Applications

Applications are programs such as text editors, drawing programs, and spreadsheets that work with the *contents* of nodes in the form of "raw data" as opposed to subgraphs. These programs are generally not written by our project. Thus, the goal for ABC is to provide an open architectural framework in which users may import familiar tools to

work with blocks of data stored within an encompassing graph structure, which they view and access through one of our browsers.

Since we are a research project in an academic setting, our systems are necessarily more constrained in their implementations than systems built by commercial developers. Consequently, the set of applications that can be included within our "open" architecture is currently limited to SmallTalk and Unix applications that function within the X-Windows environment and can be initiated with a command that includes a file name as a parameter. Nevertheless, this is a sufficiently large group that it should not inconvenience users of our system and should provide a test of our approach to issues for open hypermedia architectures.

So long as an application meets the criteria stated above, it can potentially be used on the data contents of nodes without modification. However, if the user wishes to define links to and from positions within the data -- such as a particular word within a paragraph of text -- then the application must be modified. The problem is constructing and maintaining navigational links that are *anchored* in data representations maintained by arbitrary applications. It is an unpleasant, but widely recognized, problem since it means that to support anchored links, developers of hypermedia systems, at least for the foreseeable future, must modify the source code of existing application programs if they wish to make them available to users. The long-term solution is to establish standards and encourage application and system developers to include within their programs a standard interface and set of support functions for anchors.

Our approach to the problem is to develop a general framework, or "wrapper," for applications within X-Windows. With respect to users, the wrapper provides a consistent mechanism by which users may select a segment of data and then link that segment to some other node or to some other selected data segment within the same node or a different node. With respect to the application, the wrapper expects the application to provide a mechanism by which data selected by the user can be marked and its position noted within the application's data model. With respect to ABC, the wrapper communicates with browsers and the server, as needed, to establish and complete links. The application is responsible for maintaining a table of selected positions so long as it is running; when the application is closed, the wrapper assumes responsibility for storing this table, along with associated link identifiers, in the server. When the application is initiated once again on the contents of that same node, the wrapper restores the table of positions stored in the server.

This approach accomplishes two things. First, it supports a general anchoring scheme that requires minimal customizing of the application source code. Second, it views anchor extensions of links as a property of nodes, rather than as a property of the links, themselves. This approach recognizes that anchors are meaningful only within the semantics of a particular application run on the contents of a node. Consequently, at a given level of abstraction, anchors can be ignored with respect to the graph object; for example, in the graph modes shown above, anchor extensions into the contents of nodes are not shown since they become meaningful only when the contents of a node are displayed. We believe this to be a more consistent model than one in which anchors are viewed as a basic property of links. It is also more efficient, since not every access of a link that has an anchor extension will require anchor information,. Thus, our approach is "in the spirit" of the Dexter model [Halasz & Schwartz, 1990], but it does not follow that model precisely.

CONCLUSION

In discussing the Artifact-Based Collaboration system, we have tried to show the relations among user function, system architecture, and hypermedia research issues. With respect to the server, we are addressing five basic issues: an open, extensible architecture for application programs; scale and partitioning; composite nodes and data objects;

information hiding and user comprehension; and the integrity, consistency, and completeness of the artifact as a whole. Our work with graph browsers is addressing issues concerned with the integrity of typed subgraphs, comprehension of large structures, composite objects, and reuse of high-level components. Our work with applications is concerned with an extensible hypermedia framework that can accommodate an open set of user programs and with general support for anchored links.

ACKNOWLEDGMENTS

A number of organizations and individuals have contributed to this research. Major support has come from NSF (Grant # IRI-9015443) and the IBM Corporation (SUR Agreement # 866), with additional support from ONR (Contract # N00014-86-K-00680). We are grateful to all of our faculty and student colleagues on this project who have contributed to a common body of ideas in which we all work. Especially important have been the contributions from members of the two teams developing the server and the browsers and applications: Doug Shackelford, Mike Wagner, Zhenxin Wang, Joan Boone, Jie-Shan Lin, Ta-Ming Chen, Murray Anderegg, John Menges, and J.K. Lin. Rick Snodgrass, Gordon Ferguson, and Barry Elledge made substantial contributions to earlier versions of this work. Since the graph browsers are being built using Yen-Ping Shan's Mode Development Environment (MoDE), we wish to acknowledge his contribution, as well.

REFERENCES

Bolter, J. D. & Joyce, M. (1987). Hypertext and creative writing. *Proceedings of Hypertext '87*, pp. 41-50.

Brooks, F. P., Jr. (1987). No silver bullet: Essence and accidents of software engineering. IEEE Computer, 20(4), 10-19.

Conklin, J.; & Begeman, M. L. (1987). gIBIS: A hypertext tool for team design deliberation. *Proceedings of Hypertext '87*, pp. 247-252.

Delisle N. and Schwartz, M. (1986). *Neptune: A Hypertext System for CAD Applications*. Beaverton, Oregon: Tektronix Computer Research Laboratory, Technical Report CR-85-50.

Engelbart, D.C., Watson, R.W. and Norton, J.C. (1973). The Augmented Knowledge Workshop, *AFIPS Conference Proceedings*, National Computer Conference and Exposition (June 4-8, New York, New York), pp. 9-21.

Furuta R. and Stotts, P.D. (1990). The Trellis hypertext reference model, *Proceedings of the Hypertext Standardization Workshop* (Gaithersburg, Maryland), 83-93.

Garg, P. & Scacchi, W. (1988). A software hypertext environment for configured software descriptions. *Proceedings of the International Workshop for Software Versioning and Configuration Control*, Tuebner, Stuttgart, West Germany, pp. 326-343.

Goldstein, I. P., & Bobrow, D. G. (1984). A layered approach to software design. In Barstow, D; Shrobe, H.; & Sandewall, E. (Eds.), *Interactive Programming Environments* New York: McGraw-Hill.

Halasz, F. (1987). Reflections on NoteCards: Seven issues for the next generation of hypermedia systems. *Proceedings of Hypertext '87*, pp. 345-365.

Halasz, F. and Schwartz, M. (1990). The Dexter hypertext reference model, *Proceedings of the Hypertext Standardization Workshop* (Gaithersburg, Maryland), 1-39.

Lewrentz, C. (1988). Extended programming in the large in a software development environment. *SIGSOFT Software Engineering Notes*, 13(5), 173-82.

Nagl, M. (1986). Set theoretic approaches to graph grammars. *Proceedings of 3rd International Workshop on Graph-Grammars and Their Application to Computer Science*. Springer-Verlag, pp. 41-54.

Shackelford, D. E.; Smith, J. B.; Boone, J.; & Elledge, B. (1991). *The UNC Graph Server: A Hypermedia Data Management System*. Chapel Hill, NC: UNC Department of Computer Science Technical Report # 91-019.

Smith, J.B.; Weiss, S.F.; & Ferguson, G.J. (1987). A hypertext writing environment and its cognitive basis. *Proceedings of Hypertext '87*, pp. 195-214