ABSTRACT

INTEGRATING HYPERTEXT WITH INFORMATION SYSTEMS THROUGH DYNAMIC MAPPING

by
Jiangling Wan

This dissertation presents a general hypertext model (GHMI) supporting integration of hypertext and information systems through dynamic mapping. Information systems integrated based on this model benefit from hypertext functionalities (such as linking, backtracking, history, guided tours, annotations, etc.) while preserving their own computation capabilities. Although systems supporting integration of hypertext and interface-oriented information systems do exist in hypertext literature, there is no existing model or system effectively supporting integration of hypertext and computation-oriented information systems. GHMI makes its major contributions by both extending and specifying the well-known Dexter Hypertext Reference Model. GHMI extends the Dexter model to overcome its limitations. GHMI also maps its capabilities to the extended Dexter model with appropriate specifications to meet the requirements of our dynamic mapping environment. The extended Dexter functions apply bridge laws in the hypertext knowledge base to map information system objects and relationships to hypertext constructs at run-time. We have implemented GHMI as a prototype to prove its feasibility.

INTEGRATING HYPERTEXT WITH INFORMATION SYSTEMS THROUGH DYNAMIC MAPPING

by
Jiangling Wan

A Dissertation
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Department of Computer and Information Science

May 1996

APPROVAL PAGE

INTEGRATING HYPERTEXT WITH INFORMATION SYSTEMS THROUGH DYNAMIC MAPPING

Jiangling Wan

Dr. Michael P. Bieber, Dissertation Advisor — Date
Assistant Professor, CIS Department, NJIT

Dr. Peter A. Ng, Committee Member — Date
Professor and Chairman of CIS Department, NJIT

Dr. James A.M. McHugh, Committee Member — Date
Professor and Associate Chairman of CIS Department, NJIT

Dr. Tomás Isakowitz, Committee Member — Date
Assistant Professor, Information Science Department, New York University

Dr. Fabio Vitali, Committee Member — Date
Visiting Researcher, Department of Mathematics, University of Bologna

Dr. Hua Hua, Committee Member — Date
Assistant Professor, CIS Department, NJIT

Dr. B.A. Suresh, Committee Member — Date
Assistant Professor, CIS Department, NJIT

BIOGRAPHICAL SKETCH

Author: Jiangling Wan

Degree: Doctor of Philosophy

Date: May 1996

Education:

- Doctor of Philosophy in Computer Science,
  New Jersey Institute of Technology, New Jersey, USA, 1996
- Master of Science in Computer Engineering,
  Beijing University of Posts and Telecommunications, Beijing, P.R. China, 1987
- Bachelor of Science in Computer Science,
  Tshinghua University, Beijing, P.R. China, 1985

Major: Computer Science

Publications:

J. Wan and M. Bieber, "A Logic-based Approach to Integrating Hypertext and Information Systems," *Decision Support Systems* (submitted), 1996.

J. Wan and M. Bieber, "GHMI: A General Hypertext Data Model Supporting Integration of Hypertext and Information Systems," in *Proceedings of the Twenty-Ninth Annual Hawaii International Conference on System Sciences (HICSS), Vol. 2*, pages 47-56, Maui, Hawaii, Jan. 1996.

J. Wan and M. Bieber and J.T.L. Wang and P.A. Ng, "LHM: A Logic-based Hypertext Data Model for Integrating Hypertext and Information Systems," in *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences (HICSS), Vol. III*, pages 350-359, Maui, Hawaii, Jan. 1995.

M. Bieber and J. Wan, "Backtracking in a Multiple-window Hypertext Environment," in *Proceedings of ACM European Conference on Hypertext Technology*, pages 158-166, Edinburgh, Scotland, Sept. 1994.

J. Wan and M. Bieber and J.T.L. Wang and P.A. Ng, "Document Management Through Hypertext: A Logic Modeling Approach," in *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences (HICSS), Vol. III*, pages 558-568, Maui, Hawaii, Jan. 1994.

J. Wan and M. Bieber and J.T.L. Wang and P.A. Ng, "GHMI: A General Hypertext Data Model for Integrating Hypertext and Information Systems," in *Proceedings of Workshop on Intelligent Hypertext, in Conjunction with the ACM Conference on Information and Knowledge Management*, Gaithersburg, Maryland, Dec. 2, 1994.

J. Wan and C. Gan, "Realizing Conference Calls on Digital SPC," *Telecommunications Science*, pages 25-26, 7(1), 1991.

J. Wan and C. Gan, "Modular Software Design on Call-Handling Process Management," *Telecommunications Science*, pages 19-22, 6(3), 1990.

J. Wan and C. Gan, "A Practical Computer Communication Protocol for Real-Time Systems—HCAC," *Journal of Beijing University of Posts and Telecommunications*, pages 73-78, 13(2), 1990.

This work is dedicated to
my mother, my father,
my husband Chuanyong and my son Eric

ACKNOWLEDGMENT

I would like to express my deep appreciation to Dr. Michael Bieber, my advisor, for his fruitful resources, invaluable support and constant encouragement. I'd like to thank Dr. Peter A. Ng for help on early drafts and thank Dr. Fabio Vitali for help on refining the final work. Thanks are also given to Dr. James A.M. McHugh, Dr. Tomás Isakowitz, Dr. Hua Hua and Dr. B.A. Suresh for actively participating on my committee.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

This chapter presents an overview of this dissertation, including its motivation, approach, contributions and chapter outlines.

1.1 Motivation

The overall goal of this research is to add hypertext functionality to information systems and therefore make these systems more friendly, powerful and effective for users. Incorporating direct, context-sensitive access to pieces of information and their interrelationships within information systems improves system effectiveness. We have developed a general hypertext[1] data model GHMI (i.e., a General Hypertext data Model supporting Integration) [94, 93], which supports integration of hypertext with computation-oriented information systems (called COIS throughout this thesis). COISs dynamically generate their outputs [6, 10, 80, 44], in contrast to most hypertext systems which display predeclared contents. Therefore, in COISs, users are unable to create an information network manually.

In general, there are two distinct approaches for integrating hypertext with information systems: either including hypertext functionality as part of the initial premises of the information system design, or adding hypertext functionality to existing information systems through some dynamic mapping mechanism. We adopt the second approach, i.e., integrating hypertext and information systems dynamically. Given a GHMI hypertext system, information systems' developers (or builders) integrate their systems by defining *bridge laws*. Bridge laws are schemata

[1]We do not distinguish the terms *hypertext* and *hypermedia*. We always use *hypertext* with the belief that our discussion applies to multimedia environments with proper extensions.

which specify the domain mapping from an information system to the hypertext system. This ensures that the information system remains *hypertext-unaware* and the imposed changes are minimal—two of our major contributions. We build GHMI upon the concept of *generalized hypertext* [11, 12], developed to improve the concept of *basic hypertext* employed by many existing systems. Generalized hypertext introduces dynamic mapping, which allows the hypertext system to infer links at run time based on structural specifications of the information system application.

The majority of today's hypertext systems are still designed for standalone, isolated applications [54]. They are usually non-integrated specialized systems [58, 67, 70]. To take advantage of hypertext functionalities (navigation, annotation, and structural representation) users have to give up the familiar applications they use everyday and rebuild their information framework to fit into these specialized systems. By contrast, in an integrated system, readily-available hypertext functionalities can be added to information systems with slight cooperative adjustment [57]. Recent efforts towards open hypertext systems [56, 58, 75, 78, 91] have alleviated some of the limitations of specialized systems, especially for *display-oriented* information systems which primarily facilitate accessing and managing large document-oriented information [10]. However, none of these efforts support COISs which calculate or otherwise generate their outputs dynamically as effectively as our approach does. Examples of COISs include document management systems, expert systems, decision support systems and database management systems [8, 10]. We have developed GHMI to support integration of a variety of COISs with hypertext. Systems implemented according to GHMI function within a dynamic mapping environment. Furthermore, as described in [8, 6], bridge laws enable integration with minimal change to the COIS—often the hypertext system just has to intercept internal communications with the original COIS interface.

1.2 Approach

Although hypertext has been evolving rapidly, no widely-accepted hypertext data model facilitates our goal for COIS integration. Our efforts in this direction demonstrate the power of domain mapping. In [96], we presented a logic modeling approach, which mapped the domain of document management to the domain of hypertext. We further extended this work with composite objects and higher-level constructs [95, 97]. In addition to domain mapping, hypertext systems based on our model will also provide users with a rich set of navigation facilities. For example, we incorporate *task-based backtracking* [13] to facilitate navigation within multi-window environments.

Nevertheless as we develop our own hypertext data model, we take advantage of others' research. We chose the widely-recognized Dexter Hypertext Reference Model [47] as the basis of GHMI. The Dexter model (see §2.3) establishes a robust modeling foundation through a layered system architecture. Dexter makes significant contributions to providing a common, principled interchange standard for diverse hypertext systems. Its separating hypertext into three layers makes modeling conceptually clearer and more understandable. Hypertext researchers addressed the usefulness and robustness of Dexter in a panel at the Hypertext'89 conference and in research using Dexter as a paradigm of system interchange and hypertext modeling [38, 41, 39, 37, 40, 66, 50, 62, 35]. Building a hypertext model as a Dexter-based hypertext model would enable us to share and exchange common interests and ideas with other researchers. However, Dexter is a general abstract model grown from a variety of existing systems. For our modeling goal of integrating COISs, we found some obstacles in modeling GHMI using Dexter. Dexter has problems regarding composite components, anchors and link specifiers. We needed to extend Dexter to overcome these problems. We demonstrate the compatibility of GHMI and Dexter by modeling GHMI's capabilities using the extended Dexter with appropriate specifications, including component classes, typed links, composite structures, typed anchors, navigation structures and the storage layer functions. The extended Dexter storage layer functions (i.e., the accessor function, the component resolver function and the anchor resolver function) apply bridge laws in the hypertext knowledge base to map COISs to hypertext at run-time.

In this thesis, we summarize our research to date in developing GHMI. We illustrate by mapping relational database management systems (RDBMS) to hypertext. This integration enables the RDBMS user to take advantage of hypertext functionalities (e.g., navigation, annotation, analysis support, etc., as shown in §5.2) while preserving standard RDBMS computational facilities (e.g., query processing). We also implemented GHMI in a prototype to prove its concepts and functionality.

1.3 Contributions

GHMI aims at enhancing COISs by adding hypertext functionalities through dynamic mapping facilities. Integrating with GHMI only imposes minimal changes on COISs. We view GHMI's major contributions from the following four aspects: (1) GHMI vs. Bieber et al.'s work [12, 9]: Taking its motivation from Bieber et al.'s original concept of bridge laws, GHMI extends and formalizes bridge laws within a comprehensive hypertext data model. GHMI models composites which are not found in Bieber et al.'s work. Furthermore, GHMI formalizes the dynamic mapping concept into a hypertext data model. Also, GHMI extended and implemented the general COIS/hypertext integration architecture originally proposed by Dr. Bieber but not yet implemented, as a running prototype; (2) GHMI as a hypertext data model: As a general hypertext data model for supporting hypertext and COIS integration, GHMI uniquely provides a comprehensive set of hypertext functionalities regarding hypertext objects (composites, behavioral link typing, and dynamic anchors), domain mapping mechanisms (bridge laws) and a variety of navigation features (guided-tours, task-based backtracking, history, bookmarks, overviews); (3) GHMI as a Dexter-based model: GHMI uniquely combines specific extensions and specifications on Dexter to meet the requirements of our dynamic domain mapping environment. This demonstrates both GHMI's and Dexter's robustness and generality. Extensions are introduced on Dexter's composites, link specifiers and anchors. To map all GHMI capabilities to Dexter, GHMI specifies Dexter's components, links, anchors, the resolver function and the accessor function; (4) The GHMI prototype: The GHMI prototype is the first hypertext system which implements the general architecture of supporting dynamic integration of hypertext and multiple COISs. It proves the feasibility of the architecture and the GHMI model.

1.4 An Outline

This thesis is organized as follows. Each chapter contains a summary section at its end. Chapter 2 discusses related work of other researchers regarding open hypertext systems, combining hypertext and database technologies, the Dexter Hypertext Reference Model and Dexter-based models. We motivate GHMI by identifying the limitations of these approaches. We also review other efforts on adding hypertext functionality to DBMS—our sample domain. These efforts aim at providing traditional DBMSs with a hypertext-based design and navigation environment. Chapter 3 presents GHMI's basic concepts and functionality, including an implementation system architecture and basic elements regarding components, links, anchors, navigation structures and the bridge law definition mechanism. Chapter 4 models GHMI using Dexter and builds GHMI as a Dexter-based model to ensure greater integrity in the hypertext community. After first introducing the formalized Dexter model, we present extensions to Dexter to cover GHMI and also specifications needed to map GHMI capabilities. Chapter 5 illustrates how the GHMI hypertext could improve RDBMS and how to apply the bridge law mechanism to map RDBMS to GHMI's hypertext model. Chapter 6 discusses a detailed GHMI implementation prototype architecture and reports the current GHMI implementation status. Chapter 7 concludes this thesis by discussing GHMI integration guidelines, comparing GHMI with other hypertext models and systems, outlining potential research directions based on extensions to GHMI including connecting GHMI to WWW, and identifying GHMI contributions and limitations. To demonstrate the generality of GHMI, Appendix A gives bridge law examples for mapping another COIS domain (a document document management system called TEXPROS) which is quite different from RDBMSs. Appendix B gives sample database handler source code for generating and executing bridge laws.

CHAPTER 2

MOTIVATION AND RELATED WORK

The effectiveness of information systems can be improved greatly by introducing hypertext features (or functionalities) including information structuring (i.e., information pieces are organized as a network of nodes, links and anchors in a hypertext), navigation (i.e., mechanisms for direct access to information over a hypertext network, including link traversal, guided-tours, network overviews, bookmarks, backtracking. etc.) and annotation (i.e., comments on nodes, links and anchors to record important information).

The majority of current generation hypertext systems require that applications be imported into the underlying hypertext systems. In these systems, applications need to model their data in terms of hypertext concepts (i.e., nodes, links and anchors) and import these data manually into the management of the hypertext systems. Therefore, such hypertext systems are *closed* systems which are hardly extensible to access or link objects external to them [75]. Authoring a hypertext-based application relies on the editing facilities provided by the hypertext system. And there is no way to connect existing objects without converting and importing them, and no way to support linking between existing applications. With closed hypertext systems, users may be attracted by helpful hypertext functionalities, but would have to give up (or reimplement) their current system and often much of the functionality they provide. Aiming at overcoming these limitations faced by closed hypertext systems, a variety of approaches towards developing *open* hypertext systems have appeared in the hypertext literature and open hypertext system has become a promising research direction in the hypertext field [23, 40, 75, 76, 78]. Although to date there is no widely-accepted definition for open hypertext systems, we believe it is essential for an open hypertext system to have a robust model supporting a broad range of hypertext features and to integrate existing applications and utilize data created with these applications [81, 23, 91].

We aim at developing a robust hypertext data model to support integrating hypertext with COISs. The benefits of a formal hypertext model have been addressed already in [1, 46, 60]. An abstract formal model provides a mechanism to understand and express the common structures of different hypertext systems, i.e., to construct a globally acceptable terminology from which both hypertext researchers and users can benefit [1]. A formal model also helps to separate hypertext structure from content and gives a consistent interpretation for implementation purposes [102]. Although there is no widely-accepted data model in hypertext area, there are some well-known models which are frequently cited by hypertext researchers [2, 39, 37, 40, 63, 62, 50, 35, 66]. The Dexter model is widely referenced and accepted as a common, principled interchange standard for diverse hypertext systems. Hypertext researchers addressed the usefulness of Dexter in a panel at Hypertext'89 conference and later as part of research efforts under the paradigm of system interchange and hypertext modeling [38, 41, 39, 37, 40, 66, 50, 62, 35]. Dexter's separating hypertext into three architectural layers makes modeling conceptually clearer and more understandable. Having such a model as our base enables us to share and compare our work with other researchers based on a common framework. To integrate hypertext with COISs, we need to go beyond, extending and specifying the Dexter concepts to make GHMI a comprehensive data model supporting system integration.

In this chapter, we motivate our objective of developing GHMI through a general review of related work from other researchers. We discuss related work regarding open hypertext systems and Dexter-related modeling approaches. We propose GHMI's objective as a solution to the existing limitations of these systems and models. As we use RDBMS as an example domain for illustrating GHMI domain mapping power, we also include a review on other related work concerning hypertext and DBMS.

2.1 Open Hypertext Systems

Several open hypertext approaches are reported in recent years, including Sun's Link Service [75], Microcosm [23, 24, 27], SP3 [81, 63], Chimera [5] and Multicard [78].

2.1.1 Sun's Link Service

Sun's Link Service [75] is a commercial product shipped with Sun's programming-in-the-large software development environment, the Network Software Environment (NSE). Integration with the Link Service is a standard part of each Sun workstation application. The Link Service provides an extremely loose coupling of applications and stretches openness to its limits based on a communication protocol. Applications integrate with the Link Service through a link library which implements the protocol. It allows users to make and maintain explicit and persistent inter-application relationships. The Link Service manages links and anchors while the applications are responsible for supporting operations on linked data.

The Link Service only provides link services at a primitive programming level. Its built-in hypertext functionality is very primitive. It provides only for a distributed linking mechanism and a way for representing and storing the source and destination of a link. The application is responsible to define the link-related operations on linked objects. The Link Service's hypertext concept is simply plain node, link and anchors. There is no typing or composite and the links are static and binary.

2.1.2 Microcosm

Microcosm [23, 24, 27] is an open hypertext system developed at the University of Southampton aiming at integrating third party applications. Its system architecture manages the communication between a chain of independent filters and various node content viewers. The viewers are partially or fully Microcosm-aware applications and responsible for displaying documents or other media data. Viewers should be able to communicate with Microcosm by generating messages on user actions. Messages are passed through the filter chain and handled by proper filters. Any filter can consume, pass or generate messages to the message chain. Three special filters (i.e., the Linkbase filter, the Linker and the Computed Linker) accomplish the basic linking services. At the end of the filter chain, the link dispatcher presents the user any actions contained in the resulting messages. Microcosm integrates hypertext-unaware viewers by using a shared clipboard.

As opposed to GHMI, which aims to support computation-oriented applications, Microcosm is primarily open to viewers which are display-oriented applications (IOSs in GHMI's terminology). Microcosm's system architecture does not support integrating computation-oriented applications which dynamically generate data at run-time (i.e., COISs). The Microscosm applications have to be changed to embed some macro to handle communication with the link service. A Microcosm's linear filter chain is too restrictive and inefficient. Filters have to be ordered carefully to ensure they receive all messages they expect to handle. All messages are routed through all filters regardless of their relatedness to individual filters. This heavily reduces the system performance when the message traffic is heavy and the filters are still busy on delivering unrelated messages. Such a chain structure is also problematic when two filters need to exchange messages before any actual action-invoking message is produced for the link dispatcher. A more advanced filter management structure has been proposed by the Microcosm authors to overcome these limitations [52] toward supporting distribution. In the advanced Microcosm, filters are grouped into smaller chains. Filters are asked to register message types regarding actions in a table. However, the small chains still face the limitations mentioned above. Microcosm supports no composites and its links are static, untyped and binary.

2.1.3 SP3

Since 1991, researchers in The Hypermedia Research Laboratory at Texas A&M University have been developing a series of hypermedia prototypes SP0-3 [58, 63, 81] along with a series of hyperbases (i.e., hypertext database management systems with database management facilities such as query processing, access control, concurrency control, etc.) HB0-3 to support data storage facilities. The latest SP3 defines a flexible model which extends the concepts of Dexter and supports the distribution of hypermedia across wide-area networks. SP3 employs a process-based system architecture. Links and anchors are modeled as independent processes which implement the characteristic hypermedia behaviors such as link traversal. This approach enables a wide range of flexible run-time semantics (i.e., run-time behavior of links and anchors could be defined as process properties and methods). Users can integrate with services handled by link and anchor processes. SP3 and HB3 attempt to support a hypermedia-in-the-large environment (i.e., open hypertext systems) which can not be modeled by Dexter. In SP3, applications are responsible to manage persistent selections and maintain anchor and link markers at run-time.

SP3 has no systematic support for computation-oriented applications which handle dynamically generated data. It is the application's responsibility to extend its functionality to support dynamic data. SP3 also requires applications to store their data in the hyperbase in order to benefit from special hypertext features such as versioning. SP3 models links and anchors as first-class processes. This allows the behaviors of links and anchors to be flexibly extendible at the price of managing them in an inconsistent manner. SP3 has no way to define anchors on links, as links are not first class components (i.e., independent objects with their own properties and operations).

2.1.4 Chimera

Chimera [5] is an open hypertext system developed at University of California at Irvine, which supports integrating hypertext with heterogeneous software development environments. It models hypertext using a set of concepts including objects, viewers, views, anchors, links, attribute-value pairs and hyperwebs. A distinctive feature of Chimera is that it defines links and anchors on views of objects combined with viewers (not on objects themselves). This allows viewers to implement special anchor and link behaviors. Anchors are defined and managed by viewers. A link is defined as a set of anchors which allows n-ary links.

The Chimera system architecture follows a client-server approach to meet the needs of multiple user environments. The architecture consists of a Chimera server, a process invoker, Chimera clients and external systems. The Chimera server implements the services based on Chimera hypertext concepts and manages the connections of Chimera clients. The process invoker is responsible to invoke Chimera clients. A Chimera client includes one or more viewers. The Chimera client is responsible for definition of objects, views and anchors. It also manages the communication with the Chimera server on link traversal. One advantage of client-server approach is to allow the client to be written in different languages. The communication message details between Chimera client and Chimera server are hidden from Chimera client by Chimera API and from the Chimera client by a message ADT. The Chimera's modeling links on views enables objects to be viewed from different perspectives and provides a flexible way to include new viewer-specific anchors. GHMI shares some of these ideas and differs in many others.

Chimera was developed specifically for the needs of tools in the software development environments. Its application domain is restricted on viewers which are display-oriented applications. There is no way in Chimera to support the domain of computation-oriented applications like GHMI does. Chimera hides message details from applications by using higher-level API and ADT. This allows the Chimera developer to change the message formats freely without affecting the rest of the system. But the tradeoff of this approach is that we have to modify participating applications to use Chimera's message ADT. Chimera associates anchors with views. Such views consists of object views and the viewer which displays the object views. A chimera view could contain interface objects such as buttons and windows, depending on how the viewer defines its views. This is flexible in handling multiple views of a single object. It allows the viewers to freely implement viewer-specific features at the price of managing links and anchors inconsistently which makes it difficult to extend standard features in the Chimera server. Chimera's viewers are also heavily burdened by having to define anchors and map anchors to objects.

2.1.5 Multicard

Multicard is an open hypermedia system developed within the Esprit project Multiworks [78]. Multicard provides a hypermedia system with a set of hypermedia objects, an authoring/navigation tool, a scripting language and a multimedia composition editor. It allows integration of a single hypermedia system with a various editors and applications which run as separate processes. Applications communicate with Multicard using a message passing protocol M2000. M2000 compliant editors automatically benefit from the Multicard hypermedia functionalities including linking facilities and composite structures.

The Multicard architecture consists four distinct layers: a set of hypermedia basic objects, a persistent storage platform, an authoring/navigation tool, an M2000 protocol and a series of compliant editors. The Multicard hypermedia objects include nodes, groups, anchors, links, hyper-graphs, etc. Different editors manage node contents. Groups represent composites of nodes. A Multicard link is viewed as a communication channel between endpoints and acts as a handle or port to the destination object. Scripts can be attached to Multicard's hypermedia objects (nodes, groups and anchors) to define their dynamic behavior. Scripting provides a powerful means to extend the system functionality. Even M2000 is extensible using the scripting language.

Multicard's editors are display-oriented applications and they have to be modified to be M2000 compliant to participate the integration. Multicard's links and anchors are untyped and links are binary only.

2.1.6 Conclusion

As a summary, we see these models/systems have three limitations. (1) They were designed primarily for integration with interface-oriented systems [7] (referred to as IOS, e.g., editors and document-display systems), which support interface-level functionality. None of them supports integration with COISs effectively. Computation-oriented systems (e.g., database systems, expert systems, decision support systems) generate data dynamically (e.g., query or command processing results) at run-time and COISs are primarily used for analytic functionality, rather than navigation among displayable information networks [10]. The dynamic nature of COISs requires the hypertext system to provide an efficient way to establish dynamic links among dynamic data which can not be determined statically like those managed by interface-oriented systems. While Microcosm and Multicard perhaps could be extended to serve a COIS, neither provides a systematic support to do so. (2) Current open hypertext systems are primarily link services, which support user-declared links between independent applications. Within these systems the applications are "hypertext-aware." The applications have to maintain information regarding hypertext links and anchors. Such an approach imposes changes on the applications to facilitate the process of link traversal. (3) They do not directly provide us a robust model supporting COIS integration at a system level.

To date, there is no existing hypertext model to our knowledge supporting dynamically linked COISs. Various modeling approaches have been reported in hypertext literature, examples include approaches based on logic [33], graphs [74, 88], sets [73], Petri nets [84, 32, 85], object-orientation [61] and statecharts [102]. Although there is no widely-accepted data model in hypertext area, there are some well-known models which are frequently cited by many hypertext researchers. Nevertheless as we develop our own hypertext data model, we hope to benefit from the existing work from other hypertext researchers. We choose the frequently-cited Dexter Hypertext Reference Model [46, 47] as the basis of GHMI. The next section discusses the Dexter model and other Dexter-based models and issues.

2.2 Hypertext and Databases

During our discussion in this thesis, we take relational database management systems (RDBMS) as an example domain to illustrate GHMI's goal of domain mapping and enhancing COISs with hypertext functionalities. RDBMSs are basically characterized on the basis of information management style. Navigation in a RDBMS is based on predefined queries. RDBMSs have powerful query processing abilities. The query results are dynamically generated and are not available beforehand. Hypertext is the science of relationships and is characterized by interactive access to information and relationships. Recently, hypertext researchers have been combining hypertext with database facilities. These efforts regarding RDBMS fall into the following four directions: managing hypertext data, information retrieval, hypertext application design and enhancing RDBMS with hypertext functionality.

Managing Hypertext Data. Schnase et al. [82], Hyperform [100] and Hyperbase [79] employ database facilities (e.g., query processing, access control, concurrency control, event notification) to manage data and implement their underlying hypertext data models. This approach uses database facilities to store and retrieve node, link and anchor data. It pays no attention to adding hypertext functionality to database applications themselves, which is GHMI's focus.

Information Retrieval. Many systems and models address hypertext querying as an extension to database information retrieval facilities [29, 29, 42, 28, 64, 21, 30, 65, 53, 4]. They focus on how to search for information over hypertext networks, rather than on how to map databases information spaces to hypertext networks. As future work, GHMI could incorporate information retrieval technologies to explore useful implicit relationships (e.g., through computing object similarities) and make them direct accessible through link traversals.

Hypertext Application Design. Some models and systems combine RDBMS and hypertext concepts in application design and include hypertext functionality as part of the applications. RMM [55] proposes a seven step relationship management methodology for the design and development of hypermedia applications. Hara et al. [49] presents two types of relationship abstractions (augmentation and globalization) as a facility to improve hypertext application design. Such a "design from scratch" approach, which is common to these two models, results in a hypertext system whose structure is the same as the underlying database's schema, but which is not reusable for other applications and is not able to add hypertext functionality to existing applications.

Enhancing Databases with Hypertext Functionality. This is GHMI's direction. There is some related work in this direction. (1) In [48], Hara et al. adopt a two-step approach to improve hypertext application design and mapping. At step one, the designers use the RDBMS model to design the application objects and relationships. Then at step two, they employ an SQL-like specification language to translate these objects and relationships to hypertext. The second step is similar to GHMI's bridge law mapping mechanism. This approach enables generating nodes and links dynamically. Once written, the same mapping rules can be reused for different applications. This reduces redundancies and inconsistencies resulting from individual application design. However, [48]'s focus is also primarily on application design. Its hypertext model is very primitive. Links are primarily for built-in semantic relationships. Hypertext only helps accessing these explicit relationships. It is also not clear how to implement domain translations between database and hypertext. (2) The ESPRIT Project HIFI [16] aims at providing external databases with a hypertext-based navigation interface. The hypertext interface model HDM+ is an extension of HDM [36]. The core approach of HIFI is to define a set of application mappings between HDM+ and database primitives, including the mapping between HDM+ and ER diagrams. The HIFI approach defines a hypertext interface according to the user needs rather than the structure of the underlying databases. Both HIFI and GHMI adopt a very similar approach toward mapping RDBMS dynamically. They differ in three aspects. (a) HIFI primarily focuses on capturing explicit database elements in terms of HDM+ constructs (e.g., ER relationships, semantic relationships between entities, etc). It focuses on mapping explicit and predefined object groups (e.g., entities) and relationships (e.g., entity relationships in ER diagrams) which are the static aspects of an application. Once the database is designed using HDM+ terms, all relationships become explicit. GHMI focuses on mapping implicit schematic relationships and the generic underlying database structures (e.g., databases, tables, records). (b) GHMI more faithfully preserves the original RDBMS structures which the users are familiar with while HIFI forces the user to adopt a hypertext-specific structure. The GHMI hypertext structure is defined according to the underlying database structures rather than individual application users' needs. (c) The HIFI hypertext interfaces are specialized for individual applications. For example, the interface of a medical application [51] is not reusable for a financial bank application [17]. The GHMI architecture could provide a general interface for all applications based on modeling generic RDBMS structures.

2.3 The Dexter Hypertext Reference Model

In this section, we outline the basic Dexter framework and identify its problems. Then we review some other Dexter-related models and issues.

2.3.1 An Overview of the Dexter Model

The Dexter Hypertext Reference Model [46, 47] (called Dexter throughout this thesis) is a hypertext model developed as a result of two workshops of hypertext researchers and based upon several well-known existing hypertext systems such as NoteCards [45], Neptune [25], KMS [3], Intermedia [101, 77] and Augment [26]. In Dexter, a hypertext is divided into three separate layers, namely the run-time layer, the storage layer and the within-component layer as shown in Figure 2.1. A hypertext is considered as a network of information. The run-time layer concerns the dynamic behavior of a hypertext, regarding how to present it to the user and how the user interacts with such a presentation under some interface environment. The storage layer consists of a network of *components*, which are information containers and interconnected by relational links. The within-component layer deals with the internal contents or structure of individual components. Dexter focuses on the storage layer. The reason for not modeling the within-component layer is that the range of component contents (e.g., text, graphics, animation, images, etc.) is too broad to be captured by a single generic model. A similar argument applies to the run-time layer, due to the vast diversity of user interface tools for accessing a hypertext. Nevertheless, Dexter does provide inter-layer interfaces to allow the storage layer to communicate with the other two layers. *Anchors* are employed as the interface

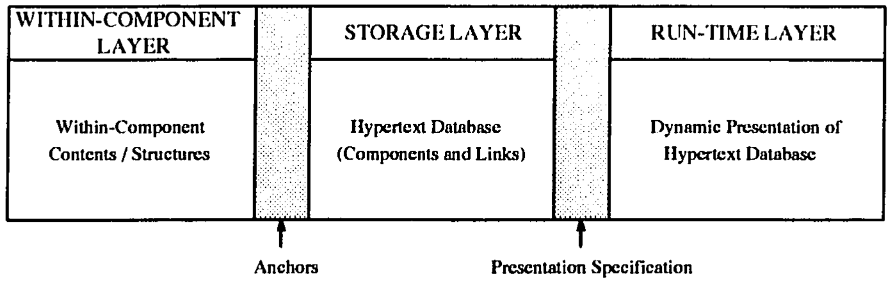

WITHIN-COMPONENT LAYER
STORAGE LAYER
RUN-TIME LAYER
Within-Component Contents / Structures
Hypertext Database (Components and Links)
Dynamic Presentation of Hypertext Database
Anchors
Presentation Specification

Figure 2.1 The Three-Layer Dexter Model

between the storage layer and the within-component layer to establish references among portions of individual components. The interface between the storage layer and the run-time layer includes a mechanism called *presentation specification* which allows the user interface presentation of a hypertext to be encoded at the storage layer.

A component can be either an atom, a link or a composite component. The atomic component is the primitive unit in Dexter. Link components represent relationships between components. A composite component is constructed from other components. The notion of composite components provides a hierarchical component structuring mechanism in Dexter and corresponds roughly to "nodes" in other hypertext systems. A Dexter component is modeled as a two-part composition: *base component* and *component information* (*COMP_INFO*). The base component is recursively defined as an atom, a link, or a sequence of other base components. The component information includes a set of attributes, a presentation specification (interpretable only in the run-time layer) and a sequence of anchors pointing to a portion of this component's contents. Every component is identified by its unique ID (UID) which is unique throughout a hypertext. The content of a link component consists of a sequence of endpoint specifications. A link endpoint is specified by an entity called *specifier* which is a combination of a component specification (*COMPONENT_SPEC*), an anchor id, a direction and a presentation specification. Span-to-span links are supported by anchors. The direction of a link endpoint could be "FROM," "TO," "BIDIRECT" and "NONE." A link must have an endpoint with direction "TO" which excludes dangling links (i.e., links missing endpoints). An anchor is a composition of an anchor *id* (*ANCHOR_ID*, identifying an anchor within a component) and an anchor *value* (*ANCHOR_VALUE*, location information within a component interpretable by the within-component layer). The Dexter storage layer also includes two fundamental functions: a *resolver* function and an *accessor* function. The resolver function is responsible for resolving the component specification in a link specifier to an explicit UID. This enables a link to point to a computed component with its specification in the link specifier. The accessor function is responsible for accessing a component given its UID.

Figure 2.2 shows an example of Dexter component, link and anchor representation. *Comp*1, *Comp*2 and *Link*1 denote component UIDs. *Comp*1 is an atomic component which defines an anchor as a portion of its content. *Comp*2 is a composite component consisting of some direct text and two atoms. *Comp*2 also defines an anchor in its text content. A third component *Link*1 represents the relationship between the two text portions which are defined as anchors in *Comp*1 and *Comp*2. *Link*1 has two endpoints represented as two specifiers. The "FROM" endpoint is anchor 1 in *Comp*1 and the "TO" endpoint is anchor 1 in *Comp*2. *Link*1 has anchor IDs instead of anchors in its specifier. The anchor value and the presentation specification (*PresentationSpec*) are denoted as black boxes indicating that their exact specifications are out of the scope of the Dexter storage layer model.

Dexter also includes a simple model for the run-time layer. In the run-time layer, the basic concept is the *instantiation* of a storage layer component. An instantiation is a mapping of a component from its storage data format to its presentation format. An entity called *session* keeps track of the dynamic mapping from components to their instantiations. A session is a run-time access environment of a

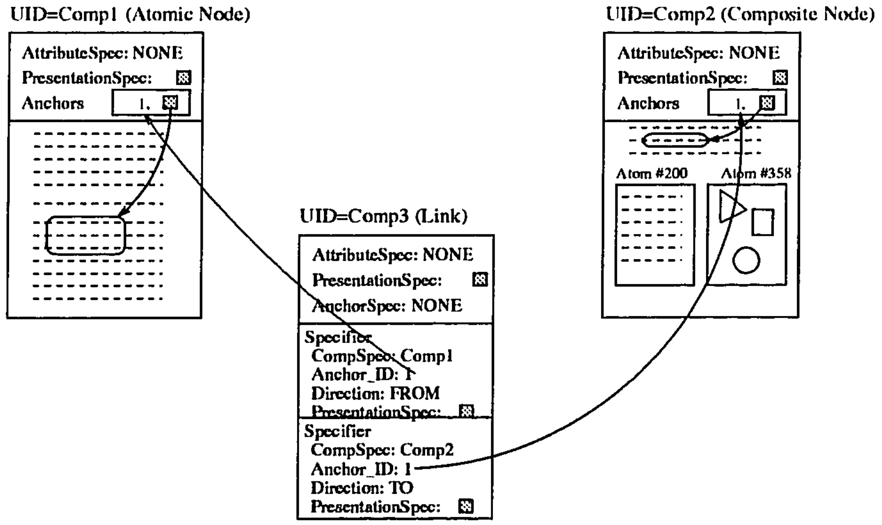


Figure 2.2 The Dexter Component Representation

hypertext. All operations within an opened session are recorded chronologically in an entity *history*.

2.3.2 Problems in Dexter

Dexter makes significant contributions to providing a common, principled interchange standard for diverse hypertext systems. Separating hypertext into three layers makes modeling conceptually clearer and more understandable. However, as a general model grown from a variety of existing systems, Dexter is sometimes too general to fit all systems. As a reference model, Dexter aims to model only the common features of different systems instead of the systems themselves in full. Therefore, Dexter is, by nature, general and incomplete. For our model focusing on integrating COISs, we find the following are problems regarding modeling GHMI in Dexter. By the term *problems* here we mean those Dexter aspects or concerns, for which we need either *extensions* or *specifications* such that they fit GHMI and our proposed hypertext system functionalities.

1. *Components.* Dexter has problems on the notion of composite components in three aspects. (1) Component Information: Dexter does not distinguish

components managed by hypertext systems and those managed by third-party applications. We can specify the component attributes to explicitly model the ownership information as well as bridge laws used to map individual COIS objects. (2) Base components: A Dexter composite component contains "bare-bone" base components which are not independent components themselves. The definition of component is recursive on *base component* rather than on *component* itself. This implies that base components in a composite component are *not* components themselves. As UIDs are associated with components only, base components have no UIDs. They can not be accessed by the *accessor* function. They can not be *external* independent components (i.e., they do not exist outside a composite component's content). On the other hand, base components have no component information. There is no way to associate attributes to base components. Base components have no anchors or presentation specifications of their own either. When we construct a composite component taking other components as base components, all other components lose their own properties (regarding attributes, anchors and presentation specification). It is also difficult to create links among base components since they are not independent components and have no UIDs. Therefore, such a notion of a composite is too restrictive. In our domain of supporting multiple COISs, we might have a composite component made up of components from different COISs (with distinct ownership properties and other COIS attributes). We also try to model the internal linking structures of composite components to facilitate navigation (e.g., create guided-tours based on the internal links of a composite). We can not effectively model these GHMI composites using Dexter. We need to extend Dexter's composite components to allow *external* components. (3) Atomic Components: Dexter does not model the content of atomic components. In our dynamic mapping environment, however, it is possible and necessary to model internal structures of atomic components to represent structured objects such as a database records. Modeling structured atomics enables defining links and anchors based on object structures (e.g., anchors on record fields or values).

2. *Links.* Dexter links have three problems. (1) Dangling links: Dexter's intolerance of these constructs has been widely criticized [39, 40, 63, 62]. In the environment of dynamic COIS mapping, a link endpoint could specify a computed component mapped from a COIS object (defined as a mapping rule). If the COIS object is deleted inside the COIS (which is transparent to the hypertext system), the execution of the mapping rule will result an empty component. This causes the link to be "dangling." If the anchor marking a link is deleted inside the COIS, the link will become dangling too. Since this situation seems often to occur, it can not be ignored by simply excluding it from a hypertext model. We need to allow these links and at the same time develop some mechanisms to handle them properly. (2) Unary links: Dexter links must at least two specifiers. However, unary links (i.e., links with only one specifier) could be useful for modeling COIS commands directly available as menu items with a specifier directed as "TO." Access to bookmarks can also be modeled as a unary link with only one "TO" specifier. (3) Typed Links: Dexter links do not explicitly support a semantic or behavioral type. It has been widely recognized that typed links reduce disorientation for users and design overhead for designers [20, 67, 73, 87]. Dexter implies that link typing is possible by attaching a "type" attribute to a component. We need to specify Dexter's component attributes to support link types explicitly. We classify links based on their behaviors. For example, links representing *ad hoc* relationships should be distinguished from those for cross-referencing, those representing the underlying hierarchical structures of COIS objects, and those resulted from

COIS-defined computation (e.g., a link with an endpoint as a query result in the domain of relational databases). We need an explicit method to identify object types based on the roles they play in the integrated hypertext system.

3. *Anchors.* Dexter's problems on anchors include two aspects: (1) External anchors: Dexter defines anchors in the content of components. It is not clear how to define anchors in base components. On the other hand, link specifiers contain an *ANCHOR_ID* which therefore must be consistent with the definition in the component embedding the anchor. A link specifier's *COMPONENT_SPEC* needs to be resolved to UIDs and therefore may lead to different UIDs in different computations. Using the actual *ANCHOR_ID* in a specifier requires an unbearable consistency burden on hypertext systems, requiring all possible components whose UIDs could be mapped from a given *COMPONENT_SPEC* to have the same anchors, or at least use the same *ANCHOR_ID* for that link. In our environment of dynamic mapping, *COMPONENT_SPEC* is frequently used in link specifiers to allow generating link endpoints dynamically. Storing *ANCHOR_ID* in link specifiers which resolve to dynamic components would impose a heavy consistency burden. It is difficult to map the specifier's *ANCHOR_ID* to the corresponding *ANCHOR_ID* in a dynamically computed component. (2) Typed Anchors: In Dexter, it is not clear how to define keyword anchors [41] and dynamic anchors. We need to extend Dexter to allow the above external anchors and classify them into three types: plain anchors, keyword anchors and dynamic anchors. Plain anchors are defined statically with explicit location information as their values. Keyword anchors represent a group of anchors with the same text value. Dynamic anchors are dynamically computed anchors. In our case of supporting COIS integration, dynamic anchors are computed at run-time along with COIS components and links. We need a mechanism to model these

anchors in the storage layer. Dynamic anchors are resolved to plain anchors or keyword anchors at run-time according to their bridge laws. We need to extend Dexter to include new resolver functions to resolve anchors from bridge laws.

4. *Domain Mapping.* Dexter was developed from closed hypertext systems. It does not model facilities for dynamic integration of hypertext and information systems. We employ a mechanism called *bridge laws* to specify domain mappings. Under our dynamic mapping environment, all components mapped from COIS objects or relationships are non-persistent *virtual* components. The hypertext system does not keep any copy of their contents. Every time they are required by the user, the system maps them by executing bridge laws. These components could be computed components if they are dynamically generated from COIS-dependent operations (e.g., database queries). Since bridge laws are invoked at run-time to generate components and links, we need to specify the semantics of Dexter's resolver and accessor functions to apply bridge laws.

2.3.3 Dexter-based Models and Systems

Over the past several years, models and systems have been developed following Dexter. Some of them applied Dexter to build their systems and made necessary extensions or specifications according to their specific needs; Others addressed Dexter-related issues regarding their experience on developing hypertext system and data models.

DHM (or DeVise hypermedia) [38, 41, 39, 37, 40] is a Dexter-based hypermedia prototype developed at Aarhus University in Denmark. DHM extends Dexter in link directionality, dangling links, external anchors, keyword anchors, external components, virtual components and computed components. [37] further extends DHM composites to include a class hierarchy and four aspects of composites contents.

Leggett and Schnase criticize Dexter's abilities on hypermedia interchange and hypermedia-in-the-large (i.e., open hypermedia systems) design [63]. They address four issues from their experience on translating Intermedia and KMS using Dexter as an exchange standard [62]. They discuss issues regarding Dexter's problems on dangling links, versioning, external components, deletion semantics for composites, composite's internal linking and navigational link semantics. In addition, Leggett et al. propose seven fundamental assumptions for hypermedia-in-the-large system design. Based on these assumptions they claim that Dexter does not support hypermedia-in-the-large and it is not profitable to further extend the Dexter model.

RHYTHM [66] is a hypertext system developed the University of Bologna in Italy. The authors believe that modeling RHYTHM using Dexter proved the usefulness, soundness and robustness of Dexter, although they made an extension on external anchors. They introduce a primitive link typing to classify links into two classes: navigation and inclusion links, but only allow binary links.

The Amsterdam Hypermedia Model (AHM) [50] is a general framework focusing on extending hypertext to hypermedia. AHM was developed as a Dexter-based model with extensions on notions of time, high-level presentation attributes and link context, and external components. Although AHM extends Dexter from a multimedia point of view, which is not the current focus of GHMI, we share common points on modeling composite contents using referencing rather than embedding other components.

Garzotto et al. [35] made extensions on Dexter's storage layer by introducing the concept of *collections* and on Dexter's run-time layer by related notions of collection-navigation and collection-synchronization. The internal structure of a collection includes two aspects: a *set* of members and a *structure* of *topologically* arranged members. Index and guided-tours are two basic collection-based navigation structures. Garzotto et al. addresses the Dexter problems on internal composite structures (only as set) and the notion of navigation structures (guided-tour and index). We can go beyond these extensions on composite structures.

2.3.4 Conclusion

Although all of the above approaches address some of the Dexter problems we identified in 2.3.2, no hypertext literature to our knowledge addresses all of the above Dexter problems satisfactorily for our needs of supporting COIS integration. None of them addresses our concern of dynamic domain mapping and the run-time layer structures supporting task-based backtracking. The other issues they addressed, such as the multimedia and collaboration related issues, are attractive but not our current focus. We will consider them in our future work on further extending GHMI.

GHMI develops its concepts and functionalities according to the requirements of supporting integration of hypertext and COISs. We map GHMI's capabilities to Dexter with appropriate extensions and specifications to overcome the above Dexter problems. The task of modeling GHMI in terms of Dexter includes two aspects: extensions on composite components, external anchors, dynamic anchors, unary links and dangling links; specifications on component attributes, atomic components, composite components, anchors, link specifiers, the resolver function and the accessor function.

2.4 Summary

This chapter motivates GHMI through a state of the art review of hypertext research on open hypertext systems, combining hypertext with database technology, the Dexter Hypertext Reference Model and Dexter-related issues.

Open Hypertext Systems. Aiming at overcoming the problems faced by closed systems, a variety of *open* hypertext systems have been reported in recent years, including Sun's Link Service [75], Microcosm [23, 24, 27], SP3 [81, 63], Chimera [5] and Multicard [78] From our perspective of supporting COIS integration, we find that these systems and their models have three limitations. (1) They were designed primarily for integration with interface-oriented systems [7]; and therefore, (2) current open hypertext systems are primarily link services, which support user-declared links between independent applications. Within these systems the applications are "hypertext-aware;" and therefore, (3) they do not directly provide a robust model to model a comprehensive set of hypertext functionalities for the hypertext system we intend to develop.

Hypertext and Databases. Hypertext is the science of relationships and is characterized on the basis of the interactive access to information and relationships. RDBMSs have powerful query processing abilities. The query results are dynamically generated and are not available beforehand. Recently, hypertext researchers have been combining hypertext with database facilities. These efforts regarding RDBMS fall into the following four directions: managing hypertext data, information retrieval, hypertext application design and enhancing RDBMS with hypertext functionality. Most other hypertext research focus on the first three. GHMI focuses on enhancing existing RDBMS with hypertext functionality.

Why Dexter. We aim to develop a hypertext system with a robust data model to support integrating hypertext with COISs. The Dexter model is widely referenced and accepted as a common, principled interchange standard for diverse hypertext systems. Hypertext researchers addressed the usefulness of Dexter in a panel at Hypertext'89 conference and later in research concerning the paradigm of system interchange and hypertext modeling [2, 39, 37, 40, 63, 62, 50, 35, 66]. Dexter's separating hypertext into three architectural layers makes modeling conceptually clearer and more understandable. Having such a model as our base enables us to share and compare our work with other researchers based on a common framework. However, to meet the requirements of dynamic COIS integration, we need go beyond and extend the Dexter concepts to develop a comprehensive data model supporting system integration facilities.

Dexter and Its Problems. The Dexter Hypertext Reference Model [46, 47] divides a hypertext into three separate layers. The run-time layer concerns the dynamic behavior of a hypertext. The storage layer consists of a network of *components* which are information containers and interconnected by relational links. The within-component layer deals with the internal contents or structure of individual components. The focus of Dexter is on the storage layer. Dexter employs *anchors* as the interface between the storage layer and the within-component layer. The interface between the storage layer and the run-time layer is a mechanism called *presentation specification.* For our modeling focused on integrating COISs, we found that Dexter has the following limitations. (1) Components: no model for component structures; subcomponents in a composite component have no component information; no component ownership information; (2) Links: no dangling links; no link typing; no unary links; (3) Anchors: no external anchors; no keyword anchors or dynamic anchors; (4) Domain mapping: not modeled; We need to both extend and specify all of these limitations in our goal of supporting COIS integration with a powerful hypertext data model.

Dexter-based Systems and Issues. Over the past several years, models and systems have been developed following Dexter, including DHM [38, 41, 39, 37, 40] Leggett and Schnase [63], RHYTHM [66], AHM [50] and Garzotto et al. [35]. They made extensions on Dexter concepts and/or specified Dexter to map their models and systems. However, no hypertext literature to our knowledge addresses all of the above Dexter limitations satisfactorily for our needs of supporting hypertext/COIS integration.

Goal of GHMI. We adopt two steps to develop GHMI as a Dexter-based model: First, we develop GHMI concepts and functionalities according to the requirements of supporting integration of hypertext and COISs. Then, we map GHMI's capabilities to Dexter with appropriate extensions and specifications to overcome the above Dexter limitations. Therefore, the task of modeling GHMI in terms of Dexter includes two aspects: *extensions* on composite components, external anchors, dynamic anchors, unary links and dangling links; and *specifications* on component attributes, atomic components, composite components, anchors, link specifiers, the resolver function and the accessor function.

CHAPTER 3

GHMI: BASIC CONCEPTS

Figure 3.1 shows the layout of our proposed hypertext data model GHMI (a General Hypertext data Model supporting Integration). After a brief discussion on the system architecture, this chapter focuses on the GHMI concepts including an object class hierarchy, components, links, anchors, dynamic mapping and the bridge law template.

3.1 A System Architecture

The purpose of this section is to demonstrate how the COIS mapping approach works from an implementation viewpoint. This will support understand our discussions regarding domain mapping in GHMI. Figure 3.1 presents a general system architecture supporting implementation of GHMI. This architecture consists of three basic layers: the computation-oriented information systems (COISs), a hypertext engine (HTE [8]) and the interface-oriented systems (IOSs). An information system typically comprises two functional components: an IOS front end and a COIS back end. By assuming that information systems are designed following a modular fashion such that their IOSs can be replaced by other IOSs, we can augment an information system with hypertext functionality by incorporating a hypertext engine between the IOS and the COIS. This means the HTE intercepts any messages the COIS would send to its interface and generates all appropriate responses. Each COIS or IOS is connected to the HTE by its own *handler*. A COIS handler is an extended portion of the COIS and is responsible for translating the messages coming out of the COIS into the COIS-HTE communication format which the HTE can handle and vice versa. Another job of the COIS handlers is to "buffer" the HTE from the

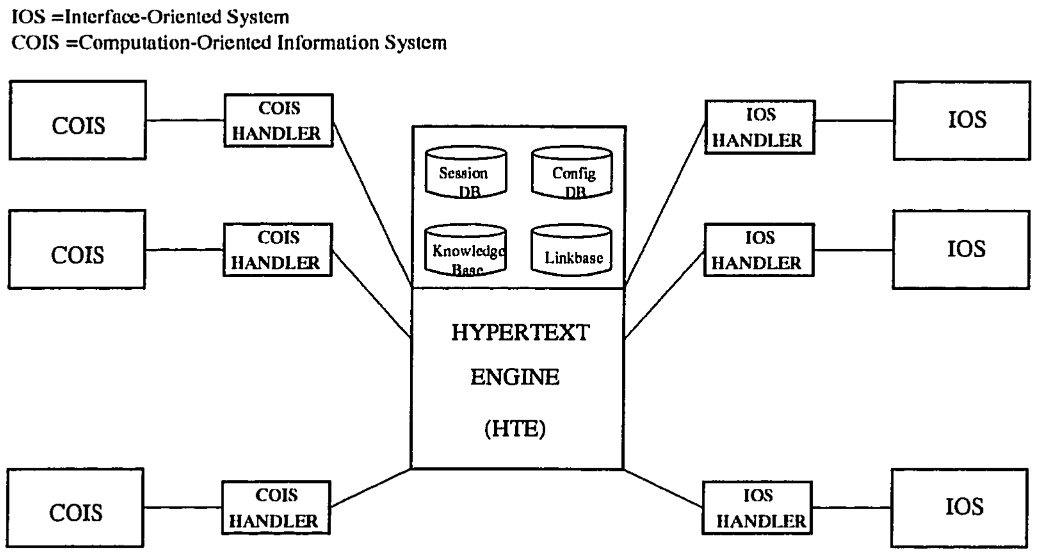


Figure 3.1 A GHMI System Architecture

COIS: if the HTE expects the COIS to perform a function it can not, the COIS handler must implement this function to ensure seamless integration. Similarly, the IOS handlers handle IOS-HTE communications and buffering. Our purpose is to design a system architecture which is general enough to apply to a variety of COISs and IOSs, which means that every COIS can be arbitrarily combined with an IOS that handles its media types. Currently our major contribution focuses on the COIS-HTE side. The HTE-IOS mapping would be another interesting research area. We are developing GHMI as a general hypertext data model supporting integration of a variety of COISs.

The HTE has a knowledge base made up of COIS-dependent mapping rules, i.e., bridge laws, which map individual COISs to hypertext. Each COIS has its own set of bridge laws. These rules are registered by the COIS builders during the progress of system set-up. To integrate a COIS with a hypertext system based on our model, the COIS builders need to write the bridge laws stored in the HTE's knowledge base and write the code for their individual COIS handler.

Besides a knowledge base, the HTE maintains three databases: a Linkbase, a Session DB and a Configuration DB. The Linkbase stores persistent hypertext data, which are not mapped from COISs (e.g., manually created static links, annotations and bookmarks.). The Session DB stores dynamic data with respect to a navigation session (e.g., history information within a session) for constructing dynamic navigation structures such as history list and backtracking. The Configuration DB maintains configuration information for COISs and IOSs.

The HTE relies on individual COIS handlers as preprocessors to facilitate COIS-HTE cooperation and is responsible for accomplishing the hypertext functionalities defined in GHMI. It should manage dynamic information exchange and identify mapped hypertext objects from COIS specifications. The HTE uses predefined bridge laws to map COIS objects to hypertext objects. When the HTE catches some user action which happens on the IOS, say, a link anchor being selected, the HTE consults its knowledge base seeking appropriate semantics of the action and identifies destination COIS objects needed or which COIS execution procedure to invoke. Communication with the COIS is then activated through the COIS handler, which executes its routines accordingly (e.g., executing the bridge laws, and/or consulting the underlying COIS database), often returning a report to display in response to the user action.

GHMI aims at providing a robust data model for representing the functionalities of the HTE toward integrating COISs with hypertext. The following sections present the basic elements of the GHMI model. In Chapter 6, we present a GHMI prototype for implementing the GHMI system architecture.

3.2 Object Classes

GHMI models objects as links and components. We employ an object-oriented approach to illustrate the GHMI object class hierarchy, as shown in Figure 3.2.

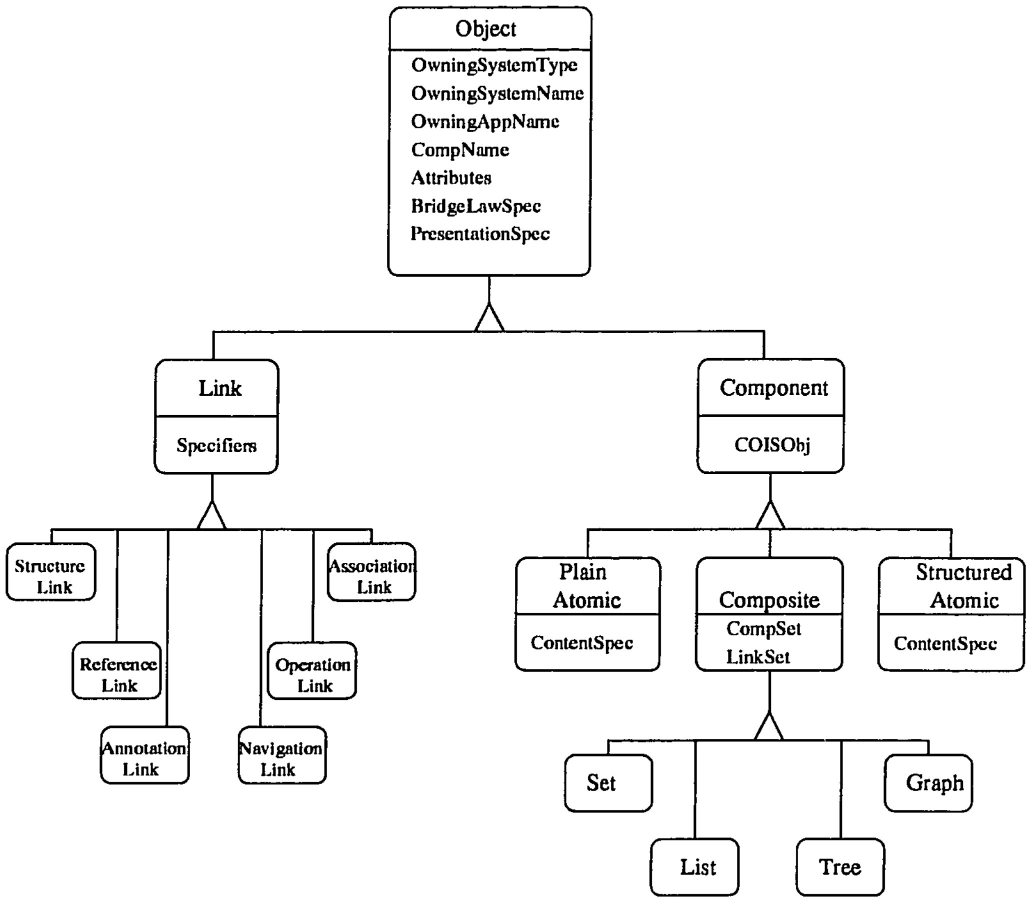


Figure 3.2 GHMI Object Class Hierarchy

Symbol $\triangle$ means a *generalization* relationship between two object classes (the upper-position class is the generalization of the lower-position class in the figure). Generalization indicates property inheritance between classes. That is, if class A is the generalization of B, then we can construct B based on A. B will inherit all the properties A has and will also have its own additional properties. These properties include attributes and methods (or operations) applicable to the individual classes.

GHMI classifies *links* into six categories (see §3.4 for details). *Components* fall into three subclasses: *Plain Atomic*, *Structured Atomic*, and *Composite*. GHMI distinguishes composites based on their internal structures: *Set*, *List*, *Tree* and *Graph* (see §3.5).

GHMI's object class classification is based on the properties and operations available on the objects of individual classes. As shown in Figure 3.2, all objects have seven common properties: *OwningSystemType*, *OwningSystemName*, *OwningAppName*, *CompName*, *Attributes*, *BridgeLawSpec* and *PresentationSpec*. The *OwningSystemType* of a component could be either "Hypertext" or a COIS handler name. COISs belonging to the same system type share a single COIS handler. *OwningSystemName* is a COIS name. *OwningAppName* is an application name within a COIS. GHMI allows an object to have a name property *CompName* to emphasize its semantic origins. An object name plays a role as a semantic type. *Attributes* is a sequence of attribute-value pairs representing additional COIS-dependent object attributes. *BridgeLawSpec* is a bridge law ID (BLID) identifying a bridge law which maps the content of the component. The presentation specification *PresentationSpec* is a specification about how a component is presented to the user at run-time. It enables encoding a component's presentation style (e.g., positions in an overview graph and window size.) prior to run-time.

GHMI explicitly distinguishes hypertext components (e.g., annotations) from those mapped from a COIS using the ownership properties (i.e., *OwningSystemType*, *OwningSystemName*, *OwningAppName*). For example, in the domain of RDBMS, a component's *OwningSystemType* could be "Database." Its *OwningSystemName* could be a general RDBMS name such as "MS-Access" or "Foxpro," etc. Its *OwningAppName* could be a specific application database name such "Small School," "GHMI Linkbase," etc. It is helpful to have such ownership information as our intention is to support multiple COISs and applications simultaneously and allow linking among them.

An object name *CompName* plays a role as a semantic type. For example, for those components mapped from database tables, we can name them as "Table" to depict their semantics in the originated COIS. Similarly, a link representing an advisor-student relationship and starting from the advisor's record could have a name "Advisor." For hypertext components, such as annotations, we can name them as "Annotation."

A bridge law is a COIS-dependent mapping rule for mapping COIS objects and relationships to hypertext constructs (i.e., components, links and anchors). Bridge laws are stored in the HTE Knowledge Base. For those components owned by "Hypertext," the *BridgeLawSpec* is *NONE*.

A GHMI link is a set of specifiers. Each link specifier contains a component specification (*CompSpec*), an anchor specification (*AnchorSpec*), a direction and a presentation specification (*PresentationSpec*). We shall discuss details on link specifiers in §3.4. GHMI anchors are defined in link specifiers as *AnchorSpec*. An *AnchorSpec* used in a link specifier combining with the *CompSpec* in the same specifier (which identifies the embedding component) provides complete information to identify an anchor in a component externally.

All GHMI components have a common property *COISObj* (see §3.5.3 for details) which is a COIS-dependent expression indicating their COIS origins (i.e., the original COIS objects they are mapped from). For components not mapped from any COISs, the *COISObj* is *NONE*. The content of a composite component consists of a set of components (*CompSet*) and a set of links (*LinkSet*). Each component in *CompSet* is either identified by a component ID or a *COISObj* expression which resolves to components dynamically by applying corresponding bridge laws. The content (*ContentSpec*) of a structured atomic component is modeled as a sequence of *attribute-value* pairs. This captures the internal structure of an atomic component. For example, a database record could be modeled as a structured atomic with a content as a sequence of field-value pairs. The content of a plain atomic is undefined in GHMI and could be some direct data content or reference to external data content.

The following sections discuss GHMI anchors, links and components in detail.

3.3 Anchors

A GHMI anchor is a portion in the content of a component which marks the endpoint of a link departing from the component. GHMI defines anchors in link specifiers as *AnchorSpec*:

$$AnchorSpec = \langle AnchorID, AnchorType, AnchorValue \rangle$$

which introduces the concept of anchor typing.

The *AnchorID* is a COIS-dependent value which uniquely identifies an anchor location within a component's content. For example, a database record value could be identified by a combination of its key value and field name. On the other hand, a text anchor in a text file can be identified by a combination of its length and offset in the file.

GHMI anchors are typed into three categories: *plain* anchors, *keyword* anchors and *dynamic* anchors. The *AnchorValue* is the anchor content (i.e., the text for a text anchor). The *AnchorID* and *AnchorValue* of different anchor types have different semantics.

- **Plain Anchors**

  A *plain* anchor is an anchor whose *AnchorID* contains explicit location information interpretable to COISs. Plain anchors are created manually and statically.

- **Keyword Anchors**

  A *keyword* anchor is an anchor whose value is a string representing the keyword (or keyphrase). The keyword indicates that its every occurrence inside the contents of the embedding component is an anchor with the same value. The ID of a keyword anchor is statically defined as "NONE" and is resolved to actual anchor IDs (i.e., locations) by the system at run-time. Keyword anchors are defined manually.

- **Dynamic Anchors**

  A *dynamic* anchor is an anchor whose value is defined in a bridge law and resolved to an explicit anchor dynamically at run-time. The ID and value of a dynamic anchor is resolved by executing a link bridge law which maps the link embedding this anchor in its "FROM" specifier. A RDBMS example of a dynamic anchor is a string in a record (probably generated from a query result) which happens to be a table's name. Such an anchor can be defined in MRDC with semantics like "A is an anchor if it is a $Value$ of a record and also the name of a table in a database." Embedding such an *AnchorSpec* in a link specifier of a bridge law defines the entire class of such anchors. We shall see examples of dynamic anchors in §5.

3.4 Links

Besides common object properties, a GHMI link has property *LinkType* representing six link categories. A link consists of a sequence of link *Specifiers* which specify the link endpoints. A hypertext under GHMI helps a COIS establish direct access to explicit and implicit relationships among underlying COIS objects. This section discusses link types and link specifiers.

3.4.1 Typed Links

Typed links provide an easier and clearer mechanism for both the readers and authors to understand a hypertext information network. Link typing enhances the power of two navigational tactics: *filtering* and *zooming*. Filtering occurs when the user is presented by the system with a subset of links which can be followed. With untyped links, however, the user could be overwhelmed by the cognitive overhead of dealing with the whole set of links outgoing from components. Filtering on link types restrict his or her navigation to link types of interest while disabling others. Links in GHMI have a property *LinkType*, representing six categories of links based on different roles they are playing in a hypertext system.

$$LinkType = \text{“}StructureLink\text{”} \mid \text{“}ReferenceLink\text{”}$$
$$\mid \text{“}AnnotationLink\text{”} \mid \text{“}AssociationLink\text{”}$$
$$\mid \text{“}NavigationLink\text{”} \mid \text{“}OperationLink\text{”}$$

Structure links represent the underlying structural inter-object relationships within a COIS domain. In a well-organized information system, among the various types of inter-object relationships, there might be distinguishable relationships which dominate the overall information organization and can be represented as structure links. For example, in a RDBMS, a database consists of tables, a table consists of records and a record consists of individual values. Such "consists of" (or its reverse direction) relationships can be mapped as structure links which allow direct access from a parent object to a child object or vice versa. Structure links are dynamic links defined by bridge laws.

Reference links depict cross-reference relationships among components, which can be generated automatically by the system according to predefined bridge laws. In the domain of RDBMS, the ER diagram itself represents a cross-referencing relationship among entities. Although these relationships are lost when we map the ER diagrams into flat database tables, GHMI could restore them as reference links. Other examples include defining a reference link from a record to another record which has the same key value. The system should be able to compute such links automatically based on their bridge laws. Therefore, reference links are also dynamic links defined by bridge laws.

Annotation links connect objects to their annotations. An annotation is a commentary document attached to an object. We separate an object from its annotation by placing the annotation in a separate atomic component and connecting it to the object through an annotation link. Unlike structure links and reference links, annotation links are pure hypertext features which have no corresponding mappings in the underlying COIS. They are static links created manually.

Association links are user-declared *ad hoc* links representing semantic relationships among objects. Users can add such links to or delete them from a hypertext network at will. Association links are non-automatable (otherwise they would be reference links). Instead, they are defined manually based on a semantic conceptualization in the user's mind which is not interpretable by the system. The user can define any links among objects and give them semantic labels. In GHMI, an association link could be an inter-COIS link which relates an object in a COIS to an object in another COIS. Association links are also static links created manually.

Navigation links are system-generated links for navigation purposes. Such links are used to construct navigation structures (e.g., guided-tours, see §3.6). Navigation links do not reflect inter-object relationships. They are dynamic links and generated automatically by the system according to the user's navigation requests. Navigation links are transparent to users. Users might have no knowledge about the existence of these links.

Operation links model operational commands and queries over a hypertext network. They are dynamic links defined by bridge laws. An operation conducted on an object can be modeled as an operation link from the object pointing to the operation results (which might be generated as destination components). Operations invoked from an interface menu item can be modeled as an operation link with no departing component. The computation of the destination components might be completed by the cooperation of the HTE and the COIS. In the case of RDBMS, operations such as database queries can be modeled as operation links. The user can access these links directly. For example, we can define a query as an operation link. Another type of operation link in RDBMS is user-declared queries. The user can define frequently-used specific queries as ready-to-follow operation links using a COIS-supported query language. When following such a link, instead of executing it directly, the hypertext engine sends the operation to the COIS for solutions. The destination of such a link would be the query results resolved dynamically by the COIS's query processing system every time this link is followed.

3.4.2 Link Specifiers

The "content" of a GHMI link is a sequence of link $Specifiers$. A $Specifier$ defines a link endpoint through four fields:

$$Specifier = CompSpec, AnchorSpec, Direction, PresentationSpec$$

CompSpec is a component specification identifying a component as a link endpoint. The GHMI concept of *CompSpec* reflects dynamic mapping between COIS objects and GHMI components. The GHMI *CompSpec* is either a hypertext object specification *HTObj* (if not mapped from COIS) or a COIS object specification $COISObj$, plus an optional sequence of ownership properties:

```
CompSpec   = HTObj
           | (COISObj,
           [OwningSystemType,OwningSystemName,OwningAppName])
```

A *CompSpec* uniquely identifies an object in a GHMI hypertext system. An *HTObj* could be an explicit global component ID (GID) or a hypertext query expression which resolves to a component ID by some hypertext query processing function. (The discussion on hypertext queries is out of the scope of this thesis.) A $COISObj$ is an expression (see §3.5.3) which resolves to a component mapped from COIS object contents at run-time through bridge law execution. If the *CompSpec* of a link does not resolve to an explicit component, the link endpoint becomes a dangling component and the link becomes a dangling link. This could happen when the corresponding COIS object is deleted without notifying the hypertext system. The system should be able to provide users with information regarding this situation. If there are no ownership properties in a specifier's *CompSpec*, the specifier inherits the link's ownership properties. §3.5.3 discusses more details about *CompSpec* in GHMI.

AnchorSpec specifies an anchor in the a link specifier to mark a link endpoint. A link specifier representing an entire component has "NONE" as its *AnchorSpec*. *Direction* defines the directional nature of the link endpoint as one of "FROM" (a departure), "TO" (a destination), "BIDIRECT" (both departure and destination), and "NONE." Such a notion of link specifiers is powerful for modeling multi-headed n-ary links (i.e., a link with more than two endpoints). Unidirectional binary links are modeled as two endpoints with one directed as "FROM" and the other directed as "TO." Bidirectional links (e.g., an association link "Co-workers") have both endpoints directed as "BIDIRECT." An operation link which is not departing from any components (e.g., "Open Database" command in RDBMS) could be modeled as a unary link with a single endpoint (e.g., the destination database of an "Open Database" command) directed as "TO."

Note that the *PresentationSpec* in a link endpoint is a link property different from the *PresentationSpec* property of the endpoint component itself. For example, to distinguish an expert-user presentation and a novice-user presentation of a component, we can encode the accesses to the component as two links with distinct *PresentationSpec* (e.g., one defines the endpoint as "editable" and the other defines it as "read-only") regardless of the component's own property *PresentationSpec*. In GHMI, embedding *PresentationSpec* in components is optional. *PresentationSpec* can also be used to define the view style of a component (see §3.5.2).

3.5 Components

GHMI classifies components into three subclasses: plain atomic components, structured atomic components and composite components.

3.5.1 Atomic Components

An atomic component in GHMI can not embed other components in its content. An atomic component could be structured or unstructured. A *plain* atomic is an unstructured atomic which has a content without any internal structure. In a multimedia environment, typical examples of plain atomics include a page of text, a picture, a raster of image, a short audio tape, a short animated sequence, etc. The content of a plain atomic component is primitive and unspecified in GHMI. Currently we only consider text atomics in GHMI and believe that the model can be extended to include other kinds of data resources.

The content of a *structured* atomic comprises a sequence of *attribute-value* pairs interpretable to COISs. Examples include a database record, a hypertext link browser (consisting of a list of link references), etc. We model components with only attributes (e.g., a database table schema which is a sequence of field names) as *NONE-value* structured atomics by specifying *NONE* in their values. For structured atomics, we can define certain structure-based operations, such as linking to or from a dynamic anchor defined on an attribute or value. Complicated component content structures, such as "table of content" or a database table, can be represented as composite components with internal structures (see below).

3.5.2 Composite Components

The concept of composite greatly improves the organization of a hypertext network. Composites provide a more powerful way to construct a hypertext network over the pure low-level node-link model and assist both users and authors at various levels. During navigation, for instance, with composites the user can *zoom* into a particular subcomponent for details or *zoom* out it to navigate along the overview structure of a composite. On the other hand, some COIS (such as DBMSs) have their own data models, i.e., objects of these COISs are well-structured. It is essential for a hypertext system to capture these COIS structures and map them faithfully to compatible hypertext structures.

A composite component (or simply *composite*) is constructed from other components. Individual components embedded in a composite could be any type of components themselves, including composites, plain atomics and structured atomics.

GHMI explicitly classifies composites based on the representation of their internal structures as *Set*, *List*, *Tree* and *Graph*. A *Set* consists of a set of components and no explicit links exist among these components. A *List* is composed of an ordered set of components connected linearly. A *Tree* is constructed from a set of components connected as a tree-like structure and has a distinguished component as its *root*. A *Graph* has components as "nodes" and links as its "edges".

One purpose of modeling composite structures is to build multiple views from a composite based on its internal structure. In GHMI, besides position information, the *PresentationSpec* can be used to define multiple views of a component. Usually, *PresentationSpec* is the same as the *CompClass*. For composite components, however, we can view them in another style coded in *PresentationSpec*. For example, in RDBMS, a database can be mapped to a Set component which consists of a set of tables (identified by their names). A table, in turn, consists of a set of records (identified by their keys). A regular Set view of a database object would be a set of table names. We can overwrite such a view by defining *PresentationSpec* as "Tree." A Tree view of a database object expands all of its tables and records. That means we would see a three-level tree: The root in level 1 is the database name itself. Level 2 contains all table names. Level 3 contains all record keys. Clicking on record keys will bring up record contents. We can also view a Set as a default guided-tour (DGT, see the next section) by specifying *PresentationSpec* as "DGT." Component view styles can be also coded in link specifier's *PresentationSpec*. When a component with its own *PresentationSpec* is defined as a link endpoint, the link specifier's *PresentationSpec* overwrites the component's *PresentationSpec*.

3.5.3 Virtual and Computed Components

From the component creation point of view, a GHMI component is either a hypertext component or COIS component. Therefore, the component specification (*CompSpec*) in GHMI is either a hypertext object (*HTObj*) or a COIS object (*COISObj*) along with ownership properties.

HTObj specifies either an explicit hypertext component by an explicit ID or a computed hypertext component by a hypertext query. A hypertext query is a COIS-independent query expression, which usually requires structural information [44]. Examples of such queries are "Find all components with annotations," "Find all components with only one departure link," "Find all unary links," etc. The hypertext engine is responsible to resolve such kind of structural queries into UIDs. This is an advanced hypertext functionality which most current hypertext systems do not support. We shall not discuss the details of hypertext queries in this thesis. We consider *CompSpec* as COISObj (along with ownership properties) only.

GHMI employs the concept of dynamic mapping. One of our major concerns is to map COIS objects to GHMI components and therefore the COIS can take advantage of hypertext functionality without changes on its underlying organization. On the other hand, the hypertext system also wants to support the computation abilities (such as query processing) of COISs fully. A COIS component is a component mapped from a COIS object dynamically through predefined bridge laws. The mapped component is not persistently stored in the HTE Linkbase. Every time it is required by the user, the HTE dynamically generates its content. Therefore, every COIS component is a *virtual* and *computed* component. We map COIS objects to GHMI components only *on-demand* at run-time according to their specifications in COISObj and corresponding bridge laws. A COIS object ($COISObj$) can be internally represented as an explicit COIS object expression:

$$COISObj = \langle COISID, COISType, COISLabel \rangle$$

Once an explicit expression of COISObj is defined, the HTE is ready to apply a bridge law (according to the COISID and the COISType) to map its content. To accomplish such a mapping, the HTE packs up a message requesting the to-be-mapped object information from the COIS by specifying its COISID and mapping rules. After a COIS object is mapped to a component, the HTE executes bridge laws to generate all link anchors departing from this component.

For dynamic links whose endpoints contain implicit information (e.g., defined in a bridge law), the HTE needs to apply a link bridge law to resolve implicit information to explicit COISObj expression. For example, in RDBMS, the specification for the endpoint of "all tables having the same key field with the current table" could be $\langle X, \text{"}Table\text{"}, Y \rangle$ in a link bridge law, where X and Y are MRDC variables. The HTE needs to resolve them to an explicit COISID and COISLabel based on the MappingRule in the bridge law in order to make the target tables directly accessible by users. The corresponding table contents are not generated until these tables' COISIDs are selected by the user.

Only the contents of hypertext components (e.g., annotation and association links) are persistent in the HTE Linkbase. The contents of COIS components are dynamically computed whenever they are selected. The entire hypertext network is generated dynamically from underlying COIS databases. Such an approach effectively separates the HTE from COISs and reduces the data consistency problem caused by HTE-transparent COIS operations (e.g., "Edit Table" and "Delete Table" in RDBMS which could be happening outside and beyond the control of the HTE).

3.6 Navigation Structures

In the previous sections, we presented GHMI's basic elements. We model navigation structures in terms of GHMI composite components in this section.

The associative nature of a hypertext network structure enables hypertext users to manage and access data stored in a hypertext database with great flexibility. It is this flexibility, however, that frequently causes user cognitive overhead and disorientation during navigation courses over the hypertext network. This classic hypertext navigation problem—user disorientation—has been identified and discussed extensively in hypertext literature [3, 18, 44, 72, 90]. Arbitrary linking even has been compared to the abuse of *GOTO*s in non-structured programming [22]. Efforts have been made to alleviate the disorientation associated with hypertext's non-restrictive linking and direct user-access features. Navigation via graphical maps and overviews [31, 59, 71, 72, 90] has been proved a useful tool in many hypertext systems such as Intermedia [77, 101], gIBIS [20, 19], NoteCards [45], PlaneText [18] and Neptune [25]. Query-based filtered browsers [45, 25], history list [72], bookmarks and Intermedia's Web View [90] are also helpful mechanisms towards disorientation reduction. Navigation via guided tours [89, 68, 42, 34], combined with other techniques, reduces both disorientation and user cognitive overhead.

Benefiting from the experience of other hypertext researchers, GHMI provides a comprehensive level of navigation structures including bookmarks, network overviews and guided-tours. We include these structures effectively modeled in terms of composite components. One major contribution of GHMI on navigation modeling is the introduction of the four guided-tour categories (query-based guided-tours, default guided-tours, user-defined guided-tours and navigation-based guided-tours) which are not found in any other hypertext literature. This section focuses on the representation of navigation structures regarding bookmarks, network overviews and guided-tours. We model some other navigation structures (e.g., sessions) and facilities (e.g., backtracking) which rely on the run-time user interaction in §3.8.

3.6.1 Bookmarks

Some components in a hypertext network may be of special importance to the user. It is helpful to provide a direct access to these components from any navigation position. These components are called *bookmarks*. Navigation links are maintained by the system to allow direct access to bookmarks. Bookmarks are special components in the hypertext network which are directly accessible from all other components. GHMI models bookmarks as a *Set* composite with an *index* link pointing to it. This index link is a unary link of type "NavigationLink" with only an endpoint directed as "TO" indicating this is a component accessible from all components (usually through a menu bar item). Users are allowed to manipulate (add or delete a bookmark) the bookmark *Set*. The content of this *Set* is a set of component specifications (*CompSet*). As a result of dynamic mapping, the content of a COIS-mapped bookmark is actually generated when the user selects it (on its icon or label).

3.6.2 Network Overviews

Users often get lost when exploring hypertext networks. A network overview [71, 90] (or simply called an *overview*) is a vision of a substructure of a hypertext network. Overviews help alleviate the network disorientation [18, 72] by giving the user a sense of context. GHMI models overviews on composite components. A component overview is constructed as a virtual component based on the component's internal structures, which could be a Set, List, Tree, or a Graphs, depending on the complexity of the original COIS object.

3.6.3 Guided-Tours

From a control point of view, navigation over a hypertext network can be user-controlled and system-controlled. In a user-controlled navigation, all paths are determined by the user through navigation commands provided by the system. In system-controlled navigation, the navigation paths are prepared by the system following some user input commands. By default, the user is not allowed to use navigation commands to choose his own paths once getting on a system-controlled navigation path (though the user may be able to overwrite this). One typical example of system-controlled navigation is a *guided-tour* [34, 42, 68, 89] (GT) which is a navigation structure built from a sequence of components as a linear path. When navigating on a GT, the user must follow the GT to access information. No branch links are available unless an explicit request is applicable to overwrite the prepared paths. The user can get on or off a GT from any other navigation pattern. At any stop of a GT, the user is allowed to invoke other links by pausing the tour and returning back later.

GHMI models a GT as a *List* composite consisting of a sequence of components and a set of links. Each link is a "NavigationLink" named "NextGtStop." A link endpoint is called a *GtStop* which can be any type of component (e.g., a link, a component, or another guided-tour). The user can only follow link "NextGtStop" linearly to access *GtStops* in the order they are connected in the GT. In a GT, two links are distinguished to represent the starting and ending *GtStops*. The starting link has only one endpoint directed as "TO" while the ending link has only one endpoint directed as "FROM." All other links in a GT have two endpoints directed as "TO" and "FROM" respectively. Figure 3.3 shows the concept of *List* representing a GT. This GT contains four *GtStops*: components $C1$, $C2$, $C3$ and $C4$, as shown in Figure 3.3(b). Internally this GT is represented as four links $L1$, $L2$, $L3$ and $L4$, as shown in Figure 3.3(a), which embed the component UIDs in their endpoint

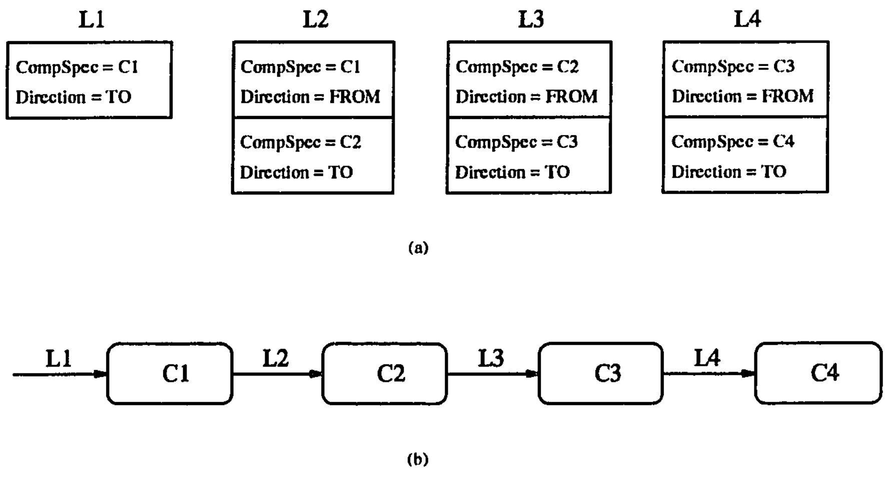


Figure 3.3 A Guided-tour Example

specifiers. In the figure we only show the *Direction* and *CompSpec* of each link and the UIDs of each *GtStop.* The content of a link endpoint is dynamically computed one by one when the GT is followed by the user. For example, a GT resulted from a RDBMS query "Find all Ph.D. students' names who take CIS 610" would be a set of records as a GT consisting of student names.

GHMI classifies guided-tours into four categories: default guided-tours (DGTs), query-based guided-tours (QGTs), user-defined guided-tours (UGTs) and navigation-based guided-tours (NGTs).

- **Default Guided-tours (DGTs)**

  DGTs are derived from the structural information of a composite. They are created automatically by the system and directly available to the user. A DGT of a composite is a *List* over links of type *NavigationLink* automatically derived from structure links of the composite . One way to obtain a DGT from a composite is to expand the breadth-first search tree on the original structure links level by level and order the resulted components in a linear manner. A DGT *GtStop* could also be another DGT if the corresponding component is a composite.

- **Query-based Guided-tours (QGTs)**

  QGTs are created by the system representing query results. The components resulting from a query are organized as *GtStops*. A *GtStop* in a QGT could be another guided-tour. If a *GtStop* is a composite, it could be targeted as a DGT of the composite instead of presenting the entire composite and expecting the user to browse it.

- **User-defined Guided-tours (UGTs)**

  The user is able to define a UGT on a set of components in the same way as defining *ad hoc* association links. In this case, the resulted links would be *ad hoc* navigation links which group participating components into a *List* composite as a UGT. Once defined, a UGT can be invoked arbitrary times until deleted by the user. The user can manipulate a UGT (e.g., annotating, deleting or adding new components, etc.) as a normal composite.

- **Navigation-based Guided-tours (NGTs)**

  The user can define an NGT based on his or her individual navigation history stored in the *History Log* (see §3.8). The user can select events from the *History Log* to construct an NGT. Once constructed, an NGT (actually its specifications) exists in the HTE Linkbase until the user deletes it explicitly. As with UGTs, the user can also manipulate NGTs at run-time.

The navigation structures (bookmarks, network overviews and a variety of guided-tour types) presented in this section help reduce user disorientation and provide the user a flexible, comprehensive and well-structured mechanism to customize individual navigation environment over a hypertext network.

3.7 A Bridge Law Template

In this section, we discuss bridge law definitions. GHMI employs bridge laws to map COIS objects and relationships to hypertext constructs. A single bridge law will map entire classes of COIS objects satisfying the bridge law's condition. All components which represent COIS objects are generated dynamically through bridge law mappings in response to user requests (e.g., traversing a link to bring up the destination component). When the user selects an object, bridge laws determine what COIS objects, operations, or relationships will be mapped from the COIS. As an early step towards demonstrating the power of domain mapping, we explored logical modeling on the representations of both TEXPROS and hypertext in [96, 97]. In our work aiming at developing a general hypertext data model, we have further refined bridge laws using logic modeling approach. This section presents a bridge law template as a standard format for defining bridge laws. We also discuss a simple mapping rule definition convention (MRDC) for defining expressions used in COIS-dependent component property specifications.

Bridge laws are stored in the HTE Knowledge Base and identified by their bridge law IDs (BLIDs). Each COIS has its own set of bridge laws written by its builders during the course of system set-up. The HTE dynamically invokes these bridge laws using argument settings as input to generate components. Defining a bridge law requires specifying the properties of the component to be mapped by this bridge law in terms of COIS-dependent mapping rules. This section presents a general template for writing bridge laws and a mapping-rule definition convention (MRDC). We shall also briefly illustrate how to use this bridge law template and MRDC expressions (see §3.7.3) to define and execute bridge laws through RDBMS examples.

We consider two types of bridge laws: component bridge laws and link bridge laws. A component bridge law maps a COIS object to a component. A link bridge

Table 3.1 A Component Bridge Law Template

CompClass	
OwningSystemType	
CompName	
PresentationSpec	
COISObj	
CompSet	
LinkSet	
ContentSpec	
MappingRule	

law maps a COIS relationship to a link. A COIS object is mapped to a component when it is selected by the user (usually as an link endpoint selected by a link traversal command). When a COIS object is mapped to a component through a component bridge law, the HTE executes link bridge laws to map all link anchors departing from the mapped component. These links are marked by anchors embedded in the component content. The mapping of link endpoints is delayed until the links are actually traversed.

3.7.1 Component Bridge Laws

Table 3.1 shows a component bridge law template which is a two-column table. The left column contains a list of component property names. To write a bridge law is to define the properties in the corresponding right-column items. A right-column item of a bridge law template could be either a constant (e.g., "Hypertext," "Table," etc.) or MRDC (see below) variables whose semantics are defined in the right column of the *MappingRule*. A *MappingRule* is a set of MRDC predicates representing COIS-dependent information.

3.7.2 Link Bridge Laws

A link bridge law defines a COIS relationship which will be mapped to a GHMI link. The HTE executes a link bridge law when the component embedding the anchor

Table 3.2 A Link Bridge Law Template

CompClass	
OwningSystemType	
CompName	
PresentationSpec	
LinkType	
$\{\text{Specifier}_i\}^+$	
MappingRule	

marking this link is brought to display. The link's endpoints are not mapped until this link is actually traversed. Table 3.2 illustrates a link bridge law template. A link bridge law template is similar to the component bridge law template, except that a link has a *LinkType* and a list of *Specifiers* instead of component properties (i.e., *COISObj*, *CompSet* and *LinkSet*). A specifier is a composition of

$$\{CompSpec, AnchorSpec, Direction, PresentationSpec\}.$$

A link bridge law defines link properties in terms of MRDC expressions.

The GHMI bridge law templates are greatly influenced by the bridge law notions of Bieber et al.'s work [12, 9], which has no composites and maps nodes, links and anchors separately, which have not been implemented. (Bieber's current prototype uses bridge laws developed specifically with this implementation in mind. While they are general enough for any COIS, they do not come from a principled model.) Bieber's bridge laws correspond to GHMI bridge laws' *MappingRule* part. GHMI bridge laws map COIS objects to more complex hypertext constructs. The GHMI *MappingRule* is more formalized and simpler (only three predicates, see §3.7.3). By modeling bridge laws in a table format combined with a simple set of predicates, GHMI makes bridge laws more understandable.

GHMI extends and formalizes the previous bridge law formats to support composites and mappings from COIS to a GHMI hypertext network. Our previous work on bridge laws [96, 97, 94, 93] was based on modeling the domain of a document management system. In this thesis we focus on the domain of RDBMS which is quite different from the document management domain. Our goal of modeling distinct domains has led us to generalize bridge law templates and prove our idea of using GHMI as a general model for all COISs. We present the GHMI version of TEXPROS bridge laws in Appendix A.

3.7.3 MRDC: A Mapping Rule Definition Convention

The major part of a bridge law template is the *MappingRule*. In order to provide a formal template to define bridge laws, we need to formalize expressions representing COIS-dependent information. Benefiting from our previous efforts on modeling bridge laws using logic, we model MRDC as a subset of Prolog which consists of a set of constant symbols, variable symbols, a small set of predicates and functions. The basic MRDC elements include the following.

1. Primitive Symbols

   - **Propositional Connectives:** $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction), $\rightarrow$ (implication), $\equiv$ (equivalence), $=$ (equality), $\neq$ (non-equality), $\forall$ (universal quantifier) and $\exists$ (existential quantifier).

   - **Set Connectives:** $\cup$, $\cap$, $\subset$, $\subseteq$, $\supset$, $\supseteq$.

   - **Variable Symbols**

     - **Simple Variables** are upper-case-leading strings (e.g., X, Y, $Z1$). When used separately, symbol '_' represents "arbitrary" or "don't care" values. MRDC has two types of variables: simple variables and list variables.

- **List Variables** are variables denoted as a list of other variables, i.e., $X = [X_1, X_2, ..., X_n]$, where X is a list variable and every $X_i (1 \leq i \leq n)$ is either a simple variable or a another list variable.

- **Constant Symbols** are strings quoted in quotation marks (e.g., "MS-Access", 'Record', etc.) which represent instantiated variable values.
- **Function Symbols** are upper-case strings, including symbol '_' (e.g., *APPLY_BL_COMP*, *GET_PROPERTY*, etc.).
- **Predicate Symbols** are lower-case-leading strings (e.g., *object*, *relation*, etc).

2. Predicates

   There are three predicates in MRDC:

   - **object(X,ClassName)**

     Predicate *object*$(X, ClassName)$ identifies an object X belonging to a class named *ClasssName*. X is a variable. *ClassName* is a constant string. For example, in a database bridge law, *object*$(X,$ '*Table*'$)$ indicates that X is a database object belonging to a class named "Table."

   - **property(X,PropertyName,Y)**

     Predicate *property*$(X, PropertyName, Y)$ indicates that object X has property named *PropertyName* and the value of this property is Y. X is a variable. *PropertyName* is a constant string. Y could be a variable, a constant string, or a function which returns a value. For example, *property*$(F,$ '*keyField*', '*SSN*'$)$ indicates that F has a property "KeyField" as "SSN."

   - **relation(X,Y,RelationName)**

     Predicate *relation*$(X, Y, RelationName)$ indicates that object X and object Y have a relationship named *RelationName*. X and Y could be variables. *RelationName* is a constant string. For example, $relation(X, Y, \text{'}Likes\text{'})$ indicates that X and Y have a relationship "Likes" (i.e., X "Likes" Y).

3. A Special Function

   $OPERATION(Z, X, Y)$ is a special function which is available in MRDC expressions. Function $OPERATION(Z, X, Y)$ identifies a COIS-supported operation on object Z. The operation's name is X and it takes Y as an argument list. X is a constant symbol representing an operation name (e.g., "Query" in a database). Y is a plain string expression interpretable to the COIS when combined with the operation name (e.g., in a database, Y could an SQL statement). Variables inside Y have prefix "$$". For example, in a database, $OPERATION$('Small School', 'Query', 'Select *Name* from *Employee* where *Salary* $\geq$ 40,000') denotes a query on database "Small School" with no variables. Expression $OPERATION$('Small School', 'Query', 'Select *Name* from $$X where *Salary* $\geq$ 40,000') contains a variable X (stands for a table name) which needs to be instantiated when this query is passed to the COIS handler. The return value of $OPERATION()$ is the operation results resolved by the underlying COIS.

3.7.4 Executing a Bridge Law

Although we define bridge laws in format of tables, this by no means implies that bridge laws are only simple "look-up" tables. A bridge law is *applied* in the HTE (i.e., preparing correct arguments) and actually *executed* in a COIS handler. Internally, prior to executing a bridge law, the COIS handler needs to translate it to a set of Prolog predicates. Therefore, the entire table of a bridge law definition implies a set of predicates. An execution of a bridge law would take given variable values (e.g., COISID, COISType, etc.) to instantiate all other free variables in the predicates.

In other words, variables are "inferred" from the predicates defined by a bridge law. This procedure is similar to running a query under Prolog. A COIS handler usually has a bridge law engine to handle bridge law execution.

The HTE employs three bridge law functions to apply bridge laws: $APPLY\_BL\_COMP()$, $APPLY\_BL\_LINK()$ and $APPLY\_BL\_ANCHOR()$, which apply a bridge law to generate components, link endpoints and dynamic anchors, respectively.

- $APPLY\_BL\_COMP(BLID, ArgumentSpec)$

  This function is responsible for instantiating a component BL to a component. $APPLY\_BL\_COMP(BLID, ArgumentSpec)$ instantiates component bridge laws in the HTE Knowledge Base to construct virtual components. $APPLY\_BL\_COMP()$ takes two parameters: a component bridge law ID specified by $BLID$ and a list of parameter specifications in $ArgumentSpec$. For example,

  $$APPLY\_BL\_COMP(BL_{Table1}, [D, T] = [`SmallSchool`, `DoctoralStudent`])$$

  applies bridge law BL_{Table1} to generate a component from the content of table "DoctoralStudent" in database "Small School."

- $APPLY\_BL\_LINK(BLID, ArgumentSpec)$

  This function is responsible for mapping a link endpoint from a link bridge law. It takes a link bridge law and a list of arguments to map a link endpoint specified in a link specifier with "TO" direction. For example,

  $$APPLY\_BL\_LINK(BL_{SameKey}, [D, T] = [`SmallSchool`, `DoctoralStudent`])$$

  applies link bridge law $BL_{SameKey}$ to generate components from tables having the same key as table "DoctoralStudent" in database "Small School."

- $APPLY\_BL\_ANCHOR(BLID, ArgumentSpec)$

  This function is responsible for mapping dynamic anchors defined in the "FROM" specifiers of a link bridge law. Like $APPLY\_BL\_LINK()$, this function also takes a link bridge law and a list of arguments. But instead of generating the link endpoints in "TO" specifiers like $APPLY\_BL\_LINK()$ does, instead it generates dynamic anchors defined in "FROM" link specifiers by applying the MappingRule in the bridge law. After mapping, a dynamic anchor is temporarily stored in the HTE Linkbase and is ready to follow as if it were a plain anchor. For example,

  $$APPLY\_BL\_ANCHOR(BL_{RefToTable},$$

  $$[D, T, K] = [`SmallSchool`, `DoctoralStudent`, `123456789`])$$

  applies link bridge law $BL_{RefToTable}$ to generate components from tables whose names appear as a value in record "123456789" in table "DoctoralStudent" of database "Small School."

3.7.5 Bridge Law Examples

Table 3.3 shows a component bridge law example BL_{Table1} and Table 3.4 shows a link bridge law example $BL_{RefToTable}$.

BL_{Table1} maps *tables* to *set* components from records, as shown in Table 3.3. The resulting component contains a set of record components mapped from database records by the above BL_{Record}. A table is identified by its table name and the database name in which it resides (i.e., $[D, T]$). The content of the resulting composite is a set of record components. $\{[D, T, K], `Record`, K]\}^*$ means 0 or more records. $object([D, T, K], `Record`)$ indicates that K is a record residing in table T of database D. The *CompSet* does not include ownership properties as the corresponding COIS objects (i.e., records) inherit these properties from their embedding table.

Table 3.3 Bridge Law BL$Table1$

CompClass	'Set'
OwningSystemType	'Database'
CompName	'Table'
COISObj	$[[D, T], \text{'}Table\text{'}, T]$
CompSet	$\{[[D, T, K], \text{'}Record\text{'}, K]\}^*$
LinkSet	'NONE'
ContentSpec	'NONE'
MappingRule	$object([D, T, K], \text{'}Record\text{'})$.

The following instantiation of BL_{Table1} maps a the content of table "Doctoral-Student:"

$$APPLY\_BL\_COMP(BL_{Table1},$$

$$[D, T] = [\text{'}SmallSchool\text{'}, \text{'}DoctoralStudent\text{'}])$$

$BL_{RefToTable}$ maps an implicit relationship between a record and a table whose name appears in the record as a value, as shown in Table 3.4. The $CompSpec$ in this link bridge law's specifiers does not include ownership properties as the corresponding endpoint COIS objects (i.e., the record and tables) inherit these properties from this link.

$BL_{RefToTable}$ defines a dynamic anchor in the departing record. The anchor's value V happens to be a table's name in the same DB. Based on such an implicit relationship, this BL constructs a reference link from the record to the table marked by its table name (highlighted as anchors) in the record's content. $BL_{RefToTable}$ is frequently used in the GHMI prototype to present a query result and other reference link destination mappings to the user (see §6). We consider a query result as a dynamic table. The user can navigate on its records via a query-based guided-tour (QGT). We can apply $BL_{RefToTable}$ to the records contained in all dynamic tables (i.e., those resulted from operation links and reference links) as well as static tables (i.e., regular tables in a DB).

Table 3.4 Bridge Law BL*RefToTable*

CompClass	‘Link‘
OwningSystemType	‘Database‘
CompName	‘RefToTable‘
LinkType	‘ReferenceLink‘
Specifier$_1$	
CompSpec	$[[D, T, K], \text{‘}Record\text{‘}, \_]$
AnchorSpec	$[[D, T, K, F], \text{‘}Dynamic\text{‘}, V]$
Direction	‘FROM‘
Specifier$_2$	
CompSpec	$[[D, V], \text{‘}Table\text{‘}, \_]$
AnchorSpec	‘NONE‘
Direction	‘TO‘
MappingRule	$object([D, T, K], \text{‘}Record\text{‘}),$ $object([D, T, K, F], \text{‘}Value\text{‘}),$ $object([D, V], \text{‘}Table\text{‘}),$ $property([D, T, K, F], \text{‘}Content\text{‘}, V).$

We illustrate more bridge laws in §5 and §A. §5 discusses how to use MRDC and other GHMI constructs to map RDBMS to hypertext. §A gives bridge law examples on another domain—TEXPROS, a document management system.

3.8 Session Structures

In modern hypertext systems, it is crucial to provide users with a friendly, flexible and reliable navigation environment over a hypertext network. Such an environment heavily relies on what run-time structures the underlying hypertext model provides. A well-organized navigation environment should be able to efficiently reduce user disorientation as much as possible. Over a session of interactive activities, the user can invoke many run-time navigation patterns such as forward browsing, backtracking and backjumping. We introduced the concept of task-based backtracking in multi-window environments in [13]. This section focuses on the run-time structures for the HTE Session DB, including event structures and system traversal logs. The HTE Session DB stores history information regarding a user navigation session based on these structures. More details can be found in [13], where we illustrated how these structures support navigation facilities and presented a preliminary algorithm for implementing task-based backtracking based on these structures.

3.8.1 The Event Structure

We define an *event* as any user action which affects the system status. These actions usually cause some change on the user interface such as creating a new window or closing an existing one. We classify events into *forward*, *backwards* and *switching* events. Link traversal is a *forward* event. Backtracking (executing a backtrack command) is a *backwards* event. Selecting, opening and closing comprise the *switching* events as each deactivates the current window and activates a different one. (Closing a window activates the window beneath it, if any.) To support different kinds of backtracking and other navigation facilities (e.g., creating history-based guided-tours), the system keeps a complete set of user event information, which we record in the following *event structure*.

We represent each event by a tuple $\langle I, A\rangle$. The event identifier I provides a unique reference to the event. A contains the set of attributes which characterizes the event. Event attributes include the following:

Event-type. An event can be one of five types:

- *traversal* – traverse a link to a new (or already displayed) window;
- *open* – create a new window (or activate an already displayed window) explicitly by executing an "open window," "open new document," or "open new component" command;
- *select* – activate an existing window directly by selecting it, not through any link traversal;

- *close* – close an existing window directly by executing a "close window" or "close component" command; and
- *backtrack* – backtrack or backjump along a link, or more generally, along a previous event by executing a "backtrack" command.

Departure-component This field contains the UID of the "departure" component from which an event originates.

Destination-component This field identifies the UID of the "destination" component that the event activates.

Subtask-log-id This field indicates the *Subtask Log* (see §3.8.2) referencing this event.

Log-index This field contains an integer indicating the event's chronological position in the *Chronological Log* (see §3.8.2). This attribute applies only to traversal events.

The system stores events in a system session structure called *System Traversal Logs*, which we describe next.

3.8.2 System Traversal Logs

To track user actions and enable multiple types of backtracking, GHMI maintains a *System Traversal Log* structure consisting of three types of traversal logs:

- **History Log**

  The *History Log* records the complete event structure for every user event, including its event identifier and all attributes. In addition to backtracking, users could employ the *History Log* to create guided tours. Experimenters could use it to trace and analyze user actions.

- **Chronological Log**

  Unlike the *History Log*, the *Chronological Log* only registers forward (traversal) events. Each entry contains an event identifier corresponding to an event in the *History Log.*

- **Subtask Logs**

  Similarly, *Subtask Logs* only contain forward events. Each *Subtask Log* contains all uninterrupted forward traversals. The system starts a new *Subtask Log* whenever a forward event happens after a backwards or switching event. Each entry contains an event identifier corresponding to an event in the *History Log.*

3.9 Summary

In this chapter, we presented GHMI's system architecture and basic concepts. The GHMI system architecture consists of three basic layers: the computation-oriented information systems (COISs), a hypertext engine (HTE) and the interface-oriented systems (IOSs), each running as independent processes. COISs and IOSs are connected to the HTE by their handlers. Currently our major contribution focuses on the COIS-HTE side. To integrate a COIS to a hypertext system based on our model, the COIS builders have to write the bridge laws stored in the HTE's Knowledge Base and write the handler code for their individual COISs. The HTE uses predefined bridge laws to map COIS to hypertext. GHMI aims at providing a robust data model for representing the functionalities of the HTE toward integrating COISs with hypertext.

GHMI presents a hierarchical object class representation. Basic GHMI concepts include dynamic anchors, behavioral link typing, composite structures, structured atomic components, virtual components, dynamic mapping and computed components. GHMI also includes several navigation structures (bookmarks, network overviews and guided-tours). GHMI enables dynamic mapping of COIS objects and relationships through a bridge law mechanism. Each bridge law is defined using a template and a simple Prolog-like mapping rule definition convention MRDC. All of these provide a powerful and comprehensive data representation framework for our platform of supporting COIS-hypertext integration. GHMI's basic concepts include the following: (1) Object class hierarchy: GHMI models objects as links and components. Components are classified into atom components and composite components. GHMI distinguishes plain atomic and structured atomic components and models four subclasses of composites based on their internal structures; (2) Anchors: GHMI models external anchors, anchor typing and the concept of dynamic anchors. Dynamic anchors are generated through bridge law mapping; (3) Links: GHMI links are classified into six categories based on the roles they play in the hypertext system; (4) Navigation structures: GHMI models navigation structures as structured composites. These structures include bookmarks, network overviews and four categories of guided-tours; (5) Virtual and computed components: In GHMI, all components mapped from COIS bridge laws are computed components and also virtual components. They are not stored in the HTE Linkbase. Every time they are required by users, the HTE applies bridge laws to dynamically generate them; (6) The bridge law template: GHMI classifies bridge laws as component bridge laws and link bridge laws. GHMI provides a bridge law template and a Prolog-like simple language MRDC for defining bridge laws; (7) Session structures: GHMI models a set of session structures (i.e., the event structure and the system traversal logs) to support dynamic navigation facilities.

Both the builders of the hypertext system and the COISs benefit from the GHMI's concept of bridge laws. Bridge laws act as the *bridges* between the hypertext system and a range of heterogeneous COISs, providing the COIS builders with a comprehensive mechanism to integrate their COISs with the hypertext system. The GHMI bridge law templates are largely influenced by the bridge law notions of Bieber et al.'s work [12, 9], which has no composites. GHMI extends and formalizes the previous bridge law formats to support composites and mappings from COISs.

GHMI aims to utilize the Dexter Hypertext Reference Model [46, 47] to build its model. We shall discuss how we apply Dexter to model GHMI with proper extensions and specifications in the next chapter.

CHAPTER 4

GHMI: A DEXTER-BASED HYPERTEXT MODEL

The Dexter Hypertext Reference Model [46, 47] is a widely recognized hypertext model serving as an interchange standard for hypertext systems. Hypertext researchers addressed the usefulness and robustness of Dexter in a panel at the Hypertext'89 conference and later in research concerning the paradigm of system interchange and hypertext modeling, including DHM [38, 41, 39, 37, 40], RHYTHM [66], AHM [50], interchange between Intermedia and KMS [62], and Garzotto et al.'s model [35]. In this chapter, we aim at modeling GHMI in the terms of Dexter to make GHMI a Dexter-based model. We first present a general review of the formalized Dexter model. Then we illustrate why Dexter is not strong enough to model GHMI, and our necessary *extensions*. Finally, we present how we model GHMI using the extended Dexter model by tailoring it with GHMI *specifications*.

4.1 The Dexter Model

In Dexter, a hypertext is divided into three separate layers, namely the run-time layer, the storage layer and the within-component layer. Dexter focuses on the storage layer. In this section, we present basic elements of the Dexter storage layer model. In the next two sections, we will extend and specify Dexter to model GHMI.

The Dexter storage layer consists of a network of *components* which are information containers and interconnected by relational links. A component contains a base component (i.e., the content of the component) along with associated component information, including a set of attributes, a presentation specification (interpretable in the run-time layer) and a set of anchors pointing to portions of this component's contents. A base component is recursively defined as an atomic, a link or a sequence of other base components. The notion of composite components provides a hierarchical component structuring mechanism. The atomic component is the primitive unit in Dexter. Links are components representing relationships among components. Defining links as components enables links to be defined among links.

Dexter is formulated in Z [83], which is a formal specification language based on typed set theory. We shall only roughly follow the Z notations used in Dexter and shall describe complex concepts in words so that understanding of our description does not rely on a knowledge of Z.

A Dexter hypertext system consists of a set of components. Every component is identified uniquely by its internal UID. An *accessor* function is responsible for accessing a component given its UID. UIDs provide a direct component addressing mechanism. In some cases, though, we need to address a component through some statement (e.g., a database SQL statement), so that the UIDs are unknown. In these cases, the UID mechanism is insufficient and Dexter provides an indirect addressing mechanism which resolves a specification to a UID and then the *accessor* function is able to access the resulting component. Therefore, Dexter includes a *resolver* function which is responsible for resolving a component specification into a UID. The UID is primitive in Dexter, i.e., it is left unspecified.

$$[UID]$$

Component specification and presentation specification are also primitive in Dexter from the set $COMPONENT\_SPEC$ and the set $PRESENT\_SPEC$ (in Z, upper-case strings in a pair of brackets represent given sets, which are primitive):

$$[COMPONENT\_SPEC, PRESENT\_SPEC]$$

Anchors specify link endpoints and are defined in components. An anchor has an *ANCHOR_ID* and an *ANCHOR_VALUE* from two given sets. The anchor value represents anchor location information within a component, and is interpretable by the within-component layer. A component maintains an anchor lookup table with each entry as a pair of *ANCHOR_ID* and *ANCHOR_VALUE*. We call such an anchors *internal* anchors as they are defined explicitly in a lookup table in components. An anchor id uniquely identifies an anchor in a component.

$$[ANCHOR\_ID, ANCHOR\_VALUE]$$

$$ANCHOR == ANCHOR\_ID \times ANCHOR\_VALUE$$

The above anchor definition can be read as "an anchor is defined as an *ANCHOR_ID* and *ANCHOR_VALUE* pair." A link consists of a sequence of *specifiers*. Each link specifier contains a component specification, an anchor specification, a presentation specification and a direction.

$$DIRECTION ::= FROM \mid TO \mid BIDIRECT \mid NONE$$

```
— SPECIFIER ——————————————
componentSpec : COMPONENT_SPEC
anchorSpec : ANCHOR_ID
presentSpec : PRESENT_SPEC
direction : DIRECTION
——————————————————————————
```

Z employs the notion of a "half-box" (open to the right) to define an object (upper-case leading strings) schema. The above half-box defines an object

SPECIFIER which has four attributes (or fields, denoted as lower-case leading strings), each being defined by corresponding given set names (i.e., upper-case strings to the right of ':' which have been defined prior to this definition).

The *COMPONENT_SPEC* in a link specifier enables a link endpoint to be defined implicitly and computed dynamically. This is a powerful mechanism for constructing computed components.

Dexter requires a link to have at least two specifiers and at least one specifier with direction "TO:"

— LINK ——————————
specifiers : seq SPECIFIER

——————

$\#specifiers \geq 2$
$\exists s : ran \;\; specifiers \bullet s.direction = TO$

——————————————

Here, *seq* stands for "a sequence of." The lower part of the above half-box contains constraints specifications on object attributes. '#' stands for "number of", *ran* stands for "in range of" and '•' stands for "such that."

A Dexter component is modeled as a two-part composition: a *compBase* and a *compInfo*. A *compBase* represents a base component which is recursively defined as an atom, a link component, or a sequence of other base components. An atom is modeled by the primitive type *ATOM*,

[*ATOM*]

We use the recursive type *BASE_COMPONENT* to represent base components recursively:

```
BASE_COMPONENT   ::= atom⟨⟨ATOM⟩⟩
                   | link⟨⟨LINK⟩⟩
                   | composite⟨⟨seq BASE_COMPONENT⟩⟩
```

Components can have arbitrary associated information as attribute-value pairs from two given sets:

$[ATTRIBUTE, VALUE]$

The *compInfo* includes a set of attributes, a presentation specification and a sequence of anchors:

```
— COMP_INFO ————————————————
attributes: ATTRIBUTE ⇸ VALUE
anchors: seq ANCHOR
presentSepc: PRESENT_SPEC
————————
#anchors = #(first(| ran anchors |))
————————————————————————————
```

Symbol '⇸' indicates a function mapping an attribute (in set *ATTRIBUTE*) to its domain value (in set *VALUE*).

The schema *COMPONENT* represents a base component and associated information:

```
— COMPONENT ————————————
compBase : BASE_COMPONENT
compInfo : COMP_INFO
————————————————————————
```

A *link component* is a component with a link as its base component:

```
— LinkComp ————————————
COMPONENT
————
compBase ∈ ran link
————————————————
```

Finally, a hypertext system can be modeled by the schema *PROTO_HYPERTEXT* which has three parts: (1) a finite set of components; (2) a *resolver* function which returns the UID for a given component specification; (3) an *accessor* function which given a UID returns a component:

```
— PROTO_HYPERTEXT ————————————
components : F COMPONENT
resolver : COMPONENT_SPEC ⤔ UID
accessor : UID ⤔ COMPONENT
————————————————————
```

F stands for "a finite set."

A Dexter *HYPERTEXT* can be constructed as an instance of the schema *PROTO_HYPERTEXT* by satisfying four constraints: (1) The *accessor* function must generate a value for every component (i.e., every component must have a UID); (2) The *resolver* function must produce all possible valid UIDs (i.e., all component specifications must resolve to existing UIDs); (3) A component can not contain itself either directly or indirectly in its base component; (4) The anchor id of a component must be the same as the anchor ids in link specifiers resolving to this component.

4.2 Extensions to Dexter

Dexter emerged from modeling existing hypertext systems. As hypertext field evolves, Dexter becomes insufficient to fit all systems especially those have emerged after Dexter. We found many obstacles in modeling GHMI using Dexter. Dexter has problems with its model for composite components, anchors and links. In this section, we discuss the problems we encountered and our solutions to them as extensions to Dexter.

4.2.1 Components

Dexter has problems on the definition for composite components. A Dexter composite component contains "bare-bone" base components which are not independent components. The definition for component is recursive on *base component* rather than on *component*. This implies that base components in a composite component are *not* components. Since UIDs are only associated with components, base components have no UIDs. Base components can not be accessed by the *accessor* function. Furthermore, base components have no $COMP\_INFO$. There is no way to associate attributes to base components. Base components have no anchors or presentation specifications of their own either. When we construct a composite component taking other components as base components, all other components lose their own properties (regarding attributes, anchors and presentation specification). It is also difficult to create links among base components since they are not independent components and have no UIDs. Therefore, such a notion of composite is too restrictive.

For example, in our domain of supporting multiple COISs, we might have a composite component made up of components from different COISs (with distinct ownership properties and other COIS attributes). We also try to model the internal linking structures of composite components to facilitate navigation (e.g., create guided-tours based on the internal links of a composite). We can not effectively model these GHMI composites in terms of Dexter. We need to extend the Dexter base component definition. The following is our solution:

```
BASE_COMPONENT  ::= atom⟨⟨ATOM⟩⟩
                  | link⟨⟨LINK⟩⟩
                  | composite⟨⟨seq COMPONENT⟩⟩
```

With this improved base component definition, a component can be made up of other independent components having UIDs and properties. This supports the concept of *external* components (or "reference vs. contain"), i.e., a component's containing other independent components can be treated as "referencing" other components instead of embedding "bare-bone" base components. (Dexter's not allowing external components has been widely criticized [38, 41, 39, 37, 40, 63, 50].) This solves the problem of constructing composite components from independent components and enables modeling internal structures of composite components.

4.2.2 Anchors

Dexter defines anchors in the content of components. Link specifiers contain an *ANCHOR_ID* which must be consistent with the definition in the component embedding the anchor. Since the *COMPONENT_SPEC* in a link specifier needs to be resolved to UIDs, it may lead to different UIDs in different computations. Using an actual *ANCHOR_ID* in a specifier requires an unbearable consistency burden on hypertext systems: all possible components whose UIDs could be mapped from a given *COMPONENT_SPEC* need to have the same anchors, or at least need to use the same *ANCHOR_ID* for that link. In our environment of dynamic mapping, *COMPONENT_SPEC* is frequently used in link specifiers to allow generating link endpoints dynamically. Storing *ANCHOR_ID* in link specifiers which resolve to dynamic components would impose a heavy consistency burden. It is difficult to map the specifier's *ANCHOR_ID* to the corresponding *ANCHOR_ID* in a dynamically computed component.

As suggested by Maioli et al. [66], we modify Dexter anchor notions to replace the $ANCHOR\_ID$ in a link specifier with an anchor specification $ANCHOR\_SPEC$ which, along with $COMPONENT\_SPEC$, resolves to anchors in the link endpoint:

```
— SPECIFIER ————————————
componentSpec : COMPONENT_SPEC
anchorSpec : ANCHOR_SPEC
presentSpec : PRESENT_SPEC
direction : DIRECTION
————————————————————
```

As we mentioned before, Dexter only supports internal anchors which are defined in components. Here we define *external* anchors in link specifiers (rather than in components) using an $ANCHOR\_SPEC$. To resolve $ANCHOR\_SPEC$ to anchors, we introduce two new resolver functions: an $AIresolver$ function and an $AVresolver$ function. Given UID (resolved by the resolver function from $COMPONENT\_SPEC$), the $AIresolver$ resolves $ANCHOR\_SPEC$ to $ANCHOR\_ID$ and the $AVresolver$ resolves $ANCHOR\_SPEC$ to $ANCHOR\_VALUE$.

$$resolver : COMPONENT\_SPEC \mapsto UID$$

$$AIresolver : UID \times ANCHOR\_SPEC \mapsto ANCHOR\_ID$$

$$AVresolver : UID \times ANCHOR\_SPEC \mapsto ANCHOR\_VALUE$$

The $AIresolver$ function maps $ANCHOR\_SPEC$ to $ANCHOR\_ID$, to retain the original Dexter model of internal anchors. In the original Dexter model, links store $ANCHOR\_IDs$ and $COMP\_INFOs$ store a sequence of anchors (i.e., a lookup table of pairs of *ANCHOR_ID* and *ANCHOR_VALUE*). Given an *ANCHOR_ID*, we can easily determine the *ANCHOR_VALUE*, which is what we need to actually determine the exact location of the link end-point. The *AVresolver* function maps *ANCHOR_SPEC* to *ANCHOR_VALUE* to introduce the concept of *external anchors*, i.e., situations in which the component does not know which of its parts have been selected as link endpoints and therefore there is no way to define anchors in the component's *COMP_INFO*. Therefore, we extend the Dexter hypertext system schema *PROTO_HYPERTEXT* by adding the two anchor resolver functions:

$$\begin{array}{|l}
\text{-- PROTO\_HYPERTEXT} \text{ ————————————} \\
\text{components} : \mathbf{F}\ \text{COMPONENT} \\
resolver : COMPONENT\_SPEC \rightarrowtail UID \\
AIresolver : ANCHOR\_SPEC \rightarrowtail ANCHOR\_ID \\
AVresolver : ANCHOR\_SPEC \rightarrowtail ANCHOR\_VALUE \\
accessor : UID \rightarrowtail COMPONENT \\
\hline
\end{array}$$

With the extended Dexter, computing a link endpoint involves resolving both components and anchors at run-time. This is exactly what we need to support dynamic mapping.

4.2.3 Links

Dexter requires links to have at least two identifiers. This excludes unary links. Dexter also excludes dangling links by requiring all links to have at least one specifier with direction "TO" and *COMPONENT_SPEC* to be resolved to existing components. Dexter's restriction on these constructs has been widely criticized [39, 40, 63, 62].

In our approach of supporting COIS and hypertext integration, we also find these restrictions are too narrow. Unary links are useful for modeling COIS commands directly available as menu items. Access to bookmarks can also be modeled as a unary link with only one "TO" specifier. On the other hand, Dexter allows neither "explicit" dangling links (i.e., all links must have at least a "TO" specifier) nor "potential" dangling links (i.e., all *COMPONENT_SPECs* must resolve to existing components). Such exclusion of dangling links is too restrictive in many cases [39, 40, 63, 62]. In the environment of dynamic COIS mapping, a link endpoint could specify a computed component mapped from a COIS object (defined as a mapping rule). If the COIS object is deleted inside the COIS (which is transparent to the hypertext system), the execution of the mapping rule will result in an empty component. This causes the link pointing to the component to become "dangling." If the anchor marking a link is deleted inside the COIS, the link becomes dangling too. Therefore, we extend Dexter's link definition as follows by reducing the minimal specifier number to 1 and removing the restrictive condition on "TO" specifiers:

```
— LINK ————————————————
specifiers : seq SPECIFIER
————————
#specifiers ≥ 1
————————————————————————
```

4.3 Specifications to Dexter

Dexter is a high level abstract reference model. It aims at capturing the common features of different hypertext systems but does not specify any systems in full. To create a model for the GHMI hypertext system using the above extended Dexter model, we need to map GHMI's capabilities to Dexter. Once mapped to Dexter, GHMI becomes a Dexter-based model which proves both GHMI's and Dexter's robustness and generality. This section illustrates our specifications of all GHMI features that fall in Dexter and therefore build GHMI as a Dexter-based model.

4.3.1 Component Information

In GHMI, we specify the *ATTRIBUTE* in Dexter *COMP_INFO* to explicitly model object properties as well as other COIS-dependent attributes.

We specify link types and component classes as follows:

LINKTYPE ::=
"StructureLink" | "ReferenceLink"
| "AnnotationLink" | "AssociationLink"
| "NavigationLink" | "OperationLink"

COMPOSITE ::= "Set" | "List" | "Tree" | "Graph"

COMPCLASS ::= "PlainAtomic" | "StructuredAtomic" | COMPOSITE

GHMI objects have specific properties. We can specify Dexter's *ATTRIBUTE* to represent them. GHMI object common properties include link type, component classes and other COIS-dependent attributes as follows:

$[VALUE]$

$[OST, OSN, OAN, CN, BLS]$

— ATTRIBUTE ————————
owningSystemType : OST
owningSystemName: OSN
owningAppName: OAN
compName : CN
bridgeLawSpec : BLS
linkType : LINKTYPE
compClass : COMPCLASS
attributes : ATTRIBUTES $\mapsto$ VALUE
————————————

A GHMI link has no *compClass* attribute and a GHMI component has no *linkType* attribute.

4.3.2 UIDs

GHMI distinguishes hypertext components from COIS components according to their origins. Hypertext components are components not mapped from COIS objects. They are identified by their UIDs as system-generated integer values, called Global IDs (GIDs). Examples of hypertext components include annotation components which contain commentary information of other components, annotation links which connect components to their annotations, association links which are created manually, etc. These objects are persistent objects in the HTE Linkbase. COIS components are mapped from COIS objects by applying bridge laws. They are not persistent in the HTE Linkbase and therefore can not be identified by simple integer IDs. Instead, a COIS object is identified uniquely by a $COISOBJ$ plus ownership properties. Therefore, GHMI specifies the Dexter UID as either a GID or a $COISOBJ$ plus ownership properties:

$$[GID, COISID, COISTYPE, COISLABEL]$$

$COISOBJ == COISID \times COISTYPE \times COISLABEL$

```
UID  ::= GID
       | (COISOBJ,
       OwningSystemType, OwningSystemName, OwningAppName)
```

COISID is an object ID within a COIS application. COISLABEL defines a display label for an object. COISTYPE is an object class name in a COIS. Bridge laws are defined on entire classes of objects. COISTYPE determines which bridge law should be applied to generate an object given COISID and OwningSystemType. OwningSystemType identifies a COIS handler which works for a group of COISs with a common data model (e.g., a single DB handler for all RDBMSs). OwningSystemName identifies individual COISs (e.g., MS-Access, Oracle, Sybase, etc.). OwningAppName identifies individual applications within a COIS. The ownership information is optional as some COIS might encode these information as a part of COISIDs.

4.3.3 Components and the Accessor Function

GHMI's components are compatible to Dexter's components. Besides the above component classes specified as a Dexter *ATTRIBUTE*, we can also specify Dexter's atoms and base components to model GHMI atomics and composites.

GHMI explicitly models atomic components as either unstructured atomics (i.e., plain atomics) or structured atomics. *contentSpec* defines the content of atomic components. The *contentSpec* of a structured atomic component is a sequence of COIS-dependent *attribute-value* pairs. The *contentSpec* of a plain atomic is primitive. It is COIS-interpretable (could be some data content or reference to external data content). We obtain this by specifying Dexter's atoms:

$[PLAIN\_ATOMIC]$

$[COIS\_ATTRIBUTE, COIS\_VALUE]$

$STRUCTURED\_CONTENT == COIS\_ATTRIBUTE \times COIS\_VALUE$

```
— STRUCTURED_ATOMIC ————————————
contentSpec : seq STRUCTURED_CONTENT
————————————————————————————————————
```

$ATOM ::= PLAIN\_ATOMIC \mid STRUCTURED\_ATOMIC$

A GHMI component could be an atom (i.e., plain atomic or structured atomic), a link (the same as a Dexter link), or a composite component. The content of a GHMI composite component contains a set of non-link components (*CompSet*) and a set of link components (*LinkSet*). Recall that Dexter defines a link component as:

```
— LinkComp ——————————
COMPONENT
—————
compBase ∈ ran link
—————————————————————
```

We define a GHMI non-link component similarly:

```
— NonLinkComp ——————————
COMPONENT
—————
compBase ∉ ran link
————————————————————————
```

We therefore specify Dexter's *base_component* to represent GHMI components:

$LinkSet ::= \langle\langle \text{seq LinkComp} \rangle\rangle$

$CompSet ::= \langle\langle \text{seq NonLinkComp} \rangle\rangle$

$$\begin{array}{ll} \text{BASE\_COMPONENT} & ::= \text{atom}\langle\langle \text{ ATOM} \rangle\rangle \\ & \mid \text{link}\langle\langle \text{LINK} \rangle\rangle \\ & \mid \text{composite}\langle\langle \text{CompSet, LinkSet} \rangle\rangle, \end{array}$$

Here we define a base component as a either an *ATOM*, a *LINK*, or a composite consisting of a *LinkSet* and a *CompSet*, which are sequences of components (link components and non-link components respectively). Therefore, in fact, a GHMI base component is still a sequence of components. This is consistent with the extended Dexter base component definition. The only difference is we explicitly distinguish link components from non-link components, and this does not violate the definition consistency.

All GHMI components mapped from COIS objects are computed components. The *accessor* function is responsible for mapping a GHMI UID (i.e., COISOBJ plus ownership properties) to actual COIS object contents by applying a bridge law. The accessor function takes a UID and maps it to the associated component. In GHMI, we need to specify the functionality of the Dexter accessor function to include applying bridge laws to obtain the content of a component. In GHMI, hypertext components are static and their UIDs are explicit. Given a UID for a hypertext component, the accessor function can directly obtain the component (i.e., its content) from the HTE Linkbase without applying any bridge law. On the contrary, COIS components are mapped at run time. The UIDs for COIS components are symbols representing a COISOBJ plus ownership information (see the above §4.3.2). Given such UIDs, the content of COIS components is not directly available from the HTE Linkbase. Instead, the accessor function needs to apply a component bridge law to generate the component. Therefore, the accessor function is equivalent to GHMI's function $APPLY\_BL\_COMP(BLID, ArgumentSpec)$ which takes a component bridge law (identified by the $BLID$) and includes a given UID in its $ArgumentSpec$ to map a component:

$$accessor == APPLY\_BL\_COMP(BLID, ArgumentSpec)$$

Therefore, a GHMI component bridge law defines a mapping from a UID to a component. This is exactly what the Dexter accessor function does. A component bridge law specifies OwningSystemType for identifying COIS handlers. Other ownership properties should be also available at the time of applying a bridge law to identify COISs and applications within individual COISs. Given a COISOBJ and ownership properties, the HTE searches the HTE Knowledge Base for a bridge law matching the COISTYPE and OwningSystemType, and passes these as parameters to the accessor function. The accessor function generates a message containing the UID and the bridge law's MappingRule and sends it to the corresponding COIS handler (identified by OwningSystemType). After receiving the responses from the COIS handler, the accessor function generates the content of the component based on the COIS handler's responses and other given information. At this time, the component is ready to be instantiated by the run-time layer for display.

4.3.4 Anchors and Anchor Resolver Functions

Dexter models an anchor as an $ANCHOR\_ID$ and $ANCHOR\_VALUE$. $ANCHOR\_ID$ provides a way to reference an internal anchor through a lookup table in a component. In GHMI, however, all anchors are external anchors. Components have no lookup tables for anchors. Therefore, GHMI's anchor ID is quite different from Dexter's concept of $ANCHOR\_ID$. A GHMI anchor ID identifies an anchor by its location information. A GHMI value is the actual anchor text. Therefore, a GHMI anchor ID (i.e., anchor locations) along with a GHMI anchor value (i.e., an anchor text) maps to Dexter's $ANCHOR\_VALUE$. GHMI's anchor types can also be included in Dexter's $ANCHOR\_VALUE$. We specify Dexter's $ANCHOR\_VALUE$ to model these typed anchors:

$[GHMI\_ANCHOR\_ID, GHMI\_ANCHOR\_VALUE]$

$ANCHOR\_TYPE ::=$ "*Plain*" | "*Keyword*" | "*Dynamic*"

$ANCHOR\_VALUE == GHMI\_ANCHOR\_ID \times ANCHOR\_TYPE \times GHMI\_ANCHOR\_VALUE$

In GHMI, unlike plain anchors and keyword anchors which are created statically, dynamic anchors are resolved from $ANCHOR\_SPEC$ through link bridge laws. A link bridge law defines a Dexter link and anchor specifications. The GHMI $ANCHOR\_SPEC$ is defined as an MRDC (i.e., the Mapping Rule definition Convention which are Prolog-like logical expressions, see §3.7.3.) anchor value expression $MRDC\_ANCHOR\_VALUE$ (i.e., an $ANCHOR\_VALUE$ expression containing unresolved MRDC variables), along with a link bridge law:

$[COMP\_BRIDGE\_LAW, LINK\_BRIDGE\_LAW]$

$[MRDC\_ANCHOR\_VALUE]$

$ANCHOR\_SPEC == MRDC\_ANCHOR\_VALUE \times LINK\_BRIDGE\_LAW$

GHMI does not need the *AIresolver* function as it does not use any Dexter *ANCHOR_ID*. GHMI relies on the *AVresolver* to resolve an *ANCHOR_SPEC* to an *ANCHOR_VALUE*. A link bridge law's MappingRule gives the mapping information for a dynamic anchor. The *AVresolver* function is responsible resolve an *ANCHOR_SPEC* to explicit *ANCHOR_VALUE*. In GHMI, *ANCHOR_SPECs* are used in "FROM" specifiers of link bridge laws. After a component is mapped and displayed on screen, the *AVresolver* is invoked to map all dynamic anchors in that component. This is done by partially applying a link bridge law which resolves only *ANCHOR_SPEC* from its MappingRule. After resolving to an explicit anchor, a dynamic anchor is temporarily stored in the HTE Linkbase as if it were a static anchor and is readily accessible. Selecting a dynamic anchor would actually invoke the execution of a link bridge law to map the "TO" specifier's *COMPONENT_SPEC* (see below §4.3.5) to a component. Therefore, the *AVresolver* function is equivalent to GHMI's function $APPLY\_BL\_ANCHOR(BLID, ArgumentSpec)$ which takes a link bridge law (identified by the *BLID*) and includes a given source COISOBJ in its *ArgumentSpec* to map a dynamic anchor. Here, the *BLID* along with the *ArguementSpec* is equivalent to the above *ANCHOR_SPEC*:

$$AVresolver == APPLY\_BL\_ANCHOR(BLID, ArgumentSpec)$$

4.3.5 Links and the Resolver Function

GHMI classifies links behaviorally into six types: structure, reference, annotation, association, navigation, and operation links. As illustrated previously, this link typing feature can be represented as an attribute *LINKTYPE* (in the above *ATTRIBUTE*):

```
LINKTYPE  ::=
              "StructureLink" | "ReferenceLink"
              | "AnnotationLink" | "AssociationLink"
              | "NavigationLink" | "OperationLink"
```

GHMI links could be static or dynamic. Static links are not mapped by bridge laws (e.g., annotation links, association links). Dynamic links are mapped by bridge laws (e.g., structure links, reference links and operation links). In a link bridge law, each link specifier contains an $ANCHOR\_SPEC$ instead of an $ANCHOR\_ID$. The end point of a dynamic link is defined as a $COMPONENT\_SPEC$ and resolved dynamically by the resolver function.

The resolver function is responsible for resolving a $COMPONENT\_SPEC$ in a link bridge law to a UID. (Then the accessor function takes the UID and generates the content of the component as described above). The $COMPONENT\_SPEC$ in GHMI only considers COIS components. (Hypertext queries or search is out of scope of this thesis.) The Dexter version of GHMI $COMPONENT\_SPEC$ is an MRDC UID (i.e., a UID expression with MRDC variables) along with a link bridge law:

$$[MRDC\_UID]$$

$$COMPONENT\_SPEC == MRDC\_UID \times LINK\_BRIDGE\_LAW$$

When following a link, the "FROM" specifier's $COMPONENT\_SPEC$ has been already resolved to an explicit UID (not an $MRDC\_UID$) prior to displaying the source component (which enables this "FollowLink" command). The resolver function executes a bridge law to instantiate $COMPONENT\_SPEC$ in "TO" specifiers to UIDs. Therefore, we specify the Dexter resolver function as GHMI's special function $APPLY\_BL\_LINK(BLID, ArgumentSpec)$ which takes a link bridge law (identified by the $BLID$) and includes a given source COISOBJ in its $ArgumentSpec$ to map a link endpoint. Here the $BLID$ along with the $ArgumentSpec$ is equivalent to the above $COMPONENT\_SPEC$:

$$resolver == APPLY\_BL\_LINK(BLID, ArgumentSpec)$$

A GHMI link bridge law defines a mapping from a *COMPONENT_SPEC* to an explicit UID in a dynamic link. When such a dynamic link is selected by the user and the UID of the current component (i.e., the link's source) is given, the HTE finds a link bridge law by matching COISTYPE in the UID against those COISTYPEs in all link bridge laws' "FROM" specifiers. After finding a match, the HTE invokes the resolver function to compute the *COMPONENT_SPEC* in the "TO" specifier of the same link bridge law. The resolver function takes all given parameters along with the content of the link bridge law and sends a request to a corresponding COIS handler for resolution. Then it collects the results from the COIS handler to map the *COMPONENT_SPEC* to one or more UIDs.

The HTE then asks the IOS to display the results (as some interface mapping of the resulted UIDs). When the user selects one of these UIDs, the HTE calls the accessor function to map its content by applying a component bridge law. At this time, the HTE finds and asks the IOS to mark up all link anchors associated with this component. The HTE calls the *AVresolver* to compute dynamic anchors (as discussed in §4.3.4). The command "FollowLink" is now ready to execute again. Although all links are marked on screen, the content of link endpoints are not computed until the user actually selects to "follow" that link.

4.4 Summary

In this chapter, we have modeled GHMI in terms of Dexter and make GHMI a Dexter-based model with extensions and specifications. We first presented a general review of the formalized Dexter model. Then we illustrated why Dexter is not strong enough to model GHMI and necessary extensions. Finally, we presented how we model GHMI in terms of the extended Dexter and tailoring it with GHMI specifications.

Dexter emerged from modeling existing hypertext systems. As the hypertext field evolves, Dexter grows more and more insufficient to fit all systems, especially those have emerged after the formalization of Dexter. We found many obstacles in modeling GHMI using Dexter. Dexter has problems on its model for composite components, anchors and links. We discussed the problems we encountered and our solutions to them as extensions to Dexter: We extend the Dexter base component to be recursively defined on components (instead of on base components) to allow base components to be independent components having UIDs and properties. This enables composite components to be constructed from external components; We introduce the concept of *ANCHOR_SPEC* and replace the *ANCHOR_ID* in link specifiers by *ANCHOR_SPEC*. This enables dynamic anchors; We also introduce two new resolver functions: an *AIresolver* which resolves *ANCHOR_SPEC* to *ANCHOR_ID* and an *AVresolver* which resolves *ANCHOR_SPEC* to *ANCHOR_VALUE*. This enables dynamic anchors to be resolved to explicit *ANCHOR_IDs* and *ANCHOR_VALUEs* at run-time; We also extend Dexter links to allow dangling links and unary links.

Dexter is a high-level abstract reference model. It aims at capturing the common features of different hypertext systems but does not specify any systems in full. To create a GHMI hypertext system using the extended Dexter model, we need to map GHMI's capabilities to Dexter. Once mapped to Dexter, GHMI becomes a Dexter-based model which proves both GHMI's and Dexter's robustness and generality. We illustrated our specifications of all GHMI features that fall in Dexter and build GHMI as a Dexter-based model:

- We specify Dexter's *ATTRIBUTE* as GHMI object common properties, link types, component classes and other COIS-dependent attributes;

- We specify Dexter's UID as either a GID (i.e., Global ID for a hypertext object not mapped from COISs) or a $COISOBJ$ (i.e, $\langle ID, Type, Label \rangle$) plus ownership properties;
- We specify Dexter's $ATOM$ to obtain GHMI's atomics;
- We specify Dexter's *base* components as a *CompSet* and a *LinkSet* which are non-link components and link components, respectively;
- We specify Dexter's anchor values as GHMI's typed anchors;
- We specify Dexter's accessor function as GHMI's function $APPLY\_BL\_COMP()$ in order to utilize GHMI component bridge laws;
- We specify the extended Dexter's *AVresolver* as GHMI's function $APPLY\_BL\_ANCHOR()$ in order to utilize GHMI anchor definitions in link bridge laws;
- We specify Dexter's resolver function as GHMI's function $APPLY\_BL\_LINK()$ in order to utilize GHMI link bridge laws.

CHAPTER 5

MAPPING RELATIONAL DATABASES TO HYPERTEXT

The purpose of mapping RDBMS to hypertext is to provide a hypertext-based front-end to external heterogeneous databases managed by RDBMSs. RDBMSs usually do not support a hypertext-based navigation style for accessing information. Instead, they are based on predefined queries. This implies that the resulting applications are difficult to use or to navigate through. As a further limitation, different databases can not be accessed unless specific *ad hoc* programs are developed. We aim at combining hypertext and RDBMS technologies. In GHMI systems, the hypertext interface has its own data model and visual structure defined in the popular hypertext style, rather than the structures of its external heterogeneous databases. In this section, we present a framework for mapping RDBMS to hypertext, based on GHMI's constructs and MRDC. Applying hypertext functionality enhances the effectiveness of RDBMS for users. After identifying how GHMI could help RDBMS, we illustrate domain mapping between RDBMS and hypertext through bridge law examples.

5.1 Identifying RDBMS Objects

The hypertext representation under GHMI helps a RDBMS user establish direct access to explicit or implicit relationships among its underlying DBMS objects. We view a relational database as a composition of five types of objects (see Figure 5.1): (1) Value: a individual value in a table; (2) Record: a set of field-value pairs in a table (i.e., a tuple); (3) Field: a field name along with a sequence of values under under that field name; (4) Table: a set of records (or fields); (5) Database: a set of tables.

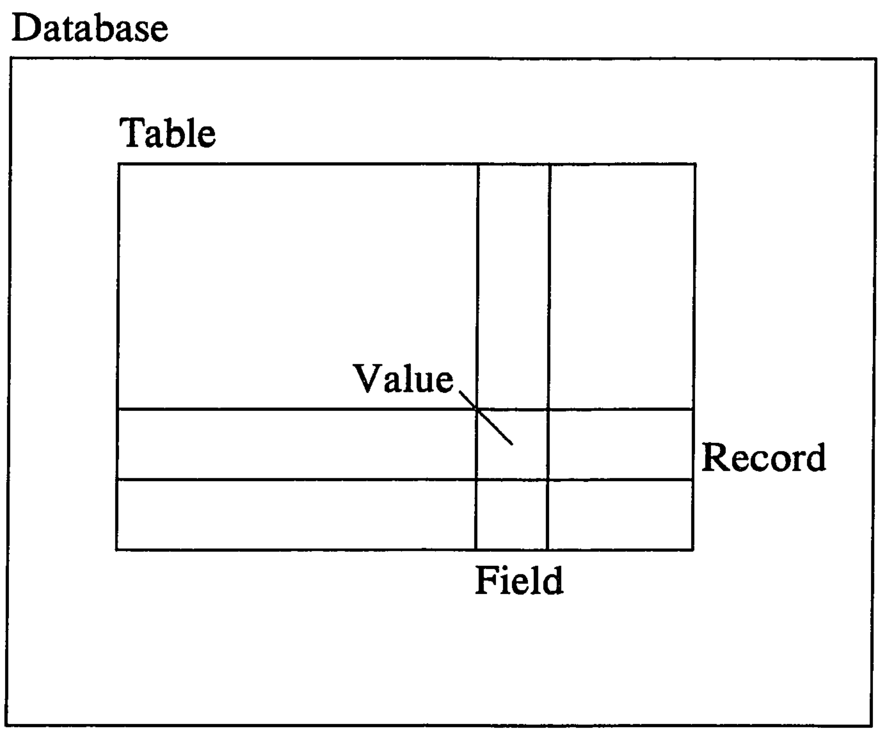


Figure 5.1 Database Objects

We can map the above objects using GHMI constructs. We map individual values to anchors, records to structured atomics, fields to structured atomics (with the same attribute for all values), tables to a *Set* of records or fields and databases to a *Set* of tables.

Figure 5.1 shows the inter-object hierarchical relationships. GHMI represents these structural relationships as structure links. We give bridge laws for mapping the RDBMS objects and structure links in §5.5. Besides these structural relationships, GHMI also helps directly access other implicit relationships as discussed in the next section.

5.2 Applying Hypertext Functionality

After mapping database objects to hypertext components, we can apply hypertext functionalities on database objects, including browsing or navigating among inter-object relationships, annotating (e.g., adding comments, bookmarks, etc), generating overviews, providing guided-tours, and supporting analysis (i.e., through connecting related inputs, computations and outputs). Links can be defined as a powerful means for directly accessing explicit and implicit inter-object relationships.

Object linking. GHMI enable the user to create and access inter-object links representing semantic relationships. Such links could be intra-database or inter-database links.

Direct access to structural relationships. GHMI helps the user directly access the structural inter-object relationships within a database shown in Figure 5.1. Once database objects are mapped to hypertext components, the user is able to access database objects by following the structure links in the details of various levels.

Direct access to schema-based relationships. The relational database model conceptually represents inter-object relationships as Entity-Relationship (ER) diagrams. When we actually implement an ER diagram within a relational database system, all information has to be mapped to independent schemata. The original ER information structure becomes *implicit.* In GHMI, we can map schemata to structured atomics (with only fields) and ER diagrams to Graphs (in which entities are components and the ER relationships are links). GHMI gives users direct access to these objects and their related implicit relationships through mapping them to reference links defined by bridge laws.

Direct access to RDBMS operations. GHMI facilitates direct access to RDBMS operations by modeling them as operation links on database objects. When users select them, GHMI does not reimplement RDBMS commands, rather it gives users direct manipulation access to them. Operation links directly access dynamic objects generated by user commands and queries. All database queries can be mapped to operation links either on particular database objects, or as menu items with no departure components and accessible from all locations (i.e., unary links). We can also define frequently-used specific queries as reference links. Once defined as an operation link, a specific query is directly executable and reusable.

Direct access to meta-information. Database objects could have two types of meta-information: annotation and system-controlled information. GHMI allows the user to manipulate annotations on objects through *annotation links.* Users can access system-controlled meta-information through operation links. Such information includes object size, field type, object description, timestamps (e.g., creation time, update time, etc.), and other object statistics—information often not directly accessible from objects.

Navigation assistance. GHMI provides RDBMS users with a variety of navigation facilities including backtracking, history list, bookmarks, network overviews and guided-tours. Most of these features are supported by navigation links. Navigation links can be defined either statically or dynamically. With a composite component, such as a table or a database, the user can simply follow the default guided-tour automatically generated by the system to explore the component's content. On the other hand, for instance, the user can select manually a small group of records in tables "Faculty" and "Student" representing a group of people involved in a project. The selected records can be connected in a guided-tour through navigation links. A guided-tour can consist of a sequence of automated queries (called a query-based guided-tour in GHMI). When such a guided-tour is followed at run-time, the queries are dynamically resolved to explicit database objects (or GHMI components mapped from them). History-based guided-tours enable the user to access session histories directly.

Analysis guidance. Hypertext can help the user control the process of a well-defined analysis procedure on databases [6]. Hypertext could guide the analyst by automatically retrieving the data needed and connecting with the appropriate analysis routines. All steps of the procedure could be annotated by the procedure builder or the analysis. Readers could select any item within the final report and get information on how it was calculated. For example, suppose that an analyst often compares the contents of two related databases DB1 and DB2 and has declared a standard procedure to assist in this process. (1) The analyst selects certain values in a database report (this report could be a table, a record, or a text file). (2) The system determines to which objects they correspond in DB1. (3) The system determines to which objects they correspond in DB2. (4) The system guides the analyst through a series of statistical analyses comparing the values from the two databases. (5) The analyst constructs a final report, in which numeric elements are highlighted as anchors. Users can select them and see the process used to calculate them. This analysis procedure can be implemented as a GHMI guided-tour. Every stop on this trail is annotatable.

5.3 The Schema DB

In this section we discuss an implementation data structure supporting schema-based relationship mapping. To take advantage of the GHMI style database (DB) schema mapping, every DB needs to a schema representation stored in a associated database called schema DB (or in the same DB with distinguished table names). Here by "primary" DB, we mean the database itself consisting of tables instantiated from its original ER diagram. By "schema DB" we mean the special DB managed by the DB handler which maintains the original ER diagram information to relate the primary DB tables to each other. Therefore, every schema DB has a primary DB associated to it. We name a schema DB using its primary DB's name plus

SYSTables

TableName	SchemaName
......	

SYSSchemata

SchemaName	SchemaType	SchemaName1	Key1	SchemaName2	Key2
......					

(a) System Tables

Entity Schemata

Key	Field1	......

Key	Field1	......

⋮

Relation Schemata

Key1	Key2

⋮

(b) Schema Tables

Figure 5.2 A Schema DB Representation

word "schema" for consistent identification and easy association. Figure 5.2 shows a general representation of a schema DB.

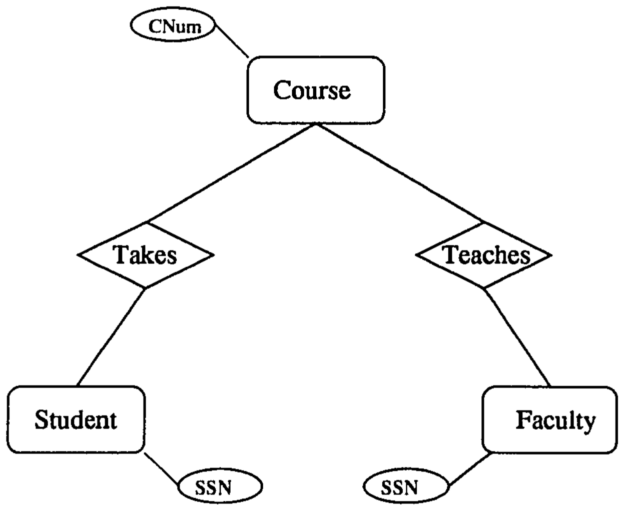
A schema DB consists of two sets of tables: the system tables (i.e., meta tables) and schema tables. (1) There are two system tables: SYSERSchemata and SYSTables. SYSERSchemata records all schemata in the schema DB derived from the ER diagram. *SchemaName* identifies each schema. *SchemaType* could be *Entity* or *Relation* representing the entities and relationships in an ER diagram respectively. To simplify discussion, we only consider binary entry relationships (other complex relationships can be decomposed into binary relationships). The other fields in *SYSERSchemata* are for the *Relation* schema only, identifying the two participant entity schema names and key field names. *SYSTables* records all tables in the primary DB and their corresponding schemata in the schema DB. (2) The schema tables of a schema DB are the actual schemata (both entities and relations) mapped from an ER diagram. A schema could have multiple instances in the primary DB (recorded in *SYSTables*).

Figure 5.3 shows a simple schema DB representation for a small DB called "SmallSchool." (a) is an ER diagram. Normally we convert it to the schemata in (b), which are instantiated to plain tables. Therefore the original ER relationships among tables are no longer directly accessible. (c) shows a corresponding schema DB which restores all ER information among tables. We can easily write a query to find the original ER relationships. In addition, based on this schema DB, we can direct access other implicit schematic relationships through bridge laws. Examples include "Find all other tables with the same schema as the current table," "Find all other tables having the same key field as the current table," etc.

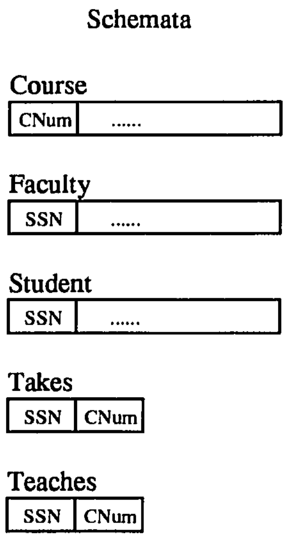
5.4 RDBMS Bridge Law Design Guidelines

The objective of designing RDBMS bridge laws is to enable direct access to RDBMS objects, relationships and meta-information through dynamically mapping them to GHMI constructs. To complete our domain modeling on RDBMS and demonstrate the power of domain mapping, we define a set of bridge laws to map RDBMS. To define bridge laws, we need to find out potential explicit or implicit relationships or objects which can be mapped by a bridge law. The following gives some guidelines for defining RDBMS bridge laws.

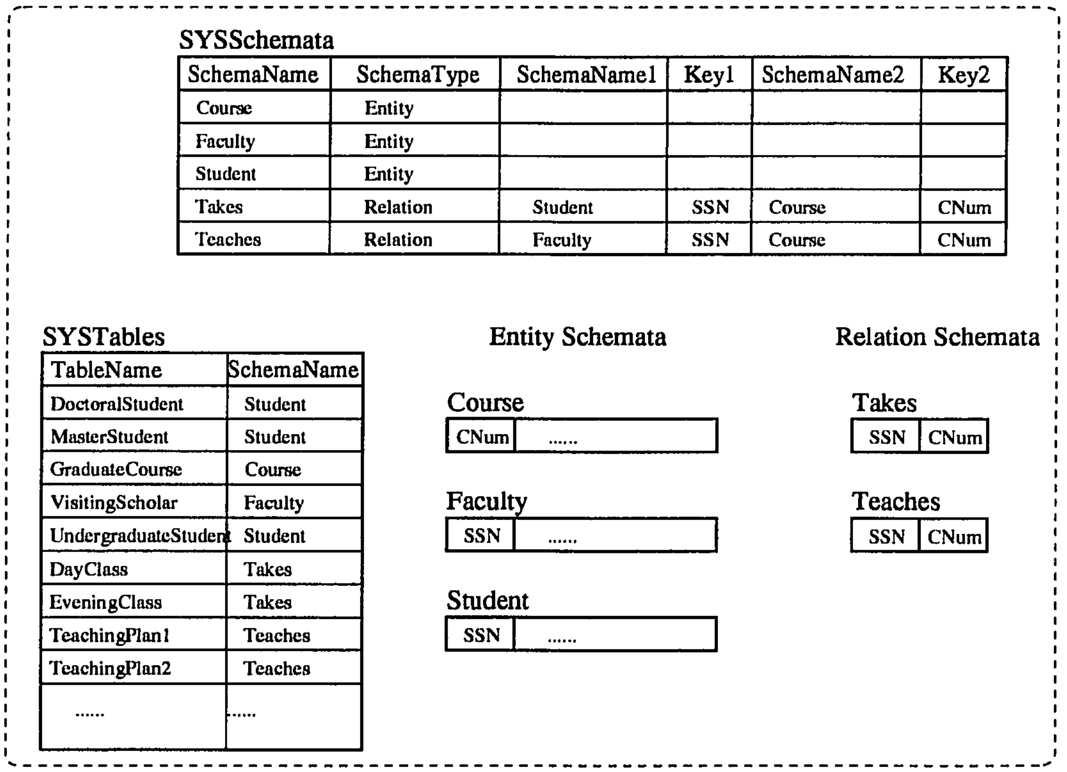
- **Object BLs.** We need BLs for the five database objects as described above (i.e., values, fields, records, tables and databases), as well as schemata and ER diagrams in the associated schema DB. Object BLs map objects' contents.
  - Map records to structured atomics



(a) An ER Diagram



(b) Schema Mapping of ER Diagram



SYSSchemata

SchemaName	SchemaType	SchemaName1	Key1	SchemaName2	Key2
Course	Entity				
Faculty	Entity				
Student	Entity				
Takes	Relation	Student	SSN	Course	CNum
Teaches	Relation	Faculty	SSN	Course	CNum

SYSTables

TableName	SchemaName
DoctoralStudent	Student
MasterStudent	Student
GraduateCourse	Course
VisitingScholar	Faculty
UndergraduateStudent	Student
DayClass	Takes
EveningClass	Takes
TeachingPlan1	Teaches
TeachingPlan2	Teaches
......	......

(c) A Schema DB

Figure 5.3 A Schema DB Example: SmallSchool-Schema

  - Map fields to structured atomics
  - Map tables to sets of records
  - Map tables to sets of fields
  - Map databases to sets of tables
  - Map schemata to structured atomics
  - Map ER diagrams to graphs

- **Structure BLs.** We need bridge laws to map objects upwards to their embedding composites. Structure BLs would include: mapping records to tables, records to a DB, fields to tables, fields to a DB and tables to DB.

- **Operation BLs.** We need BLs for SQL queries and ODBC operations. From the implementation point of view, these operations should include all operations supported by ODBC. Frequently used specific queries can also be mapped to operations links.

- **Schema-based BLs.** We store schemata as tables. Therefore, all BLs on regular tables should apply to schema DB too. Besides these schema table BLs, we need BLs to map implicit inter-object relationships implied by the schemata in the schema DB of a primary DB. We can map these relationships to reference links.

- **Meta-information BLs.** Certain users such as developers should be able to access object statistics, such as field type, field size, record size, table size (number of records in a table), DB size (number of tables in a DB), referential constraints, etc. Accessible meta-information also includes dynamic information supported by ODBC (e.g., such as "updatable"). Bridge laws help access these by defining reference links.

5.5 RDBMS Bridge Laws

In the following subsections, we present bridge laws for mapping RDBMS objects, structural relationships, operations, schematic relationships and meta-information.

5.5.1 Object Bridge Laws

1. BL_{Record}: *Mapping Records to Structured Atomic components*, as shown in Table 5.1. The GHMI HTE instantiates the bridge law for each required record mapping. We specify the component class (*CompClass*) as 'StructuredAtomic.' The *CompName* 'Record' indicates this bridge law applies to record objects. The hypertext identifier ($COISID$) is a composition of $[D, T, K]$, where symbols D, T and K are defined in the *MappingRule* part of the bridge law. In *MappingRule*, predicate *object*$([D, T, K], \text{'}Record\text{'})$ indicates D, T, K is a record object which is internally identified by its key K, embedding table T and embedding database D. The content C of record K is represented by predicate *property*$([D, T, K], \text{'}Content\text{'}, C)$. From this example, we can see that hypertext system uniquely represents the $COISID$ of a record by a combination of the embedding table's $COISID$ and the record's key value. The DB handler would find out from this mapping rule that D stands for a database object, T is a table and K is the record's key value. We represent the component set (*CompSet*) and the link set (*LinkSet*) as "NONE" because this mapped component is 'atomic' with no other components or links in its content.

2. BL_{Field}: *Mapping Fields to Structured Atomic components*, as shown in Table 5.2. This bridge law is similar to BL_{Record}. The only difference is that a field is identified by its field name instead of a key value of a record and the COISType is 'Field' instead of 'Record.'

Table 5.1 Bridge Law BL*Record*

CompClass	'StructuredAtomic'
OwningSystemType	'Database'
CompName	'Record'
COISObj	$[[D, T, K], \text{'}Record\text{'}, K]$
CompSet	'NONE'
LinkSet	'NONE'
ContentSpec	C
MappingRule	$object([D, T, K], \text{'}Record\text{'})$, $property([D, T, K], \text{'}Content\text{'}, C)$.

Table 5.2 Bridge Law BL*Field*

CompClass	'StructuredAtomic'
OwningSystemType	'Database'
CompName	'Field'
COISObj	$[[D, T, K], \text{'}Field\text{'}, K]$
CompSet	'NONE'
LinkSet	'NONE'
ContentSpec	C
MappingRule	$object([D, T, K], \text{'}Field\text{'})$, $property([D, T, K], \text{'}Content\text{'}, C)$.

Table 5.3 Bridge Law BL$Table1$

CompClass	'Set'
OwningSystemType	'Database'
CompName	'Table'
COISObj	$[[D,T],\textit{'Table'},T]$
CompSet	$\{[[D,T,K],\textit{'Record'},K]\}^*$
LinkSet	'NONE'
ContentSpec	'NONE'
MappingRule	$object([D,T,K],\textit{'Record'})$.

3. BL_{Table1}: *Mapping Tables to Set components from records*, as shown in Table 5.3. The resulting component contains a set of record components mapped from database records by the above BL_{Record}. A table is identified by its table name and the database in which it resides (i.e., $[D,T]$). The content of the resulting composite is a set of record components. $\{[D,T,K],\textit{'Record'},K]\}^*$ means 0 or more K. $object([D,T,K],\textit{'Record'})$ indicates that K is a record residing in table T of database D.

   The following instantiation of BL_{Table1} maps the content of table "Doctoral-Student:"

$$APPLY\_BL\_COMP(BL_{Table1},$$

$$[D,T] = [\textit{'SmallSchool'},\textit{'DoctoralStudent'}])$$

4. BL_{Table2}: *Mapping Tables to Set components from fields*, as shown in Table 5.4. The resulting component contains a set of field components mapped from database fields by the above BL_{Field}. This bridge law is similar to BL_{Table1} except it provides another perspective of viewing a table and an alternative way to navigate a composite's content. Here K represents a field object and 0 or more K's are mapped to the content of a table.

Table 5.4 Bridge Law BL$Table2$

CompClass	'Set'
OwningSystemType	'Database'
CompName	'Table'
COISObj	$[[D,T], \text{'}Table\text{'}, T]$
CompSet	$\{[[D,T,K], \text{'}Field\text{'}, K]\}^*$
LinkSet	'NONE'
ContentSpec	'NONE'
MappingRule	$object([D,T,K], \text{'}Field\text{'}).$

Table 5.5 Bridge Law BL$Database$

CompClass	'Set'
OwningSystemType	'Database'
CompName	'Database'
PresentationSpec	'Tree'
COISObj	$[[D], \text{'}Database\text{'}, D]$
CompSet	$\{[[D,T], \text{'}Table\text{'}, T]\}^*$
LinkSet	'NONE'
ContentSpec	'NONE'
MappingRule	$object([D,T], \text{'}Table\text{'}).$

5. $BL_{Database}$: *Mapping Databases to Set components*, as shown in Table 5.5. The resulting component is labeled by its name and contains a set of table components mapped from database tables by the above BL_{Table1} or BL_{Table2}. With this bridge law, we view a database as a composite *Set* consisting of table components. A database object is identified by its name.

6. BL_{Schema}: *Mapping Table Schemata to Structured Atomic components*, as shown in Table 5.6.

7. $BL_{ERDiagram}$: *Mapping an* ER diagrams *to Hybridgraphs*, as shown in Table 5.7.

Table 5.6 Bridge Law BL*Schema*

CompClass	'StructuredAtomic'
OwningSystemType	'Database'
CompName	'Schema'
COISObj	$[[D, T], \text{'}Schema\text{'}, T]$
CompSet	'NONE'
LinkSet	'NONE'
ContentSpec	C
MappingRule	$object([D, T], \text{'}Schema\text{'})$, $property([D, T], \text{'}Content\text{'}, C)$.

Table 5.7 Bridge Law BL*ERDiagram*

CompClass	'Graph'
OwningSystemType	'Database'
CompName	'ERDiagram'
PresentationSpec	'Graph'
COISObj	$[[D, \text{'}ERDiagram\text{'}], \text{'}ERDiagram\text{''}, \_]$
CompSet	$\{[[D, S], \text{'}Schema\text{'}, \_]\}^*$
LinkSet	$\{[[[D, S1], \text{'}Schema\text{'}, \_], \_, \text{'}From\text{'}]$, $[[[D, S2], \text{'}Schema\text{'}, \_], \_, \text{'}To\text{'}]\}^*$
ContentSpec	'NONE'
MappingRule	$object(D, \text{'}Database\text{'})$, $object([D, S1], \text{'}Schema\text{'})$, $object([D, S2], \text{'}Schema\text{'})$, $object([D, S], \text{'}Schema\text{'})$, $relation(S1, S2, \text{'}ERRelation\text{'})$.

An instantiation of this BL in the above example schema DB would be:

$$APPLY\_BL\_COMP(BL_{ERDiagram}, D = \text{`}SmallSchool\text{`})$$

The variables would be instantiated as:

D = 'Small School',

S = {'Student', 'Course', 'Faculty'}

$\{[S1, S2]\} = \{[\text{`}Student\text{`}, \text{`}Course\text{`}], [\text{`}Faculty\text{`}, \text{`}Course\text{`}]\}$

A graphical view of an ER diagram component would be similar to Figure 5.2(a).

5.5.2 Structure Link Bridge Laws

Structure links help direct access RDBMS objects through their structural relationships. We do not need a structure link to access records in a table because the records are contained as the table's content and can be accessed by applying BL_{Table1}. However, we need to access in the reverse direction: from records to tables, from tables to databases, etc. We give five structure link bridge laws, mapping access from records to their containing table, fields to their containing table, tables to their containing database, records to their containing database and fields to their containing database.

1. $BL_{RecordToTable}$: *Accessing a table from its records*, as shown in Table 5.8.

   The following instantiation of $BL_{RecordToTable}$ maps a record with key "123456789" to its containing table "MasterStudent."

$$APPLY\_BL\_LINK(BL_{RecordToTable},$$

$$[D, T, F1] = [\text{`}SmallSchool\text{`}, \text{`}MasterStudent\text{`}, \text{`}123456789\text{`}])$$

2. $BL_{FieldToTable}$: *Accessing a table from its fields*, as shown in Table 5.9.

Table 5.8 Bridge Law BL*RecordToTable*

CompClass	'Link'
OwningSystemType	'Database'
CompName	'RecordToTable'
LinkType	'StructureLink'
Specifier$_1$	
CompSpec	$[[D, T, F1], \text{'}Record\text{'}, \_]$
AnchorSpec	'NONE'
Direction	'FROM'
Specifier$_2$	
CompSpec	$[[D, T], \text{'}Table\text{'}, \_]$
AnchorSpec	'NONE'
Direction	'TO'
MappingRule	$object([D, T, F1], \text{'}Record\text{'}).$

Table 5.9 Bridge Law BL*FieldToTable*

CompClass	'Link'
OwningSystemType	'Database'
CompName	'FieldToTable'
LinkType	'StructureLink'
Specifier$_1$	
CompSpec	$[[D, T, F1], \text{'}Field\text{'}, \_]$
AnchorSpec	'NONE'
Direction	'FROM'
Specifier$_2$	
CompSpec	$[[D, T], \text{'}Table\text{'}, \_]$
AnchorSpec	'NONE'
Direction	'TO'
MappingRule	$object([D, T, F1], \text{'}Field\text{'}).$

Table 5.10 Bridge Law BL*TableToDatabase*

CompClass	'Link'
OwningSystemType	'Database'
CompName	'TableToDatabase'
LinkType	'StructureLink'
Specifier$_1$ CompSpec AnchorSpec Direction	 $[[D, F1], \text{'}Table\text{'}, \_]$ 'NONE' 'FROM'
Specifier$_2$ CompSpec AnchorSpec Direction	 $[[D], \text{'}Database\text{'}, \_]$ 'NONE' 'TO'
MappingRule	$object([D, F1], \text{'}Table\text{'})$.

3. $BL_{TableToDatabase}$: *Accessing a database from its tables*, as shown in Table 5.10.

4. $BL_{RecordToDatabase}$: *Accessing a database from its records*, as shown in Table 5.11.

5. $BL_{FieldToDatabase}$: *Accessing a database from its fields*, as shown in Table 5.12.

5.5.3 Operation Link Bridge Laws

We can map a generic query to an operation link.

- BL_{Query}: *Mapping a query to an Operation Link*, as shown in Table 5.13.

BL_{Query} in Table 5.13 maps a query represented by SQL string Q to a component. The following instantiation of BL_{Query} maps an operation link generating all student names from table "DoctoralStudent."

$$APPLY\_BL\_LINK(BL_{Query},$$

$$[F1, Q] = [\text{'}SmallSchool\text{'}, \text{'}SELECT\ Name\ FROM\ DoctoralStudent\text{'}])$$

Table 5.11 Bridge Law BL*RecordToDatabase*

CompClass	'Link'
OwningSystemType	'Database'
CompName	'RecordToDatabase'
LinkType	'StructureLink'
Specifier_1 CompSpec AnchorSpec Direction	 $[[D, T, F1], \textit{'Record'}, \_]$ 'NONE' 'FROM'
Specifier_2 CompSpec AnchorSpec Direction	 $[[D], \textit{'Database'}, \_]$ 'NONE' 'TO'
MappingRule	$\mathit{object}([D, T, F_1], \textit{'Record'}).$

Table 5.12 Bridge Law BL*FieldToDatabase*

CompClass	'Link'
OwningSystemType	'Database'
CompName	'FieldToDatabase'
LinkType	'StructureLink'
Specifier_1 CompSpec AnchorSpec Direction	 $[[D, T, F1], \textit{'Field'}, \_]$ 'NONE' 'FROM'
Specifier_2 CompSpec AnchorSpec Direction	 $[[D], \textit{'Database'}, \_]$ 'NONE' 'TO'
MappingRule	$\mathit{object}([D, T, F1], \textit{'Field'}).$

Table 5.13 Bridge Law BL$Query$

CompClass	'Link'
OwningSystemType	'Database'
CompName	'Query'
LinkType	'OperationLink'
Specifier$_1$ CompSpec AnchorSpec Direction	 $[F1, \text{'}Database\text{'}, \_]$ 'NONE' 'FROM'
Specifier$_2$ CompSpec AnchorSpec Direction	 $[[F1, F2], \text{'}Table\text{'}, \_]$ 'NONE' 'TO'
MappingRule	$object(F1, \text{'}Database\text{'})$, $property(F1, \text{'}Operation\text{'}, \text{'}Query\text{'})$, $object([F1, F2], \text{'}DynamicTable\text{'})$, $property(F2, \text{'}Content\text{'}, OPERATION(F1, \text{'}Query\text{'}, Q))$.

The result is a dynamic table holding the student names. The table does not exist in the DB prior to the execution of this bridge law. This is why we include an object predicate $object([F1, F2], \text{'}DynamicTable\text{'})$ instead of $object([F1, F2], \text{'}Table\text{'})$ (which represents a DB fact that $F2$ is an existing table) in the above $MappingRule$. All dynamic tables have a property "Content" whose value is some MRDC function specification and is evaluated upon execution of the bridge law. (We will see similar results as dynamic tables when we discuss reference links bridge laws later in this section.) The query expression Q in BL_{Query} could be arbitrarily complicated as long as it is a valid SQL statement. When the RDBMS sends back the query result (along with the COISIDs) and marks the COISType as "Table," the HTE applies BL_{Table} to map it to a Set component as if it is a regular table. These dynamically generated records are organized into a guided-tour (i.e., a query-based guided-tour) facilitating user navigation.

Besides the generic query, we can also map frequently accessed specific queries to operations links as variants of the above BL_{Query}. Consider the following specific queries on our sample DB "Small School:"

(1) Given a course, find all dayclass students taking that course.

(2) Find all professors (names) teaching undergraduate courses.

(3) Find all professors a student takes courses with.

(4) Find all courses a student taking.

(5) Find all courses taken by undergraduates.

These queries could be frequently used for access cross-table information. BLs enable such queries to be defined as "ready-to-execute" commands modeled as operation links. We take (1) as an example. Table 5.14 shows bridge law $BL_{CourseQuery}$ which maps the above query (1).

$BL_{CourseQuery}$ is actually an instantiation of BL_{Query}. The following instantiation of $BL_{CourseQuery}$ maps all students taking course "CIS610":

$$APPLY\_BL\_LINK(BL_{CourseQuery}, N = \text{`}CIS610\text{`})$$

The above instantiation is equivalent to the following instantiation of BL_{Query}:

$APPLY\_BL\_LINK(BL_{Query},$

[F1,N,Q] = ['Small School',

'CIS610',

'SELECT Name FROM

DoctoralStudent, MasterStudent,

Undergraduatestudent, DayClass WHERE

DayClass.CNum = $$ N AND

(DayClass.SSN = DoctoralStudent.SSN

OR DayClass.SSN = MasterStudent.SSN

DayClass.SSN = UndergraduateStudent.SSN)'])

Table 5.14 Bridge Law BL*CourseQuery*

CompClass	'Link'
OwningSystemType	'Database'
CompName	'CourseQuery'
LinkType	'OperationLink'
Specifier$_1$ CompSpec AnchorSpec Direction	 [[*'SmallSchool'*], *'Database'*, _] 'NONE' 'FROM'
Specifier$_2$ CompSpec AnchorSpec Direction	 [[*'SmallSchool'*, *F*2], *'Table'*, _] 'NONE' 'TO'
MappingRule	*object*(*'SmallSchool'*, *'Database'*), *property*(*'SmallSchool'*, *'Operation'*, *'Query'*), *object*([*'SmallSchool'*, *F*2], *'DynamicTable'*), *property*(*F*2, *'Content'*, *OPERATION*(*'SmallSchool'*, *'Query'*, 'SELECT Name FROM DoctoralStudent, MasterStudent, UndergraduateStudent, DayClass WHERE DayClass.CNum = $$ N AND (DayClass.SSN = DoctoralStudent.SSN OR DayClass.SSN = MasterStudent.SSN DayClass.SSN = UndergraduateStudent.SSN)')).

5.5.4 Schema-Based Bridge Laws

Bridge laws help directly access schema-related implicit relationships. Consider the following examples:

(1) Given a table, find all tables which have the same schema.

(2) Given a table, find all tables having the same key field.

(3) Given a table, find all tables which have an ER relationship with it.

(4) Given a record, find all tables whose names appear in this record.

(5) Given a non-key field, find all other tables which have it as a key field (i.e., the non-key field in a table is a foreign key field in other tables).

(6) Given a record and a non-key value, find all other records which have this value as their key value (i.e., this value is a foreign key value).

(7) Given a schema, find all tables under this schema.

(8) Given an application database, find its ER diagram.

(9) Given a record, find all records which have an ER relationship with it.

(10) Given a table, find all tables which have an indirect ER relationship with it (i.e., ER relatioship through transitivity).

(11) Find all tables which include a given table's fields.

(12) Find all tables which include all of a given table's fields except X.

We give bridge laws for (1) to (7).

1. $BL_{SameSchema}$: *Given a table, find all tables which have the same schema,* as shown in Table 5.15.

   An instantiation of this BL in the above example schema DB would be:

$$APPLY\_BL\_LINK(BL_{SameSchema},$$

$$[D, F1] = [\text{'}Small\ School\text{'}, \text{'}MasterStudent\text{'}])$$

Table 5.15 Bridge Law BL$SameSchema$

CompClass	'Link'
OwningSystemType	'Database'
CompName	'SameSchema'
LinkType	'ReferenceLink'
Specifier$_1$ CompSpec AnchorSpec Direction	 $[[D, F1], \text{'}Table\text{'}, \_]$ 'NONE' 'FROM'
Specifier$_2$ CompSpec AnchorSpec Direction	 $[[D, F2], \text{'}Table\text{'}, \_]$ 'NONE' 'TO'
MappingRule	$object([D, F1], \text{'}Table\text{'}),$ $object([D, F2], \text{'}Table\text{'}),$ $object([D, S], \text{'}Schema\text{'}),$ $relation([D, F1], S, \text{'}HasSchema\text{'}),$ $relation([D, F2], S, \text{'}HasSchema\text{'}).$

Variables would be instantiated as:

S = 'Student'

$F2$ = {'DoctoralStudent', 'MasterStudent', 'UndergraduateStudent'}

The resulted tables would be three tables "MasterStudent," "DoctoralStudent" and "UndergraduateStudent." Whenever a bridge law execution results in multiple tables, the DB handler organizes them into a dynamic table with each table name highlighted as dynamic anchors. Each dynamic anchor in a dynamic table marks a reference link $RefToTable$ (see below $BL_{RefToTable}$). The user can select on any of these anchors to access the underlying table's content.

2. $BL_{SameKey}$: *Given a table, find all tables having the same key field,* as shown in Table 5.16.

Table 5.16 Bridge Law BL*SameKey*

CompClass	'Link'
OwningSystemType	'Database'
CompName	'SameKey'
LinkType	'ReferenceLink'
Specifier$_1$ CompSpec AnchorSpec Direction	 $[[D, F1], \text{'}Table\text{'}, \_]$ 'NONE' 'FROM'
Specifier$_2$ CompSpec AnchorSpec Direction	 $[[D, F2], \text{'}Table\text{'}, \_]$ 'NONE' 'TO'
MappingRule	$object([D, F1], \text{'}Table\text{'})$, $object([D, F2], \text{'}Table\text{'})$, $property([D, F1], \text{'}KeyField\text{'}, K)$, $property([D, F2], \text{'}KeyField\text{'}, K)$.

An instantiation of $BL_{SameKey}$ in DB *SmallSchool* would be:

$$APPLY\_BL\_LINK(BL_{SameKey},$$

$$[D, F1] = [\text{'}SmallSchool\text{'}, \text{'}VisitingScholar\text{'}])$$

Variables are instantiated as:

K = 'SSN',

$F2$ = {'DoctoralStudent', 'MasterStudent', 'UndergraduateStudent'}

3. $BL_{ERRelation}$: *Given a table, find all tables which have an ER relationship with it*, as shown in Table 5.17.

An instantiation of $BL_{ERRelation}$ in the above $SmallSchool - Schema$ would be:

$$APPLY\_BL\_LINK(BL_{ERRelation},$$

$$[D, F1] = [\text{'}SmallSchool\text{'}, \text{'}GraduateCourse\text{'}])$$

Table 5.17 Bridge Law BL*ERRelation*

CompClass	'Link'
OwningSystemType	'Database'
CompName	'ERRelation'
LinkType	'ReferenceLink'
Specifier_1 CompSpec AnchorSpec Direction	 $[[D, F1], \textit{'Table'}, \_]$ 'NONE' 'FROM'
Specifier_2 CompSpec AnchorSpec Direction	 $[[D, F2], \textit{'Table'}, \_]$ 'NONE' 'TO'
MappingRule	$object([D, F1], \textit{'Table'})$, $object([D, F2], \textit{'Table'})$, $relation([D, F1], S1, \textit{'HasSchema'})$, $relation([D, F2], S2, \textit{'HasSchema'})$, $relation(S1, S2, \textit{'ERRelation'})$.

Variables are instantiated as:

$S1$ = {'Student', 'Faculty'}, $S2$ = 'Course'

$F2$ = {'DoctoralStudent', 'MasterStudent', 'VisitingScholar', 'Undergraduat-eStudent'}

The query result contains table names which are organized in records of a dynamic table. The user can access these tables by selecting on their names which are highlighted as anchors in the dynamic table indicating reference link "RefToTable," mapped by $BL_{RefToTable}$ (see below).

4. $BL_{RefToTable}$: *Given a record, find all tables whose names appear in this record,* as shown in Table 5.18.

This BL defines a dynamic anchor in the departing record. The anchor's value V happens to be a table's name in the same DB. As represented by predicate

Table 5.18 Bridge Law BL$RefToTable$

CompClass	'Link'
OwningSystemType	'Database'
CompName	'RefToTable'
LinkType	'ReferenceLink'
Specifier$_1$ CompSpec AnchorSpec Direction	 $[[D,T,K],\text{'}Record\text{'},\_]$ $[[D,T,K,F],\text{'}Dynamic\text{'},V]$ 'FROM'
Specifier$_2$ CompSpec AnchorSpec Direction	 $[[D,V],\text{'}Table\text{'},\_]$ 'NONE' 'TO'
MappingRule	$object([D,T,K],\text{'}Record\text{'}),$ $object([D,V],\text{'}Table\text{'}),$ $property([D,T,K,F],\text{'}Value\text{'},V).$

$property([D,T,K,F],\text{'}Value\text{'},V)$, this value V is identified by a key value K and a field name F in record $[D,T,K]$. Based on such an implicit relationship, this BL constructs a reference link from the record to the table marked by its table name in the record content. $BL_{RefToTable}$ is frequently used in the GHMI prototype to present a query result and other reference link destination mappings to the user. We consider a query result as a dynamic table. The user can navigate on its records via a query-based guided-tour (QGT). We can apply $BL_{RefToTable}$ to the records contained in all dynamic tables (i.e., those resulted from operation links and reference links) as well as static tables (i.e., regular tables in a DB).

5. $BL_{ForeignKeyField}$: *Given a non-key field, find all other tables which have it as a key field (i.e., the non-key field in a table is a foreign key field in other tables)*, as shown in Table 5.19.

Table 5.19 Bridge Law BL*ForeignKeyField*

CompClass	'Link'
OwningSystemType	'Database'
CompName	'ForeignKeyField'
LinkType	'ReferenceLink'
Specifier$_1$ CompSpec AnchorSpec Direction	 $[[D, T1, F1], \text{'}Field\text{'}, \_]$ 'NONE' 'FROM'
Specifier$_2$ CompSpec AnchorSpec Direction	 $[[D, T2], \text{'}Table\text{'}, \_]$ 'NONE' 'TO'
MappingRule	$object([D, T1, F1], \text{'}Field\text{'})$, $object([D, T2], \text{'}Table\text{'})$, $property([D, T1], \text{'}KeyField\text{'}, K1)$, $property([D, T2], \text{'}KeyField\text{'}, F1)$, $K1 \neq F1$.

$BL_{ForeignKeyField}$ reveals an implicit relationship "foreign key" in a database: a table's ($T1$) non-key field ($F1$) happens to be another table's ($T2$) key field.

6. $BL_{ForeignKeyValue}$: *Given a record and a non-key value, find all other records which have this value as their key value (i.e., this value is a foreign key value),* as shown in Table 5.20.

$BL_{ForeignKeyValue}$ reveals an implicit relationship regarding foreign key values: a non-key value V, identified by key value $V1$ and field name $F1$ in record $[D, T1, V1]$ (with key field $K1$), happens to be the key value of another record $[D, T2, V]$ (with key field $K2$). The non-key nature of V is indicated by $K1 \neq F1$ where $K1$ is the record's key field name and $F1$ is the value's field name. Direct selecting the anchor defined in the departure record will dynamically lead to a sequence of destination records.

Table 5.20 Bridge Law BL$ForeignKeyValue$

CompClass	'Link'
OwningSystemType	'Database'
CompName	'ForeignKeyValue'
LinkType	'ReferenceLink'
Specifier$_1$ CompSpec AnchorSpec Direction	 $[[D, T1, V1], \text{'}Record\text{'}, \_]$ $[[D, T1, V1, F1], \text{'}Dynamic\text{'}, V]$ 'FROM'
Specifier$_2$ CompSpec AnchorSpec Direction	 $[[D, T2, V], \text{'}Record\text{'}, \_]$ 'NONE' 'TO'
MappingRule	$object([D, T1, V1], \text{'}Record\text{'})$, $object([D, T2, V], \text{'}Record\text{'})$, $object([D, T1, F1], \text{'}Field\text{'})$, $property([D, T1], \text{'}KeyField\text{'}, K1)$, $property([D, T2], \text{'}KeyField\text{'}, K2)$, $property([D, T1, F1], \text{'}Value\text{'}, V)$, $property([D, T2, K2], \text{'}Value\text{'}, V)$, $K1 \neq F1$.

Table 5.21 Bridge Law BL$SchemaToTable$

CompClass	'Link'
OwningSystemType	'Database'
CompName	'SchemaToTable'
LinkType	'ReferenceLink'
Specifier$_1$ CompSpec AnchorSpec Direction	 $[[D, F1]], 'Schema', \_]$ 'NONE' 'FROM'
Specifier$_2$ CompSpec AnchorSpec Direction	 $[[D, F2], 'Table', \_]$ 'NONE' 'TO'
MappingRule	$object([D, F1], 'Schema'),$ $object([D, F2], 'Table'),$ $relation([D, F2], F1, 'HasSchema').$

7. $BL_{SchemaToTable}$: *Given a schema, find all tables under this schema*, as shown in Table 5.21.

   An instantiation of this BL in the above example schema DB would be:

$$APPLY\_BL\_LINK(BL_{SchemaToTable},$$

$$[D, F1] = ['Small\ School', F1 = 'Student'])$$

   The variables would be instantiated as:

   $F2$ = {'DoctoralStudent', 'MasterStudent','UndergraduateStudent'}

8. $BL_{DBToERDiagram}$: *Given an application database, find its ER diagram*, as shown in Table 5.22.

   An instantiation of this BL in the above example schema DB would be:

$$APPLY\_BL\_LINK(BL_{DBToDiagram}, D = 'Small\ School')$$

Table 5.22 Bridge Law BL$DBToERDiagram$

CompClass	'Link'
OwningSystemType	'Database'
CompName	'DBToERDiagram'
LinkType	'ReferenceLink'
Specifier$_1$	
CompSpec	$[D, 'Database', \_]$
AnchorSpec	'NONE'
Direction	'FROM'
Specifier$_2$	
CompSpec	$[[D, 'ERDiagram'], 'ERDiagram', \_]$
AnchorSpec	'NONE'
Direction	'TO'
MappingRule	$object(D, 'Database')$.

The execution of $BL_{DBToERDiagram}$ will invoke the execution of $BL_{ERDiagram}$ to map the destination ER diagram to a Graph.

5.5.5 Meta-information BLs

We can define bridge laws to directly access meta-information on DB objects. Such information could relate to object statistics, such as field type, field size, record size, table size (number of records in a table), DB size (number of tables in a DB), etc. This also could include dynamic information supported by ODBC (e.g., such as "updatable"). Bridge laws help access these information through mapping them to reference links. The Database handler dynamically obtains these data and puts them in a dynamic table. The following are meta-information bridge laws for DB objects: database, table, records and fields. These BLs define reference links from the objects to their metainformation. The Database handler is responsible for generating each type of meta-information when executing these bridge laws. Each of the following bridge laws returns all meta-information of an object. (To obtain specific mete-information, we can define other specific bridge laws.)

Table 5.23 Bridge Law BL*MetaRecord*

CompClass	'Link'
OwningSystemType	'Database'
CompName	'MetaRecord'
LinkType	'ReferenceLink'
Specifier$_1$ CompSpec AnchorSpec Direction	 $[[D,T,F1],\text{'}Record\text{'},\_]$ 'NONE' 'FROM'
Specifier$_2$ CompSpec AnchorSpec Direction	 $[[D,F2],\text{'}Table\text{'},\_]$ 'NONE' 'TO'
MappingRule	$object([D,T,F1],\text{'}Record\text{'})$, $object([D,F2],\text{'}DynamicTable\text{'})$, $property([D,T,F1],\text{'}Operation\text{'},\text{'}MetaInformation\text{'})$, $property([D,F2],\text{'}Content\text{'}$, $OPERATION([D,T,F1],\text{'}MetaInformation\text{'},\text{'}Record\text{'}))$.

1. $BL_{MetaRecord}$: *Mapping Record meta-information*, as shown in Table 5.23. Record meta-information examples include the record key field name, key value, number of values, time stamps (creating, updating, accessing times), attributes, non-key field name, etc.

2. $BL_{MetaField}$: *Mapping Field meta-information*, as shown in Table 5.24. Field meta-information examples include field type, field size, etc.

3. $BL_{MetaTable}$: *Mapping Table meta-information*, as shown in Table 5.25. Table meta-information examples include number of records, number of fields, key field name, referential constraints, timestamps, etc.

4. $BL_{MetaDatabase}$: *Mapping Database meta-information*, as shown in Table 5.26. Database meta-information examples include number of tables, time stamps, database handler name, access control, etc.

Table 5.24 Bridge Law BL$MetaField$

CompClass	'Link'
OwningSystemType	'Database'
CompName	'MetaField'
LinkType	'ReferenceLink'
Specifier_1 CompSpec AnchorSpec Direction	 $[[D, T, F1], \text{'}Field\text{'}, \_]$ 'NONE' 'FROM'
Specifier_2 CompSpec AnchorSpec Direction	 $[[D, F2], \text{'}Table\text{'}, \_]$ 'NONE' 'TO'
MappingRule	$object([D, T, F1], \text{'}Field\text{'})$, $object([D, F2], \text{'}DynamicTable\text{'})$, $property([D, T, F1], \text{'}Operation\text{'}, \text{'}MetaInformation\text{'})$, $property([D, F2], \text{'}Content\text{'}$, $OPERATION([D, T, F1], \text{'}MetaInformation\text{'}, \text{'}Field\text{'}))$.

Table 5.25 Bridge Law BL$MetaTable$

CompClass	'Link'
OwningSystemType	'Database'
CompName	'MetaTable'
LinkType	'ReferenceLink'
Specifier_1 CompSpec AnchorSpec Direction	 $[[D, F1], \text{'}Table\text{'}, \_]$ 'NONE' 'FROM'
Specifier_2 CompSpec AnchorSpec Direction	 $[[D, F2], \text{'}Table\text{'}, \_]$ 'NONE' 'TO'
MappingRule	$object([D, F1], \text{'}Table\text{'})$, $object([D, F2], \text{'}DynamicTable\text{'})$, $property([D, F1], \text{'}Operation\text{'}, \text{'}MetaInformation\text{'})$, $property([D, F2], \text{'}Content\text{'}$, $OPERATION([D, F1], \text{'}MetaInformation\text{'}, \text{'}Table\text{'}))$.

Table 5.26 Bridge Law BL*MetaDatabase*

CompClass	'Link'
OwningSystemType	'Database'
CompName	'MetaDatabase'
LinkType	'ReferenceLink'
Specifier$_1$ CompSpec AnchorSpec Direction	 $[[F1], 'Database', \_]$ 'NONE' 'FROM'
Specifier$_2$ CompSpec AnchorSpec Direction	 $[[F1, F2], 'Table', \_]$ 'NONE' 'TO'
MappingRule	$object(F1, 'Database')$, $object([F1, F2], 'DynamicTable')$, $property(F1, 'Operation', 'MetaInformation')$, $property([F1, F2], 'Content'$, $OPERATION(F1, 'MetaInformation', 'Database'))$.

In this section, we presented bridge law examples for RDBMS. We do not mean to enumerate all possible BLs a RDBMS user might have. The user can add new BLs at any time.

5.6 Summary

In this chapter we demonstrated how GHMI can be used to provide an independent application (i.e., a COIS) with supplemental hypertext functionality. We presented the mapping from relational databases to hypertext, based on the GHMI model. We use RDBMS as our application domain. Applying hypertext functionality enhances the effectiveness of RDBMS for users. The hypertext representation under GHMI helps a RDBMS user establish direct access to explicit or implicit relationships among underlying RDBMS objects. Such help includes direct access to structural inter-object relationships, direct access to ER relationships, direct access to RDBMS operations, direct access to metainformation (e.g., annotation and system information such as object size, field type, object description and timestamps.) on objects selected by users, navigation assistance, and analysis guidance.

We discuss a data structure (i.e., the Schema DB) which implements schematic bridge law (BL) mapping. To complete our domain modeling on RDBMS and demonstrate the power of domain mapping, we defined a set of bridge laws to map a RDBMS, including the following categories:

- **Object BLs.** We need BLs to map the five database objects (i.e., values, fields, records, tables and databases) as well as the schemata and ER diagrams in the corresponding schema DB. Object BLs map objects' contents.
  - Map records to structured atomics
  - Map fields to structured atomics
  - Map tables to sets of records
  - Map tables to sets of fields
  - Map database to sets of tables
  - Map schemata to structured atomics
  - Map ER diagrams to graphs
- **Structure BLs.** We need bridge laws to map objects upwards to their embedding composites. Structure BLs would include: mapping record to table, record to DB, field to table, field to DB and table to DB.
- **Operation BLs.** We need BLs for SQL queries and ODBC operations. From the implementation point of view, these operations should include all operations supported by ODBC. Frequently used specific queries can also be mapped to operations links.
- **Schema-based BLs.** We store schemata as tables. Therefore, all BLs on regular tables should apply to schema DB too. Besides these schema table BLs, we BLs to map implicit inter-object relationships implied by the schemata in

the schema DB of a primary DB. We can map these relationships to reference links.

- **Meta-information BLs.** Certain users such as developers should be able to access object statistics, such as field type, field size, record size, table size (number of records in a table), DB size (number of tables in a DB), referential constraints, etc. Accessible meta-information also includes dynamic information supported by ODBC (e.g., such as "updatable"). Bridge laws help access these by defining reference links.

CHAPTER 6

IMPLEMENTATION: THE GHMI PROTOTYPE

In this chapter, we present an implementation prototype to prove the correctness and robustness of the GHMI model. We discuss the instantiated implementation architecture and its individual components.

6.1 The Implementation Architecture

Figure 6.1 shows GHMI's prototype implementation architecture as an instantiated implementation architecture of Figure 3.1 in Chapter 3. The prototype architecture comprises a hypertext engine (HTE), three COISs (MS Access, TEXPROS and MS-DOS) and an IOS (i.e., Interface-Oriented System). Every COIS or IOS connects to the HTE through its own handler. Handlers translate the HTE's messages to a format the COIS or IOS understands, and vice versa. COIS handlers translate bridge laws to access COIS' operations, objects and data. To integrate a COIS, the only change this architecture requires of the COIS is that its communications path be routed through the handlers [10]. COIS developers or builders very familiar with the COIS must write the COIS handler, as well as bridge laws for each class of objects and relationships accessible to users. RDBMS is so well-understood that we were able to do this ourselves for MS Access. The complexity of the bridge laws and COIS handlers depends on the COIS' complexity. The following subsections describe each functional component in the architecture.

6.1.1 The Hypertext Engine

The HTE consists of four databases (DBs) and six managers. The Knowledge Base stores bridge laws for mapping individual COISs to hypertext. The Linkbase contains

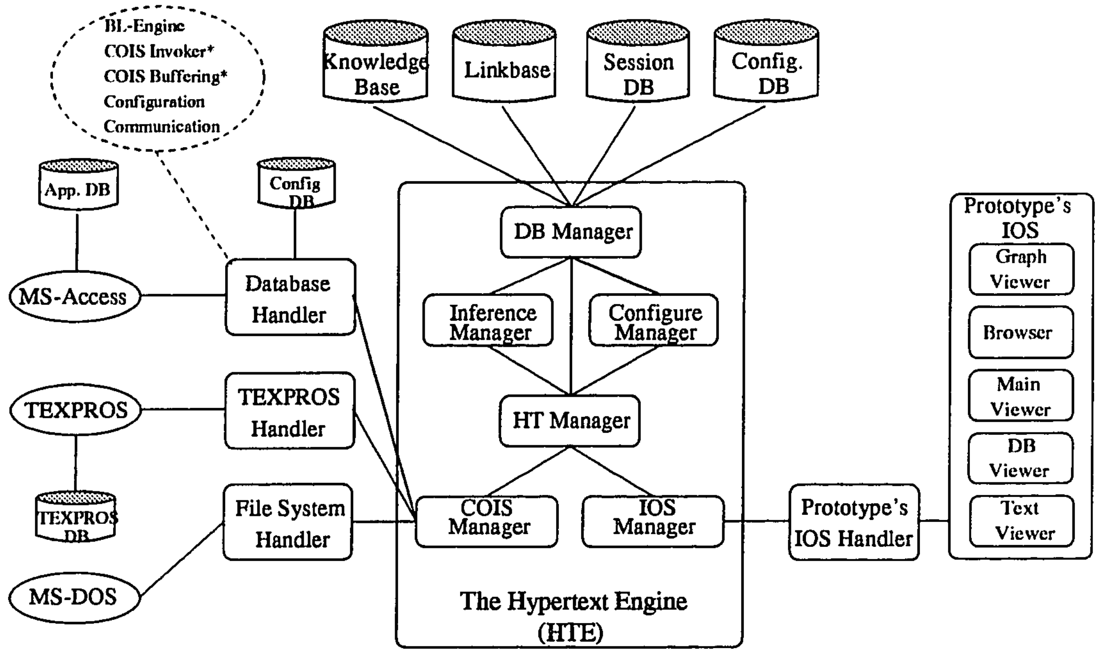


Figure 6.1 The GHMI Implementation Architecture

persistent COIS-independent data including links, anchors, annotation components, guided-tours and bookmarks. The Session DB contains navigation-related dynamic structures including the Traversal Logs (i.e., History Log, the Chronological Log and the Task Logs [13]. The Configuration DB contains COIS/IOS configuration data including handler identities and available COIS/IOS commands. The DB Manager manages manipulation of the four HTE DBs. The Inference Manager validates and invokes bridge laws. The Configuration Manager is responsible for COIS/IOS configuration and invoking the COIS handlers at run time if they are not active. The COIS Manager handles communication with COIS handlers. It encodes and decodes messages according to the communication protocol. Similarly, the IOS Manager handles communication with IOS handlers. The central part of the HTE is the HT (i.e, HyperText) Manager which manages the implementation of all GHMI hypertext functionalities.

The Knowledge Base maintains three bridge law tables: SYSBLComponents, SYSBLLinks and SYSBLSpecifiers. SYSBLComponents contains component bridge laws. SYSBLLinks and SYSBLSpecifiers together contain link bridge laws. SYSBLLinks contains link header information (e.g., including OwningSystemType, LinkType, etc). SYSBLSpecifiers contains definition and mapping rules for individual specifiers of each link in SYSBLLinks.

The Linkbase contains persistent data. Corresponding tables include SYSLinks, SYSSpecifiers, SYSAnchors, SYSComponents, SYSBookmarks and SYSGTs. In GHMI's dynamic mapping environment, all components mapped from COISs are *virtual* components. The HTE does not store their contents persistently in the Linkbase. The only persistent components are annotations, which are not mapped from COISs. The HTE stores other components in the Linkbase only when they are registered in persistent navigation structures by the user at run-time (e.g., guided-tours and bookmarks). When a component is brought to display, the HTE stores it in the traversal logs. Similarly to components, not all links are persistent. Only association links and annotation links which are hypertext-owned are persistent links. Other links become persistent only when they are embedded in the content of persistent components (e.g., guided-tours). The Linkbase has two tables for links: SYSLinks and SYSSpecifiers. All persistent links are stored in table SYSLinks. Each link entry has an ID and contains general header information in GHMI (e.g., owning system, class name, BLID, etc). The link specifiers are stored in a separate table SYSSpecifiers with each entry storing link directionality, filtering, component ID, and SYSLinks entry ID. Separating link specifiers from links ensures the implementation of n-ary links. An n-ary link can have n entries in SYSSpecifiers. Anchors are stored in SYSAnchors. Plain and keyword anchors are persistent in nature. Dynamic anchors are not persistent.

The Session DB contains session-based dynamic structures (i.e., traversal logs) consisting of three tables: SYSHistoryLog, SYSChronologicLog and SYSTaskLogs. SYSHistoryLog keeps a complete record of the user navigation history represented as event structures (see [13]). SYSChronologicLog contains a subset of entries in SYSHistoryLog which consists of components not generated from backtracking. SYSTaskLogs groups navigation history in a task-based manner to support task-based backtracking in multi-window environments [13].

The Configuration DB maintains three tables: SYSCOISs, SYSCommands and SYSIOSs. SYSCOISs contains registered COIS handler's information including name, path, registration time, status, etc. The SYSCommands contains COIS commands including command name, owning system name, parameter number and parameter types. The HTE implements these commands as operation links. SYSIOSs contains information similar to the SYSCOISs for all participating IOS handlers.

The DB Manager manages manipulations on the HTE DBs. It consists of subroutines supporting operations on database objects (databases, tables, records), including open, close, insert, edit, find, list-all, query, etc. These routines are written in Visual Basic and ODBC. Therefore, although we implement these databases on MS Access, the subroutines are essentially portable to any other RDBMS supporting ODBC, such as Oracle, Paradox, FoxPro, dBaseIII and dBaseIV. The current prototype only supports MS Access. It is easy to extend it to support others provided their proper ODBC drivers are available.

The Inference Manager validates and invokes bridge laws. There are two modules in the Inference Manager: a BL-parser and a BL-invoker. The BL-parser parses a bridge law to ensure its syntax correctness. When mapping an object, the BL-invoker finds the proper BL according to the object's COISType and OwningSystemType from SYSBLComponents and sends it to the HT Manager. The HT

Manager then sends the BL to the COIS Manager as a message for the corresponding COIS handler. The BL-invoker is also responsible for collecting BL executing results and mapping them to GHMI components by invoking other BLs (if necessary).

The Configuration Manager manages the static and dynamic system configuration. The static configuration includes the registration of COIS/IOS handlers, COISs/IOSs, COIS commands, bridge laws, etc., during the process of system setup. The dynamic configuration includes dynamic manipulation of the configuration information, message address validation, and inactive application (COIS/IOS handlers) invoking.

The COIS Manager is responsible for communicating with the COIS handlers. It has two modules: an API module and a DDE module. The API (Application Programmer Interface) routines perform standard message manipulation (e.g., set a tag, get a tag value, etc.). The DDE (Dynamic Data Exchange) routines conduct the actual inter-process communication for sending messages to and receiving messages from specific destinations through Windows95's DDE protocol.

The IOS Manager performs similarly as the COIS Manager except that it manages communication with IOS handlers instead of COIS handlers.

The HT Manager is the control center of the HTE. It implements of all GHMI hypertext functionalities (e.g., link traversal, linking, anchoring, annotating, navigating on guided-tours, backtracking, history,etc.) by managing and cooperating with other HTE managers.

6.1.2 The COISs

In this prototype, we consider three diverse COISs: a database system (MS Access), a document management system TEXPROS and a file system (MS-DOS). Our focus is on MS Access. We map objects and relationships defined by bridge laws in §5.5 and [92]. TEXPROS is still under development. We only map its objects (i.e., folders, frame templates and frame instances) and file structures to components according to bridge laws [92]. Although file systems are hardly COISs, we include them to demonstrate our system functionalities on supporting text documents and annotations. We model files as plain atomic components. Annotations could be modeled as atomic components with text contents.

6.1.3 The IOS

We only have one IOS in this prototype as our focus is on the COIS-HTE side. The current IOS consists of five viewers: a Text Viewer, a DB Viewer, a Browser, a Graph Viewer and a Main Viewer. Each viewer has its own menu items for viewer-specific commands. Some standard commands are common to all viewers, including History, Backtracking, Overview, Bookmark, GuidedTour, etc. An IOS usually has one Main Viewer and multiple other viewers simultaneously.

The Text Viewer is responsible for displaying unstructured text data. It should be able to: (1) display text content; (2) tell the starting position and length of the text selected by the user; (3) highlight a certain portion of the text based on starting position and length sent by the IOS handler. The menu items of the Text Viewer include AddAnchor, DeleteAnchor, StartLink, EndLink, DeleteLink, ShowLink, ShowBookmark, ShowGuidedTour, etc.

The DB Viewer should be able to display data in the format of a table (i.e., a spreadsheet or its simulation). It should at least: (1) display a table in a two-dimension table with a row header and a column header; (2) identify user selection on an table item (corresponding to a DB record value) by a row header (corresponding to a DB record key) and a column header (corresponding to a DB field name). This requires every row or column to have a unique header and matches the case of a DB record. The menu items for the DB Viewer are the same as the menu for the Text Viewer but their internal processing could be different. For example, the identification of a user selection is different in the Text Viewer (by a string starting position and length) and the DB Viewer (by a row header and a column header). Thus the event reports for the HTE would have different contents.

The Browser displays overview information in a tree-like structure, including: (1) Overviews of components in a tree-like structure; (2) A list of all available links on a selected object, including destinations, starting anchors, ending anchors and link types. The user should be able to select (double click on an item) or delete (click one and select a Delete menu item) an item in the list. Selecting an item triggers a link traversal; (3) The history list of a user navigation session: The user can select an item (double click) to trigger a backjump to a previous component; (4) A list of bookmarks. User commands on bookmarks include AddCurrent, Delete and GoTo. Double click on a selected item is equivalent to a GoTo command; (5) A list of guided-tours for user to navigate and manipulate.

The Graph Viewer displays graphical data in the GHMI system (e.g., images, graphical overviews of composites, etc.) which can not be displayed in the Browser. The graph viewer should be able to: (1) display images as atomic components; (1) display a composite as links and components in its LinkSet and CompSet; (2) enable selection on a component to see its content; (3) enable adding or deleting components and links; (4) enable position adjusting on components; (5) enable saving changes on component's screen positions. A typical graphical example in the current prototype is an ER diagram in a RDBMS which is mapped to a Graph component. Such a composite is displayed in the Graph Viewer as list boxes (representing entities or schemata) connected by directed lines (representing relationships). The user is able to add/delete entities (i.e., schemata) or relationships, change and save presentation positions of entities. Selecting an entity (i.e., schema) will enable the user to see all tables under this schema (mapped as a reference link).

Every time the system is activated, the user reaches the Main Viewer. When the user selects a current COIS, the system will display the overview of the COIS on the Browser. The Main viewer is responsible for: (1) displaying configuration commands and dialogues for COIS and IOS handlers; (2) inputing and editing of bridge laws; (3) setting up a current COIS by switching among registered COISs; (4) guiding the user to input the proper parameters required by a COIS command; (5) displaying dialogues for link creation; (6) displaying all other information not displayed in the other three viewers (e.g., error or warning messages). The menu commands in the Main Viewer correspond to the above functionalities.

Figure 6.2 and Figure 6.3 are example screen dumps of the GHMI prototype showing the IOS viewers.

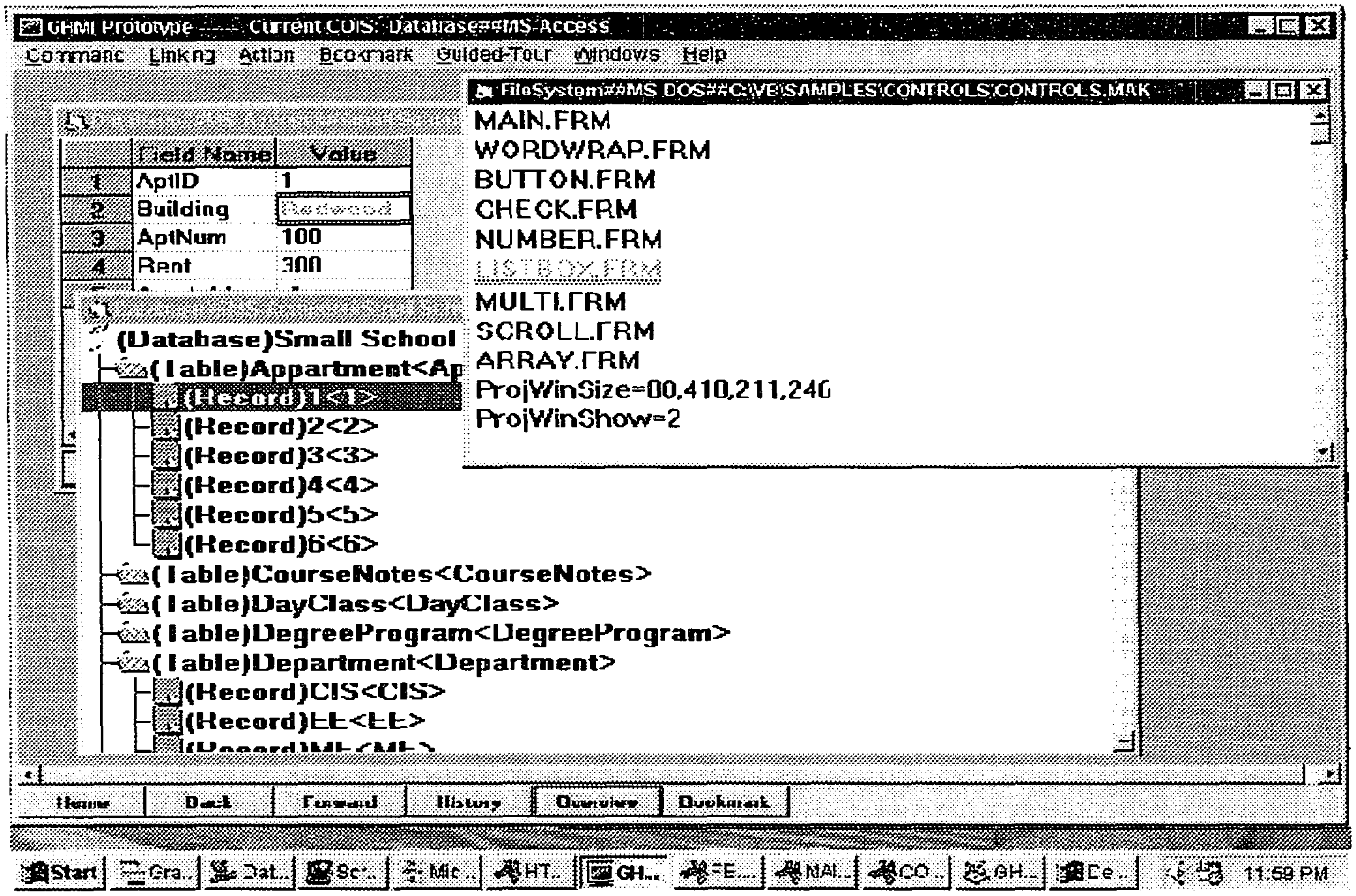
Figure 6.2 shows an example screen dump of the IOS. The screen shows four viewers, each being a separate window. The Main Viewer on the back is a root window covering all the other windows. The active window at this moment is the Text Viewer window on top of the others. It has an anchor highlighted on string "LISTBOX.FRM." The text window identifies its anchors by offsets and text lengths. The Main Viewer menu items are always for the current active window (in this case, the Text Viewer window). Directly under the text window is the Browser window which is able to show tree-like structures for component overviews. The current content of the Browser is the overview of a composite component mapped from a database object "Small School." Under the Browser is the DB Viewer window, which contains a spreadsheet and is displaying a database record. The DB Viewer displays records as a two-column spreadsheet corresponding to field name and value pairs. The current record on the DB Viewer has an anchor on text value "Redwood." The bottom part of the Main viewer contains buttons for navigation facilities, including jumping to a landmark which shuts down all child windows(button "Home"), backtracking (button "Back"), forwarding on backtracking (button "Forward"), displaying component overviews (button "Overview") and editing bookmarks (button "Bookmark").

Figure 6.3 shows another similar screen dump. This screen contains the Graph Viewer, the Main Viewer and the Browser. The graph viewer is displaying a graphic representation of the ER diagram in Figure 6.4. The small scrollable list boxes represent entities (i.e., table schemata) and the lines represent relationships.

6.1.4 The COIS Handlers

The COIS Handlers are essential components of the GHMI prototype. They actually execute bridge laws to generate responses to HTE requests. A COIS handler usually has four modules addressing four aspects of its responsibilities. (1) Executing bridge laws: This includes two submodules: a general bridge law engine (BL-Engine) and a COIS invoker. The BL-Engine translates bridge laws to executable codes (e.g., Prolog queries). The COIS invoker actually invokes COIS routines combining with the output of the BL-Engine to produce results of the bridge law execution. (2) Buffering COISs: The COIS buffering module of a COIS handler implements functions that the original COIS does not provide but are required by a bridge law execution (e.g., retrieving implicit relationships and object statistics); (3) Communicating with the HTE: The communication module is responsible for communicating with the HTE following the GHMI protocol and formats. (4) Managing the configuration of COISs: A COIS handler managing multiple COISs should maintain a configuration database and invoke inactive COISs when a bridge execution needs COISs' participation. All of the five modules written for a COIS handler can be made as APIs and reusable for other COIS handlers, except the the COIS buffering module. We have made these APIs in the current prototype.

This prototype has three COIS handers: a database (DB) handler, a TEXPROS handler and a file system handler. The TEXPROS handler handles TEXPROS bridge


GHMI Prototype ---- Current COIS: Database##MS-Access
Command Linking Action Bookmark Guided-Tour Windows Help
FileSystem##MS DOS##C:\VB\SAMPLES\CONTROLS\CONTROLS.MAK
Home
Back
Forward
History
Overview
Bookmark
Start
Gra..
Dat..
Sc..
Mic..
HT..
GH..
=E...
NAI..
CO..
GH..
Ce..
11:59 PM

	Field Name	Value
1	AptID	1
2	Building	Redwood
3	AptNum	100
4	Rent	300

```
(Database)Small School
├(Table)Appartment<Ap
│ ├(Record)1<1>
│ ├(Record)2<2>
│ ├(Record)3<3>
│ ├(Record)4<4>
│ ├(Record)5<5>
│ └(Record)6<6>
├(Table)CourseNotes<CourseNotes>
├(Table)DayClass<DayClass>
├(Table)DegreeProgram<DegreeProgram>
├(Table)Department<Department>
│ ├(Record)CIS<CIS>
│ ├(Record)EE<EE>
│ └(Record)ME<ME>
```

```
MAIN.FRM
WORDWRAP.FRM
BUTTON.FRM
CHECK.FRM
NUMBER.FRM
LISTBOX.FRM
MULTI.FRM
SCROLL.FRM
ARRAY.FRM
ProjWinSize=00,410,211,240
ProjWinShow=2
```

Figure 6.2 GHMI Prototype IOS Screen Example 1

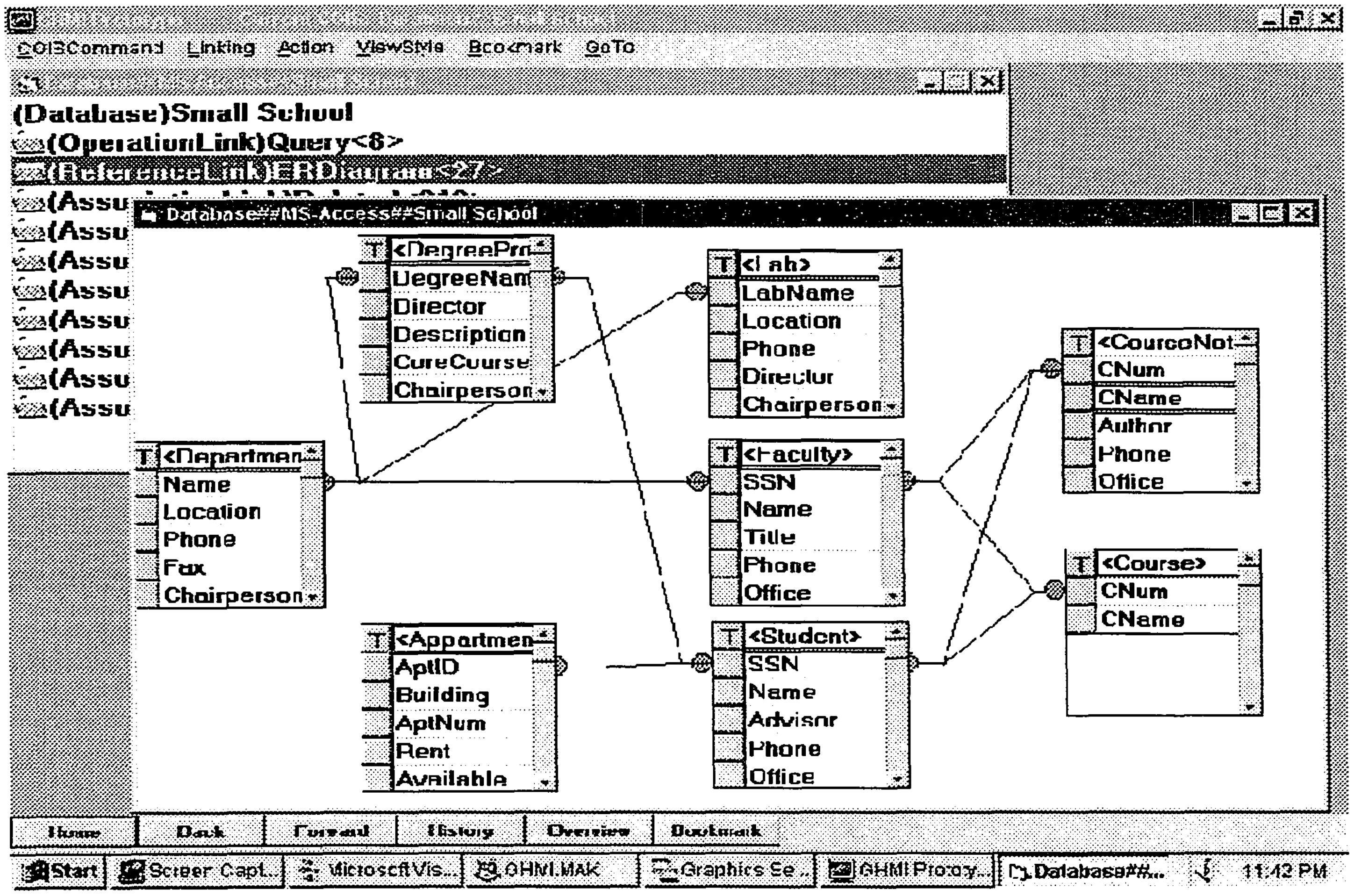


Figure 6.3 GHMI Prototype IOS Screen Example 2

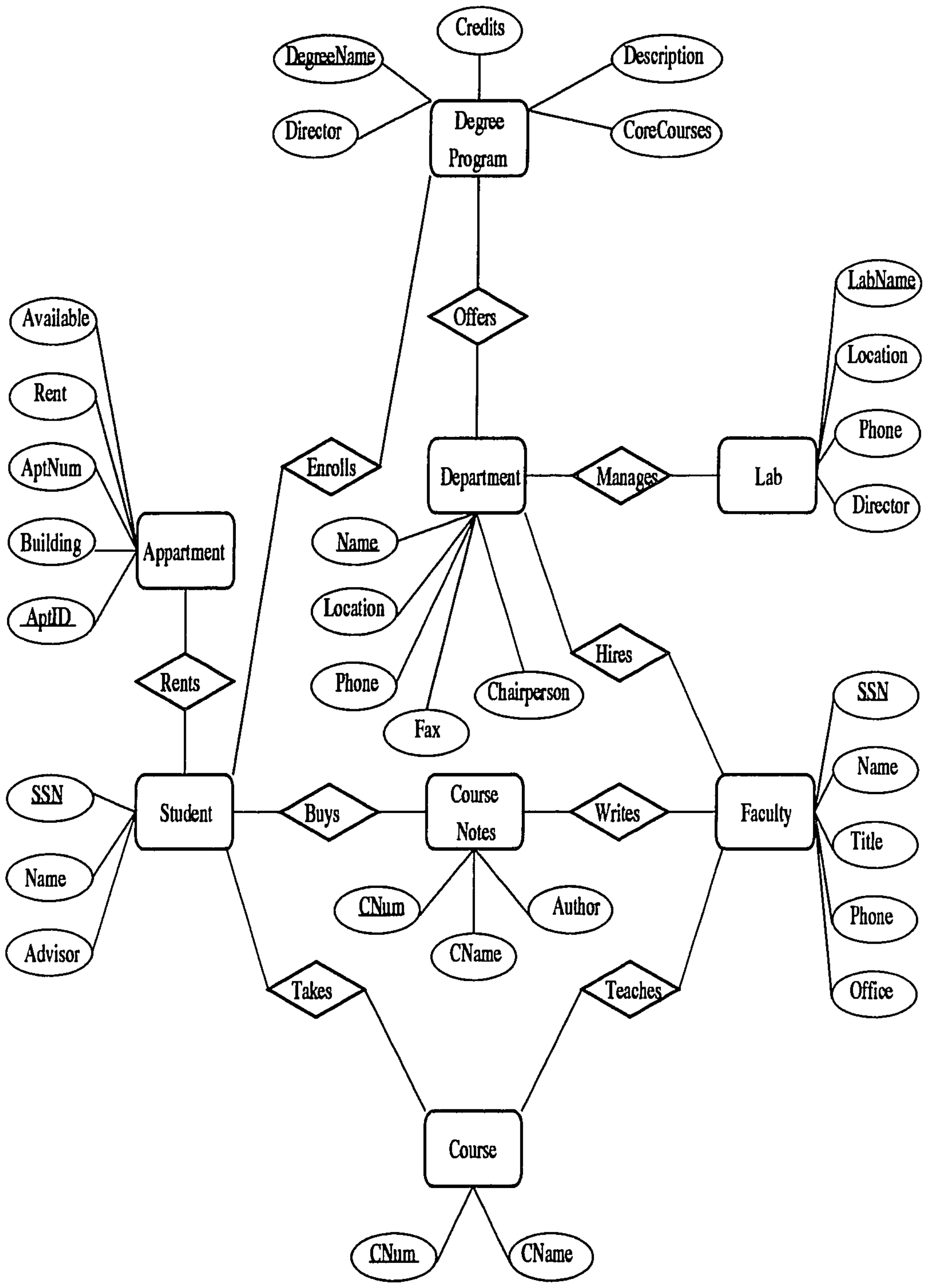


Figure 6.4 An ER Diagram Example

law mappings and communications. The file system handler directly accesses files in MS-DOS on response to a bridge laws execution. A COIS handler working for multiple COISs (e.g., the DB Handler works for all RDBMSs) maintains a configuration DB for its COISs' information.

This prototype focuses on the DB handler which employs a BL-Engine written in Prolog to execute bridge laws. After receiving a bridge law along with parameters from the HTE, the DB handler analyzes it and decides what actions it should take. This might be one of these three cases: (1) For a bridge law which requires purely object mappings (e.g., an object bridge law), the DB handler translates the bridge law into an executable Prolog predicate along with instantiated variables from the HTE parameters. Then the DB handler passes the predicate to the BL-Engine and invokes the BL-Engine to execute it. The BL-Engine executes this predicate based on predefined DB facts and a set of inference rules (managed by the COIS buffering module) and sends back the results to the DB handler. The DB handler collects the results and sends them to the HTE. (2) For a bridge law relating a COIS operation (e.g., an operation link bridge law), the DB handler's COIS buffing module either directly executes, or asks the COIS to execute this operation to generate proper responses. (3) For a bridge law which involving both of bridge law execution (by the BL-Engine) and COIS operations (e.g., a reference link bridge law with function MRDC *OPERATON*() in its *MappingRule*), the DB handler combines the results generated from the BL-Engine and the COIS buffing module and creates proper responses for the HTE.

6.1.5 The IOS Handler

This prototype only has one IOS which is written in Visual Basic. The IOS handler dispatches messages to the proper viewer based on specifications coded in the messages: command dialogues are for the Main Viewer; text messages are for the

Text Viewer; database tables, records and fields are for the DB Viewer; overviews and link browsers are for the Browser; graphical data are for the Graph Viewer.

6.2 Inter-process Communication

This section discusses the inter-process communication protocol and a communication scenario for dynamic mapping.

6.2.1 The Communication Protocol

In this prototype, the HTE, the COIS handlers and the IOS handler run as independent processes. They communicate with each other through message passing at run-time via a common message bus. Figure 6.5 shows the communication architecture. When a process (i.e., the HTE or a COIS/IOS handler) wants to send a message to another process, it prepares a message with specific destination's identity and puts the message onto the message bus. Whenever the message bus has a message, each process checks whether it is the receiver. If not, it just ignores the message. Otherwise, it handles the message and generates appropriate responses on the message bus for the sender.

We employ a communication protocol GHMICP (i.e., GHMI Communication Protocol) for interprocess communications. In GHMICP, a message consists of a sequence of tag-value pairs. Message lengths and contents vary according to message types. We classify messages into four categories based on their directions: IOS-HTE messages (from IOS handler to HTE), HTE-IOS messages (from HTE to IOS handlers), HTE-COIS messages (from HTE to COIS handler) and COIS-HTE messages (from COIS handler to HTE). (We do not consider COIS-COIS or IOS-IOS communications in this prototype.) Table 6.1 shows GHMICP messages. Column "Tag" shows the message tags a message contains and column "Sample Value" gives an example value. Column "Msg Type" indicates the message types to which this

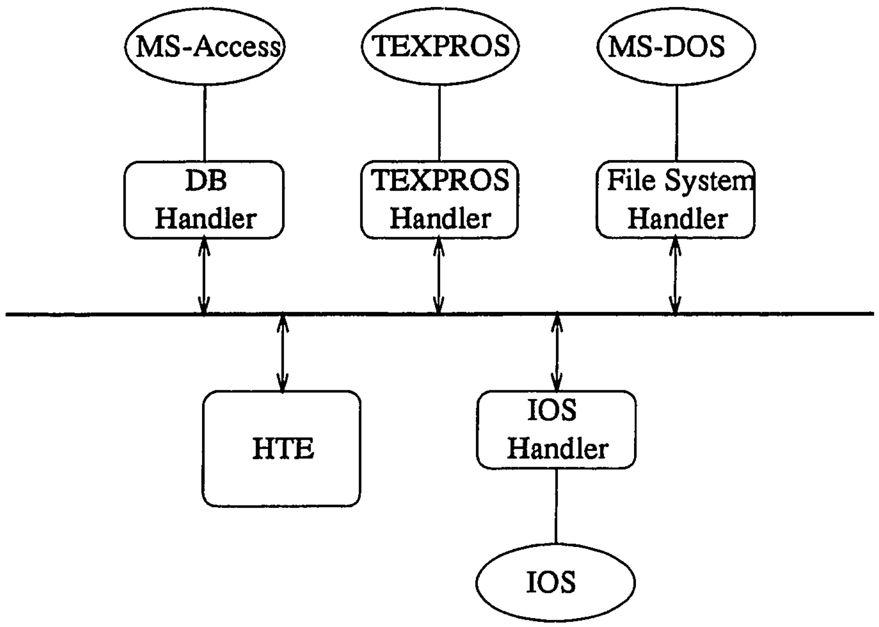


Figure 6.5 The Process Communication Architecture

corresponding tag-value pair applies. "IH" stands for IOS-HTE messages. "HI" stands for HTE-IOS messages. "HC" stands for HTE-COIS messages. "CH" stands for COIS-HTE messages. "All" indicates the tag applies to all message types.

A message consists of the following types of tags. (1) address: "Sender" and "Receiver" apply to all messages; (2) COIS identity: "SystemType" and "SystemName" identify a COIS by its type and name. "AppName" identifies the application name within a COIS. For example, In Table 6.1, "SmallSchool" is an application in "MS Access". COISs of the same type share a single COIS handler. For example, as shown in Figure 6.1, all the application DBs share a single DB handler; (3) event report: The IOS handler manages user events. Whenever a user selects an object on screen, the handler sends an event report message to the HTE. Tag "Event" indicates the event name and tag "Button" indicates the corresponding button (or menu item, depending on the IOS implementation) name. If the user does not select any button, the message will contain no "Button" tag; (4) object identity: Object entity information "COISID," "COISType" and "COISLabel" correspond to GHMI's

Table 6.1 GHMICP Messages

Tag	Sample Value	Msg Type
Sender	"DB Handler"	All
Receiver	"HTE"	
SystemType	"Database"	All
SystemName	"MS Access"	
AppName	"Small School"	
Viewer	"DBViewer"	
Event	"SelectButton"	IH
Button	"ShowLink"	
Command	"Show"	HC, HI
COISID	"SYSLinks"	All
COISType	"Table"	
COISLabel	"Student"	CH
TotalObjs	"2"	CH, HI
ObjIndex	"1"	
AnchorDataType	"DB"	HI, IH
AnchorCOISID	"SmallSchool##GraduateCourse ##SSN##123456789"	
AnchorStart	"20"	
AnchorLength	"8"	
AnchorText	"Hello"	
Content	record content	All

explicit COIS object expression (i.e., $\langle COISID, COISType, COISLabel\rangle$ identifies a COIS object). "COISID" and "COISType" apply to all messages to ensure object identity. "COISLabel" applies to COIS-HTE messages only. The HTE is responsible for defining proper display message settings (based on object's *PresentationSpec*) when an object is sent to the IOS for display; (5) message grouping: Message grouping enables a process to pass a group of messages in response to a single request (e.g., a database query may result in a set of records). "TotalObjs" indicates the total number of messages in this group while "ObjIndex" indicates the index of the current message within the message group. Message grouping tags are available to all messages. The message receiver (e.g., the HTE or handlers) is responsible for keeping track of message groups. Usually, a the message sender becomes a message receiver after sending out a message requesting responses. It will wait until all responding messages of a single group have been received; (6) anchor identity: The anchor related tags identify anchors. "AnchorDataType" indicates the anchor's data type, which could be a DB anchor or a text anchor. "AnchorStart," "Anchor-Length" and "AnchorText" apply to text anchors. Text anchors can be embedded in all COIS objects (e.g., text files, database tables or records). (We intend to extend GHMI for multi-media anchors). "AnchorCOISID" defines a DB anchor. For example, "SmallSchool##DoctoralStudent##123456789##Name" defines an anchor as a value in table "DoctoralStudent" of database "SmallSchool" with key value "123456789" and field name "Name;" (7) command: "Command" indicates an HTE command to a COIS or an IOS; (8) content: "Content" specifies command parameters or actual data associated with other tags in a message. It could be a file name, plain text, record content, SQL statement, command parameters, etc. For example, a database "Query" event can be accompanied by an "Content" as a SQL statement.

6.2.2 Dynamic Mapping: A Communication Scenario

Figure 6.6 shows a dynamic mapping and interprocess communication scenario in the GHMI prototype.

The user selects an object or an anchor on screen (at 1). The IOS handler makes an event report message to the HTE (at 3,4,5). The HTE finds out (from SYSAnchors) which object this selection stands for (at 3) and finds out all links available on this object marked by the selected anchor (at 4, 5). Static links (i.e., association links and annotation links) can be found in the Linkbase (at 4). Dynamic links (i.e., operation links, reference links and structure links) can be found in the Knowledge Base based on the object's COISType (at 5). The HTE then sends a "ShowLink" command to the IOS handler to display all available links on the Browser (at 6,7). The user can then select one of these links to follow (at 8). After receiving a user selection on the Browser, the IOS handler makes another message to report the link selection event (at 9). The HTE then finds a link BL which maps this link from the Knowledge Base (at 10, assuming that the selected link is a dynamic link). The HTE sends this BL to the COIS handler asking to map the link (at 11, 12). The COIS handler either directly obtains the requested endpoint from the COIS database or invokes the proper COIS routines to compute the link endpoint (at 13). The COIS handler makes the resulting object expressions (in terms of $\langle COISID, COISType, COISLabel\rangle$) as messages and sends them to the HTE (at 14). The HTE finds BLs for mapping the resulted COIS objects (at 15) and asks the COIS handler again to execute an object BL to generate object contents (at 16, 17). The COIS handler generates the requested object content and sends it to the HTE (at 18, 19). The HTE then maps the COIS objects to hypertext components (at 20), stores them in the Session DB (at 21), and sends them to the IOS handler for display (at 22,23 24). The HTE's next task is to find all links departing from the current component. These links could be static or dynamic. For static links found in the Linkbase, the HTE needs to find all the anchors (at 24). For dynamic links with dynamic anchors, the HTE finds link bridge laws applying to the current component and sends them to COIS handler to compute the anchors (at 25, 26, 27 28, 29). The HTE collects all the results and asks the IOS to mark up the link anchors on screen (at 30, 31). After the mark up (at 32), the system is ready for another round of dynamic mapping triggered by a user selection on screen.

6.3 Implementing GHMI Functionalities

This section discusses how this prototype implements GHMI's functionalities.

6.3.1 Components

Components mapped by bridge laws are not persistent in the Linkbase. The HTE stores their specifications (i.e., parameters and identifiers which are enough for regenerating component contents) in the SYSHistoryLog within a session. The *PresentationSpec* can be used to specify the view style of a component. For example, the user can view a Set component as either a Set view or a Tree view. A Set view expands a Set by one level (i.e., without further expanding its subsets). Using a Tree view, however, the user can see a global overview of a Set in a single view with each subset expanded upon clicking. The following paragraphs discuss components mapped from the three COISs in the prototype.

- **RDBMS Components.** For the RDBMS domain, we need bridge laws to map:
  - Values to anchors.
  - Records to structured atomics
  - Fields to structured atomics
  - Tables to sets (of records)
  - Tables to sets (of fields)

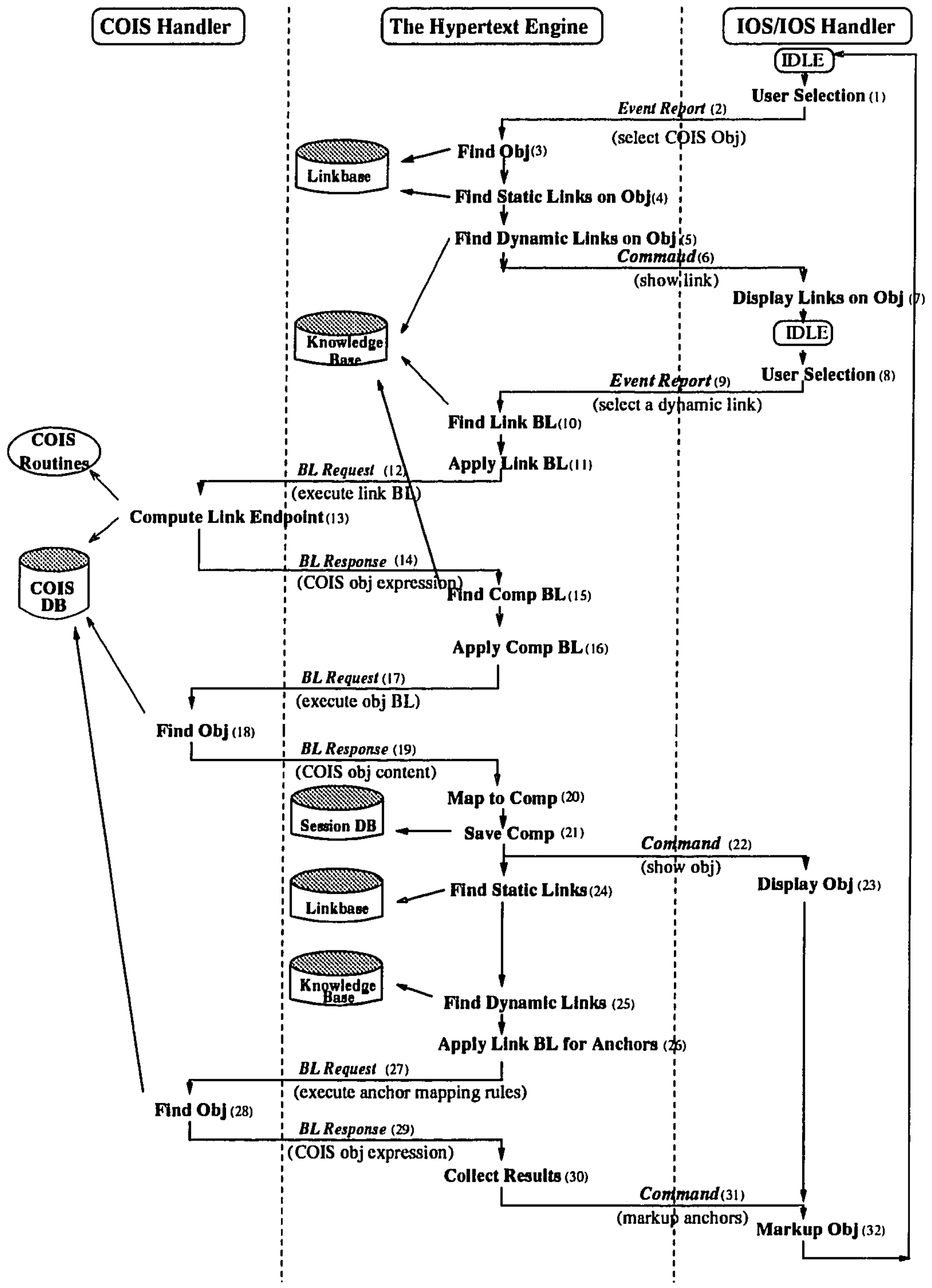


Figure 6.6 A Communication Scenario

- Database to sets (of tables)
- Schemata to structured atomics
- ER diagrams to graphs

A database or a table is displayed either as a Set view or a Tree view. GHMI bridge laws rely on object identifiers. The basic assumptions for identifying database objects are: (1) every DB has a unique name (and path) throughout the scope of a DB handler; (2) every table has a unique name within a DB; (3) every field has a unique name in a table; (4) every record has a unique key value within a table.

- **TEXPROS Components.** TEXPROS objects (see Appendix A) include folders, frame templates, frame instances and original documents. GHMI models folders as Set components, frame templates and frame instances as structured atomic components, and original documents as plain atomic components. An overview of a TEXPROS folder could be a Set view or a Tree view which is mapped at run-time. The frame templates and frame instances are stored in the TEXPROS DB which are also accessible as an application of the DB handler. Original documents are under management of MS-DOS and are therefore accessible through the file system handler too. We include TEXPROS bridge laws in Appendix A.

- **File System Components.** For the domain of file systems, we only model one type of objects: files (advanced mapping would distinguish directories from files). File bridge laws map files to plain atomics and file system overviews to Tree components. A file system is not a real COIS example since it has few computation features. We model plain files because they are basic structure of hypertext annotations in GHMI. By partially modeling file system, we can prove our concept of mapping annotations and plain atomic components which are not available in the RDBMS domain. Other advanced features provided by file managers are out of the scope of this prototype.

- **Hypertext Components.** Hypertext components are annotation components which have internal file names and therefore are treated as file objects under management of the file system. All such components are persistent objects in the Linkbase.

6.3.2 Anchors

The user should be able to define anchors on: any text of a file, the file itself, a record value, a record, a field, a table and an entire DB. Text anchors are identified by <start, length> and record value anchors are identified by their field name and key value. Static anchors include plain anchors and keyword anchors. Users can define these static anchors manually at run-time. Static anchors are persistent in the Linkbase while dynamic anchors are not. Dynamic anchors are defined by bridge laws in content of dynamic links' specifiers and computed at run-time.

6.3.3 Typed Links

GHMI supports COISs with the following six link types. Our discussion focuses on supporting RDBMSs.

– Structure links: All structure links are mapped from COISs. They are not persistent in Linkbase. In the case of DB links, all DB objects should able to reference upwards to their embedding objects through structure links. We map the following structure links (accessing objects in the reverse direction has been modeled as accessing the contents of composites by object bridge laws):

$record \rightarrow table, record \rightarrow DB, field \rightarrow table, field \rightarrow table, table \rightarrow DB.$

– Association links: Association links are hypertext-owned persistent links stored in the Linkbase. They could be inter-COIS or intra-COIS links. The user is able to define ad hoc links along with anchors across all application objects at run-time. Clicking on anchors should lead link traversal to the anchor position of destination component. These links do not need bridge laws.

– Reference links: These links are not persistent. Implicit schematic relationships we lost when mapping ER diagrams to tables could be restored through bridge law mapping as reference links. We consider examples such as direct ER relationships in the ER diagram, tables with the same schema, tables containing the same key field, etc.

– Annotation links: These are persistent links owned by hypertext. We do not need bridge laws for this kind of links. They are static and are created by the system when the user add annotations to components.

– Operation links: Operation links model hypertext operations and COIS operations. They should model all RDBMS and ODBC operations.

– Navigation links: These could be virtual links when dynamically generated on user requests. They also could be persistent links when the user explicitly requests (e.g, when being included in a UGT). These links do not need bridge laws. When the user defines a navigation structure (e.g., a guided-tour), the HTE adds navigation links for accessing this structure automatically. The HTE could generate both navigation structure and navigation links (e.g., for default guided-tours) automatically. The components in a GT could be dynamic. They are generated at run-time through bridge laws when the users actually traverse them on a GT.

6.3.4 Navigation

GHMI supports six navigation features: browsing, backtracking, history, guided-tours, bookmarks and overviews. Simple link traversal implements browsing. The Session DB structures support backtracking and history. Guided-tours, bookmarks and overviews are modeled and implemented as as composite components.

6.3.4.1 Guided-tours This subsection discusses guided-tours (GTs) and their construction algorithms. GHMI models four types of guided-tours as List components which consist of a set of components and a set of links. Each link contains a set of specifiers representing endpoints in the component set. Each component in a GT is either static or dynamic. Static components have their contents in the Linkbase. A table SYSGTs stores all static GTs (see below). GTs can be nested with arbitrary depths. A stop in a GT implies a DGT (i.e., Default Guided-Tour) if it is a composite. The DGT of a GT is itself. All component contents of dynamic GTs are dynamically computed every time the component is displayed on screen. Such computation is based on the component specification used to generate the component originally.

DGTs are dynamic and are generated automatically for composites on their structure links when the user explicitly asks to traverse this composite through DGT. DGTs are not stored in SYSGTs. The user invokes a DGT by selecting a composite and a DGT button on screen. The system then computes the DGT by applying bridge laws. NGTs (i.e., Navigation-based Guided-Tours) are static and manually specified by the user from the history list. The user only needs to select an item from the history and click an corresponding "add to NGT" menu item. Then an NGT is generated and added into table SYSGTs. Every NGT could have a user-defined semantic name for future reference. QGTs (i.e., Query-based Guided-Tours) are dynamic. Whenever the user makes a query resulting a set of objects, the system generates a QGT to hold them into a single composite component. This component is dynamic and is not stored in the Linkbase unless the user explicitly saves it as a UGT (i.e., User-defined Guided-Tour). UGTs are static. The user can add any current objects including all of the above guided-tours into a customized collection, i.e., a UGT.

The following gives algorithms for creating GTs and navigating on GTs. We assume that these standard functions have been defined: *Set_GetNextComp*(A) returns and removes a component from a set of components in A. *List_FindLast*(L) returns the component pointed by the last link in a List L. *Set_AddLink*(X, L) adds link L to a link set X. *Set_AddComp*(X, C) adds component C to a component set X.

1. Create DGTs

   - DGT_On_Set() creates a DGT on a Set component.

```
List DGT_On_Set(Component aComp)
begin
  /* Declare variables */
  Component C1;
  List GT;
  Link aLink;
  /* Get component */
  GT.LinkSet = NONE;
  C1 = Set_GetNextComp(aComp.CompSet);
  /* Add component and link to GT */
  if C1 <> NONE then
  begin
    /* Add GT's first component */
    aLink.Type = "NavigationLink";
    aLink.Content =
    $\{\langle CompSpec = C1, Direction = "to"\rangle\}$;
    Set_AddLink(GT.LinkSet, aLink);
    Set_AddComp(GT.CompSet, C1);
```

```
        C1 = Set_GetNextComp(aComp.CompSet);
        /* Add GT's other components */
        while C1 <> NONE do
        begin
          aLink.Type = "NavigationLink";
          aLink.Content =
            {$\langle CompSpec = List\_FindLast(GT), Direction = "from" \rangle$,
            $\langle CompSpec = C1, Direction = "to" \rangle$};
          Set_AddLink(GT.LinkSet, aLink);
          Set_AddComp(GT.CompSet, C1);
          C1 = Set_GetNextComp(aComp.CompSet);
        end
    end
    GT.OwningSystemType = "Hypertext";
    GT.OwningSystemName = "GHMI Prototype";
    GT.CompName = "DGT";
    return(GT);
end
```

- DGT_On_List() creates a DGT on a List component by copying the original component with changes on its properties.

```
List DGT_On_List(List Comp)
begin
    List GT;
    GT = Comp;
    for all aLink in GT.LinkSet do
        aLink.Type = "NavigationLink";
```

```
  /* Change properties */
  GT.OwningSystemType = "Hypertext";
  GT.OwningSystemName = "GHMI Prototype";
  GT.CompName = "DGT";
  return(GT);
end
```

- DGT_On_Tree() creates a DGT on a Tree component by constructing a DGT on its breadth-first search sequence. Assume that function *Graph_BreadthFirstSearch*() returns a List composite with an ordered set of components as its content.

```
List DGT_On_Set(Tree Comp)
begin
  List GT, C1;
  C1 = Graph_BreadthFirstSearch(Comp);
  GT = DGT_On_List(C1);
  return(GT);
end
```

- DGT_On_RootedDAG() creates a DGT on a RootedDAG component by constructing a DGT on its breadth-first search sequence

```
List DGT_On_Set(RootedDAG Comp)
begin
  List GT, C1;
  /* ordered output of breadth-first-search */
  C1 = Graph_BreadthFirstSearch(Comp);
  GT = DGT_On_List(C1);
```

```
        return(GT);
    end
```

2. Create NGTs

   $NGT()$ takes a GT (initially as NONE) and a user selected component (from the History Log) and adds the component into the GT.

```
List NGT(List GT, Component aComp)
begin
    Link aLink;
    If aComp <> NONE then
    begin
        aLink.Type = "NavigationLink";
        aLink.Content =
            $\{\langle CompSpec = List\_FindLast(GT), Direction = "from" \rangle,$
            $\langle CompSpec = aComp, Direction = "to" \rangle\};$
        Set_AddLink(GT.LinkSet, aLink);
        Set_AddComp(GT.CompSet, aComp);
    end
    GT.OwningSystemType = "Hypertext";
    GT.OwningSystemName = "GHMI Prototype";
    GT.CompName = "NGT";
    return(GT);
end
```

3. Create UGTs

   $UGT()$ takes a GT (initially as NONE) along with a user selected component and adds the component into the GT.

```
List UGT(List GT, Component aComp)
begin
  Link aLink;
  If aComp <> NONE then
  begin
    aLink.Type = "NavigationLink";
    aLink.Content =
      $\{\langle CompSpec = FindLast(GT), Direction = "from" \rangle,$
      $\langle CompSpec = aComp, Direction = "to" \rangle\}$;
    Set_AddLink(GT.LinkSet, aLink);
    Set_AddComp(GT.CompSet, aComp);
  end
  GT.OwningSystemType = "Hypertext";
  GT.OwningSystemName = "GHMI Prototype";
  GT.CompName = "UGT";
  return(GT);
end
```

4. Create QGTs

   QGT() constructs a GT (initially as NONE) from a set of components (from any query result). It calls *UGT*() to add component one by one and then changes GT name to QGT.

```
List QGT(ComponentSet aCompSet)
begin
  List GT;
  Component aComp;
  GT = NONE;
```

```
    While aCompSet is not empty do
        begin
            aComp = Set_GetNextComp(aCompSet);
            GT = UGT(GT, aComp);
        end
    GT.CompName = "QGT";
    return(GT);
end
```

5. Navigating on a GT

   NavigateOnGT() navigates on a GT through following the links in its *LinkSet*. We assume: *HT_ApplyBLByObj*() applies bridge laws (according to the COISType of the object) to generate the destination component. *HT_ShowComponent*(*Comp*) sends a component content to a proper IOS viewer for display; *List_FindFirst*(L) is a standard List operation which returns the first component in List L; *List_FindNext*(L, C) is a standard List operation which returns the component next to C in List L. If C is NONE, it returns the first component.

```
Navigate_On_GT(List GT)
begin
    Component aComp, NewComp;
    If GT = NONE then return
    /* Find and show first component */
    aComp = List_FindFirst(GT);
    HT_ApplyBLByObj(aComp);
    HT_ShowComponent(aComp);
    /* Find and show next component */
    aComp = List_FindNext(GT, aComp);
    while aComp <> NONE do
    begin
       HT_ApplyBLByObj(aComp);
       HT_ShowComponent(aComp);
       aComp = List_FindNext(GT, aComp);
    end
end
```

6.3.4.2 Bookmarks Bookmarks are persistent in Linkbase. GHMI models bookmarks as a special Set object. The user clicks on "bookmark" button for accessing a bookmark overview on the Browser, on which the user can perform operations including adding current, deleting a selected item and jumping to a selected item. All bookmarks are stored in a table SYSBookmarks in the Linkbase. Its entries are copied from the SYSHistoryLog in the Session DB.

6.3.4.3 Overviews Overviews are virtual and computed components modeled as Trees or Graphs. The HTE can derive object overviews from their structure links by applying bridge laws. We implement overviews for composite components as trees (on the Browser) and graphs (on the Graph Viewer) derived from their CompSet and LinkSet.

6.3.4.4 Backtracking and History The Session DB stores GHMI's run-time layer structures include SYSHistoryLog, SYSChronologicLog and SYSTaskLogs. SYSHistoryLog contains complete information regarding generated components and their instantiations in terms of event structures. SYSChronologicLog and SYSTaskLogs have the same structure as the history log.

6.4 Current Implementation Status

To completely implement the prototype proposed in this chapter is far more than a year's work for one person. Nevertheless, we were able to implement those essential parts that we believe are enough to serve as a proof of concept for GHMI. In this section, we summarize our current implementation status in terms of the GHMI prototype architecture and GHMI functionality.

The implementation architecture. The current prototype includes all components of the implementation architecture in Figure 6.1: three COIS handlers, an IOS (with five viewers), an IOS handler, six HTE managers, four HTE DBs, the DB handler's Configuration DB, an example application DB ("Small School"), and a TEXPROS DB. Although we explicitly define one application DB with ER schemata to prove our RDBMS mapping, all of the DBs in Figure 6.1 (i.e., the Knowledge Base, the Linkbase, the Session DB, the HTE Configuration DB, the DB handler's Configuration DB and the TEXPROS DB) are also treated as normal application DBs under the management of the DB handler. All functionalities GHMI adds to the application DB apply to all of these DBs too. The HTE Inference Manager is partially implemented as a BL-invoker (the missing part is the BL-parser). The HTE Configuration Manager is completed on the COIS side and is incomplete on the IOS side as we only have one IOS. The communication protocol GHMICP and its API (i.e., standard functions) are fully implemented. The IOS is fully implemented with all viewers and is able to communicate with the HTE through DDE. The DB handler is also completed (including a Prolog BL-Engine). The TEXPROS handler and the File System handler work for object mapping and generating overview trees and communicating with the HTE. The prototype also supports configuration of multiple COISs. The HTE is able to activate COISs if they are not running when a message exchange is needed.

The HTE, the IOS, the DB handler, the TEXPROS handler and the file system handler run as independent processes and communicate to each other through DDE. The prototype starts from the IOS. The IOS is responsible for activating the HTE. Upon receiving a user event, the IOS contacts the HTE for processing. The HTE then activates proper COIS handlers to generate responses.

Components. The current prototype supports mapping of these GHMI components: plain atomics (text files), structured atomics (DB records, DB fields), Set (database, tables, query results), List (guided-tours), Trees (tree-overviews) and Graph (ER diagrams). We wrote bridge laws to map these components from COIS objects.

Links. The current prototype supports creation and traversal of five GHMI link types (except annotation links): association, structure, reference, operation and navigation. Association links can be manually created as intra-COIS or inter-COIS links. We map structure, reference and operation links using bridge laws. Reference links can be mapped automatically in dynamically generated documents (e.g., query resulted tables).

Anchors. We implemented two of the three GHMI anchor types: plain anchors and dynamic anchors. Plain anchors are created manually and can be embedded in association links. Dynamic anchors are generated from link bridge laws. The anchors in a dynamically generated table are dynamic anchors. Whenever a dynamic table is resulted from an operation link or a reference link, the HTE instructs the IOS to mark it up with dynamic anchors for further access.

Navigation. The current prototype includes these navigation features: browsing (i.e., link traversal), history (accessible from all viewers), backtracking (chronological only), overview (for Sets, Trees, and Graphs), bookmarks (accessible from all viewers) and guided-tours (default guided-tours and query-based guided-tours only).

Bridge Laws. We wrote bridge laws for three COISs: RDBMS, File System and TEXPROS.

(1) RDBMS: We implemented the most of RDBMS bridge laws defined in §5.5:

- Object BLs: BL_{Record}, BL_{Table1}, $BL_{Database}$, BL_{Schema}, $BL_{ERDiagram}$
- Structure link BLs: $BL_{RecordToTable}$, $BL_{TableToDatabase}$
- Operation link BLs: BL_{Query}
- Reference link BLs: $BL_{SameSchema}$, $BL_{SameKey}$, $BL_{RefToTable}$, $BL_{ERRelation}$, $BL_{DBToERDiagram}$, $BL_{SchemaToTable}$, $BL_{MetaTable}$, $BL_{MetaDatabase}$

(2) TEXPROS: We implemented BL_{folder} and BL_{Fi} in §A. This enables us to explore the TEXPROS folders and frame instances from a hierarchical overview (viewed as a tree).

(3) File system: We have one bridge law BL_{File} for mapping text files to plain atomic components.

6.5 Summary

In this chapter, we presented the GHMI prototype implementation details. The prototype architecture comprises a hypertext engine (HTE), three COISs (MS Access, TEXPROS and MS-DOS) and an IOS. Every COIS or IOS connects to the HTE through its own handler. Handlers translate the HTE's messages to a format the COIS or IOS understands, and vice versa. COIS handlers translate bridge laws (in the HTE's MRDC format) to access COIS' operations, objects and data. To integrate a COIS, the only change this architecture requires of the COIS is that its communications path be routed through the handler [10]. Developers and builders very familiar with the COIS must write the COIS handler, as well as bridge laws for each class of objects or relationships accessible to users. RDBMS is so well-understood that we were able to do this ourselves for MS Access. The complexity of the bridge laws depends on the COIS' complexity.

The HTE consists of six managers and four databases (managed under MS Access). The managers are: the COIS Manager, the IOS Manager, the HT Manager, the Inference Manager, the Configuration Manager and the DB Manager. The databases are: the Knowledge Base, the Linkbase, the Session DB and the Configuration DB.

In this prototype, we consider three diverse COISs: a database system (MS Access), a document management system TEXPROS and a file system (MS-DOS). Our focus is on MS Access. We map objects and relationships defined by bridge laws in §5.5. TEXPROS is still under development. We only map its objects (i.e., folders, frame templates and frame instances, see §A) and file structures to components according to the bridge laws in §A. Although file systems are hardly COISs, we include them to demonstrate how GHMI supports text documents and annotations. We model text files as plain atomic components.

We only have one IOS in this prototype, as our focus is on the COIS-HTE side. The current IOS consists of five viewers: a Text Viewer, a DB Viewer, a Browser, a Graph Viewer and a Main Viewer. Each viewer has its own menu items for viewer-specific commands. Some standard commands are common to all viewers, including History, Backtracking, Overview, Bookmark, GuidedTour, etc.

The HTE and the handlers run as independent processes. They communicate with each other at run time through message passing. We employ a communication protocol GHMICP (i.e., GHMI Communication Protocol) for interprocess communications. In GHMICP, a message consists of a sequence of tag-value pairs. Message lengths and contents vary according to message types (i.e., message sender and receiver types).

Although this prototype is not completely implemented, the current implementation does include all essential parts to serve as a proof of the GHMI concepts proposed in this dissertation. We summarized our current implementation status in terms of GHMI prototype architecture and GHMI functionality. These include all components of the implementation architecture in Figure 6.1. In terms of GHMI functionality, the current prototype supports: (1) components: plain atomic components (text files), structured atomic components (from DB records, DB fields), Set components (database, tables, query results), List components (guided-tours), Tree components (tree overviews) and Graph components (graphical overviews); (2) Links: five GHMI link types (except annotation links): association, structure, reference, operation and navigation links; (3) anchors: plain anchors and dynamic anchors; (4) navigation: browsing (i.e., link traversal), history (accessible from all viewers), backtracking (chronological only), overviews (for Sets, Trees and Graphs), bookmarks (accessible from all viewers) and guided-tours (default guided-tours and query-based guided-tours); (5) bridge laws: bridge laws for mapping the above functionalities. These include most of bridge laws we defined in §5.5 and §A.

As part of our future work, we plan to continue implementing this prototype to make it a complete GHMI hypertext system.

CHAPTER 7

SUMMARY AND FUTURE WORK

In this chapter, we discuss guidelines for using GHMI, compare GHMI with other systems and models, identify both GHMI's major contributions and limitations, and briefly outline future research that could emanate from GHMI.

7.1 Guidelines: Using GHMI

In §5.4, we discussed the guidelines for integrating RDBMS with hypertext. This section discusses general COIS integration guidelines. To integrate a COIS with a GHMI hypertext system, the COIS builders (or developers) need to follow the following steps.

1. *Study the GHMI Model*

   To add the full GHMI functionalities to a COIS, it is essential for the COIS builders to have a good understanding of the GHMI model. The first step toward building an integration system is to study the GHMI hypertext concepts, including components, links, anchors, navigation features and especially the bridge law template.

2. *Identify Potential GHMI Constructs*

   The next step is to identify COIS objects, relationships, meta-information and operations (or commands), which could be mapped to GHMI constructs (i.e., component classes and typed links).

   - **Identify COIS Objects:** We need to identify all COIS objects which might be mapped to GHMI components and therefore made directly accessible to users.

- **Identify Useful Relationships:** We need to identify both explicit and implicit relationships. Mapping these relationships to GHMI links makes them directly accessible to users. GHMI enables mapping COIS relationships to appropriate link types based on their behaviors (e.g., mapping a hierarchical relationship to a *structure* link; mapping an implicit relationship to a *reference* link). Mapping implicit relationships to reference links makes them "explicit" and directly accessible.
- **Identify Meta-information:** Certain users such as developers should be able to access meta-information associated with an object, such as the object type, time stamps, referential constraints, etc. GHMI could help users access these information by mapping them to reference links.
- **Identify Useful Operations:** COISs usually provide powerful object manipulation operations (e.g., open, delete, modify, query, compute). GHMI enables direct access to these operations by mapping them to *operation* links.

3. *Software Engineering*

After identifying the potential GHMI constructs, the next step is to write the mapping rules (i.e., bridge laws) and the COIS handler code.

- **Write Bridge Laws.** We need to write all bridge laws to map the above identified potential GHMI constructs using the GHMI bridge law template. Component bridge laws map GHMI component and link bridge laws map GHMI links and dynamic anchors. It would be necessary to understand some simple syntax of Prolog as bridge laws' *MappingRule* part employs a Prolog-like syntax. The complexity of bridge laws depends on the complexity of the COIS.

- **Write the COIS Handler.** Writing the COIS handler code is the most difficult and tedious work in this integration procedure. The COIS handler actually executes bridge laws to generate responses to HTE requests. The responsibilities of a COIS handler include: (1) executing bridge laws; (2) managing the configuration of COISs; (3) calling COIS routines to generate responses for HTE requests; (4) communicating with the HTE. Bridge law execution is accomplished by a bridge law engine (probably written in some Prolog-like language because heavy inference functionality is involved in executing a bridge law's *MappingRule*). Skillful experience of some programming language(s) might be essential for this step too. Understanding the GHMI communication protocol is also important for building the communication module.

7.2 Comparison with Other Systems and Models

GHMI and its prototype share ideas and common constructs with other systems developed by hypertext researchers, especially in the field of providing hypertext functionality to third-party applications and Dexter-based modeling.

7.2.1 Open Hypertext Systems

We compare GHMI with other open hypertext systems including Sun's Link Service [75], Microcosm [23, 24, 27], SP3 [58, 63, 81], Chimera [5] and Multicard [78]. We compare GHMI with these systems and models with respect to three aspects: the application domain, the system architecture and the hypertext model.

Figure 7.1 shows the architecture and application domain comparison and Table 7.2 shows the hypertext data model comparison with open systems and models.

Sun's Link Service. GHMI shares with the Sun's Link Service [75] separating links from application data but provides more complex hypertext features. (1) Appli-

Features / Systems	Applications		Distinctive Features	Advantages	Potential Limitations
	Domain	Hypertext Aware			
GHMI-based (Wan'96)	Computation-oriented	No	- Dynamic content mapping - Application unaware integration	- Supporting integration with COISs at system level	- Bridge laws could be complicated - Dynamic mapping: slow
Sun's Link Service (Pearl'89)	Interface-oriented	Yes	- Linking by protocol library	- Flexible, least restrictive openness	- Primitive functionality - Application invokes each other
Microcosm (Fountain et al.'90, (Davis et al.'92, Hill & Hall'94)	Interface-oriented	Yes	- Extensible filter chain	- Easy extending - Easy message sharing	- Inefficient for message exchanging - Not supporting dynamic data
SP3 (Leggett et al.'91,94)	Interface-oriented	Yes	- Process-based modeling - Hyperbase support	- Easy for defining dynamic link behaviors - Easy for versioning	- Heavy performance (slow) - Inconsistent link behaviors - Applications have to store data in the hyperbase
Chimera (Anderson et al.'94)	Viewers	Yes	- View-based modeling: - ADT for communication	- Easy for defining viewer-specific link behaviors - Higher level protocol	- Inconsistent link behaviors - More changes on applications to follow ADT
Multicard (Rizk'92)	Editors	Yes	- Scripting language	- Easily extendable funcitonality	- Applications heavily burdened for writing scripts - Inconsistent link behaviors

Figure 7.1 Architecture Comparison with Open Hypertext Systems

Aspects / Models	Links			Anchors		Composites
	Typed	n-ary	Dynamic	Typed	Dynamic	
GHMI	Yes	Yes	Yes	Yes	Yes	Yes
Microcosm	No	No	No	Yes	No	No
Sun's Link Service	No	No	No	No	No	No
SP3	No	Yes	No	No	No	Yes
Chimera	No	Yes	No	No	No	Yes
Multicard	No	No	No	No	No	No

Figure 7.2 Model Comparison with Open Hypertext Systems

cation domain: The domain of the Link Service could be any application running on a Sun workstation while GHMI is designed specifically for computation-oriented applications. However, the Link Service only provides link services at a very primitive level. It only maintains link sources and destinations. GHMI provides applications with a much richer set of hypertext features. (2) Architecture: The Link Service was provided as a standard feature on Sun workstations. Therefore it is open to applications at the programming level and its built-in hypertext functionality is very primitive. The application is responsible to define the link-related operations on linked objects. (3) Hypertext model: The Link Service's hypertext model is simply plain node, link and anchors. There is no typing or composites. Links are static and only binary links. (4) Link Traversal: Link Service's applications are link-aware (i.e., applications have to manage link information) while GHMI's applications are not. In GHMI, applications are link-unaware and the hypertext engine is responsible for invoking applications.

Microcosm. GHMI and Microcosm [23, 24, 27] have many common points. They both separate links and anchors from application objects and both adopt a message-based API to establish hypertext-application communication with a similar message format (i.e., a tag-value pair format). They also share anchor concepts: Microcosm's specific anchors and local anchors are compatible with GHMI's plain anchors and keyword anchors respectively. However, GHMI differs from Microcosm in many ways. (1) Application domain: As opposed to GHMI, which aims at supporting computation-oriented applications, Microcosm is primarily open to viewers which are display-oriented applications (i.e., IOSs in GHMI's terminology). Microcosm's system architecture does not support integrating computation-oriented applications (i.e., COISs) which dynamically generate data at run-time. (2) Architecture: Microcosm applications have to be changed to embed some macros to handle communication with the link service, while GHMI's architecture requires separate handlers to handle communication and thus the applications remain unchanged. Microcosm's linear "filter" message passing chain is too restrictive and inefficient. "Filters" (program modules) have to be ordered carefully to ensure they receive all messages they expect to handle. GHMI adopts a message bus and allows modules to communicate with each other by routing through the hypertext engine. (3) Hypertext Model: Microcosm's links are static, untyped and binary. GHMI allows n-ary links and a broader range of behavioral link types. GHMI also supports dynamic links which are mapped from COIS domains at run-time through bridge laws. Microcosm's anchors are static while GHMI allows dynamic anchors to be inferred through bridge laws. Furthermore, Microcosm does not have a model for composites. (4) Link traversal: Microcosm's viewers are responsible for communicating with Microcosm. Integrating with independent viewers is still an ongoing issue. The authors proposed a mechanism to integrate hypertext-unaware viewers [24] which supports anchors through content search instead of identifying them using some underlying COIS objects IDs. In GHMI, however, COIS handlers located between the COIS and the hypertext engine handle the communication details. The COIS itself is hypertext-unaware. The hypertext model is hidden from the COISs. Such an approach enables effectively integrating existing applications with minimum changes. Changes are imposed on the handlers only.

SP3. GHMI and SP3 [58, 63, 81] both address issues regarding the Dexter model but GHMI follows a quite different approach. (1) Application domain: In SP3, there is no systematic support for computation-oriented applications which handle dynamically generated data. The application has the responsibility to extend its functionality to support dynamic data. (2) Architecture: The application needs to communicate with other hypertext components using IPC (inter-process communication). GHMI takes this burden off applications and puts it on their handlers, enabling applications to remain unchanged (except to communicate with their handlers). SP3 requires applications to store application data in order to benefit from special hypertext features such as versioning. Instead of storing application data, GHMI dynamically maps applications to hypertext. Versioning is not available in GHMI yet. (3) Hypertext model: Both GHMI and SP3 support n-ary links. In contrast to SP3's modeling links and anchors to be first-class processes, GHMI models links and anchors as objects managed by the hypertext engine. This allows links and anchors to be handled in a consistent manner. On the other hand, SP3 has no way to define anchors on links, as links are processes instead of first class components. GHMI models links as components. All operations on components also apply to links. (4) Link traversal: SP3's applications have to maintain link-related data which implies more changes would be made when integrating existing applications. GHMI's applications (COISs) are hypertext-unaware. They have no knowledge of links or anchors. Application objects are mapped to hypertext objects dynamically through bridge laws. (5) Other features: SP3 supports versioning, distribution and collaboration which are not in current stage of GHMI development.

Chimera. (1) Application domain: Chimera [5] was developed specifically for the needs of tools in software development environments. Its application domain is restricted to viewers which are display-oriented applications. There is no way in Chimera to support the domain of computation-oriented applications like GHMI does. (2) Architecture: GHMI uses a message-based API to support inter-process communication. The message format is simply ASCII tags. In contrast, Chimera hides message details by a using higher-level API and ADT (i.e., Abstract Data Type). This allows the Chimera developers to change message formats freely without affecting the rest of the system. But the tradeoff of this approach is that applications have to be changed to use the message ADT. (3) Hypertext model: Chimera associates anchors with views including an object view and the viewer displaying the object view. The Chimera concept of views is independent of where it is stored. A Chimera view could contain interface objects such as buttons and windows, depending on how the viewer defines its views. This is flexible in handling multiple views of a single object. There is no analogous concept of such anchors in GHMI. The way to multiply view an object in GHMI is to define a link pointing to a component with proper presentation specifications. Chimera's approach allows the viewers to freely implement viewer-specific features at the price of managing links and anchors inconsistently, which makes it difficult to extend standard features in the Chimera server. (4) Link traversal: GHMI's applications are hypertext-unaware and do not participate in link traversal. Chimera's viewers, however, are heavily burdened to define anchors and map anchors to objects.

Multicard. (1) Application domain: In contrast to GHMI, Multicard [78] is primarily open to editors which are display-oriented applications. The authors of

Multicard mentioned that Multicard can provides integration with large range of applications from basic text editors to sophisticated systems such as expert systems and object-oriented database systems. But it is still an ongoing issue and it is not clear how to support these dynamic systems at the system level. In [4], the authors connect Multicard to an object-oriented database system O2 to support querying hyperdocuments. They only *use* database systems to implement their hypertext facilities, rather than take database systems as an application domain and add hypertext functionality to them. (2) Architecture: Multicard's editors have to be modified to be use M2000 to participate in integration. GHMI does this by separating the applications from their handlers. An application is hypertext unaware and thus minimal changes are imposed for cooperating with its own handler. The domain mapping between application objects and hypertext objects happens at run-time by applying appropriate bridge laws. (3) Hypertext model: Multicard includes a simple version of composites which is a node hierarchy consisting of nested nodes, similar to GHMI's Tree composites. Multicard's links and anchors are untyped and links are binary only. Multicard's script-attached links are similar to GHMI's concept of operation links. GHMI's concept of bridge laws is similar to the Multicard script language in the sense of defining dynamic behavior of operation links. The behavior of a GHMI operation link is specified in bridge laws. The difference is that GHMI provides bridge laws for the purpose of mapping applications, while Multicard aims at providing a tool to extend its system functionality.

7.2.2 The Dexter-based Models

The Dexter model [47] is widely referenced and accepted as a common, principled interchange standard for diverse hypertext systems. Its separating hypertext into three layers makes modeling conceptually clearer and more understandable. Having such a model as our base enables us to share and compare our work with other researchers based on a common framework. Over the past several years, models and systems have been developed following the Dexter approach. Figure 7.3 shows the comparison of GHMI with other Dexter-based models.

DHM (or DeVise hypermedia) [38, 41, 39, 37, 40] is a Dexter-based hypermedia prototype developed at Aarhus University in Denmark. DHM extends Dexter in link directionality, dangling links, anchor typing, structures and component contents. Besides Dexter's four constant link directions (i.e., "From", "To", "Binary", "None"), DHM employs a broader concept including three orthogonal notions of link directionality: semantic directions, creation directions and traversal directions. In contrast to Dexter, DHM allows dangling links which have no "To" directions. DHM's anchors are typed to include whole-component anchor, marked anchors and unmarked anchors. By storing the references instead the contents of components, DHM supports linking to objects created by external applications. GHMI have many similarities with DHM. GHMI's external components and keyword anchors are similar to DHM's. GHMI also models component's internal structures and shares the concern of distinguishing hypertext-managed components from application-managed components with DHM. In [37], Grønbæk further extends DHM composites to a class hierarchy and four aspects of composite contents. The class hierarchies of GHMI and DHM are similar but follow different perspectives. DHM focuses on modeling the entire storage layer and run-time layer objects while GHMI separates the navigation structures from the underlying classes with the belief that the navigation structures can be modeled with the underlying component structures. DHM provides an architecture for cooperative work support [39] which is not the current focus of GHMI. DHM's structure dimension of modeling composite's contents is similar to GHMI's composite subclasses. DHM's virtual computed composites are similar to GHMI's computed components. However, GHMI's component content computation could

Extensions or Issues / Models	Anchors	The Storage Layer				Presentation Specification	The Run-time Layer	Distinctive Features
		Components	Links	Navigation	Other			
Dexter (Halasz & Schwartz)	- <ID, Value> pairs - Embedded in components	- Atomic, composite - Computed components - Embedding anchors and other components	- Links as components - N-ary links: a sequence of specifiers - Directionality: 4 constants			- Mandatory	- Session (simple)	- Hypertext interchange - Standard model
GHMI (Wan'96)	- Three types (plain, keyword, dynamic) - Dynamic anchors - External anchors	- Class hierarchy - Composite structures - External components - Dynamic mapping	- Six behavior-based types - Filtering - Dynamic mapping - Dangling links	- Guided tours, bookmarks, overviews, history, browsing	- Hypertext knowledge base	- Optional - Multiple views of composites	- Event structures - System traversal logs	- Integraion with dynamic information systems
DHM (Gronbak et al.'92,'94)	- Typed (marked, unmarked) - Keyword anchors	- Class hierarchy - External components - Virtual and computed components	- Dangling links - Directionality: 3 notions	-Guided tours, tabletops, link browsers	- Hierarchical structuring (hierarchy by reference & hierarchy by inclusion)	- Optional (as window position and size)	- Class hierarchy (session and instantiation)	- Object-oriented design - Cooperative hypermedia systems
"Issues in..." (Leggett & Killough'91)		- Versioning - External components	- Dangling links - Link behaviors					- Hypermedia interchange: transfer Intermedia to KMS
SP3 (Leggett & Schnase'94)	- Open anchor types - Modeled as processes	- External components - Composite internal linking	- Open link types - Modeled as processes		- Seven assumptions for hypermedia-in-the-large system design		- Context: across-hypetext links	- Collaboration - Hypermedia-in-the-large
RHYTHM (Maioli et al.'94)	- External anchors	- Files, documents, versions	- Link directionality: inclusion, from, to - Two link types: inclusion and navigation links		- Explicit attributes: root, author, num-version, sys-admin		- Handlers: complex sturcture	- Versioning
AHM (Hardman et al.'94)	- A list of <Comp ID, Anchor ID>	- External components - Synchronization arcs - Composite types: parallel and choice	- Link context: comp collection affected by a link traversal		- Collection and synchronization	- Atomics: channel, duration - Composites: sybchronization arcs		- Extending hypertext to hypermedia
"Adding..." (Garzotto et al.'94)		- Collections: inner structure and operations				- Guided-tours, indices - Collection-navigation - Collection-synchronization		- Multimedia support

Figure 7.3 Comparison with Other Dexter-based Models

involve dynamic mapping rule execution in applications while DHM's computation takes place within the hypertext domain.

Leggett and Schnase criticizes Dexter's abilities on hypermedia interchange and hypermedia-in-the-large (i.e., open hypermedia systems) design [63]. They address four issues from their experience on translating Intermedia and KMS using Dexter as an exchange standard [62], including underlying model confliction, multidestinational links, link directionality and methods of defining hypermedia boundaries. Although they consider Dexter a robust model for hypermedia systems as an interchange standard, they discuss issues regarding Dexter's problems on: not allowing dangling components, no notion of versioning, no external components, no notion of deletion semantics for composites, no notion of composite's internal linking and restrictive navigational link semantics. GHMI addresses all of these issues except versioning. In addition, Leggett et al. proposes seven fundamental assumptions for hypermedia-in-the-large system design. Based on these assumptions they claim that Dexter does not support hypermedia-in-the-large and it is not profitable to further extend the Dexter model. GHMI addresses similar issues on broader link services (by providing a larger range of hypertext functionalities) and heterogeneous application support. However, GHMI differs from Leggett et al.'s work in three major ways. (1) different focuses: Leggett et al. focuses on issues for general hypermedia-in-the-large system design while GHMI focuses on supporting dynamic mapping of computation-oriented applications; (2) different perspectives of viewing Dexter: GHMI follows only the *spirit* of Dexter on layered modeling and consistent representation of hypertext elements as storage layer components. Taking Dexter as a base did not prevent us from extending and specifying Dexter to fit our needs. It is unnecessary to recommend the termination of Dexter; (3) different models for link and anchor behaviors: Leggett et al.'s SP3 employs a process-based design by modeling links and anchors as processes and allowing open types. Such an approach allows broader and extensible application integration at the price of heavy performance (especially in distributed systems), inconsistent link/anchor behaviors and heavy application burden (the applications have to define link and anchor behaviors). In contrast, in GHMI, the applications are hypertext-unaware.

RHYTHM [66] is a hypertext system developed at the University of Bologna in Italy. The authors believe that modeling RHYTHM using Dexter proved the usefulness, soundness and robustness of Dexter, although they made a few extensions. RHYTHM components are files, documents and versions which can be mapped to Dexter components. Files are entities storing actual data. A version is an entity showing data to users through a list of references. Versions are collected in large entities called documents which establish relations between them. RHYTHM allows only binary links and divides links into two disjoint classes: navigation links and inclusion links. Versions are composites made exclusively of links and can include previous versions through inclusion links. Navigation links include all binary links other than inclusion links. RHYTHM extends the Dexter anchor concept to support external anchors. GHMI has similarities with RHYTHM concerning the concepts of external anchors, computed anchors, anchor resolver and link typing, but differs from RHYTHM in three aspects: (1) RHYTHM does not explicitly model keyword anchors. Furthermore, RHYTHM's concept of computed anchors is similar to GHMI's dynamic anchors. But RHYTHM does not include a mechanism to define computed anchors. GHMI defines dynamic anchors using bridge laws. (2) RHYTHM's restriction on binary links is too narrow for modeling complex links in large hypertext application environments; (3) RHYTHM includes a primitive notion of link typing with a distinction of navigation links and inclusion links while GHMI models a broader range of behavioral link types. RHYTHM's inclusion links are a subset of GHMI's structure links.

The Amsterdam Hypermedia Model (AHM) [50] is a general framework focusing on extending hypertext to hypermedia. AHM was developed as a combination of the Dexter model and the CMIF multimedia model [14] with extensions on Dexter by introducing the notions of time, high-level presentation attributes and link context. AHM extends Dexter's presentation specification on atomic components to include channel and duration information. Channels define global attributes in documents, including media-type independent specifications (e.g., background, foreground and highlight colors) and media-type dependent specifications (e.g., font and size for texts; scaling factor for graphs; volume for voices; etc.). An AHM composite does not contain any direct data. Instead, a composite references its data via an atomic component. A composite's content contains a collection of atomic or other composites. The presentation specification of composites contains a collection of synchronization arcs which are structures defining relative ordering information. AHM introduces the notion of *link context* which is a component containing a group of composites or atomics affected by a linking operation at run-time. Link context allows a "follow link" operation to affect only part of a document structure. Nevertheless AHM extends Dexter from a multimedia point of view which is not the current focus of GHMI, both models share common points on external components.

Garzotto et al. [35] made extensions on Dexter's storage layer by introducing the concept of *collections* and on Dexter's run-time layer using related notions of collection-navigation and collection-synchronization. A collection is a composite consisting of member nodes (or components). The internal structure of a collection includes two aspects: a *set* of members and a *structure* of *topologically* arranged members. These structures are similar to GHMI's Set and Graph composites. Operations on a collection include definition of the member set, definition of its internal structure and definition of the association node which represents the collection. Indices and guided-tours are two basic collection-based navigation structures. GHMI shares the notion of guided-tours, especially the nested guided-tours [34] and defines a richer set of guided-tour categories. Another difference is that Garzotto et al. models navigation as an extension to the Dexter run-time layer with the consideration of active media while GHMI models guided-tours and other navigation structures using the storage layer constructs.

7.2.3 GHMI and WWW

Although both provide hypertext features, the World-Wide-Web (WWW) and GHMI are quite different in their design purpose and system architecture. The WWW provides a world-wide access and browsing environment in a hypertext manner. GHMI aims at providing integrating COISs with hypertext and providing COISs with hypertext functionalities dynamically. We view their differences from the following aspects. (1) Data Model: Unlike GHMI, the WWW's hypertext data model is simple. It consists of plain nodes and binary links. It has neither structures nor composites. The WWW does not support bidirectional links as it employs HTML which embeds links only in departure documents. Links are also unlabeled and untyped (neither semantically nor behaviorally). This could cause navigation disorientation by overwhelming users with a vast of structure-less information. GHMI's model improves this situation by including composite structures, n-ary links, bidirectional links, and behavioral link typing. (2) Navigation: GHMI shares WWW with the functionalities on link traversal (i.e., browsing), querying (COIS supported), history, backtracking and bookmarks. When navigating around the WWW, however, users can be easily disoriented and lost due to its insufficient navigation structures and tools. Another tradeoff of most WWW viewers is their single-window environment which worsens disorientation; GHMI provides a richer set of navigation facilities, including guided-tours, overviews and task-based backtracking, which are not available in the WWW. (3) Document Markup: WWW forces its authors to use a markup language HTML to reproduce their documents in order to be accessible through WWW viewers. This reduces the WWW's openness and flexibility severely. All links and anchors are static and have to be encoded in an HTML format in application documents' content. There is no easy way to link existing documents dynamically or even add a link at run-time manually. All documents have to be rewritten to embed static links and anchors which are read-only at run-time. To implement these linking features, users have to write specific programs using some script language (e.g., CGI scripts), which is complex and overwhelming to average authors. GHMI integrates COIS applications through dynamic mapping and supports dynamic links in a much easier way. It separates links from the original documents and therefore does not impose any markup on original applications. This enables dynamic links and anchors to be generated at run-time. (4) Distribution: The major distinctive feature of the WWW lies in its world-wide distribution and ability of interoperating among a large range of heterogeneous hardware and software environments. This is not available in current GHMI. It would be an interesting future research to enhance GHMI by making the WWW an IOS, combining the WWW's distribution with GHMI functionalities.

7.3 GHMI Contributions and Limitations

In this thesis, we presented a general hypertext model GHMI, which is a Dexter-based hypertext model supporting integration of hypertext and computation-oriented information systems (COISs). This section summarizes GHMI's major contributions as well as its limitations.

7.3.1 GHMI Contributions

GHMI aims at enhancing COISs by adding hypertext functionalities through dynamic linking facilities. Integrating with GHMI only imposes minimal changes on COISs. We view GHMI's major contributions from the following four points of views: (1) GHMI vs. Bieber et al.'s work [12, 9]: Taking its motivation from Bieber et al.'s original concept of bridge laws, GHMI extends and formalizes bridge laws within a comprehensive hypertext data model. GHMI models composites which are not found in Bieber et al.'s work. Furthermore, GHMI formalizes the dynamic mapping concepts into a hypertext data model (through its MRDC processing). Also, GHMI extended and implemented the general system architecture in Figure 3.1, originally proposed by Dr. Bieber but not yet implemented, as a running prototype; (2) GHMI as a hypertext data model: As a general hypertext data model for supporting hypertext and COIS integration, GHMI uniquely provides a comprehensive set of hypertext functionalities regarding hypertext objects (composites, behavioral link typing, and dynamic anchors), domain mapping mechanisms (bridge laws) and a variety of navigation features (guided-tours, task-based backtracking, history, bookmarks, overviews); (3) GHMI as a Dexter-based model: GHMI uniquely combines specific extensions and specifications on Dexter to meet the requirements of our dynamic domain mapping environment. This demonstrates both GHMI's and Dexter's robustness and generality. Extensions are introduced on Dexter's composites, link specifiers and anchors. To map all GHMI capabilities in terms of Dexter, GHMI specifies Dexter's components, links, anchors, the resolver function and the accessor function; (4) The GHMI Prototype: The GHMI prototype is the first hypertext system which implements the general architecture of supporting dynamic integration of hypertext and multiple COISs. It proved the feasibility of the architecture and the GHMI model.

7.3.2 Potential Limitations

The GHMI approach has limitations in three aspects: (1) Object Identities: GHMI relies on resolvable [47] COIS object identifiers to map explicit COIS objects to hypertext objects. This approach benefits from the fact that object identities are widely adopted in systems with the increasingly-popular object-oriented designs. (2) Software Engineering: The COIS builders have to write bridge laws and the COIS handlers. They need to learn the bridge law template syntax. The complexity of bridge laws and COIS handlers depends on the complexity of COISs. In complicated COISs, writing bridge laws and COIS handlers could be difficult and tedious. The burden of writing COIS handers could be greatly reduced by providing (by the GHMI developers) a set of built-in APIs for those modules common to all COIS handlers. Actually this is possible for all the COIS handler modules except the COIS buffering module and the COIS invoker (see §6.1.4). (3) Speed: The speed of dynamic mapping could be slow. Bridge law mapping involves HTE-COIS communication and COIS program execution, which could be time-consuming. Speed depends on how much inference the bridge laws do and how much COIS execution is needed to generate outputs. In certain environments (such as real-time applications) when speed is the highest concern, GHMI's approach might not be satisfactory, although software and hardware optimization could help some.

7.4 Future Research

GHMI is a robust model for supporting COIS-hypertext integration. Extensions in several directions can be made to enhance the current version of GHMI and its prototype resulting from this dissertation. This section outlines the future work we plan to pursue after this dissertation.

Implementation Issues. (1) Unimplemented Features: We plan to continue implementing those features defined in the GHMI model but not included in the GHMI prototype. This includes: system configuration, bridge law configuration, annotations, navigation based guided-tours, user-defined guided-tours, task-based backtracking, bridge law parser, etc. (2) Dynamically updating objects: The current GHMI prototype does not consider how to dynamically update objects which are currently displayed on screen. In the current GHMI prototype, neither the Linkbase nor the Session DB stores these objects' interface-related data (e.g., at which window and what position the objects are displayed). The current GHMI assumes that data processing is triggered by user events in the IOSs. However, in some information systems, internal triggers will cause events to occur (e.g., an office automation system might include a week trigger for display all meetings for the week every Monday at 8am). In other systems, some items on the screen need to be updated automatically (e.g., a financial system interface with stock prices fluctuating over time). GHMI needs to provide a mechanism to facilitate these situations. (3) Destructive operations: Current GHMI prototype maps operation results as dynamic tables. It does not consider the situation of a destructive operation which generates no explicit results but might delete COIS objects without notifying the HTE. This could make the data in the Linkbase out of date (e.g., links to a deleted COIS object). Updating data as a result of outside changes is still an open issue in the area of open hypertext systems [23, 24, 27, 81, 63, 5, 78]. GHMI assumes the COIS handler is responsible for notifying the HTE about such operations. If any dangling component occurs (e.g., link traversal, backtracking or accessing annotations to a deleted COIS object), the HTE gives the user a warning message and deletes the corresponding link resolving to the dangling component. The potential dangling objects due to such destructive operations can not be found until the user follows a link pointing to them. The COIS could notify the HTE on such an action. This prototype does not require COISs to report destructive operations. An effective solution is not available in current GHMI and is open for future exploration.

Distribution and WWW. In recent years, delivering electronic information via computer networks has been gaining significant growth. There are numerous hypertext systems operating in a distributed environment [3, 75, 100, 52, 15]. The most well-known and widely used distributed hypertext system is WWW, which provides a robust navigation environment among a large range of heterogeneous information resources. A WWW document with hypertext links is constructed based on a markup language HTML which is a simple version of SGML. We could combine GHMI's functionalities with WWW's distribution feature by extending the GHMI model and making it WWW-compatible. Our first plan toward this goal is to develop a WWW handler which connects GHMI to WWW and make it directly accessible from the Internet. We can build basing on the current prototype by replacing (or adding, if we want both) the current IOS handler with a WWW handler written as a CGI script and the current IOS with a WWW browser (e.g., Netscape). Such replacement will not affect any code in the HTE and the COIS/COIS handlers (except to add some HTE configuration information). The WWW handler would intercept messages previously sent to the current IOS handler. It would convert object contents to HTML documents with embedded links by combining object content messages with anchor markup messages from the HTE. The five current IOS viewers could be re-implemented in WWW browsers (e.g., Netscape) by means of WWW scripting languages (e.g., Pearl, JAVA scripts). Menu items, buttons, trees and graphs can be easily implemented using these languages.

Hypertext Searching and Querying. GHMI allows a component specification to be either a COIS query or a hypertext query. A COIS query is dynamically resolved by the COIS and the results are mapped to hypertext components. The mechanisms supporting hypertext query processing are left out of the current GHMI. Issues regarding searching and querying on hypertext structures have been addressed by hypertext researchers [29, 42, 28, 64, 21, 30, 65, 53, 4]. In GHMI's dynamic mapping environment, structural querying (or search) on a hypertext network becomes complicated as the hypertext network is not directly available prior to dynamic content mapping. The hypertext engine would have to execute bridge laws to map all components to build the hypertext network (or mapped a subset of components, depending on particular queries) prior to resolving a hypertext query. An alternative way which avoids heavy bridge law execution is that the hypertext engine employs some *query mapping* mechanism to translate a hypertext query to a COIS query and relies on the COIS to resolve the query. Such an approach works on the assumption that the COIS has some query processing ability and the query translation is less complicated than the generation of the entire hypertext network through bridge law execution combined with structural search of the hypertext objects (e.g., annotations, association links).

Versioning. Versioning is an important feature of hypertext systems and has been included in some systems [69, 43, 81, 63]. Versioning enables users to access and manipulate a history of information changes to their hypertext network. The current GHMI does not support versioning. We can extend GHMI's component properties to include versioning information. Each component could have its own version history, probably modeled as a linear *List* or a more complicated *version Tree*. Versioning with composites could be modeled at two levels: versioning on a composite itself as a whole, and versioning on its individual subcomponents and links. In GHMI's COIS integrating environment, another unsolved issue is: should versioning be supported by the HTE or the COISs? In either way, GHMI needs to identify versioning properties and include them in the bridge law template.

Collaboration. Supporting cooperative work on a shared hypertext network among multiple users is another important feature of today's hypertext systems [81, 63, 38, 86]. Currently GHMI does not include collaboration supporting. We need to extend GHMI to include notions of managing asynchronous access on a single component or an entire hypertext network. The collaboration supporting mechanisms typically include ownership identity, locking, transaction management, concurrency control and event notification.

Multimedia. There are existing hypertext models and systems supporting multimedia [50, 35, 14]. Examples of multimedia components include a CAD picture, a raster image, a short video clip, a short audio tape, a short animated sequence, etc. Currently GHMI only considers text components. The component framework in GHMI can be extended to support multimedia. GHMI's structured composites allow a separation of component contents from their hypertext representations. For example, a data file storing a raster image can be represented as an atomic which contains the reference to it. The collections of run-time multimedia presentations can also be modeled as structured composites in GHMI. For example, when the user wants to watch a movie while reading a text caption and listening to an audio tape, the collected presentation of these three types of media can be modeled as a structured composite. We need to extend the component properties to support time synchronization and media-related attribute specifications. We also need to support anchors in multimedia components.

7.5 Summary

In this chapter, we discuss guidelines for using GHMI, compare GHMI with other systems and models, identify both GHMI's major contributions and limitations, and briefly outline future research that can emanate from GHMI.

To integrate a COIS with a GHMI hypertext system, the COIS builders (or developers) need to follow these guidelines. (1) Study the GHMI Model: The first step toward building an integration system is to study the GHMI hypertext concepts, including components, links, anchors, navigation features and especially the bridge law template. (2) Identify Potential GHMI Constructs: This step is to identify COIS objects, relationships, meta-information and operations (or commands), which could be mapped to GHMI constructs (i.e., component classes and typed links). (3) Software Engineering: This step is to write the mapping rules (i.e., bridge laws) and the COIS handler code.

GHMI and its prototype share ideas and common constructs with other systems developed by hypertext researchers, especially in the field of providing hypertext functionality to third-party applications and Dexter-based modeling. We compared GHMI (and the current GHMI prototype) with open hypertext systems (including Sun's Link Service [75], Microcosm [23, 24, 27], SP3 [58, 63, 81], Chimera [5] and Multicard [78]), the Dexter-based models and the WWW.

We identified GHMI's major contributions and limitations. GHMI's contributions include four aspects: GHMI vs. Bieber et al.'s work, GHMI as a general hypertext model, GHMI as a Dexter-based model and the GHMI prototype. GHMI's limitations include three aspects: relying on object identities, heavy software engineering for bridge laws and COIS handlers and slow speed for dynamic mapping.

GHMI is a robust model for supporting COIS/hypertext integration. Extensions in several directions can be made to enhance the current version of GHMI and its prototype resulting from this dissertation. These issues include improving implementation, connecting GHMI to WWW, hypertext searching and querying, versioning, collaboration and multimedia.

Conclusion Remarks

In this thesis, we presented a general hypertext model GHMI, which is a Dexter-based hypertext model supporting integration of hypertext and computation-oriented information systems (COISs). GHMI enhances COISs by adding hypertext functionalities through dynamic mapping mechanisms. Integrating with GHMI only imposes minimal changes on COISs. GHMI extends and specifies the original Dexter model with additional concepts which are fundamental to our goal of dynamically adding hypertext functionalities to COISs. We also proved the feasibility and utility of the GHMI concepts by implementing it as a prototype. In our future research we shall enhance GHMI by following several directions.

APPENDIX A

SECOND MODELING DOMAIN: TEXPROS

This chapter demonstrates how to apply GHMI to model a COIS through an example system called TEXPROS [99], which is an intelligent document management system developed by researchers in our institute.

A.1 TEXPROS's Data Model

TEXPROS is a personal document processing system combining filing and retrieval systems. It supports storing, extracting, classifying, categorizing, retrieving and browsing information from a variety of documents. Documents are grouped into *classes.* Each class is associated with a semantic document *type* to describe the common properties for the class of documents. A data structure called a *frame template* represents the document class *type.* A *frame template* can be instantiated by filling its attributes with values extracted from the original document. The instantiated object is a *frame instance*, representing a synopsis of a single document, rather than its original contents. Window 2 in Figure A.2 shows an instantiated frame instance for the frame template of type "Assistantship." The template's left-hand column contains its attributes and the right-hand column contains the frame instance's values.

A *folder*, identified by its title, is a logical repository of documents comprising a set of frame instances. Folders represent the user's logical file structures. They are connected via the "Depends On" relationships. A folder depending on another folder is called a *subfolder* or *child* folder while the depended-on folder is called a *parent* folder. Subfolders are categorized via some user-declared *criteria.* A folder could depend on multiple parent folders. Figure A.1 shows a simplified logic file

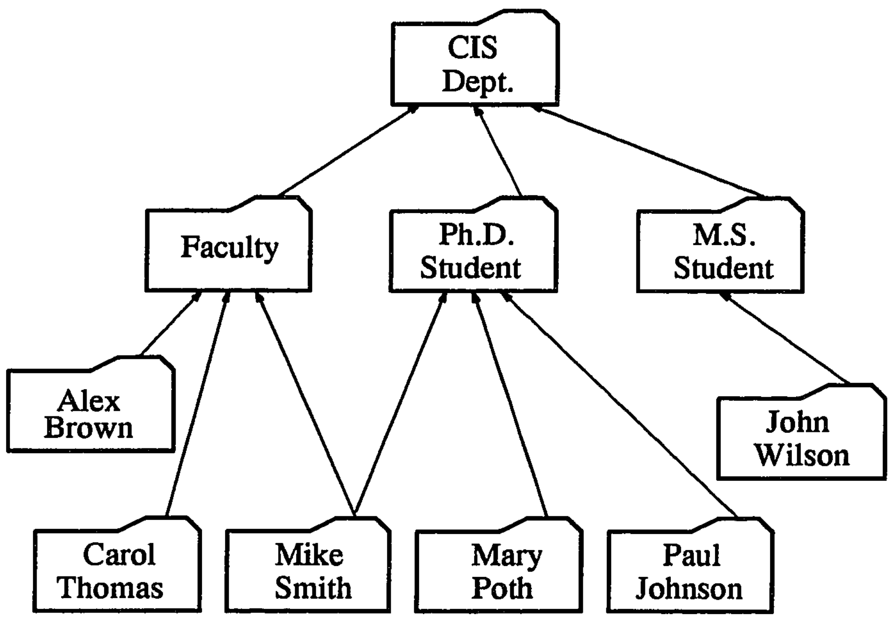


Figure A.1 A Department Chairperson's Logical File Structure

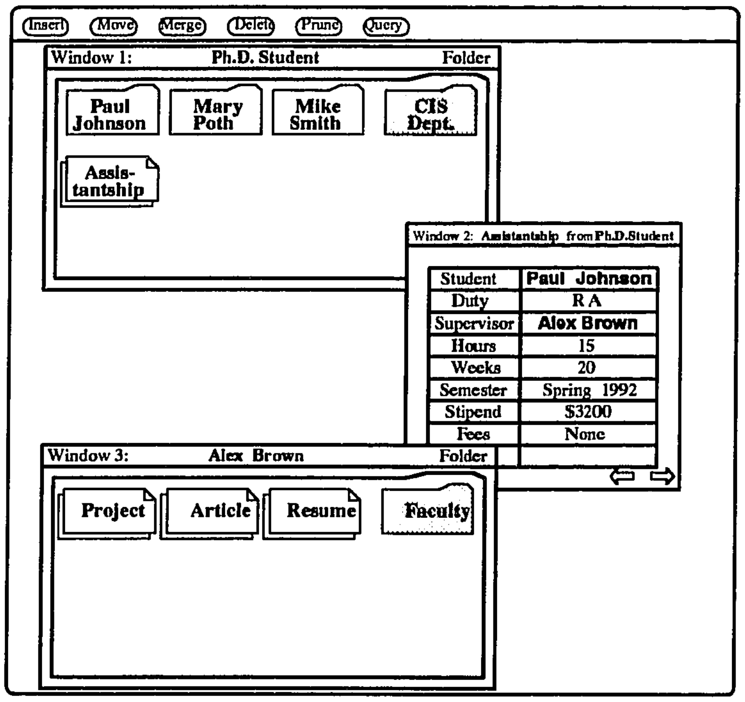
structure—the hypothetical folder organization of a department chairperson. Arrows signal the relationship "Depends On." A set of operations can be applied on folders to manipulate the file structure. Operations include *insert*, *move*, *merge*, *prune*, *query* and *delete*. Formal definitions and semantic descriptions of operations can be found in [98].

A.2 Mapping TEXPROS to GHMI

In this subsection, we identify potential bridge laws which map TEXPROS to GHMI components and links.

- **Object BLs**

  TEXPROS objects include frame templates, frame instances, original documents and folders. We can write BLs to map frame templates to structured atomics, frame instances to structured atomics, original documents to plain atomics and folders to Sets (containing folders and frame instances).



Window 2: Assistantship fromPh.D.Student

Student	Paul Johnson
Duty	RA
Supervisor	Alex Brown
Hours	15
Weeks	20
Semester	Spring 1992
Stipend	$3200
Fees	None

Figure A.2 An Example Hyperdocument for TEXPROS

- **Structure BLs**

  Structural relationships help directly access a component's embedding composite. We can map TEXPROS's "Depends On" as a structure link from a child folder to a parent folder. Other structural relationships include frame instances to folders. Accessing a folder from its residing frame instances is another example of structural relationship.

- **Operation BLs** All TEXPROS operations can be mapped as GHMI operation links, including *insert*, *move*, *merge*, *prune*, *query* and *delete*. Specific queries on folders, such as "Find all Ph.D. students who have financial support" could be mapped to operational links departing from folder "Ph.D. Student" and ending with a list of computed folder titles.

- **Reference BLs** Implicit relationships among TEXPROS objects can be modeled by explicit GHMI reference links, which provide a direct access to these relationships. Consider the following examples:

(1) Given a frame instance, find all folders whose titles appear in this frame instance as a value.

(2) Given a frame instance, find all other frame instances of the same frame template.

(3) Given a frame instance, find its frame template.

(4) Given a frame instance, find its original document.

(5) Given a frame template, find all of its frame instances.

Suppose, as shown in Figure A.2, that a frame instance in the "Ph.D. Student" folder contains a reference to "Alex Brown." "Alex Brown" is also a folder, but is not connected to the "Ph.D. Student" folder by a "Depends On" relationship. TEXPROS's original model included no way of capturing or representing such an "implicit" relationship among folders. Once mapped to reference links, GHMI allows direct access to folder "Alex Brown" from folder "Ph.D. Student."

- **Meta-information BLs** Meta-information for TEXPROS objects includes file structure size (number of folders), number of frame instances in a folder, number of subfolders in a folder, object timestamps, etc.

A.3 Bridge Law Examples

This section presents some bridge law examples to illustrate the domain mapping from TEXPROS to GHMI hypertext.

1. BL_{folder}: *Mapping Folders to Set components*, as shown in Table A.1.

Table A.1 Bridge Law BL*Folder*

CompClass	'Set'
OwningSystemType	'TEXPROS'
CompName	'Folder'
PresentationSpec	'Tree'
COISObj	$[F, \text{'}Folder\text{'}, F]$
CompSet	$\{[N1, \text{'}Folder\text{'}, N1], [N2, \text{'}FrameInstance\text{'}, T]\}^*$
LinkSet	'NONE'
ContentSpec	'NONE'
MappingRule	$object(F, \text{'}Folder\text{'})$, $object(N1, \text{'}Folder\text{'})$, $object(N2, \text{'}FrameInstance\text{'})$, $relation(N1, F, \text{'}DependsOn\text{'})$, $property(N2, \text{'}Type\text{'}, T)$, $relation(N2, F, \text{'}ResidesIn\text{'})$.

The following instantiation of BL_{folder} maps folder 'Ph.D. Student' to a *Set* in Figure A.2 with a content including all of its three subfolders:

$$APPLY\_BL\_COMP(BL_{Folder}, F = \text{'}Ph.D.\ Student\text{'})$$

The resulting component will have COISObj as

$$[\text{'}CIS\ Dept.\text{'}, \text{'}CIS\ Dept.\text{'}, \text{'}Folder\text{'}]$$

2. BL_{Fi}: *Mapping* Frame Instances *to* Structured Atomic *components*, as shown in Table A.2.

This bridge law is executed to map frame instances when the user selects a frame instance icon to explore its contents. For example, when the user selects the icon labeled as "Assistantship" in Window 1 of Figure A.2, the hypertext engine executes BL_{Fi} to map the individual frame instances one by one. Prior to this mapping, every such frame instance has been denoted as a COISObj in

Table A.2 Bridge Law BLFi

CompClass	'StructuredAtomic'
OwningSystemType	'TEXPROS'
CompName	'FrameInstance'
COISObj	$[F, 'FrameInstance', T]$
CompSet	'NONE'
LinkSet	'NONE'
ContentSpec	C
MappingRule	$object(F, 'FrameInstance')$, $property(F, 'Content', C)$, $property(F, 'Type', T)$.

the *CompSet* of folder "Ph.D. Student" previously mapped by BL_{Folder} (see Table A.1). Therefore, the instantiation of BL_{Fi} will be expressed as:

$$APPLY\_BL\_COMP(BL_{Fi},$$

$$COISObj = [F, 'FrameInstance', 'Assistantship'])$$

where COISID F is extracted from *CompSet* of folder "Ph.D. Student."

3. $BL_{DependsOn}$: *Mapping relationship "DependsOn" to a Structure link*, as shown in Table A.3.

   This BL allows a child folder to access its parent folder following the direction of "Depends On." The "FROM" and "TO" endpoints of this link are specified using two variable names $F1$ and $F2$, which represent COIS objects defined in the *MappingRule*. The *MappingRule* of this BL consists of three predicates. $object(F1, 'Folder')$ and $object(F2, 'Folder')$ indicate $F1$ and $F2$ are two existing folders in TEXPROS database. $relation(F1, F2, 'DependsOn')$ indicates folder $F1$ "depends on" folder $F2$. When the content of $F1$ is mapped to a component and is on display, the hypertext engine executes this BL to map all "DependsOn" links departing from $F1$. The following instantiation of $BL_{DependsOn}$ maps a link "Depends On" marked by icon "CIS

Table A.3 Bridge Law BL*DependsOn*

CompClass	‘Link‘
OwningSystemType	‘TEXPROS‘
CompName	‘DependsOn‘
LinkType	‘StructureLink‘
Specifier_1 CompSpec AnchorSpec Direction	 $[F1, \text{‘}Folder\text{‘}, F1]$ ‘NONE‘ ‘FROM‘
Specifier_2 CompSpec AnchorSpec Direction	 $[F2, \text{‘}Folder\text{‘}, F2]$ ‘NONE‘ ‘TO‘
MappingRule	$object(F1, \text{‘}Folder\text{‘})$, $object(F2, \text{‘}Folder\text{‘})$, $relation(F1, F2, \text{‘}DependsOn\text{‘})$.

Dept.” in Window 1 of Figure A.2:

$$APPLY\_BL\_LINK(BL_{DependsOn}, F1 = \text{‘}Ph.D.Student\text{‘})$$

4. $BL_{RefToFolder}$: *Given a frame instance, find all folders whose titles appear in this frame instance as a value*, as shown in Table A.4.

   The “FROM” endpoint’s *CompSpec* is specified by $F1$ and the “TO” endpoint’s *CompSpec* is specified by $F2$. The “FROM” endpoint has anchors of type “Dynamic” extracted from the content of $F1$.

   Bridge law $BL_{RefToFolder}$ is automatically executed to generate all “RefTo-Folder” links departing from a frame instance when its is mapped to an atomic component. All corresponding anchors are highlighted in some manner to single the existence of these reference links. In Figure A.2, when the user reaches Window 2, all folder titles occurring in the content of the frame instances of type “Assistantship” are highlighted as anchors marking links “RefToFolder.”

Table A.4 Bridge Law BL$RefToFolder$

CompClass	'Link'
OwningSystemType	'TEXPROS'
CompName	'RefToFolder'
LinkType	'ReferenceLink'
Specifier$_1$ CompSpec AnchorSpec Direction	 $[F1, `FrameInstance', \_]$ $[A, `Dynamic', V]$ 'FROM'
Specifier$_2$ CompSpec AnchorSpec Direction	 $[F2, `Folder', \_]$ 'NONE' 'TO'
MappingRule	$object(F1, `FrameInstance')$, $object(F2, `Folder')$, $object(A, `Value')$, $relation(A, F1, `InContent')$, $object(F2, `Folder')$, $property(F2, `FolderTitle', V)$.

Let $FiID$ represent the frame instance in Window 2 of Figure A.2. The following instantiation of $BL_{RefToFolder}$ generates all links marked by anchors representing folder titles in Window 2 (e.g., "Alex Brown"):

$$APPLY\_BL\_LINK(BL_{RefToFolder}, F1 = FiID)$$

APPENDIX B

SAMPLE SOURCE CODE FOR DATABASE HANDLER

B.1 Prolog BL-Engine Code

```
\small
/* This is the Prolog BL_Engine program */
/* (original tt.pl)
/* Last modified: 4/18/96 */
/****************************************/
/* Loop until flag is set */
/* BLEngine Starts here */
/* It calls BL.pl */
/* If not run, check flagtext.pl */
:-initialization(startup).

startup:-
load_files([facts]),
repeat,
see('dirty.pl'), /* dirty flag from VB */
read(X),
seen,
check1(X),

see('flagtext.pl'), /* flag from VB */
read(X),
seen,
check(X),
fail.

/* If data is ready from VB, exectue BL */
check1(X):- X = 'Y',
see('dirty.pl'),
find('N')
-> seen;
task1.


/* If data is ready from VB, exectue BL */
check(X):- X = 'Y',
see('flagtext.pl'),
find('N')
-> seen;
task.


/* Old data, wait for new */
failed:- print('Old data'),nl,
seen.

/* Load and set flag */
task1:- seen,
load_files([facts]),
tell('dirty.pl'),
write(`N. `),
told.


/* Set flag */
task:- print(`I got it`), nl,
seen,
execute_bl,
tell('flagtext.pl'),
write(`N. `),
told.

/* Execute bl in bl.pl automatically upon loading */
execute_bl:-
load_files([facts]),
load_files('bl.pl').

/* ===================*/
```

B.2 Visual Basic BL-Engine Code

```
\small
'/* This is the VB_BL_Engine subroutines in VB's DBHand.bas */
'/* It cooperates with the Prolog_BL_Engine to */
'/* generate and execute bridge laws. */
'/* Last modified: 4/18/96 */

Option Explicit
'/* Const for table field types */
Const FIELD_BOOLEAN = 1
Const FIELD_BYTE = 2
Const FIELD_INTEGER = 3
Const FIELD_CURRENCY = 5
Const FIELD_TEXT = 10
Const FIELD_MEMO = 12

Global gHandDB As Database

Sub BLEngine_BLComp (Msg$, AppDB As Database)
'/* Create a Comp BL in file bl.pl; */
'/* Clear and  set 'Y. ' in flagtext.pl ; */
'/* Wait and collect result in CompSet.pl, LinkSet.pl and Content.pl */
'/* from Prolog when 'N. ' is set in flagtext.pl; */
    Dim Predicate$, CompSet$, linkset$, Content$, Content1$
    Dim BLCOISID$, BLCOISType$, BLCOISLabel$
    Dim COISObj$, BLCOISObj$, COISID$, COISType$, COISLabel$, AppName$,
TmpType$
    Dim BLName$, FileName$, MappingRule$
    Dim List1$, List2$, List3$
    Dim DBName$, TableName$, RecordKey$, FielName$
    Dim nl, Spaces$
    Dim p1%, FNum1%, Fnum2%, ObjCount%

    nl = Chr(13)
    Spaces$ = "    "

    '/* Get input from Msg */
    BLName$ = Msg_GetTagValue(Msg$, "blName")
    BLCOISID$ = Msg_GetTagValue(Msg$, "blcoisid")
    BLCOISType$ = Msg_GetTagValue(Msg$, "blcoistype")
    BLCOISLabel$ = Msg_GetTagValue(Msg$, "blcoislabel")
    If BLCOISLabel$ = "" Then
BLCOISLabel$ = "_"
    End If
    COISID$ = Msg_GetTagValue(Msg$, "coisid")
    '/* Trim spaces around "\\" */
    p1% = InStr(1, COISID$, "\\ ")
    While p1 <> 0
```

```
COISID$ = Mid(COISID$, 1, p1 + 1) & Mid(COISID$, p1 + 3)
p1% = InStr(1, COISID$, "\\ ")
    Wend
    COISType$ = Msg_GetTagValue(Msg$, "coistype")
    COISLabel$ = Msg_GetTagValue(Msg$, "coislabel")
    If COISLabel$ = "" Then
COISLabel$ = "_"
    End If
    CompSet$ = Msg_GetTagValue(Msg$, "compSet")
    linkset$ = Msg_GetTagValue(Msg$, "LinkSet")
    Content$ = Msg_GetTagValue(Msg$, "ContentSpec")
    MappingRule$ = Msg_GetTagValue(Msg$, "MappingRule")

    '/* COISObj = "['COISID', 'COISType', 'COISLabel']"
    '/* Construct COISObj */
    's:- apply_bl('BL_Table1',[['Small School','Appartment'],'Table',_],
[CompSet,[],[]]).
    'Call DB_SeparateCOISID(COISID$, DBName$, TableName$, RecordKey$,
FieldName$, TmpType$)
    Call BLEngine_BuildCOISObj(COISObj$, COISID$, COISType$, COISLabel$,
True)
    Call BLEngine_BuildCOISObj(BLCOISObj$, BLCOISID$, BLCOISType$,
BLCOISLabel$, False)
    '/* Open file */
    FNum1% = FreeFile
    FileName$ = "c:\wan\ghmi\Pro386w\blengine\bl.pl"
    Open FileName$ For Output As FNum1

    '/* Create file header */
    '/* Idea: bridge_law_comp(BLName, COISObj, [CompSet, LinkSet,
ContentSpec]) */
    '/* Make ":-initialization(apply_bl('bl_databse',['Small School',
'Database','Small School'], [CompSet,[],[]])). "
    List1$ = "[]"
    List2$ = "[]"
    List3$ = "[]"
    If CompSet$ <> "NONE" Then
List1$ = "CompSet"
    End If
    If linkset$ <> "NONE" Then
List2$ = "LinkSet"
    End If
    If Content$ <> "NONE" Then
List3$ = "Content"
    End If
    Predicate$ = ":-initialization(apply_bl(" & BLName & "," & COISObj$ &
","
```

```
    Predicate$ = Predicate$ & "[" & List1$ & "," & List2$ & "," & List3$ &
"])). "
    Print #FNum1, Predicate$, nl

    '/* Make: 'apply_bl('bl_dataase',[D,'Database',_],[CompSet,LinkSet,
ContentSpec]):-"
    '/* apply_bl(BLName, COISObj, [CompSet, LinkSet, Content]):-"
    Predicate$ = "apply_bl(BLName," & BLCOISObj$ & ", [CompSet, LinkSet,
Content]):-"
    Print #FNum1, Predicate, nl

    '/* Make: "bagof([[D|T],'Table',T],compset([[D|T],'Table',T]),CompSet),
"
    If CompSet$ <> "NONE" Then
'/* Handle only single CompSet only */
p1% = InStr(1, CompSet$, "*")
If p1% <> 0 Then
    '/* Eliminate "*" at end */
    CompSet$ = Mid(CompSet$, 1, Len(CompSet$) - 1)
End If
Predicate$ = "bagof(" & CompSet$ & ", compset(" & CompSet$ & "),
CompSet), "
Print #FNum1, Spaces, Predicate, nl
    End If
    If linkset$ <> "NONE" Then
'/* Handle only single CompSet only */
p1% = InStr(1, linkset$, "*")
If p1% <> 0 Then
    '/* Eliminate "*" at end */
    linkset$ = Mid(linkset$, 1, Len(linkset$) - 1)
End If
Predicate$ = "bagof(" & linkset$ & ", linkset(" & linkset$ & "),
LinkSet), "
Print #FNum1, Spaces, Predicate, nl
    End If
    Content1$ = Content$
    If Content$ <> "NONE" Then
'/* Handle only single Content only */
p1% = InStr(1, Content$, "*")
If p1% <> 0 Then
    '/* Eliminate "*" at end */
    Content$ = Mid(Content$, 1, Len(Content$) - 1)
End If
'/* For records, add BLCOISID to Content "C" ***/
Content1 = BLCOISID$
Content1$ = "[" & Content1$ & "," & Content$ & "]"
Predicate$ = "bagof(" & Content$ & ", content(" & Content1$ & "),
 Content), "
 Print #FNum1, Spaces, Predicate, nl
     End If

     '/* Make control predicates */
     Predicate$ = Spaces$ & "fcreate('compset.pl',0), " & nl
     Predicate$ = Predicate$ & Spaces$ & "tell('compset.pl'), " & nl
     Predicate$ = Predicate$ & Spaces$ & "write(CompSet),nl, " & nl
     Predicate$ = Predicate$ & Spaces$ & "told, " & nl
     Predicate$ = Predicate$ & Spaces$ & "fcreate('linkset.pl',0), " & nl
     Predicate$ = Predicate$ & Spaces$ & "tell('linkset.pl'), " & nl
     Predicate$ = Predicate$ & Spaces$ & "write(LinkSet),nl, " & nl
     Predicate$ = Predicate$ & Spaces$ & "told, " & nl
     Predicate$ = Predicate$ & Spaces$ & "fcreate('content.pl',0)," & nl
     Predicate$ = Predicate$ & Spaces$ & "tell('content.pl'), " & nl
     Predicate$ = Predicate$ & Spaces$ & "write(Content),nl, " & nl
     Predicate$ = Predicate$ & Spaces$ & "told. " & nl
     Print #FNum1, Predicate$

     '/* Make: compset(CompSet):- MappingRule
     Predicate$ = "compset(" & CompSet$ & "):-" & nl
     Predicate$ = Predicate$ & Spaces$ & MappingRule$ & " " & nl
     Print #FNum1, Predicate$

     Predicate$ = "linkset(" & linkset$ & "):-" & nl
     Predicate$ = Predicate$ & Spaces$ & MappingRule$ & " " & nl
     Print #FNum1, Predicate$

     Predicate$ = "content(" & Content1$ & "):-" & nl
     Predicate$ = Predicate$ & Spaces$ & MappingRule$ & " " & nl
     Print #FNum1, Predicate$
     Close #FNum1

     '/* Invoke BLEngine *./
     '/* Return when data ready */
     Call BLEngine_SendProlog

     '/* Construct CompSet, LinkSet, Content from Prolog files */
     Call BLEngine_GetPrologData(CompSet$, linkset$, Content$)

     '/* Count Objects */
     p1% = InStr(1, CompSet$, "<")
     ObjCount% = 0
     While p1% <> 0
ObjCount% = ObjCount% + 1
p1% = InStr(p1 + 1, CompSet$, "<")
     Wend
```

```
    '/* Update Msg */
    AppName$ = Msg_GetTagValue(Msg$, "AppName")
    Call Msg_SetTagValue(Msg$, "COISID", AppName$)
    'Call Msg_SetTagValue(Msg$, "COISTYPE", "Database")
    Call Msg_SetTagValue(Msg$, "CompSet", CompSet$)
    Call Msg_SetTagValue(Msg$, "LinkSet", linkset$)
    Call Msg_SetTagValue(Msg$, "Content", Content$)
    Call Msg_SetTagValue(Msg$, "objindex", "1")
    Call Msg_SetTagValue(Msg$, "totalobjs", Trim(Str(ObjCount%)))


End Sub

Sub BLEngine_BLLink (Msg$, AppDB As Database)
'/* Create a Link BL in file bl.pl; */
'/* Clear and  set 'Y. ' in flagtext.pl ; */
'/* Wait and collect result in CompSet.pl, LinkSet.pl and Content.pl */
'/* from Prolog when 'N. ' is set in flagtext.pl; */
'/*
'/* BLCOISObj: 'From' specifier's CompSpec
'/* BLCOISObj2: 'To'  specifier's CompSpec
'/* Note: Here, CompSet.pl contains 'TO' BLspecifier's CompSpec */
'/* Another sub convert CompSet to table QueryResults */

    Dim Predicate$, CompSet$, linkset$, Content$, Content1$
    Dim BLCOISID$, BLCOISType$, BLCOISLabel$
    Dim BLCOISID2$, BLCOISType2$, BLCOISLabel2$
    Dim COISObj$, BLCOISObj$, BLCOISObj2$, COISID$, COISType$, COISLabel$,
AppName$, TmpType$
    Dim BLName$, FileName$, MappingRule$
    Dim List1$, List2$, List3$
    Dim DBName$, TableName$, RecordKey$, FielName$
    Dim nl, Spaces$
    Dim p1%, FNum1%, Fnum2%, p2%

    nl = Chr(13)
    Spaces$ = "    "

    '/* Get input from Msg */
    BLName$ = Msg_GetTagValue(Msg$, "blName")
    BLCOISID$ = Msg_GetTagValue(Msg$, "blcoisid")
    BLCOISType$ = Msg_GetTagValue(Msg$, "blcoistype")
    BLCOISLabel$ = Msg_GetTagValue(Msg$, "blcoislabel")
    '/* New: Find 'To' specifier's obj */
    BLCOISID2$ = Msg_GetTagValue(Msg$, "blcoisid2")
    BLCOISType2$ = Msg_GetTagValue(Msg$, "blcoistype2")
    BLCOISLabel2$ = Msg_GetTagValue(Msg$, "blcoislabel2")

    If BLCOISLabel$ = "" Then
BLCOISLabel$ = "_"
    End If
    If BLCOISLabel2$ = "" Then
BLCOISLabel2$ = "_"
    End If

    COISID$ = Msg_GetTagValue(Msg$, "coisid")
    '/* Trim spaces around "\\" */
    p1% = InStr(1, COISID$, "\\ ")
    While p1 <> 0
COISID$ = Mid(COISID$, 1, p1 + 1) & Mid(COISID$, p1 + 3)
p1% = InStr(1, COISID$, "\\ ")
    Wend
    COISType$ = Msg_GetTagValue(Msg$, "coistype")
    COISLabel$ = Msg_GetTagValue(Msg$, "coislabel")
    If COISLabel$ = "" Then
COISLabel$ = "_"
    End If
    CompSet$ = Msg_GetTagValue(Msg$, "compSet")
    linkset$ = Msg_GetTagValue(Msg$, "LinkSet")
    Content$ = Msg_GetTagValue(Msg$, "ContentSpec")
    MappingRule$ = Msg_GetTagValue(Msg$, "MappingRule")

    '/* COISObj = "['COISID', 'COISType', 'COISLabel']"
    '/* Construct COISObj: actual parameters */
    's:- apply_bl('BL_Table1',[['Small School','Appartment'],'Table',_],
[CompSet,[],[]]).
    'Call DB_SeparateCOISID(COISID$, DBName$, TableName$, RecordKey$,
FieldName$, TmpType$)
    Call BLEngine_BuildCOISObj(COISObj$, COISID$, COISType$, COISLabel$,
True)
    Call BLEngine_BuildCOISObj(BLCOISObj$, BLCOISID$, BLCOISType$,
BLCOISLabel$, False)
    Call BLEngine_BuildCOISObj(BLCOISObj2$, BLCOISID2$, BLCOISType2$,
BLCOISLabel2$, False)

    '/* New: Mimic CompSet of BLEngine_BLComp */
    '/* Using both 'From' and 'To' specifiers
    CompSet$ = BLCOISObj & "," & BLCOISObj2$

    '/* Open file */
    FNum1% = FreeFile
    FileName$ = "c:\wan\ghmi\Pro386w\blengine\bl.pl"
    Open FileName$ For Output As FNum1
    '/* Create file header */
    '/* Idea: bridge_law_comp(BLName, COISObj, [CompSet, LinkSet,
ContentSpec]) */
    '/* Make ":-initialization(apply_bl('bl_databse',['Small School',
'Database','Small School'], [CompSet,[],[]]). "
    List1$ = "[]"
    List2$ = "[]"
    List3$ = "[]"
    If CompSet$ <> "NONE" Then
List1$ = "CompSet"
    End If
    If linkset$ <> "NONE" Then
List2$ = "LinkSet"
    End If
    If Content$ <> "NONE" Then
List3$ = "Content"
    End If
    Predicate$ = ":-initialization(apply_bl(" & BLName & "," & COISObj$ &
","
    Predicate$ = Predicate$ & "[" & List1$ & "," & List2$ & "," & List3$ &
"])). "
    Print #FNum1, Predicate$, nl

    '/* Make: 'apply_bl('bl_dataase',[D,'Database',_],[CompSet,LinkSet,
ContentSpec]):-"
    '/* Using BLCOISObj: 'From' specifier
    '/* apply_bl(BLName, BLCOISObj, [CompSet, LinkSet, Content]):-"
    Predicate$ = "apply_bl(BLName," & BLCOISObj$ & ", [CompSet, LinkSet,
Content]):-"
    Print #FNum1, Predicate, nl

    '/* Make: "bagof([[D|T],'Table',T],compset([[D|T],'Table',T]),CompSet),
"
    '/* New: "bagof(BLCOISObj2,compset(BLCOISObj + BLCOISObj2),CompSet), "
    If CompSet$ <> "NONE" Then
'/* Handle only single CompSet only */
p1% = InStr(1, CompSet$, "*")
If p1% <> 0 Then
    '/* Eliminate "*" at end */
    CompSet$ = Mid(CompSet$, 1, Len(CompSet$) - 1)
End If
Predicate$ = "bagof(" & BLCOISObj2$ & ", compset(" & CompSet$ & "),
CompSet), "
Print #FNum1, Spaces, Predicate, nl
    End If
    If linkset$ <> "NONE" Then
```

```
'/* Handle only single CompSet only */
p1% = InStr(1, linkset$, "*")
If p1% <> 0 Then
    '/* Eliminate "*" at end */
    linkset$ = Mid(linkset$, 1, Len(linkset$) - 1)
End If
Predicate$ = "bagof(" & linkset$ & ", linkset(" & linkset$ & "),
LinkSet), "
Print #FNum1, Spaces, Predicate, nl
    End If
    Content1$ = Content$
    If Content$ <> "NONE" Then
'/* Handle only single Content only */
p1% = InStr(1, Content$, "*")
If p1% <> 0 Then
    '/* Eliminate "*" at end */
    Content$ = Mid(Content$, 1, Len(Content$) - 1)
End If
'/* For records, add BLCOISID to Content "C" ***/
Content1 = BLCOISID$
Content1$ = "[" & Content1$ & "," & Content$ & "]"
Predicate$ = "bagof(" & Content$ & ", content(" & Content1$ & "),
Content), "
Print #FNum1, Spaces, Predicate, nl
    End If

    '/* Make control predicates */
    Predicate$ = Spaces$ & "fcreate('compset.pl',0), " & nl
    Predicate$ = Predicate$ & Spaces$ & "tell('compset.pl'), " & nl
    Predicate$ = Predicate$ & Spaces$ & "write(CompSet),nl, " & nl
    Predicate$ = Predicate$ & Spaces$ & "told, " & nl
    Predicate$ = Predicate$ & Spaces$ & "fcreate('linkset.pl',0), " & nl
    Predicate$ = Predicate$ & Spaces$ & "tell('linkset.pl'), " & nl
    Predicate$ = Predicate$ & Spaces$ & "write(LinkSet),nl, " & nl
    Predicate$ = Predicate$ & Spaces$ & "told, " & nl
    Predicate$ = Predicate$ & Spaces$ & "fcreate('content.pl',0)," & nl
    Predicate$ = Predicate$ & Spaces$ & "tell('content.pl'), " & nl
    Predicate$ = Predicate$ & Spaces$ & "write(Content),nl, " & nl
    Predicate$ = Predicate$ & Spaces$ & "told. " & nl
    Print #FNum1, Predicate$

    '/* Make: compset(CompSet):- MappingRule
    Predicate$ = "compset(" & CompSet$ & "):-" & nl
    Predicate$ = Predicate$ & Spaces$ & MappingRule$ & " " & nl
    Print #FNum1, Predicate$

    Predicate$ = "linkset(" & linkset$ & "):-" & nl
    Predicate$ = Predicate$ & Spaces$ & MappingRule$ & " " & nl
    Print #FNum1, Predicate$

    Predicate$ = "content(" & Content1$ & "):-" & nl
    Predicate$ = Predicate$ & Spaces$ & MappingRule$ & " " & nl
    Print #FNum1, Predicate$
    Close #FNum1

    '/* Invoke BLEngine *./
    '/* Return when data ready */
    Call BLEngine_SendProlog

    '/* Construct CompSet, LinkSet, Content from Prolog files */
    Call BLEngine_GetPrologData(CompSet$, linkset$, Content$)

    '/* If only one object in CompSet, directly show it */
    '/* Otherwise build SYSQueryResults from CompSet */
    p1% = InStr(1, CompSet, "<")
    If p1% <> 0 Then
p2% = InStr(p1 + 1, CompSet, "<")
If p2% <> 0 Then
    '/* Construct table SYSQueryResult from CompSet */
    '/* Update and send back Msg inside it */
    Call BLEngine_BuildQueryResult(Msg$, AppDB, CompSet$)
Else
    '/* Build CompBL result for single object */
    '/* This is useful for structure links */
    Call BLEngine_BuildCompBL(Msg$, AppDB, CompSet)
End If

    End If


    '/* Update Msg */
    'AppName$ = Msg_GetTagValue(Msg$, "AppName")
    'Call Msg_SetTagValue(Msg$, "COISID", AppName$)
    'Call Msg_SetTagValue(Msg$, "CompSet", CompSet$)
    'Call Msg_SetTagValue(Msg$, "LinkSet", linkset$)
    'Call Msg_SetTagValue(Msg$, "Content", Content$)

End Sub

Sub BLEngine_BuildCOISObj (COISObj$, COISID$, COISType$, COISLabel$,
AddQuotes%)
'/* Build COISObj using nested "[", "]" */
'/* For BLCOISObj, not adding ' */
    Dim p1%, p2%, Counter%, i%
    Dim TempID$, DBName$, TableName$, RecordKey$, FieldName$

    '/* Build ID */
    TempID$ = ""

    Call DB_SeparateCOISID(COISID$, DBName$, TableName$, RecordKey$,
FieldName$, "")

    If DBName$ <> "" Then
If AddQuotes% = True Then
    TempID$ = TempID$ & "'" & DBName & "'"
Else
    TempID$ = TempID$ & DBName
End If
    End If
    If TableName$ <> "" Then
If AddQuotes% = True Then
    TempID$ = TempID$ & ",'" & TableName & "'"
Else
    TempID$ = TempID$ & TableName
End If
    End If
    If RecordKey$ <> "" Then
If AddQuotes% = True Then
    TempID$ = TempID$ & ",'" & RecordKey$ & "'"
Else
    TempID$ = TempID$ & RecordKey$
End If
    End If
    If FieldName$ <> "" Then
If AddQuotes% = True Then
    TempID$ = TempID$ & ",'" & FieldName & "'"
Else
    TempID$ = TempID$ & FieldName
End If
    End If
    TempID$ = "[" & TempID$ & "]"

    '/* Build COISObj */
    If AddQuotes% = True Then
COISObj$ = "[" & TempID$ & ",'" & COISType$ & "'," & COISLabel$ &
"]"
    Else
COISObj$ = "[" & TempID$ & ",'" & COISType$ & "'," & COISLabel$ & "
]"
    End If
End Sub

Sub BLEngine_BuildCompBL (Msg$, AppDB As Database, CompSet$)
'/* New: Build LinkBL result with single Obj in CompSet */
'/* as if we were applying a CompBL */
'/* Build table SYSQueryResult from CompSet$ */
'/* Format: CompSet = <Small School\\MasterStudent,Table,_8905874>,<...>
'/* Called by BLEngine_BLLink() for reference links */
'/* If only one object in CompSet, directly show table or database
    Dim COISID$, COISType$
    Dim p1%, p2%, p3%

    '/* Find tablenames or db names */
    p1% = InStr(1, CompSet$, "<")
    p2% = InStr(p1, CompSet$, ",")
    p3% = InStr(p2 + 1, CompSet$, ",")
    If p1% <> 0 And p2% <> 0 And p3 <> 0 Then
COISID$ = Mid(CompSet$, p1 + 1, p2 - p1 - 1)
COISType$ = Mid(CompSet$, p2 + 1, p3 - p2 - 1)
    End If

    '/* Update Msg */
    Call Msg_SetTagValue(Msg$, "COISID", COISID$)
    Call Msg_SetTagValue(Msg$, "COISType", COISType$)

    Select Case LCase(COISType$)
    Case "table"
Call DBHand_BLTable(Msg$, AppDB)
    Case "record"
Call DBHand_BLRecord(Msg$, AppDB)
    Case "field"
Call DBHand_BLField(Msg$, AppDB)
    Case "database"
Call DBHand_BLDatabase(Msg$, AppDB)
    End Select

End Sub

Sub BLEngine_BuildQueryResult (Msg$, AppDB As Database, CompSet$)
'/* Build table SYSQueryResult from CompSet$ */
'/* Format: CompSet = <Small School\\MasterStudent,Table,_8905874>,<...>
'/* SYSQueryResult fields: <SYSID, TableName>
'/* Called by BLEngine_BLLink() for reference links */
'/* If only one object in CompSet, directly show table or database
    Dim ThisTableName$, SQL$, COISID$
    Dim DS As Dynaset
    Dim KeyValue%
    Dim NewTableName$, AppName$
    Dim NewField1 As New Field
    Dim NewField2 As New Field
    'Dim NewField3 As New Field
    Dim NewTable As New Tabledef
    Dim p1%, p2%

    '/* Perpare queryresult table */
    NewTableName$ = "SYSQueryResult"

    '/* Delete SYSQueryResult */
    'On Error Resume Next
    AppDB.TableDefs.Delete NewTableName$
    'On Error GoTo 0
    '/* Create a new table */
    NewTable.Name = NewTableName$
'     '/* Add a Key field */
    NewField1.Name = "SYSID"
    NewField1.Type = FIELD_TEXT       'Variant integer
    NewField1.Size = 50
    NewTable.Fields.Append NewField1
    '/* Add another field */
    NewField2.Name = "TableName"
    NewField2.Type = FIELD_TEXT       'text
    NewField2.Size = 50
    NewTable.Fields.Append NewField2
    '/* Add table to database */
    AppDB.TableDefs.Append NewTable
    '/* Open SYSQueryResult */
    SQL$ = "SELECT * FROM SYSQueryResult"
    Set DS = AppDB.CreateDynaset(SQL$)
    KeyValue% = 1

    '/* Find all tablenames */
    p1% = InStr(1, CompSet$, "<")
    p2% = InStr(p1, CompSet$, ",")
    While p1% <> 0 And p2% <> 0
COISID$ = Mid(CompSet$, p1 + 1, p2 - p1 - 1)
'/* Get Table Name */
Call DB_SeparateCOISID(COISID$, "", ThisTableName$, "", "", "")
'/* Add table name */
DS.AddNew
'/* Add Keyfield first */
DS.Fields(0) = Trim(Str(KeyValue%))
'/* Add a row */
DS.Fields(1) = ThisTableName$
DS.Update
```

```
KeyValue% = KeyValue% + 1
'/* Find next table name */
p1% = InStr(p2 + 1, CompSet$, "<")
If p1 > 0 Then p2% = InStr(p1, CompSet$, ",")
    Wend

    '/* Update Prolog DB
    Call DBHand_Initialize

    '/* Update Msg */
    AppName$ = Msg_GetTagValue(Msg$, "AppName")
    Call Msg_SetTagValue(Msg$, "COISID", AppName$ & "\\" & NewTableName$)
    Call Msg_SetTagValue(Msg$, "COISType", "table")

    '/* Call DBHand_BLTable() */
    Call DBHand_BLTable(Msg$, AppDB)

    On Error Resume Next
    DS.Close

End Sub

Sub BLEngine_FactsDB ()
'/* generate DB facts */
    Dim HandDB As Database
    Dim AppDB As Database
    Dim SQL$, FileName$, TableName$, RecordKey$, Atom$
    Dim DBName$, DBPath$, RowContent$, Content$, KeyValue$
    Dim DS As Dynaset   '/for HandDB
    Dim DS1 As Dynaset  '/for appDB
    Dim DS2 As Dynaset  '/for tables
    Dim nl
    Dim TableID%, i%, FNum%

    nl = Chr(10)
    FNum% = FreeFile

    '/* Open file */
    FileName$ = "c:\wan\ghmi\Pro386w\blengine\facts.pl"
    Open FileName$ For Output As FNum%

    '/* Find DBs */
    Set HandDB = OpenDatabase("c:\wan\ghmi\cois\rdbms\dbhand.mdb", False,
False)
    SQL$ = "SELECT * FROM DBApps WHERE SystemName = 'Small School'"
    Set DS = HandDB.CreateDynaset(SQL$)
    If DS.EOF Then
```

```
DS.Close
Exit Sub
    End If

    '/* to save space, work for Small School only */
    If Not DS.EOF Then
DBName = "Small School"
DBPath$ = DS!DBFullPath
Set AppDB = OpenDatabase(DBPath$, False, False)
'/* Create a Prolog Fact */
Atom$ = "object('" & DBName$ & "','Database'). " & nl
Print #FNum, Atom$

'/* Find tables */
For TableID% = 0 To AppDB.TableDefs.Count - 1
    '/* Skip system tables */
    If LCase(Left(AppDB.TableDefs(TableID%).Name, 4)) <> "msys"
Then
'/* Find table name */
TableName$ = AppDB.TableDefs(TableID%).Name

'/* Add a Prolog fact */
'/* object(D,T,'Table').
Atom$ = "object('" & DBName$ & "','" & TableName$ & "',
'Table'). " & nl
Print #FNum, Atom$

'/* Find records */
SQL$ = "SELECT * FROM " & TableName$
Set DS1 = AppDB.CreateDynaset(SQL$)
While Not DS1.EOF
    '/* Create facts on records */
    RecordKey$ = DB_FindKeyField(AppDB, TableName$)
    KeyValue$ = Trim(Str(DS1(RecordKey$)))

    '/* Add an atom */
    '/* property(D,T,'KeyField',K).
    Atom$ = "property('" & DBName$ & "','" & TableName$ &
"','KeyField','" & RecordKey$ & "'" & "). " & nl
    Print #FNum, Atom$

    '/* Add an atom */
    '/* object(D,T,R,'Record').
    Atom$ = "object('" & DBName$ & "','" & TableName$ &
"','" & KeyValue$ & "','Record'). " & nl
    Print #FNum, Atom$
    '/* Find record contents */
    '/* Majorly copied from old DBHand_BLRecord */
    Content$ = ""
    If Not DS1.EOF Then
'/* Find field names */
RowContent$ = ""
For i% = 0 To DS1.Fields.Count - 1
    RowContent$ = RowContent$ & MSG_COL_SEP &
DS1.Fields(i%).Name
Next i%
RowContent$ = Mid(RowContent, Len(MSG_COL_SEP) + 1)
& MSG_COL_SEP & MSG_ROW_SEP
Content$ = Content & RowContent$
    End If

    '/* Build record content */
    If Not DS1.EOF Then
RowContent$ = ""
For i% = 0 To DS1.Fields.Count - 1
    RowContent$ = RowContent$ & MSG_COL_SEP &
DS1.Fields(i%)
Next i%
RowContent$ = Mid(RowContent, Len(MSG_COL_SEP) + 1)
& MSG_COL_SEP & MSG_ROW_SEP
Content$ = Content & RowContent$
    End If

    '/* Add an atom */
    '/* property(D,T,R,'Content',C).
    '/* Format: Content = field1@@field2@@...@@##value1@@va
lue2@@... */
    '/* This can be directly used by DBHand_BLRecord */
    Atom$ = "property('" & DBName$ & "','" & TableName$ &
"','" & KeyValue$ & "','Content','" & Content$ & "'). " & nl
    Print #FNum, Atom$

    '/* Add an atom */
    '/* relation(D,T,S,'HasSchema').
    If LCase(TableName$) = "scmsystables" Then
Atom$ = "relation('" & DBName$ & "','" &
(DS1!TableName) & "','" & (DS1!SchemaName) & "','HasSchema'). " & nl
Print #FNum, Atom$
    End If

    '/* Add an atom */
    '/* object(D,S,'Schema').
    '/* relation(S1,S2,'ERRelation').
    If LCase(TableName$) = "scmsyserschemata" Then
Atom$ = "object('" & DBName$ & "','" &
(DS1!SchemaName) & "','Schema'). " & nl
Print #FNum, Atom$

'/* Add ER relation() */
If (DS1!SchemaType) = "Relation" Then
    Atom$ = "relation('" & (DS1!SchemaName1) &
"','" & (DS1!SchemaName2) & "','ERRelation'). " & nl
    Print #FNum, Atom$
End If
    End If

    '/* Get next record */
    DS1.MoveNext
Wend
On Error Resume Next
DS1.Close
    End If
'/* Get next table
Next TableID%
On Error Resume Next
AppDB.Close
'DS.MoveNext
    End If
    Close #FNum

    On Error Resume Next
    DS.Close
    HandDB.Close
    AppDB.Close
End Sub

Sub BLEngine_GetPrologData (CompSet$, linkset$, Content$)
'/* Construct BL execution result from Prolog files */
'/* files: compset.pl, linkset.pl, content.pl */
    Dim FNum%
    Dim FileName$

    CompSet = ""
    linkset = ""
    Content = ""

    '/* Open file */
    On Error Resume Next
    FileName$ = "c:\wan\ghmi\pro386w\blengine\compset.pl"
    FNum% = FreeFile
```

```
     Open FileName$ For Input As #FNum
     Line Input #FNum, CompSet$
     Close #FNum
     '/* Polish output from Prolog to HTE format
     Call BLEngine_PolishCompSet(CompSet$)

     '/* Open file */
     FileName$ = "c:\wan\ghmi\pro386w\blengine\linkset.pl"
     FNum% = FreeFile
     Open FileName$ For Input As #FNum
     Line Input #FNum, linkset$
     '/* Polish output from Prolog to HTE format
     Call BLEngine_PolishLinkSet(linkset$)

     '/* Open file */
     FileName$ = "c:\wan\ghmi\pro386w\blengine\Content.pl"
     FNum% = FreeFile
     Open FileName$ For Input As #FNum
     Line Input #FNum, Content$
     '/* Polish output from Prolog to HTE format
     Call BLEngine_PolishContent(Content$)

End Sub

Sub BLEngine_PolishCompSet (PrologOutPut$)
'/* Polish Prolog output CompSet to HTE format */
'/*[[[Appartment],Table,[Appartment]],[[CourseNotes],Table,[CourseNotes]]]
*/
    Dim p1%, p2%, p3%
    Dim S$

    S$ = PrologOutPut$

    '/* Eliminate outer "[", "]" */
    S$ = Mid(S, 2, Len(S$) - 2)

    '/* Replace "[]", to "\\"  */
    p2% = InStr(1, S$, "[[")
    While p2 <> 0
Mid(S, p2, 2) = " ["
p3 = InStr(p2, S, ",")
p1 = InStr(p2, S, "]")
Mid(S, p1, 1) = " "
While (p3 <> 0 And p3 < p1)
    S = Mid(S, 1, p3 - 1) & "\\" & Mid(S, p3 + 1)
    p3 = InStr(p3, S, ",")
Wend
```

```
p2% = InStr(1, S$, "[[")
    Wend
    '/* Replace "[]" to "<>" */
    p2% = InStr(1, S$, "[")
    While p2 <> 0
Mid(S, p2, 2) = "<"
p2% = InStr(p2 + 1, S$, "[")
    Wend
    p2% = InStr(1, S$, "]")
    While p2% <> 0
Mid(S, p2, 2) = ">"
p2% = InStr(1, S$, "]")
    Wend

    '/* Remove spaces */
    p2% = InStr(1, S$, " ")
    While p2% <> 0
If (p2 = 1) Then
    S = Mid(S, 2)
Else
'/* if " ," ", " " <", " <" " >" "> " */
If Mid(S, p2 - 1, 1) = "," Or Mid(S, p2 + 1, 1) = "," Then
    S = Mid(S, 1, p2 - 1) & Mid(S, p2 + 1)
Else
    If Mid(S, p2 - 1, 1) = "<" Or Mid(S, p2 + 1, 1) = "<" Then
S = Mid(S, 1, p2 - 1) & Mid(S, p2 + 1)
    Else
If Mid(S, p2 - 1, 1) = ">" Or Mid(S, p2 + 1, 1) = ">" Then
    S = Mid(S, 1, p2 - 1) & Mid(S, p2 + 1)
End If
    End If
End If
End If

p2% = InStr(p2 + 1, S$, " ")
    Wend

    PrologOutPut$ = S


End Sub

Sub BLEngine_PolishContent (Content$)
'/* Polish content output from Prolog */
'/* Remove "[]" */
    Content$ = Mid(Content$, 2, Len(Content) - 1)
```

```
 End Sub

 Sub BLEngine_PolishLinkSet (linkset$)
 '/* Polish link set */
     Call BLEngine_PolishCompSet(linkset$)
 End Sub

 Sub BLEngine_SendProlog ()
 '/* Communicate with BL engine */
     Dim FileName$
     Dim FNum1%, Fnum2%
     Dim FileSize1%, FileSize2%

     '/* Initialize for consistency */
     'Call DBHand_Initialize

     '/* Delete files */
     On Error Resume Next
     FileName$ = "c:\wan\ghmi\Pro386w\blengine\compset.pl"
     Kill FileName$
     FileName$ = "c:\wan\ghmi\Pro386w\blengine\linkset.pl"
     Kill FileName$
     FileName$ = "c:\wan\ghmi\Pro386w\blengine\content.pl"
     Kill FileName$

     '/* Rewrite file */
     FileName$ = "c:\wan\ghmi\Pro386w\blengine\flagtext.pl"
     FNum1 = FreeFile
     'Open FileName$ For Random Access Read Write As #Fnum2
     Open FileName$ For Output As #FNum1
     Print #FNum1, "Y. "
     Close #FNum1
     FileSize1 = FileLen(FileName$)

     '/* Wait until data ready from Prolog */
     FileSize2 = FileLen(FileName$)
     While FileSize2 <= FileSize1
FileSize2 = FileLen(FileName$)
DoEvents
     Wend

End Sub
```

REFERENCES

1. F. Afrati and C. Koutras, "A Hypertext Model Supporting Query Mechanisms," in *HYPERTEXT: CONCEPTS, SYSTEMS AND APPLICATIONS, Proceedings of European Conference on Hypertext (ECHT'90)* (A. Rizk, N. Streitz, and J. André, eds.), Cambridge University Press, Versailles, France, pp. 52–66, Nov. 1990.

2. R. Akscyn, F. Halasz, T. Oren, V. Riley, and L. Welch, "Interchanging Hypertexts," *Panel of the Hypertext'89 Conference, ACM, Pittsburgh*, 1989.

3. R. Akscyn, D. McCracken, and E. Yoder, "KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations," *Communications of the ACM*, vol. 37, no. 7, pp. 820–835, 1988.

4. B. Amann, M. Scholl, and A. Rizk, "Querying Typed Hypertexts in Multicard/O2," in *Proceedings of the ACM European Conference on Hypermedia Technologies (ECHT'94)*, Edinburgh, Scotland, pp. 198–205, Sept. 1994.

5. K. Anderson, R.N.Taylor, and J. E.J. Whitehead, "Chimera: Hypertext for Heterogeneous Software Environments," in *Proceedings of the ACM European Conference on Hypermedia Technologies (ECHT'94)*, Edinburgh, Scotland, pp. 94–107, Sept. 1994.

6. M. Bieber, "Automating Hypermedia for Decision Support," *Hypermedia*, vol. 4, no. 2, pp. 83–110, 1992.

7. M. Bieber, "Providing Information Systems with Full Hypermedia Functionality," in *Proceedings of the Twenty-Sixth Hawaii International Conference on System Sciences (HICSS), Vol. III*, Maui, Hawaii, Jan. 1993.

8. M. Bieber, "On Integrating Hypermedia into Decision Support and Other Information Systems," *Decision Support Systems*, vol. 14, pp. 251–267, 1995.

9. M. Bieber and T. Isakowitz, "Bridge Laws in Hypertext: A Logic Modeling Approach," tech. rep., New York University, Center for Research on Information Systems, July 1991. (Technical Report #STERN IS-91-17).

10. M. Bieber and C. Kacmar, "Designing Hypertext Support for Computational Applications," *Communications of the ACM*, vol. 38, no. 8, pp. 99–107, 1995.

11. M. Bieber and S. Kimbrough, "On Generalizing the Concept of Hypertext," *Management Information Systems Quarterly*, vol. 16, no. 1, pp. 77–93, 1992.

12. M. Bieber and S. Kimbrough, "On the Logic of Generalized Hypertext," *Decision Support Systems*, vol. 11, pp. 241–257, 1994.

13. M. Bieber and J. Wan, "Backtracking in a Multiple-window Hypertext Environment," in *Proceedings of the ACM European Conference on Hypermedia Technologies (ECHT'94)*, Edinburgh, Scotland, pp. 158–166, Sept. 1994.

14. D. Bulterman, "Specifying and Support of Adaptable Networked Multimedia," *ACM Multimedia Systems*, vol. 1, no. 2, pp. 68–76, 1993.

15. A. Burger, B. Meyer, C. Jung, and K. Long, "The Virtual Notebook System," in *Hypertext'91 Proceedings*, San Antonio, TX, pp. 395–402, Dec. 1991.

16. U. Cavallaro, "HIFI - Hypertext Interface to External Databases," in *Designing User Interfaces for Hypermedia* (W. Schuler, J. Hannemann, and N. Streitz, eds.), Springer, Germany, pp. 219–224, 1995.

17. U. Cavallaro and M. Tentori, "HIFINBIPOP - Hypertext Interface to Financial Data in BIPOP Bank (Italy)," in *Designing User Interfaces for Hypermedia* (W. Schuler, J. Hannemann, and N. Streitz, eds.), Springer, Germany, pp. 236–246, 1995.

18. E. Conklin, "Hypertext: A Survey and Introduction," *IEEE Computer*, vol. 20, no. 9, pp. 17–41, 1987.

19. J. Conklin and M. Begeman, "gIBIS: A Hypertext Tool for Exploratory Policy Discussion," *ACM Transactions on Information Systems*, vol. 6, no. 4, pp. 303–331, 1988.

20. J. Conklin and M. Begeman, "gIBIS: A Tool for All Reasons," *Journal of the American Society for Information Science*, vol. 20, pp. 200–213, 1989.

21. W. Croft and H. Turtle, "A Retrieval Model for Incorporating Hypertext Links," in *Hypertext'89 Proceedings*, Pittsburgh, pp. 213–224, Nov. 1989.

22. A. V. Dam, "Hypertext'87 Keynote Address," *Communications of ACM*, vol. 31, no. 7, pp. 887–895, 1988.

23. H. Davis, W. Hall, I. Heath, G. Hill, and R. Wilkins, "Towards an Integrated Information Environment with Open Hypermedia Systems," in *Proceeding of the ACM Conference on Hypertext*, Milan, Italy, pp. 181–190, Nov. 1992.

24. H. Davis, S.Knight, and W.Hall, "Light Hypermedia Link Services: A Study of Third Party Application Integration," in *Proceedings of the ACM European Conference on Hypermedia Technologies (ECHT'94)*, Edinburgh, Scotland, pp. 41–50, Sept. 1994.

25. N. Delisle, "Neptune: A Hypertext System for CAD Applications," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, Washington, D.C., pp. 132–143, 1986.

26. D. Englebart, "Collaboration Support Provisions in Augment," in *OAC Digest: Proceedings of the 1984 AFIPS Office Automation Conference*, Los Angeles, C.A., pp. 51–58, Feb. 1984.

27. A. Fountain, W. Hall, I. Heath, and H. Davis, "MICROCOSM: An Open Model for Hypertext With Dynamic Linking," in *HYPERTEXT: CONCEPTS, SYSTEMS AND APPLICATIONS, Proceedings of European Conference on Hypertext (ECHT'90)* (A. Rizk, N. Streitz, and J. André, eds.), Cambridge University Press, Versailles, France, pp. 298–311, Nov. 1990.

28. E. Fox, Q. Chen, and R. France, "Integrating Search and Retrieval with Hypertext," in *Hypertext/Hypermedia Handbook* (E. Berk and J. Devlin, eds.), McGraw-Hill Publishing Co., Inc., New York, 1991.

29. H. Frei and D. Stieger, "Making Use of Hypertext Links when Retrieving Information," in *Proceeding of the ACM Conference on Hypertext*, Milan, Italy, pp. 102–111, Nov. 1992.

30. M. Frisse and S. Cousins, "Information Retrieval From Hypertext: Update on the Dynamic Medical Handbook Project," in *Hypertext'89 Proceedings*, Pittsburgh, pp. 199–212, Nov. 1989.

31. M. Frisse, S. Cousins, and S. Hassan, "WALT: A Research Environment for Medical Hypertext," in *Hypertext'91 Proceedings*, San Antonio, pp. 389–394, Dec. 1991.

32. R. Furuta and D. Stotts, "The Trellis Hypertext Reference Model," in *Proceeding of the Fisrt Hypertext NIST Standardization Workshop* (J. Moline, D. Benigni, and J. Baronas, eds.), US Government Printing Office, Washington, Gaithersburg, MD, pp. 83–93, Jan. 1990.

33. P. Garg, "Abstraction Mechanisms in Hypertext," *Communications of the ACM*, vol. 31, no. 7, pp. 862–870, 1988.

34. F. Garzotto, L. Mainetti, and P. Paolini, "Navigation Patterns in Hypermedia Data Bases," in *Proceedings of the Twenty-Sixth Hawaii International Conference on System Sciences (HICSS), Vol. III*, Maui, Hawaii, Jan. 1993.

35. F. Garzotto, L. Mainetti, and P. Paolini, "Adding Multimedia Collections to the Dexter Model," in *Proceedings of the ACM European Conference on Hypermedia Technologies (ECHT'94)*, Edinburgh, Scotland, pp. 70–80, Sept. 1994.

36. F. Garzotto, P. Paolini, and D. Schwabe, "HDM - A Model-Based Approach to Hypertext Application Design," *ACM Transactions on Information Systems*, vol. 11, no. 1, pp. 1–26, 1993.

37. K. Grønbæk, "Composites in a Dexter-Based Hypermedia Framework," in *Proceedings of the ACM European Conference on Hypermedia Technologies (ECHT'94)*, Edinburgh, Scotland, pp. 59–69, Sept. 1994.

38. K. Grønbæk, J. A. Hem, O. L. Madsen, and L. Sloth, "Designing Dexter-based Cooperative Hypermedia Systems," in *Hypertext'93 Proceedings*, Seattle, WA, pp. 25–38, Nov. 1993.

39. K. Grønbæk, J. A. Hem, O. L. Madsen, and L. Sloth, "Systems: A Dexter-based Architecture," *Communications of the ACM*, vol. 37, no. 2, pp. 65–74, 1994.

40. K. Grønbæk and R. Trigg, "Design Issues for a Dexter-based Hypermedia System," in *Proceeding of the ACM Conference on Hypertext*, Milan, Italy, pp. 191–200, Nov. 1992.

41. K. Grønbæk and R. Trigg, "Design Issues for a Dexter-based Hypermedia System," *Communications of the ACM*, vol. 37, no. 2, pp. 41–49, 1994.

42. C. Guinan and A. Smeaton, "Information Retrieval from Hypertext Using Dynamically Planned Guided Tours," in *Proceeding of the ACM Conference on Hypertext*, Milan, Italy, pp. 122–130, Nov. 1992.

43. A. Haake, "CoVer: A Contextual Version Server for Hypertext Applications," in *Proceeding of the ACM Conference on Hypertext*, Milan, Italy, pp. 43–52, Nov. 1992.

44. F. Halasz, "Reflections on NoteCards: Seven Issues for the Next Generation of Hypermedia Systems," *Communications of the ACM*, vol. 31, no. 7, pp. 836–852, 1988.

45. F. Halasz, T. Moran, and R. Trigg, "NoteCards in a Nutshell," in *Proceeding of the 1987 ACM Conference of Human Factors in Computer Systems (CHI+GI87)*, Totonto, Ontario, pp. 45–52, April 1987.

46. F. Halasz and M. Schwartz, "The Dexter Hypertext Reference Model," in *Proceeding of the First Hypertext NIST Standardization Workshop* (J. Moline, D. Benigni, and J. Baronas, eds.), Gaithersburg, MD, pp. 95 133, Jan. 1990.

47. F. Halasz and M. Schwartz, "The Dexter Hypertext Reference Model," *Communications of the ACM*, vol. 37, no. 2, pp. 30–39, 1994.

48. Y. Hara and R. Botafogo, "Hypermedia Databases: A Specification and Formal Language," in *DEXA'94*, pp. 521–529, 1994.

49. Y. Hara, A. Keller, and G. Wiederhold, "Relationship Abstractions for an Effective Hypertext Design: Augmentation and Globalization," in *DEXA '91*, pp. 270–274, 1991.

50. L. Hardman, D. C. Bulterman, and C. V. Rossum, "The Amsterdam Hypermedia Model: Adding Time and Context to the Dexter Model," *Communications of the ACM*, vol. 37, no. 2, pp. 50–62, 1994.

51. K. Hertwig, "The ESPRIT Project HIFI Medical Application," in *Designing User Interfaces for Hypermedia* (W. Schuler, J. Hannemann, and N. Streitz, eds.), Springer, Germany, pp. 225–235, 1995.

52. G. Hill and W.Hall, "Extending the Microcosm Model to a Distributed Environment," in *Proceedings of the ACM European Conference on Hypermedia Technologies (ECHT'94)*, Edinburgh, Scotland, pp. 32–40, Sept. 1994.

53. K. Hirata, Y. Hara, N. Shibata, and F. Hirabayashi, "Media-based Navigation for Hypermedia Systems," in *Hypertext'93 Proceedings*, Seattle, WA, pp. 159–173, Nov. 1993.

54. T. Isakowitz, "Hypermedia, Information Systems and Organizations: A Research Agenda," in *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences (HICSS), Vol. III*, Maui, Hawaii, pp. 370–379, Jan. 1993.

55. T. Isakowitz, E. Stohr, and P. Balasubramanian, "RMM: A Methodology for Structured Hypermedia Design," *Communications of the ACM*, vol. 38, no. 8, pp. 34–44, 1995.

56. C. Kacmar, "Supporting Hypermedia Services in the User Interface," *Hypermedia*, vol. 5, no. 2, pp. 85–101, 1993.

57. C. Kacmar, "A Process Approach for Providing Hypermedia Services to Existing, Non-Hypermedia Applications," *Journal of Electronic Publishing: Organization, Dissemination and Design*, (Forthcoming).

58. C. Kacmar and J. Leggett, "PROXHY: A Process-Oriented Extensible Hypertext Architecture," *ACM Transactions on Information Systems*, vol. 9, no. 4, pp. 399–419, 1991.

59. G. Landow, "Popular Fallacies About Hypertext," in *Designing Hypermedia for Learning* (D. Jonassen and H. Mandl, eds.), Springer-Verlag, New York, pp. 39–59, 1990.

60. D. Lange, "A Formal Model for Hypertext," in *Proceeding of the First Hypertext NIST Standardization Workshop* (J. Moline, D. Benigni, and J. Baronas, eds.), Gaithersburg, MD, Jan. 1990.

61. D. Lange, "Object-Oriented Hypermodeling of Hypertext Supported Information Systems," in *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences (HICSS), Vol. III*, Maui, Hawaii, Jan. 1993.

62. J. Leggett and R. Killough, "Issues in Hypertext Interchange," *Hypermedia*, vol. 3, no. 3, pp. 159–186, 1991.

63. J. Leggett and J. Schnase, "Viewing Dexter with Open Eyes," *Communications of the ACM*, vol. 37, no. 2, pp. 77–86, 1994.

64. D. Lucarella, "A Model for Hypertext-Based Information Retrieval," in *HYPERTEXT: CONCEPTS, SYSTEMS AND APPLICATIONS, Proceedings of European Conference on Hypertext (ECHT'90)* (A. Rizk, N. Streitz, and J. André, eds.), Cambridge University Press, Versailles, France, pp. 81–94, Nov. 1990.

65. D. Lucarella, S.Parisotto, and A. Zanzi, "MORE: Multimedia Object Retrieval Environment," in *Hypertext'93 Proceedings*, Seattle, WA, pp. 39–50, Nov. 1993.

66. C. Maioli, W. Penzo, S. Sola, and F. Vitali, "Using a Reference Model for Information Systems Compatibility," in *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences (HICSS), Vol. III*, Maui, Hawaii, pp. 376–385, Jan. 1994.

67. K. Malcolm, S. Poltrock, and D. Schuler, "Industrial Strength Hypermedia: Requirements for a Large Engineering Enterprise," in *Hypertext'91 Proceedings*, pp. 313–328, December 1991.

68. C. Marshall and P. Irish, "Guided Tours and On-Line Presentations: How Authors Make Existing Hypertext Intelligible for Readers," in *Hypertext'89 Proceedings*, Pittsburgh, pp. 15–42, Nov. 1989.

69. K. Østerbye, "Structure and Cognitive Problems in Providing Version Control for Hypertext," in *Proceeding of the ACM Conference on Hypertext*, Milan, Italy, pp. 33–42, Nov. 1992.

70. N. Meyrowitz, "The Missing Link: Why We're All Doing Hypertext Wrong," in *The Society of Text: Hypertext, Hypermedia, and the Social Construction of Information* (E. Barrett, ed.), MIT Press, Cambridge, pp. 107–114, 1989.

71. R. Minch, "Application and Research Areas for Hypertext in Decision Support Systems," *Journal of Management Information Systems*, vol. 6, no. 3, pp. 119–138, 1990.

72. J. Nielsen, "The Art of Navigating Through Hypertext," *Communications of the ACM*, vol. 33, no. 3, pp. 296–310, 1990.

73. H. Parunak, "Don't Link Me In: Set Based Hypermedia for Taxonomic Reasoning," in *Hypertext'91 Proceedings*, San Antonio, pp. 233–242, Dec. 1991.

74. H. Parunak, "Ordering the Information Graph," in *Hypertext/Hypermedia Handbook* (E. Berk and J. Devlin, eds.), McGraw-Hill Publishing Co., Inc., New York, pp. 299–325, 1991.

75. A. Pearl, "Sun's Link Service: A Protocol for Open Linking," in *Hypertext'89 Proceedings*, Pittsburgh, pp. 137–146, Nov. 1989.

76. V. Quint and I. Vatton, "Combining Hypertext and Structured Documents in Grif," in *Proceeding of the ACM Conference on Hypertext*, Milan, Italy, pp. 23–32, Nov. 1992.

77. V. Riley, "An Interchange Format for Hypertext Systems: The Intermedia Model," in *Proceeding of the First Hypertext NIST Standardization Workshop* (J. Moline, D. Benigni, and J. Baronas, eds.), Gaithersburg, MD, pp. 213–222, US Government Printing Office, Washington, Jan. 1990.

78. A. Rizk and L. Sauter, "Multicard: An Open Hypermedia System," in *Proceeding of the ACM Conference on Hypertext*, Milan, Italy, pp. 4–10, Nov. 1992.

79. H. Schütt and N. Streitz, "Hyperbase: A Hypermedia Engine based on a Relational Database Management System," in *HYPERTEXT: CONCEPTS, SYSTEMS AND APPLICATIONS, Proceedings of European Conference on Hypertext (ECHT'90)* (A. Rizk, N. Streitz, and J. André, eds.), Cambridge University Press, Versailles, France, pp. 95–108, Nov. 1990.

80. J. Schnase and J. Leggett, "Computational Hypertext in Biological Modelling," in *Hypertext'89 Proceedings*, Pittsburgh, pp. 181–197, Nov. 1989.

81. J. Schnase, J. Leggett, D. Hicks, P. Nürnberg, and J. A. Sánchez, "Open Architectures for Integrated Hypermedia-based Information Systems," in *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences (HICSS), Vol. III*, Maui, Hawaii, pp. 386–395, Jan. 1994.

82. J. Schnase, J. Leggett, D. Hicks, and R. Szabo, "Semantic Data Modeling of Hypermedia Association," *ACM Transaction on Information Systems*, vol. 11, no. 1, pp. 27–50, 1993.

83. J. Spivey, *The Z Notation*, Prentice-Hall International, Hertfordshire, England, 1989.

84. P. Stotts and R. Furuta, "Petri-Net-Based Hypertext: Document Structure with Browsing Semantics," *ACM Transaction on Information Systems*, vol. 7, no. 1, pp. 3-29, 1989.

85. P. Stotts, R. Furuta, and J. Ruiz, "Hyperdocuments as Automata: Trace-based Browsing Property Verification," in *Proceeding of the ACM Conference on Hypertext*, Milan, Italy, pp. 272-281, Nov. 1992.

86. N. Streitz, J. Haake, J. Hannemann, A. Lemke, W. Schüler, and M. Thüring, "SEPIA: A Cooperative Hypermedia Authoring Environment," in *Proceeding of the ACM Conference on Hypertext*, Milan, Italy, pp. 11-22, Nov. 1992.

87. M. Thüring, J. Hannemann, and J. Haake, "Hypermedia and Cognition: Designing for Comprehension," *Communications of the ACM*, vol. 38, no. 8, pp. 57-66, 1995.

88. F. Tompa, "A Data Model for Hypertext Database Systems," *ACM Transactions on Information Systems*, vol. 7, no. 1, pp. 85-100, 1989.

89. R. Trigg, "Guided Tours and Tabletops: Tools for Communicating in a Hypertext Environment," *ACM Transactions on Information Systems*, vol. 6, no. 4, pp. 398-414, 1988.

90. K. Utting and N. Yankelovich, "Context and Orientation in Hypermedia Networks," *ACM Transactions on Information Systems*, vol. 7, no. 1, pp. 58-84, 1989.

91. A. Vanzyl, "Open Hypermedia Systems Comparisons and Suggestions for Implementation Strategies," in *Proceedings of the ECHT'94 Workshop on Open Hypermedia*, Edinburgh, Scotland, Sept. 1994.

92. J. Wan, "Integrating Hypertext with Information Systems through Dynamic Mapping," Ph.D. Dissertation, New Jersey Institute of Technology, CIS Department, Newark NJ 07102, April 1996.

93. J. Wan and M. Bieber, "GHMI: A General Hypertext Data Model Supporting Integration of Hypertext and Information Systems," in *Proceedings of the Twenty-Ninth Annual Hawaii International Conference on System Sciences (HICSS), Vol. 2*, Maui, Hawaii, pp. 47-56, Jan. 1996.

94. J. Wan, M. Bieber, P. Ng, and J. Wang, "GHMI: A General Hypertext Data Model for Integrating Hypertext and Information Systems," in *Proceedings of Workshop on Intelligent Hypertext,*, in Conjunction with the ACM Conference on Information and Knowledge Management (CIKM'94), Gaithersburg, Maryland, Dec. 2 1994.

95. J. Wan, M. Bieber, J. Wang, and P. Ng, "A Logic-based Approach to Integrating Hypertext and Information Systems," *Decision Support Systems*, (submitted).

96. J. Wan, M. Bieber, J. Wang, and P. Ng, "Document Management Through Hypertext: A Logic Modeling Approach," in *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences (HICSS), Vol. III*, Maui, Hawaii, pp. 558–568, Jan. 1994.

97. J. Wan, M. Bieber, J. Wang, and P. Ng, "LHM: A Logic-based Hypertext Data Model for Integrating Hypertext and Information Systems," in *Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences (HICSS)*, Maui, Hawaii, pp. 350–359, Jan. 1995.

98. J. Wang, F. Mhlanga, Q. Liu, W. Shang, and P. Ng, "An Intelligent Documentation Support Environment," in *Proceedings of the Fifth International Conference on Software Engineering and Knowledge Engineering*, San Francisco, CA, pp. 429–436, June 1993.

99. J. Wang and P. Ng, "TEXPROS: An Intelligent Document Processing System," *International Journal of Software Engineering and Knowledge Engineering*, vol. 2, no. 2, pp. 171–196, 1992.

100. U. Wiil and J. Leggett, "Hyperform: Using Extensibility to Develop Dynamic, Open and Distributed Hypertext Systems," in *Proceeding of the ACM Conference on Hypertext*, Milan, Italy, pp. 251–261, Nov. 1992.

101. N. Yankelovich, B. Haan, N. Meyrowitz, and S. Drucker, "Intermedia: The Concept and the Construction of a Seamless Information Enviroment," *IEEE Computer*, vol. 21, no. 1, 1988.

102. Y. Zheng and M. Pong, "Using Statecharts to Model Hypertext," in *Proceeding of the ACM Conference on Hypertext*, Milan, Italy, pp. 242–250, Nov. 1992.