Chapter X

A KNOWLEDGE-BASED APPROACH TO ENGINEERING INFORMATION RETRIEVAL AND MANAGEMENT

By

William J. Rasdorf

and

Bruce R. Watson

X.1 Introduction

The operation of a computer-aided engineering environment often depends on an organization's underlying databases which act as its information sources. Such databases routinely undergo dynamic change. Changes in a database schema commonly result from the evolution of a design, from changes in the design process itself, and from changes in other subsequent downstream processes. Such continuing changes must be reflected in the database schemas and they subsequently require that application programs be updated and that online users be educated on a continuing basis.

This chapter describes a knowledge-based database interface designed to help alleviate these problems. The interface attempts to capture the knowledge that an experienced human user incorporates in his search for data in a database, i.e., it seeks to identify and use the generic knowledge needed to operate a database management system (DBMS) to retrieve data. This knowledge is used by the interface to enable both the online users and the application programs to request data without knowing the data's exact location or precisely how to ask for it. Further, the interface makes use of optional SYNONYM relations that allow the user to request data without knowing the exact name by which it is stored in the database.

X.1.1 Scenario

The use of computers in civil engineering is commonplace and computer applications continue to grow in number. The use of integrated computer-aided design systems which incorporate knowledge-based expert systems (ESs), database management systems (DBMSs), and traditional numerical analysis tools is more recently emerging in response to the need to control information flow and to facilitate information management. These systems can support a host of applications and users as Figure 1 shows.

In an integrated environment, many users, both individuals and application programs, must have access to one or more of the organization's databases to provide, use, or modify data. A typical example is a design engineer who, working at a computer-aided design (CAD) workstation, must continually access and use data. In initial design stages, for example, he must establish structural configuration, geometry, and loading conditions. This information can then be used to

Figure 1: General Computer-Aided Engineering Environment

analyze the structure to determine its forces, moments, and deflections. Simultaneously, project managers, mechanical engineers, electrical engineers, and others are making design decisions, running application programs, and drawing upon the other resources they have in hand to complete the design process.

A similar scenario exists in a parallel and, in many ways, comparable design/construction environment – manufacturing. Suppose that a mechanical engineer has specifications for a new part to be designed. He must first check a design database to determine whether or not a similar part already exists. The materials database must then be checked to determine what materials are in stock that may be used to build the part. When the part is designed, it is added to the design database by inserting a new record describing the part into one or more of the database's relations. If the part being designed is the first of a new class of parts, a new relations will need to be added to the design database schema to represent the new class of parts.

These two examples illustrate the similarities between the processes of construction, manufacturing, and assembly. The ideas presented in this chapter, therefore, are as applicable to industrial, mechanical, and aerospace engineering as they are to civil and structural engineering.

X.1.2 Problem Definition

One of the major problems facing engineering organizations is how to train personnel to use the proliferating number of computers and programs that are available for engineering computing. On a more limited scale, just teaching users how to use the various DBMSs that exist is a challenge since each has its own data definition language (DDL), data query language (DQL), and data manipulation language (DML). Many DBMSs even have a separate interface for application programs called the application program interface (API) which enables conventional programming languages to access the database. Because there are many databases that can be present in an engineering environment, it is a difficult task for an administrator to be familiar with the operation of all of them.

In addition to the difficulties associated with using database management systems, there are difficulties that are associated with structuring the information stored in databases. In design, for example, database schemas are often dynamic in nature; new structures, components, materials, catalogs, etc. must be incorporated as correctly and efficiently into the database as they are into the as-built structure. Users must be continuously informed of changes, application programs must be maintained, and the database administrator (DBA) must modify the various database schemas.

The problem to be addressed in this chapter is that it is difficult for designers and application programs to get the data that they need, when they need it, from the multiple heterogeneous DBMSs that have evolved in design and construction organizations. The diversity and dispersion of the participants in the building industry hinders a full integration of the knowledge embodied in the building process.

X.1.3 Problem Solution

One solution to the problem of efficient, accurate data retrieval is to build a general interface between users and the many sources of data available to them. This in itself is not a new suggestion; many researchers have addressed parts of this problem. In general, the interfaces that they have developed to date are best suited to environments where the structure of the database is static and does not change over time. What this chapter proposes that is different from existing work is an interface which handles the dynamic nature of design databases, enabling a designer to obtain the most accurate and up-to-date information as the structure and content of the underlying databases change. The designer should be able to simply specify his data needs and the interface should be able to interpret and satisfy those needs. To do so requires a knowledge-based approach. The structure of an ES designed to deal flexibly with engineering information needs is discussed in the following sections.

X.2 Background

As stated earlier, a number of researchers have addressed the problem of locating data that is stored in multiple heterogeneous databases. This section reviews some of the more prominent user-application-DBMS interface projects. Presented are a brief overview of KADBASE, Multibase, Mermaid, Serius-Delta, DAVID, IMDAS, FRED, and IISS. Similarities and differences between these systems are also discussed.

X.2.1 KADBASE

The KADBASE project [14,25,26,27] at Carnegie-Mellon University is a recent investigation into the problem of interfacing users, application programs, and expert systems to a number of databases. The KADBASE architecture consists of three basic parts. The first part is the Knowledge-Based System Interface (KBSI). A KBSI module is attached to each process (where a process is an identifiable computing entity, such as a user terminal session or an application program) that wishes to access data that is stored somewhere in the system. The job of a KBSI module is to take the user's queries, process them into an intermediate internal query language, and send the reformulated queries to the Network Data Access Manager (NDAM), the second component of the KADBASE architecture. NDAM is the actual interface. NDAM receives queries from the requesting KBSI, uses a global schema to locate where in the system the data is stored, issues a local query to the Knowledge-Based Database Interface (KBDBI) in question to get the data, and coordinates the transfer of data back to the requesting process. The KBDBI is the third part of KADBASE. A KBDBI is attached to each DBMS in the system and acts as a front-end processor which takes queries from NDAM and reformulates them to meet the specific requirements of the DQL of the DBMS to which it is attached. Quite simply, it is a query translator.

In summary, KADBASE utilizes knowledge-based back-ends for the applications (KBSI), knowledge-based front-ends for the DBMSs (KBDBI), and a knowledge-based interface between applications and DBMSs (NDAM) that locates data using a global schema and an intermediate internal query language. Each front-end application can use its own query language as can each back-end DBMS. Translations, or mappings, are therefore made on both sides of the interface.

X.2.2 Multibase

The Computer Corporation of America has undertaken research on a project named Multibase [30]. There are two main components in the Multibase architecture, a schema design aid and a run-time query processing system.

The schema design aid is a tool used by one who is integrating a collection of databases into a common data source (this is typically a DBA function). The purpose of the schema design aid is to help the DBA build a global schema and to help him define the mappings between the local database schemas and the global schema.

The run-time query processing system takes user queries (called global queries) and maps them across the global schema to the local schemas. It decides how to access the data and it issues to the local DBMS, through the local DBMS's API, the commands for obtaining the data. Query

optimization and resolution of database inconsistencies (such as data stored in more than one database or in different forms) is also performed by the run-time query processing system.

Multibase is designed primarily for network model databases. Multibase does not require front-end interfaces for the local DBMSs, nor does it require back-end interfaces for the processes. Unlike KADBASE, which uses a sequence (translator, interface, translator) of separate modules, Multibase does all of its translations and processing in a single module. It does, however, require that the local schemas be duplicated in the global schema and that users use a single, standard global query language known as DAPLEX.

X.2.3 Mermaid

The Mermaid project at System Development Corporation [5] is another attempt to allow a user to access multiple heterogeneous databases. Mermaid is "a front-end to relational DBMSs" [5]. It utilizes a "split architecture" in its implementation that is similar to the Multibase and the Serius-Delta architectures in that it is composed of a common user interface and a distributed back-end to handle the interface to the individual databases.

As with KADBASE and Multibase, Mermaid makes use of a global schema which replicates the local database schemas. This global schema is used by the Mermaid interface to map a global query to a particular local schema of an underlying database. Like Multibase, Mermaid makes use of a single input language for users called Ariel. It translates this user interface language into an intermediate language called DIL which is then translated into the local DBMS's DQL. One of the primary thrusts of Mermaid is the efficient implementation of queries across multiple heterogeneous DBMSs (where the DBMSs themselves may be capable of handling data distributed across multiple computers). Mermaid is designed to interface with the DBMSs even if they reside on different computers with different operating systems.

X.2.4 Serius-Delta

Serius-Delta [8] is a project being conducted at INRIA in France. It permits data access via the use of a variety of query languages which are translated into an intermediate internal query language called PETAL. Serius-Delta also uses a global schema to map from a global PETAL query to a local query on an underlying DBMS. The Serius-Delta project is similar to the Mermaid project in that it acts as a front-end to multiple underlying DBMSs. Serius-Delta, however, does not place as much emphasis on query optimization as the Mermaid project, but it does permit multiple external query languages as KADBASE does and as the remainder of the survey projects discussed below do not.

X.2.5 DAVID

The Distributed Access View Integrated Database (DAVID) is a project under development at the National Aeronautics and Space Administration (NASA) [15]. The DAVID project's intent is to provide an easy-to-use system through which one can uniformly access multiple distributed and/or heterogeneous databases. DAVID can be viewed as a DBMS of views which is built on top of

already existing DBMSs. It therefore contains a global schema that is thought to be similar in nature to those discussed previously. Communication with the individual databases is through a common DQL system that is referred to as "database logic" [15]. DAVID's job is to perform the external-to-conceptual mappings and the query translations so that the user does not need to know where the data is located or how to physically access it.

X.2.6 IMDAS

The National Bureau of Standards (NBS) in Gaithersburg, MD and the University of Florida are developing the Integrated Manufacturing Database Administration System (IMDAS) [3]. IMDAS is a distributed interface which allows users to access distributed and/or heterogeneous databases. The interface itself is distributed over computer systems and is organized in a hierarchical structure. It is therefore neither fully distributed nor is it centrally controlled. NBS refers to it as a hybrid architecture [3].

IMDAS is designed to be used in Computer-Integrated Manufacturing (CIM) systems. In particular, it will be used in NBS's Automated Manufacturing Research Facility (AMRF). NBS chose a hybrid approach to facilitate a hierarchical architecture that is able to respond to the time-critical data requirements of the manufacturing processes while maintaining system-wide reliability. Its hierarchical distributed nature makes it an ideal candidate testbed for investigating computer-aided building design applications.

As with many of the other projects, IMDAS uses a global DQL. It also uses a global schema to help it find data in the distributed systems. A unique feature of the NBS approach is that the global schema is automatically generated during system startup. The local schemas are integrated into the next higher layer of the IMDAS hierarchy as the individual computers and programs which make up IMDAS (including the back-end databases) come on line. When all of the components which make up IMDAS are online, IMDAS has an integrated global schema similar to the global schemas used by the previously mentioned projects.

The distributed, hierarchical architecture of IMDAS enables it to run (in a degraded mode) while individual components are offline, thus maintaining IMDAS availability. A prominent feature of IMDAS is its ability to handle queries at as low a level as possible, i.e., to provide as direct a path to the data as possible. If a query is issued that requires access to a DBMS which resides on a local computer, IMDAS is able to obtain the required data from the local DBMS without passing the query up through the hierarchical control structure. This helps reduce network traffic and increase system reliability.

X.2.7 FRED

GTE Laboratories' Knowledge-Based Systems Department is developing a FFrontEnd for Databases (FRED) [16]. A major thrust of the program is to create an Intelligent Database Assistant (IDA) that features a natural-language user interface.

FRED is implemented on a Xerox 1108 AI workstation and resides between the user and GTE Terminet medical information databases. The DBMSs that FRED is currently being interfaced with include Oracle, Focus, and ADR Dataquery. FRED consists of five major components: a

Communication Monitor (CM), a Natural Language Processor (NLP), a Query Planner (QP), a Database Processor (DP), and the Domain Specific Knowledge Base (DSKB).

The CM is the component that controls the intermodule interactions and the user/FRED interactions. The NLP is responsible for parsing the user's input query into a form usable by FRED. The results of this step comprise the input to the QP which plans a virtual query. This virtual query (in a special language called V/DELPHI [16]) is based on a universal relation view of the various databases that make up the entire system. This requires that FRED have access to a unified global representation of the various database schemas. Next, the virtual query produced by the QP is input into the DP. The DP is a translator that is responsible for generating a query in the target DBMS's own DQL. The last component of the FRED system is the DSKB. All of the knowledge used by the intelligent modules discussed above is located here.

X.2.8 IISS

The Integrated Information Support System (IISS) [17] is a U. S. Air Force project that was begun in 1978. It is intended to be a generic system that deals with the problems of multiple heterogeneous computers with multiple heterogeneous DBMSs operating within the scope of a single business. IISS provides users with access to the multiple machines and DBMSs and attempts to make all of the machines and data appear to the user as a single machine containing all of the data in a single, uniform format. IISS is implemented with four major subsystems. These are the Common Data Model (CDM), the Network Transaction Manager (NTM), Communications (COMM), and the User Interface (UI).

The CDM and its processors are responsible for making the data residing on the distributed machines and DBMSs look as if the data were in a single database. This is accomplished with the aid of a "conceptual schema" which is a composite of the local database schemas. The NTM is the message-passing coordinator of the IISS system. It is responsible for controlling the interactions and information flow between the IISS subsystems. COMM is the subsystem that allows the different machines in the IISS network to communicate. It uses an NBS/OSI architecture and supports both Manufacturing Automation Protocol (MAP) and Technical and Office Protocol (TOP). The UI is the subsystem that provides users with access to data anywhere it may reside in the system. It provides a virtual terminal facility and other utilities that aid the user in obtaining information from the various data management systems.

X.2.9 Summary and Comparison

The above descriptions present the similarities and differences between some of the recently developed DBMS interface systems. All but two of these efforts (KADBASE and Serius-Delta) require that the user learn and use a special-purpose global query language to communicate with the interfaces. Most of them perform a translation from the global query format into an internal query format to facilitate the mapping from a global schema to the local database schemas. Some even require a translation from application-dependent query formats to the global query format. Finally, each project depends on incorporating each local database schema into a single, common global schema which, unfortunately, results in schema duplication and its attendant consistency problems. The global schema is then used by the interface program to locate the data that the

user has requested.

The primary difference between these systems and the Adaptive Database Interface (ADI) is that ADI is an attempt to capture the knowledge of an experienced DBMS user and apply that knowledge to find and retrieve the engineering data that the requester is seeking. The goal of this investigation was to first determine what knowledge is required for a human user to operate a DBMS, and then to build a knowledge-based interface that emulates the human user. To do this, ADI must acknowledge the inherently dynamic nature of a database's schema and be able to respond to it. As such, ADI does not use a global schema and, therefore, it does not inherit the consistency problems that are just as inherent in duplicating a local schema as they are in duplicating data.

X.3 ADI Overview

The Adaptive Database Interface (ADI) is a knowledge-based database interface which attempts to satisfy engineering information needs in much the same way that a human expert would. A human expert at using databases understands how to access, interpret, and manipulate a database's schema and data. In particular, the expert knows how to use the schema to determine whether or not the required data exists in the database. Furthermore, the expert is able to make judgments about the naming conventions that the DBA has used in defining a database schema, enabling him to understand, for example, that *beam* and *girder* are equivalent entities. A human expert will also be able to satisfy a user's query in a reasonably short period of time; this is one of the qualities that makes him an expert.

The key objective of the ADI research effort is to identify, formalize, and encode the knowledge that an expert uses to operate one or more DBMSs. However, the design of ADI has other goals as well as the capture of human knowledge. From a practical point of view, it is felt that ADI should acknowledge the dynamic nature of the local design database schemas. This precludes the use of a permanent global schema as a tool to locate entities in various local databases. If the local schema is dynamic, the duplication of schemas caused by use of a permanent global schema can result in system-wide inconsistency when a local schema is changed but the global schema is not correspondingly modified in a timely fashion. Maintaining system consistency in support of schema duplication places a very large workload on the DBA. ADI was therefore designed to eliminate the need for a permanent global schema.

ADI is a layered software system that can grow in ability with successive enhancements. The first stage of the development of the system consisted of implementing the basic capabilities of query translation and naming convention hiding. The next stage of development will incorporate the more advanced capabilities of natural language processing, context sensitivity, and query abstraction.

X.3.1 Basic Capabilities

The most basic capabilities that ADI must possess are the ability to translate and build queries and the ability to work properly without forcing users to know the naming conventions of the target database. With these functions in place, ADI is "usable," and demonstrates the concepts on which it is built. These capabilities serve as a base on which the advanced capabilities can be built.

The query translation problem is twofold. First, ADI must be able to translate a user's query into its own internal format. Second, ADI must be able to translate a query from its internal format into the format of the target database's DBMS. ADI should have the ability to handle more than one target DBMS.

Hiding the database's naming conventions from the users is desirable because it frees them from having to contend with the exact names as they appear in the database. It allows the users to use names with which they are familiar rather than names that may have been arbitrarily selected by the DBA. ADI accomplishes this by using an optional SYNONYM relation in each database. If a SYNONYM relation exists for a given database, then ADI can consult it to define attributes from a user's query in terms of the actual names as they appear in the target database.

X.3.2 Advanced Capabilities

In order for ADI to plan the order of search of the various databases that are available to satisfy a user's data request, it must possess a knowledge of semantic interpretation. This requires ADI to have natural-language processing abilities, the ability to solve queries in the context of the current query session, and the ability to break abstract queries into multiple, simpler queries.

Understanding natural language requires both syntactic and semantic knowledge of the language of choice and general world knowledge of the topic of interest [28]. A natural-language understanding capability will enable ADI to decide where the required data resides: first by translating a user's English query to the relational algebra format of ADI, then by using the content of the translated query to interpret the request. In doing so, ADI will be able to determine that a user is asking for information about material properties, for instance, and will therefore look in the materials and design databases rather than look in the finance and personnel databases. A syntactic translation, therefore, proceeds a simple semantic interpretation. In this case it is assumed that there is a one-to-one correspondence between the items in the English query and the items in the relational algebra query.

Language can be said to be efficient. By efficient it is meant that the same expressions may be used over and over again by different people in different situations and at different times to mean different things [4]. This property may be referred to as the context of an expression. The ability to use the context of the current query session will allow ADI to further narrow its database search paths and to dynamically improve its own performance by filling in gaps in a user's query with information from his previous queries. For example, if the user's previous queries have been about material properties in the materials database, ADI might assume that the current query is about materials and use that assumption to complete a partially specified query. This will allow ADI to process incomplete queries in much the same way as a human expert rather than rejecting the queries or asking the user for more information.

Finally, ADI should have the ability to handle multiple levels of query abstraction. This will allow ADI to interpret the meaning of high-level queries such as:

What is the size of the pipe?

For queries such as this, ADI should be able to combine conceptual decomposition with context sensitivity to determine, for example, whether the user wants the physical length, the outside diameter, the weight, or some combination of these attributes that describe size. To do so requires that ADI first interpret the meaning of the query and then translate it into one or more relational algebra format queries that will enable it to satisfy the original query. A semantic interpretation must therefore be made before the correct syntactic translations can be generated. In this case the correspondence between the items in the English and relational algebra queries is a one-to-many rather than a one-to-one mapping.

X.3.3 Assumptions

A number of assumptions were considered and fundamental design decisions were made when developing the implementation plan for ADI. The most significant of these, relative to engineering

data-management needs, are enumerated below.

- Commercially available DBMSs are to be used with no alterations or additions permitted.
- The introduction of ADI into a computing environment must not *require* the modification of previously existing application programs which access one or more of the DBMSs. In practice, products which require modification of existing programs meet with very low acceptance.
- Similarly, no interference with the actions of experienced users of the DBMSs is permitted. While ADI supplies users with a single common DQL, users who already know the DQL of an existing DBMS may prefer to continue to use the local DBMS directly. New users may prefer to use the single, common DQL supplied by ADI. In either case, the user's freedom of choice and the continuity of existing operations must be maintained.
- ADI must enable both new and experienced users to find and retrieve data without having to know where the data is stored locally or what its specific naming conventions are.
- Finally, ADI must have a minimal impact on the host organization's DBA. In particular, ADI should not introduce system-wide inconsistencies that the DBA will have to resolve on a continuing basis.

X.4 Implementation

The actual implementation of ADI is described in this section. Included are discussions of the computer system environment in which ADI currently operates, the relational DBMS model on which it is based, a portion of the database used to test ADI, the structure of the ADI interface between users and the computing environment, ADI's theory of operation, the methods ADI employs to build a variety of queries, and the current status of ADI.

X.4.1 Computer System Environment

The target computer environment for which ADI is being developed consists of a Digital Equipment Corporation (DEC) VAXstation-II running the VMS operating system. ADI is written in three languages: FORTRAN, C, and OPS83 [11]. The initial target DBMS is Boeing Computer Service's Relational Information Manager (RIM) version 5.0 [7].

X.4.2 Relational Database Management System

The RIM DBMS is based on the relational data model [6]. Associated with the data model are a set of operators that perform the insertion, deletion, modification, and retrieval functions. These concepts are discussed briefly in the following sections.

X.4.2.1 The Relational Database Model

A data model defines the overall logical structure of a database [35]. It provides a structural framework into which the data is placed. A relational data model is a collection of interrelated relations represented in two-dimensional tabular form [6] as shown in Table 1.

TUBES

Size	X dim	Y dim	Wall thk	Weight	Area	Ix	Iy
3x2	3.00	2.00	0.1875	5.59	1.64	1.86	0.98
4x2	4.00	2.00	0.1875	6.86	2.02	3.87	1.29
4x3	4.00	3.00	0.2500	10.50	3.09	6.45	4.10

Table 1: TUBES Relation from the PIPING Database

Table 1 is the TUBES relation from a structural engineering PIPING database. This table illustrates the structure of a relation. The rows of a relation are called tuples and its columns are called attributes. The domain of an attribute is the set of allowable values the attribute may possess. The values for attribute X dim in Table 1 are drawn from the domain consisting of the set of positive real numbers. Each tuple represents an individual component and contains a value for each attribute. All tuples are distinct; duplicates are not permitted [6]. Tuples and domains have no order; they

may be arbitrarily interchanged without changing the data content or the meaning of the relation. Tuples are accessed by means of a key which may be a single attribute or a combination of attributes that uniquely identifies a tuple.

A standard shorthand notation to represent relations is as follows:

RELATIONname (ATTRIBUTEname1, ATTRIBUTEname2, ...)

with the TUBES relation of Table 1 being represented as:

TUBES (Size, X.dim, Y.dim, Wall.thk, Weight, Area, Ix, Iy),

The name of the relation is listed first followed in parentheses by the names of all of its attributes. The underlined attribute of a relation is the key.

The overall logical structure of a database is called its schema. When all relations have been defined and their attributes and associated properties specified the schema is completely defined. An introduction to relational databases using structural engineering examples was presented in reference [29]. Discussions of their role and incorporation into structural design systems are given in references [9,10].

X.4.2.2 Relational DBMS Operations

The data insertion, deletion, and modification operations are part of the Data Manipulation Language (DML). These database manipulation functions have not yet been built into ADI. ADI uses the RIM Data Query Language (DQL) operators of Select and Join [7].

The Select operator is used to selectively display data from a single relation that meets the selection criteria. The criteria are represented in the Select command by boolean predicates in a where clause. All tuples in the relation for which the where clause evaluates to true are retrieved. As an example, suppose that the TUBES relation from the structural engineering database of Table 3 is to be queried. The designer wants to know which tubes have an wall thickness that is less than .20 inch. The RIM query:

SELECT SIZE, WALL.THK FROM TUBES WHERE WALL.THK LT .20

will display the SIZE and WALL.THK attributes of the subset of tuples from the TUBES relation that have wall thicknesses up to .20 inch.

The Join operator is used to create a new relation in the database by merging two existing relations. The joining is controlled by naming a common attribute in each of the relations to be joined and by specifying a boolean relationship between these attributes. The result of a join is a new relation in the database that contains all of the attributes and data from both of the original relations. Using the structural engineering database of Table 3, assume that the user wants to know the sequence of pipes, and the length of each pipe. To obtain this information the following queries are issued:

```

JOIN SCHEDULE USING PIPE WITH ARRANGEMENT USING PIPE
FORMING TEMP WHERE EQ
SELECT ALL FROM TEMP

```

This results in the creation of a new relation (TEMP) and displays all of its attributes and tuples as shown in Table 2.

TEMP

Pipe	Size	Length	Total Force	Next Pipe	X_left	X_right	Y_left	Y_right
P1	4	10.0	200.0	P2	0.0	10.0	0.0	0.0
P2	4	5.0	500.0	P3	10.0	10.0	0.0	5.0
P3	2	15.0	150.0	P4	10.0	25.0	5.0	5.0
P4	$\frac{1}{2}$	25.0	50.0	none	25.0	40.0	5.0	25.0

Table 2: TEMP Relation Resulting from Join Operation

X.4.3 Example Database

The examples used in the remainder of this section are based on a small structural engineering database of piping information (extracted from the American Institute of Steel Construction Manual for the Design, Fabrication, and Erection of Structural Steel for Buildings [1]), which includes the following relations [23]:

```

SCHEDULE (Pipe, Size, Length, Total_force),
PIPES (Size, Outside_dia, Inside_dia, Wall_thk, Weight, Area, I),
TUBES (Size, X_dim, Y_dim, Wall_thk, Weight, Area, Ix, Iy),
LOADS (Pipe, Load_type, Force), and
ARRANGEMENT (Pipe, Next_pipe, X_left, X_right, Y_left, Y_right).

```

The attributes are defined as follows:

Pipe	unique pipe identification number
Size	nominal pipe diameter in inches
Length	pipe length in feet
Total_force	total tensile force applied to pipe in pounds
Outside_dia	outside diameter of the pipe in inches
Inside_dia	inside diameter of the pipe in inches
X_dim	width of the tube in the X direction in inches
Y_dim	height of the tube in the Y direction in inches
Wall_thk	thickness of the pipe wall in inches
Weight	pipe weight in pounds per linear foot
Area	cross-sectional area of the pipe in square inches
Ix	moment of inertia of the tube in X direction in inches**4

Iy	moment of inertia of the tube in Y direction in inches**4
I	moment of inertia of the pipe in inches**4
Load_type	type of loading applied (e.g. dead load, wind load, etc.)
Force	magnitude of the applied tensile load in pounds
Next_pipe	pipe to the right of a given pipe
X_left	X coordinate of the left end of the pipe
X_right	X coordinate of the right end of the pipe
Y_left	Y coordinate of the left end of the pipe
Y_right	Y coordinate of the right end of the pipe

The SCHEDULE relation contains the schedule of pipes used on a specific project. Pipe indicates the label of each pipe, Size and Length specify its dimensions, and Total_force indicates the sum of the axial loads applied to it. The relation LOADS specifies the type of each individual load applied to a pipe (there may be more than one) and its magnitude.

The PIPES and TUBES relations describe the physical characteristics of various stock members including their wall thickness, weight, area, and moment of inertia. Additionally, for pipes, the inside and outside diameters are specified, and for tubes the X and Y outside dimensions are given.

The relation ARRANGEMENT describes the plan layout of the piping configurations. Connectivity between pipe sections is given as well as end point coordinate locations.

The data contained in the example piping database is shown in Table 3. This database adequately demonstrates the functionality of ADI.

X.4.4 Interface Structure

Figure 2 shows ADI's relationship to the other components of an engineering information system. ADI is intended to be used by both designers at terminals and by application programs. As the figure shows, ADI's position within an organization's computer system is between its users and its databases. ADI communicates with the DBMSs through the DBMS's API and its DQL.

In the initial application, ADI uses RIM as the target DBMS. ADI is viewed simply as another application program using the RIM API [7]. It neither alters RIM, nor does it interfere with existing application programs that might use the RIM API. In this manner ADI expands an organization's data retrieval capabilities without disrupting any of its ongoing operations.

ADI uses a query language based on relational algebra rather than a formal language such as SQL [6] because of its simplicity, generic nature, its wide acceptance in the field of database research, and the ongoing effort to establish it as a standard. For an example of the use of the query language, consider the structural engineering PIPING database discussed above. To use ADI to determine the nominal size, area, and moment of inertia of available pipe sections that have a diameter less than 3.00 inches, a user or application program would issue the following relational algebra style query:

```
select size, area, inertia where od < 3.0
```

SCHEDULE

Pipe	Size	Length	Total Force
P1	4	10.0	200.0
P2	4	5.0	500.0
P3	2	15.0	150.0
P4	$\frac{1}{2}$	25.0	50.0

PIPES

Size	Outside dia	Inside dia	Wall thk	Weight	Area	I
$\frac{1}{2}$	0.840	0.622	0.109	0.85	0.25	0.017
2	2.375	2.067	0.154	3.65	1.07	0.666
4	4.500	4.026	0.237	10.79	3.17	7.230

TUBES

Size	X dim	Y dim	Wall thk	Weight	Area	Ix	Iy
3x2	3.00	2.00	0.1875	5.59	1.64	1.86	0.98
4x2	4.00	2.00	0.1875	6.86	2.02	3.87	1.29
4x3	4.00	3.00	0.2500	10.50	3.09	6.45	4.10

LOADS

Pipe	Load type	Force
P1	live load	200.0
P2	dead load	200.0
P2	live load	300.0
P3	dead load	150.0
P4	dead load	50.0

ARRANGEMENT

Pipe	Next pipe	X left	X right	Y left	Y right
P1	P2	0.00	10.00	0.00	0.00
P2	P3	10.00	10.00	0.00	5.00
P3	P4	10.00	25.00	5.00	5.00
P4	none	25.00	40.00	5.00	25.00

Table 3: Example Structural Engineering PIPING Database

Figure 2: ADI Environment

One of the key features of ADI is its ability to directly access a database's schema. This feature enables ADI to quickly locate data when presented with a query such as that given above. To insure that the names and locations of the attributes are known while respecting the dynamic nature of the databases' schemas, ADI reads the database schemas directly into ADI working memory and creates a time stamp, also in working memory, that identifies the name of the database, the name of the DBMS, and the date and time that the schema was read in. Note that there is *no global schema*; the local database schema is read, interpreted, and used at the time that an initial query is made. Subsequent accesses to the same local database can use the same internal representation of the local schema, provided that it has been in working memory for less than a predetermined length of time¹. System consistency is thus assured.

The internal representation of each attribute in the database is a unique identity, as shown in Figure 3. Attributes are identified by attribute name, relation name, the database in which they reside, and the DBMS which operates on that database.

In the event that an attribute name that is given in a user's query cannot be found directly by examining the internal representation of the schema, ADI reads in an optional SYNONYMS relation, if it exists, from the database (see Table 4). The information contained in the SYNONYMS relation equates any number of synonyms with the actual attributes of an underlying local database thereby

¹By accessing the working memory representation of the schema, load time is reduced; however, after a specified period of time the working memory representation of the schema must be discarded. The next access to that database causes the rebuilding of the working memory representation of the local schema. This is more reasonable than reading the local schema into working memory each time the database is to be accessed, yet it insures periodic reinitialization so that working memory remains current.


```

type attribute = element
{
    name:      symbol;
    relation:  symbol;
    database:  symbol;
    DBMS:      symbol;
}

```

Figure 3: Definition of ADI Schema Representation

equates any number of synonyms with the actual attributes of an underlying local database thereby allowing ADI to determine, for example, if attributes *girder* and *beam* are equivalent within the context of that particular database. A SYNONYMS relation provides one means of hiding naming conventions from database users. If the SYNONYMS relation does not exist, ADI must ask the user for definitions of unknown attributes.

<i>Synonym</i>	<i>Actual</i>
girder	beam
pipe_number	pipe
pipe_name	pipe
inertia	i
od	outside_dia
id	inside_dia

Table 4: Selected Synonyms from the PIPING Database

X.4.5 Theory of Operation

In ADI the knowledge that a human uses to locate database relations to satisfy queries is stored in rules in a knowledge base. In OPS83 and other expert system frameworks, these rules are cast in the standard “if ... then ...” form of production rules [13]. ADI uses the knowledge embedded in this framework to determine which relations to query to obtain the data from the underlying databases. The knowledge base also contains knowledge about how to create joins and projections to assemble the requested collection of data.

Figure 4 shows an example OPS83 rule from the ADI knowledge base. This rule is used to break a query into separate elements in ADI’s working memory so that they can be interpreted prior to searching the database. For every attribute in a query, ADI creates an *attrib* in working memory. ADI uses all of the attributes available from a user’s query, including the attributes in the where clause. Since an attribute may appear in a user’s query more than once, ADI must avoid making duplicate *attribs* in working memory. The rule in Figure 4 is used to determine if the attribute in the where clause of the user’s query can be used as an *attrib* in working memory. In particular, this rule keeps ADI from making duplicate *attribs* in working memory; human experts do the same thing. In a human expert this knowledge is not given much thought. It is, nonetheless,

an important part of the process of searching through the database schema to find the requested data. For a complete listing of the ADI knowledge base, see reference [33].

```
rule adi_va_2
--
-- Make the whereclause attribute a query attribute. This is
-- done to insure that we use all the information that is
-- available to us in the search for relation(s) to query.
--
{
    &1 (goal action = verify_attributes; status = active);
    &2 (wclause);
    ~ (attrib name = &2.watt);
    -->
    make (attrib name = &2.watt; set_by = |ADI|);
};
```

Figure 4: A Rule From the ADI Knowledge Base

The rule consists of the conditions necessary to invoke the rule, and the actions to be taken if the conditions are met. The conditions state that in order for this rule to be fired, the working memory must contain a *goal* element of active status whose action is to verify attributes. There must also exist an element known as a *wclause*, and there must not be an *attrib* element whose name is the same as the name of the *wclause* element. If the above conditions are met, then ADI can create an *attrib* in working memory that has the same name as the *wclause*.

The first step in interpreting a user's query is to accept the input, check it for form and completeness, and parse it into ADI's internal representation. ADI's internal representation consists of *attribs* and *wclauses*. The example query from the previous section (*select size, area, i, where od < 3.00*) would be parsed by ADI into three *attribs* (*SIZE*, *AREA*, and *INERTIA*) and a single *wclause* (*OD LT 3.00*).

Once the parser has interpreted the query, the *attribs* must be matched against the actual names used in the database. ADI does this by first reading the database's schema into working memory as described above. Next, it tries to find a match between the *attrib* and a schema attribute name. If a match is not found, ADI tries to match the *attrib* against a schema relation name. If neither of these attempts is successful, ADI looks for the *SYNONYMS* relation in the database. If it is found, the relation is read into working memory just as the schema was read. ADI then tries to match the *attrib* to a synonym in the *SYNONYMS* relation. If a match is found, the *attrib* is replaced by the actual attribute name. For example, suppose that ADI working memory contains the elements shown in Figure 5. The *attrib* *OD* does not match against any schema attribute name. It does, however, match against a synonym element. When ADI finds this match, it modifies the *attrib* name from *OD* to *OUTSIDE.DIA* which is the actual attribute name in the database. The result of the modification is shown in Figure 6.

If ADI is not able to verify that the *attrib* name is an actual attribute name, a relation name, or a synonym, it must ask the user for a definition of the *attrib*. If the user cannot define the *attrib*,

```

attrib
  name      = OD

synonym
  syn       = OD
  actual    = OUTSIDE_DIA

attribute
  name      = OUTSIDE_DIA
  relation  = PIPES
  database  = PIPING
  DBMS      = RIM

```

Figure 5: ADI Working Memory Before Synonym Match

```

attrib
  name      = OUTSIDE_DIA

synonym
  syn       = OD
  actual    = OUTSIDE_DIA

attribute
  name      = OUTSIDE_DIA
  relation  = PIPES
  database  = PIPING
  DBMS      = RIM

```

Figure 6: ADI Working Memory After Synonym Match

ADI must fail the query.

X.4.6 Query Building

Once ADI has parsed the user's query and verified the *attrs*, it can decide how to get the requested information from the system. There are, of course, many cases to consider and many ways to solve the problem. ADI solves the problem using a sequential method similar to human problem-solving methods. ADI handles three major classes of queries: single-relation queries, queries which require joins between relations, and random queries.

X.4.6.1 Single-Relation Queries

The simplest user query is one containing the key word **select** followed by a relation name (or a synonym for a relation name). One such query might be:

```
select pipes
```

ADI recognizes this single-relation class of query rapidly, notes that the user wishes to retrieve all of the attributes in the relation, and generates commands using the local DBMS's DQL to retrieve the data. ADI would generate the following commands for the RIM DBMS and PIPING database to satisfy this data request:

```
OPEN PIPING
SELECT ALL FROM PIPES
```

A slightly more complex case arises when the user's query specifies the names of attributes from a relation in addition to the relation name. Such a query might be:

```
select size, pipes, od, id
```

ADI cannot tell from the context whether the user is interested in only the named attributes **size**, **od** and **id**, or is in fact interested in the contents of the entire **PIPES** relation which is also specified. ADI makes conservative assumptions and judgments, so it will generate a request for the entire content of the relation:

```
OPEN PIPING
SELECT ALL FROM PIPES
```

Another case of a single-relation query is one in which only the names of attributes are specified. To satisfy this type of query ADI must find the single, unique relation which contains all of the *attrs*. For example, the user query given previously:

```
select size, area, i, where od < 3.00
```

will result in the ADI generated commands:

```
OPEN PIPING
SELECT SIZE, AREA, I FROM PIPES WHERE OUTSIDE_DIA LT 3.0
```

This is the case of a single-relation projection.

In all of the single-relation queries, ADI tries to find a unique relation in the database from which it can extract the required design data. If it cannot find a unique relation, it will try to form a join of two relations as discussed below.

X.4.6.2 Joins

Joins are a more interesting and complicated class of query. A join is necessary when a user's query requires information from more than one relation. A join is accomplished by combining the relations over an attribute which is common to all of the participating relations. The result of the join is a new, temporary relation. The required data is then projected from this temporary relation. After the data has been extracted, the temporary relation is deleted from the database.

As an example of a query that requires a join on two relations, consider the following:

```
select pipe_number, length, next_pipe
```

As discussed previously, ADI will parse the query and verify the *attrs*. ADI will then try to find a single relation to satisfy the request as discussed in the previous section. When it is unable to find a single relation, ADI will try to find relations that contain at least two of the requested *attrs*. In this case, the attributes PIPE and LENGTH occur in relation SCHEDULE and the attributes PIPE and NEXT_PIPE occur in relation ARRANGEMENT.

Because two distinct relations have been found, ADI will generate the following RIM commands to join them and satisfy the user's data request:

```
OPEN PIPING
JOIN SCHEDULE USING PIPE WITH ARRANGEMENT USING PIPE
  FORMING ADITMP1
SELECT PIPE LENGTH NEXT_PIPE FROM ADITMP1
REMOVE ADITMP1
```

The data to be returned will tell the requester all of the pipe numbers, their associated lengths, and the next pipe to which they connect.

X.4.6.3 Random Queries

A random query is one for which ADI cannot find any relationships between its attributes and cannot, therefore, satisfy the user's data request. Using the RIM DBMS and the PIPING database as examples again, such a query might be:

```
select od, width
```

ADI will parse this query, read in the PIPING schema and the SYNONYMS relation, and find that the *attrs* OUTSIDE.DIA and X.DIM are indeed attribute names in the PIPING database. These attributes do not occur together in any of the same relations, however. Since the attributes exhibit no relationship to one another, ADI is unable to create any commands to retrieve meaningful data using this combination of attributes. The query fails and a message is issued to the user explaining the lack of known relationships.

Closer examination of the example will show why the query must fail. While there is no relation in the database that contains both `OUTSIDE_DIA` and `X_DIM`, there are two relations that contain one of them. These are `PIPES` and `TUBES`, the schemas of which are shown in Figure 7. For these relations, *most* of the attributes are common to both relations.

SIZE	X_DIM	Y_DIM	WALL_THK	AREA	IX	IY

Relation TUBES

SIZE	OUTSIDE_DIA	INSIDE_DIA	WALL_THK	WEIGHT	AREA	I

Relation PIPES

Figure 7: Schema of Two Relations from the PIPING Database

Given that the user has requested `OUTSIDE_DIA` and `X_DIM`, how can ADI generate commands to retrieve meaningful data? If ADI were to join these two relations on the key (in this case `SIZE`, which acts as a pipe identification number), no matches will take place during the join because pipes and tubes have different identification numbers. The result is an empty join. If ADI should join on `AREA`, the result of the join would be information about pipes and tubes which have the same area. If ADI should join on `WALL_THK`, the result of the join would be information about pipes and tubes which have the same wall thickness. Clearly, ADI has no way to determine which join will result in information that first, is meaningful to the user, and second, is what he wants. The query must fail because the two requested attributes have no meaningful relationship to each other.

To clarify an ambiguous random query such as this, the user must specify a third attribute – one common to both relations. As explained in the previous section on joins, a user's query must give ADI at least this amount of information so that it can decide how to perform a join between relations.

X.4.7 Current Status

An initial prototype of ADI was completed using the expert system framework M.1 [34]. The prototype was a demonstration effort that established a proof of concept upon which the current OPS83 ADI program is based. At the present time the OPS83 prototype includes a forward chaining inference engine that fires the production rules, the interfaces to RIM and to the human user at the keyboard, and debugging tools that allow the knowledge engineer to examine the OPS83 conflict set and working memory.

The inference engine incorporates a means of determining the "cost" of firing a rule which enables, during the conflict-resolution procedure, those rules with the lowest cost to be fired before those with a higher cost. This method of controlling when rules within a conflict set (those that

match a given goal) are fired eliminates, to some extent, the requirement for rules in the knowledge base whose sole function is to direct the flow of control of the problem-solving process [2].

The ADI user interface (called the parser) has been completed. The parser is used by ADI to gather input from the user. It accepts the user's query and the answer to questions that ADI may ask. The parser incorporates some aspects of a "user friendly" environment in that it accepts a variety of word separators (usually punctuation such as commas, but extended to cover most nonalphanumeric characters). The parser is of the free-format variety and places no restrictions on the use of interword spacing. It also checks queries to determine their conformance to the relational algebra style and converts the user's query into the internal representation used by ADI.

ADI incorporates an interface to the API library of RIM [7]. This interface is used by ADI when it reads the schema or the SYNONYMS relation from a database. The interface consists of algorithmic subroutines which call the subroutines contained in the RIM API.

The ADI knowledge base contains a portion of the knowledge that a human user uses to locate, access and retrieve design data from an engineering database [34]. ADI can now solve all forms of the single-relation query, it can solve problems involving random queries, and it can handle two-relation joins. ADI's output consists of the correct sequence of commands to retrieve data from a database. At present, ADI does not invoke the RIM DQL directly.

X.5 System Evolution

Before ADI is completed its functionality must be increased. ADI must, for example, be able to use the schema manipulation operations *join* and *project* in support of performing a complete set of multi-relation data accesses. These database schema manipulation capabilities and others are being added to ADI on a continuing basis.

In addition to query capabilities, ADI must also be able to handle database *insert*, *delete*, and *update* operations. Additional levels must be added to the knowledge base to achieve this functionality. These data manipulation operations are required of a complete database interface and constitute the DML. At the present time, ADI handles only DQL operations. Future development should extend its capabilities to include DML operations.

To be fully functional, the ADI system must be able to create and communicate with subprocesses. In order for ADI to access the PIPING database using the RIM DBMS, for example, ADI must create a subprocess for RIM and communicate with RIM through the RIM DQL. This is necessary since the RIM API does not support a *join* operator. Other DBMSs that do offer a full set of operators through the API will not require a subprocess.

Since there are normally several DBMSs in an engineering environment, ADI should be extended to allow access to several commercial DBMSs. These DBMSs should be representative of the different commonly available models (relational, network, and hierarchical) as well.

When a fully functional system is developed, performance studies will be run. Of interest is the performance of ADI versus other expert systems, DBMS interfaces, and human experts for satisfying engineering information needs.

X.6 Observations

Work to date with ADI has yielded some interesting results and a few observations. First, a very important and often overlooked characteristic of an expert is the speed with which the expert can accomplish a task or solve a problem. The OPS83 version of ADI is a compiled expert system. The compiler takes great advantage of pointers and stores and accesses knowledge in somewhat the same way that a DBMS stores and accesses data. Although ADI cannot be as fast as a DBMS's own DQL, it does seem to be considerably faster than other expert systems [12]. In addition, the enhanced performance of OPS83 may offset the speed lost by copying the local database schema into working memory on the first query to a database.

Writing device drivers that connect ADI to the underlying DBMSs is a task of varying difficulty. Nearly all widely available commercial DBMSs support interfaces to applications written in a variety of programming languages. The difficulty of interfacing to the DBMS is related to the complexity of the local DBMS's API. In many cases, it may be more straightforward to run the DBMS as a subprocess to ADI and communicate with it through the DBMS's own DQL rather than through the API. The choice of whether to use the API or the DQL is mostly one of implementation. The decision must be made on an individual basis when considering the use of various DBMSs. There is nothing inherent in ADI which is partial to either approach.

The ADI system does *not* use a global schema to create a global picture of the available data resource. ADI, therefore, avoids schema redundancy and inconsistency. By avoiding the need for a global schema, ADI decreases the workload on the DBA. The DBA does not have to continuously update the global schema to reflect changes that have been made to the local schemas. ADI picks up these changes automatically from the local schemas themselves.

ADI has presently satisfied most of the design goals that were established for the prototype. It accomplishes data hiding by the use of synonyms. It permits the addition of new DBMSs with little impact on users or current applications programs. Finally, it does more than simple query translation. ADI contains a portion of the knowledge necessary to *operate* a DBMS.

X.7 Summary

Building design, construction, operation, maintenance, and control are all processes that have achieved various levels of computer use. Although the degree of computerization varies significantly, one common aspect of the computing needs of each process is an abundance of data in the form of tables, standards, project definition information, catalogs, etc. In most cases this data is stored in files which are independently used for input to stand-alone single-process application programs, such as a structural analysis application. The utility of these independent files is limited.

As concepts of integration of engineering applications evolved, the use of databases and DBMSs increased. Two issues of significant concern emerged. First, there is a need to retrieve data from many independent, possibly widely distributed databases. Second, there is a need for a uniform means of doing so. This chapter described a knowledge-based expert system that provides access to and integration of the many underlying databases needed to support the building design/construction process. The unique aspect of the expert system presented in this chapter is its capture of the knowledge that a human possesses for operating large computer programs. In doing so it formalized the levels of complexity of that knowledge and pointed out the multidisciplinary applications of the research results.

X.8 Biographical Sketches

William J. Rasdorf, P. E. is an Assistant Professor of Civil Engineering and Computer Science at North Carolina State University. He received BS and MS degrees in Architectural Engineering from Pennsylvania State University and practiced as a consulting structural engineer designing commercial buildings. He then received MS and PhD degrees in Civil Engineering from Carnegie-Mellon University specializing in computing applications in civil engineering. He was recently awarded a U. S. National Science Foundation Presidential Young Investigator Award. His research interests include engineering databases, management of engineering data, expert systems development, and artificial intelligence applications in problem solving, analysis, and design.

Bruce R. Watson, P. E. is a graduate student in the Integrated Manufacturing Systems Engineering program at North Carolina State University. He received a BS degree in Mechanical Engineering from North Carolina State University and practiced as a manufacturing engineer in factory automation before returning to graduate school. His interests lie in the application of computers in engineering situations including factory automation, the collection and management of engineering data, expert system design, and the application of artificial intelligence techniques to real-world problems.

References

- [1] American Institute of Steel Construction, 1980, *Specification for the Design, Fabrication, and Erection of Structural Steel for Buildings*, American Institute of Steel Construction, Chicago, IL, Eighth Edition.
- [2] Balaban, D. and Nelson, D., June 1985, "Flat is not Necessarily Good," *Proceedings of Computers in Engineering Conference 1985*, Volume 3, pages 361-364.
- [3] Barkmeyer, E., Mitchell, M., Mikkilineni, K., Su, S., and Lam, H., April 1986, "An Architecture for Distributed Data Management in Computer Integrated Manufacturing," National Bureau of Standards, Gaithersburg, MD, United States Department of Commerce report number NBSIR 86-3312.
- [4] Barwise, J., and Perry, J., 1983, *Situations and Attitudes*, The MIT Press, Cambridge, MA.
- [5] Brill, D. and Templeton, M., April 1984, "Distributed Query Processing Strategies in MERMAID, a Frontend to Data Management Systems," *Proceedings of International Conference on DATA ENGINEERING 1984*, pages 211-218.
- [6] Date, C. J., 1981, *An Introduction to Database Systems*, Addison-Wesley Publishing Co., Inc., Reading, MA, Third Edition.
- [7] Erickson, W. J., Gray, F. P., and Limbach, G., 1981, *Relational Information Management System*, User Guide Version 5.0, Boeing Commercial Airplane Co., Seattle, WA.
- [8] Esculier, C. L., 1984, "The SIRIUS-DELTA Architecture: A Framework for Co-Operating Database Systems," *Computer Networks*, Volume 8, pages 43-48.
- [9] Fenves, S. J., and Rasdorf, W. J., 1985, "Treatment of Engineering Design Constraints in a Relational Database," *Engineering with Computers*, Volume 1, Number 1, pages 27-37.
- [10] Fenves, S. J., and Rasdorf, W. J., October 1982, "Role of Database Management Systems in Structural Engineering," *Proceedings of the IABSE Colloquium on Informatics in Structural Engineering*, International Association of Bridge and Structural Engineers (IABSE), Bergamo, Italy, pages 229-242.
- [11] Forgy, C. L., March 1985, *OPS83 User's Manual and Report*, Production Systems Technologies, Inc., Pittsburg, PA.
- [12] Forgy, C. L., Gupta, A., Newell, A., and Wedig, R., August 1984, "Initial Assessment of Architectures for Production Systems," *Proceedings of the National Conference on Artificial Intelligence*, American Association for Artificial Intelligence, pages 116-120.
- [13] Hayes-Roth, F., Waterman, D. A., and Lenat, D. B., editors, 1983, *Building Expert Systems*, Addison-Wesley Publishing Co., Inc.
- [14] Howard, C. and Rehak, D., April 1985, "Knowledge-Based Database Management for Expert Systems," *SIGART Newsletter*, Number 92, pages 95-97.
- [15] Jacobs, B. E., 1985, *Applied Database Logic I*, Prentice Hall, Inc., Englewood Cliffs, New Jersey.

- [16] Jacobson, G., Lafond, C., Nyberg, E., and Shapiro, G., 1986, "An Intelligent Database Assistant," *IEEE Expert*, pages 65-78.
- [17] Judson, D. L., 1986, "Integrated Information Support Systems," *CIM Technology*, pages 23-30.
- [18] Lafue, G. M. E., November 1983, "Basic Decisions about Linking an Expert System with a DBMS: A Case Study," *Database Engineering*, Volume 6, Number 4, pages 56-64.
- [19] Maher, M., and Howard, C., April 1985, "Accessing Design Databases from Expert Systems," *SIGART Newsletter*, Number 92, pages 99-100.
- [20] Parker, D., July 1985, "Knowledge-Based Management Systems Simplify Engineering and Design," *Systems and Software*, pages 109-114.
- [21] Rasdorf, W., and Parks, L., February 1986, "Expert Systems and Engineering Design Knowledge," *Proceedings of the Ninth Conference on Electronic Computation*, American Society of Civil Engineers, Birmingham, AL, pages 28-42.
- [22] Rasdorf, W., and Salley, G., 1985, "Generative Engineering Databases - Toward Expert Systems," *Computers and Structures*, Pergamon Press, Volume 20, Numbers 1-3, pages 11-15.
- [23] Rasdorf, W., Ulberg, K., and Baugh, J., 1986, "A Structure-Based Model of Semantic Integrity Constraints for Relational Databases," *Engineering with Computers*, Springer Verlag, Volume 2, Numbers 1.
- [24] Rasdorf, W., and Watson, B., December 1986, "ADI: An Adaptive Database Interface for Dynamic Databases," *Proceedings of the 1986 ASME Symposium on Knowledge-Based Expert Systems for Manufacturing*, American Society of Mechanical Engineers, Production Engineering Division, Anaheim, CA.
- [25] Rehak, D., and Howard, C., November 1985, "Interfacing Expert Systems with Design Databases in Integrated CAD Systems," *Computer-Aided Design*, Volume 17, Number 9, pages 443-454.
- [26] Rehak, D., Howard, C., and Sriram, D., September 1984, "Architecture of an Integrated Knowledge Based Environment for Structural Engineering Applications," *Knowledge Engineering in Computer-Aided Design*, Budapest, Hungary.
- [27] Rehak, D., Howard, C., and Sriram, D., December 1983, "An Integrated Knowledge Based Structural Engineering Environment," Unpublished project report, *Department of Civil Engineering*, Carnegie-Mellon University, Pittsburgh, PA.
- [28] Rich, E., 1983, *Artificial Intelligence*, McGraw-Hill, Inc., New York, NY.
- [29] Schaefer, M. J., Rehak D. R., and Fenves, S. J., May 1984, "Introduction to Relational Databases Using Structural Engineering Examples," *Journal of Technical Topics in Civil Engineering*, Volume 110, Number 1, pages 1-18.
- [30] Smith, J. M., Bernstein, P. A., Dayal, U., Goodman, N., Landers, T., Lin, K. W. T., and Wong, E., 1981, "Multibase - Integrating Heterogeneous Distributed Database Systems," *AFIPS National Computer Conference Proceedings*, Volume 50, pages 487-499.

- [31] Vassiliou, Y., Matthias, J., and Clifford, J., November 1983, "Expert Systems for Business Applications - A Research Project at New York University," *Database Engineering*, Volume 6, Number 4, pages 50-55.
- [32] Vossen, G. and Brosda, V., May 1985, "A High-Level User Interface for Update and Retrieval in Relational Databases - Language Aspects," editor: S. Navathe *Proceedings of ACM-SIGMOD 1985*, pages 437-442.
- [33] Watson, B. R., and Rasdorf, W. J., August 1986, "Adaptive Database Interface: A Knowledge-Based Data Retrieval System," User Guide version 2.0, *Integrated Manufacturing Systems Engineering Institute*, North Carolina State University, Raleigh, NC 27695.
- [34] Watson, B. R., September 1986, "ADI: An Adaptive Database Interface for Dynamic Databases," Final Project Report, *Integrated Manufacturing Systems Engineering Institute*, North Carolina State University, Raleigh, NC 27695.
- [35] Ullman, J. D., 1980, *Principles of Database Systems*, Computer Science Press, Inc., Potomac, MD.