The Abstract Data Interface

by

Brian T. Wong

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 23, 2001

Author……………………………………………………………………………………
Department of Electrical Engineering and Computer Science
May 23, 2001

Certified by………………………………………………………………………………
Dr. Amar Gupta
Thesis Supervisor

Accepted by……………………………………………………………………………...
Arthur C. Smith
Chairman, Department Committee on Graduate Students

The Abstract Data Interface
by
Brian T. Wong

Submitted to the
Department of Electrical Engineering and Computer Science

May 23, 2001

in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

ABSTRACT

Data interoperability between computer systems is a critical goal for businesses. This thesis attempts to address data interoperability by proposing a design in which data producers produce an Abstract Data Interface to expose data. The interface allows data consumers to determine semantic matches, after which data producers and consumers can provide input to an Interface Generator that resolves the schematic differences. The result of the Interface Generator is an Interface that enables unfettered, interoperable data exchange between a data producer-data consumer pair.

Thesis Supervisor: Dr. Amar Gupta

Title: Co-Director, Productivity From Information Technology (PROFIT) Initiative, Sloan School of Management

Acknowledgements

I would like to begin by taking a moment to thank, in no particular order, all the people who helped me to accomplish a work of this magnitude. Without their patience and advice I would not have been able to persevere and succeed in this endeavor.

I could not have started, let alone finish, this thesis without the guidance of several people at the MITRE Corporation in Bedford. Scott Renner lent me his invaluable knowledge and direction that helped me to nail down my thoughts. As my supervisor, he also provided me with all the resources I needed to get the job done, as well as a productive work environment. Ed Housman gave me an essential introduction to the data engineering field. Arnie Rosenthal and Frank Manola provided me with bountiful brainstorming sessions that filled my head full of wonderful ideas. And of course, Beth Roberts put up with me as her officemate. You all have my gratitude.

I am also indebted to Dr. Amar Gupta, who made this entire project possible. He gave me indispensable insights into improving my research techniques and formalizing my ideas. His leadership ensured that I never lost sight of my goals.

Last but certainly not least, I would like to thank my family and friends, who were so patient and supportive of me through it all. They gave me the strength to endure through the stressful times and the happiness to enjoy the good times. Thank you.

Cambridge, Massachusetts BTW
May 23, 2001

Table of Contents

Table of Figures

1 Introduction

With the information age and the accompanying rapid advances in information technology has come an overwhelming abundance of data. However, storage of these data are far from the only challenge in dealing with these plentiful data. Perhaps more importantly, a greater challenge is the ability to share and exchange this data with others. Although the development of database management systems has increased the utility of a large store of data, such systems have not solved the problem of having a great number of separate such stores in a large company or community. Users would like to access and manipulate data from several databases, and applications may require data from a wide variety of databases. Since these databases are generally created and administered independently, physical and logical differences are common between databases [Hammer and McLeod, 1993].

Solutions to these challenges have a wide area of applicability, including telecommunications [Wu, 1999], digital libraries [Stern, 1999], finance [Bressan and Goh, 1998], taxonomy [Kitakami *et al.*, 1999], biotechnology [Jones and Franklin, 1998], geoinformation [Leclercq *et al.*, 1999], transportation [Bishr *et al.*, 1999], shipping [Phoha, 1999], and medicine [Hasselbring, 1997]. As a result, a great deal of research has been focused on changing the way in which data are accessed from databases. Instead of accessing and manipulating individual databases in isolation, the computing environment should support interoperation, permitting data consumers to access information from a variety of data sources. The interoperability should be supported without modifying the databases or reducing the autonomy of the databases, and in such a way that is relatively transparent to users and applications [Zisman and Kramer, 1996].

In this section, a brief history of research in the field of data interoperability is presented. Approaches to solving some of the problems in this field will also be discussed.

1.1 History of Data Interoperability Research

In the early days of computing, there was no formalized notion of a database. However, by the 1960's, people had begun to notice the difficulty of having thousands of programs access information held in external files. It had become clear that a set of utility functions placed between the programs and the data would help control the complexity resulting from this kind of data access. These utility functions came to be known as *access methods* and represented a first step towards removing the responsibility for managing data from the application program. [Bruce, 1992]

Computing technology progressed through a series of steps in which each achieved an increasing degree of separation between the program functions and the data access functions. Hardware-specific constraints came to be removed from the tasks of programmers. The term, "database," emerged to capture the notion that information stored within a computer could be conceptualized, structured, and manipulated independently of the specific machine on which it resided. [CICC, 1999]

A significant amount of database research during this period caused the database to evolve quickly. The hierarchical model, network model, and relational model, along with a host of other types of database models, were invented. In 1970, Dr. Edgar "Ted" Codd of IBM research wrote a landmark paper that spawned the birth of the relational database model. The paper, entitled, *A Relational Model of Data for Large Shared Data Banks*, outlined a way to use relational calculus and algebra to allow non-technical users to store and retrieve large amounts of information. Codd envisioned a database system where a user would access data using English-like commands, and where information would be stored in tables. [Codd, 1970]

In the late 1970's and 1980's, several projects and prototype development efforts to support access to heterogeneous, distributed databases were started, mostly focusing on providing methodologies for relational database design. This work addressed methodologies and mechanisms to support the integration of individual, user-oriented schemas into a single global conceptual schema. In 1986, Batini, Lenzerini, and Navathe wrote an important paper entitled, *A Comparative Analysis of Methodologies for Database Schema Integration*. This paper described some of the causes for schema diversity, and then investigated twelve of the integration methodologies and compared them on the basis of five commonly accepted integration activities: pre-integration, comparison of schemas, conforming of schemas, merging, and restructuring. [Hammer and McLeod, 1993] However, the paper did not include any of the existing specialized languages or data structures for automating these integration activities, and so the survey does not directly address the diversity problem described. Also, only a few of the methodologies presented specific tools or procedures to carry out the process of resolution beyond renaming, redundancy elimination, and generalization; more difficult resolution tasks such as translating integrity constraints, language, and data structure incompatibilities were not addressed. Furthermore, as stated by the authors:

> "None [of the methodologies] provide an analysis or proof of the completeness of the schema transformation operations from the standpoint of being able to resolve any type of conflict that can arise." [Batini, Lenzerini, and Navathe, 1986]

The absence of such analysis or proof suggests that none of the methodologies is based on any established mathematical theory; instead, they are a result of a consensus on schemas achieved by changing the view of some users.

Subsequent research data heterogeneity used varying levels of mathematical grounding and formal database theory. Several papers provided an intuitive classification of types of semantic data heterogeneity, schematic data heterogeneity, and metadata. [Kashyap and Sheth, 1996] Although not backed by formal proofs, these classifications provide a useful contribution to the study of the data heterogeneity problem because they allow researchers to focus on smaller instances of the problem. Other researchers eventually came to use more theoretical designs to specify and solve the data interoperability problem. One team of researchers investigated the use of information capacity equivalence to determine the correctness of transformed schemas by examining common tasks that require schema integration and translation, based on a set of formal definitions. [Miller, Ioannidis, and Ramakrishnan, 1993] Another approach investigated used a logic that reasons about the similarity of names in local domains, using statistical methods and a set of formal logic axioms. [Cohen, 1998]

Of the more recent methodologies to solving data heterogeneity, the tightly-coupled federation and loosely-coupled federation approaches are among the most well known. In a tightly-coupled federation, the administrator or administrators are responsible for creating and maintaining the federation and actively controlling the individual component databases. [Sheth and Larson, 1990] In a loosely-coupled federation, the user bears the responsibility of creating and maintaining the federation and no control is enforced by the federated system or by any of the administrators. The main differences between the two strategies involve who resolves conflicts and when. [Goh, 1997]

Creating a tightly-coupled federation requires schema integration. Accessing data in a loosely-coupled federation requires that a view over multiple databases be defined, or that a query using a multi-database language be defined. Schema integration, multiple database view definition, and multi-database language query definition are all affected by the problems of semantic and schematic heterogeneity.

More concrete examples of federated database systems will be presented in a later chapter. The next sections will discuss the data interoperability problem, and then the problems of semantic and schematic heterogeneity, which are subsets of the data interoperability problem.

1.2 Data Interoperability

Formally defined, data interoperability is the ability to correctly interpret data that crosses system or organizational boundaries. [Renner, 1995] From this definition, one could conclude that data interoperability problems are obstacles that frustrate efforts to correctly interpret data that crosses system or organizational boundaries. However, not all such problems are strictly data interoperability problems. Indeed, the definition of data interoperability can be justifiably vague because it must include many different kinds of interoperability, all of which are necessary.

Consider a computer user unable to download a paper containing needed information due to impaired network service. The impaired network service qualifies as a data interoperability problem because it hampers the ability of the user to interpret the data that he or she is transferring from an outside entity, across a system boundary, to himself or herself. However, this problem is primarily a communication interoperability problem, which can be solved if the viability of the underlying communications network is established.

In the same example, the computer user discovers that the paper he or she downloaded does not contain the information he actually needed. The incomplete information also qualifies as a data interoperability problem for the same reasons that the impaired network service did. However, it can be cast as a process interoperability problem, which can be solved if the user and the author of the paper properly communicate their requirements and resources.

These examples demonstrate that data interoperability includes other kinds of interoperability. Therefore, any solution that aims to solve data interoperability problems must also provide for a solution of other types of interoperability problem as well. At a minimum the solution should specify how solutions to other interoperability problems would work alongside it; at the other extreme, the overall solution could include the solutions of the other interoperability problems within itself. For instance, an interoperability solution for the example of the user downloading the paper may include assumptions of reliable network services and predetermined agreement on what is required and what will be delivered.

In solving the data focus of the data interoperability problem, the fundamental question is that of identifying objects in different data sources that are semantically related, and then of resolving the schematic differences among the semantically related objects. Two sub-problems of data interoperability are semantic heterogeneity and semantic heterogeneity. In this thesis, the term *semantic heterogeneity* will be used to refer to the identification of semantically or conceptually related objects in different databases, whereas the term *schematic* *heterogeneity* will be used to refer to the resolution of differences and similarities among semantically related objects. A solution for data interoperability must address these two critical problems. [Kashyap and Sheth, 1996]

It should be noted that, in the literature, different researchers have assigned various definitions to the terms *semantic heterogeneity* and *schematic heterogeneity.* Some of those definitions may differ from the ones used in this thesis. What is referred to in this thesis as semantic heterogeneity has been called *fundamental semantic heterogeneity*, *process interoperability,* and *semantic mismatch*. [Coulomb, 1997; Renner, 1999] What is referred to as schematic heterogeneity has been called *semantic heterogeneity*, *domain incompatibility*, and *data interoperability.* [Hammer and McLeod, 1993; Sheth and Kashyap, 1996] Nevertheless, this thesis will consistently use the terms as described in the previous paragraph, unless specified otherwise.

The next section will discuss semantic heterogeneity and schematic heterogeneity in turn, followed by approaches to each of these problems.

1.3 Semantic Heterogeneity

The *semantic heterogeneity* problem is the problem of identifying semantically related objects in different databases. Intuitively, two objects are semantically related if they represent the same real-world concept. A distinction between the *real world* and the *model world* helps in characterizing semantic heterogeneity. Objects in the model world are representations of things in the real world. Therefore, a *semantic match* is present between two objects in the model world if both objects correspond to the same real world object; a *semantic mismatch* is present instead if the two objects correspond to different real world objects. [Sheth and Kashyap, 1992] However, it turns out that determining presence of semantic match or semantic mismatch is not as straightforward as one might think from this description.

One simple example of semantic heterogeneity is a type of search problem known as the *text database discovery problem*. With the availability and popularity of online document stores, such as Dialog, Mead Data Central, Archie, WAIS, and World Wide Web, users do not have the problem of finding documents; rather, they have the problem of deciding which documents are the most relevant. Naï ve search methods that depend on word frequency can be helpful in some cases, but in other cases they fail completely. For example, a document can have a high frequency for a particular word that a user is searching for. However, the semantic meaning that the user is searching for might not match the semantic meaning of the word as used in that document, in which case the document would have little or no relevance. [Gravano, Garcia-Molina, and Tomasic, 1994]

One approach recognizes description overlap, which occurs when two different databases contain descriptions of identical real-world properties, and contends that a *property equivalence assertion* must be defined when description overlap occurs. In this approach, when a property equivalence assertion occurs between two different domains, a *conversion function* must be defined to map the two domains to each other, and then a *decision function* is applied to choose between the two properties when the values from each database disagree. [Vermeer and Apers, 1996]

Another approach defines a value known as *semantic proximity* that measures the degree to which two objects are semantically similar. In this approach, the semantic proximity of two objects is a function of the respective contexts of the objects, the abstraction used to map the domains of the objects, the domains of the objects themselves, and the states of the objects. Using a more formal definition of semantic proximity, this approach can help to identify the strongest semantic relationships to the weakest ones: semantic equivalence, semantic relationship, semantic relevance, semantic resemblance, and semantic incompatibility. [Sheth and Kashyap, 1992]

For the purposes of this thesis, the semantic heterogeneity problem will be treated largely as a process problem. As such, the implicit assumption is made that people, not automated computer systems, will be held responsible for resolving the problem of semantic heterogeneity. The assumption, however, does allow for the use of computer systems to help people to make better or more informed choices about semantic heterogeneity.

1.4 Schematic Heterogeneity

From the viewpoint of an application or other data consumer, identifying the data that are required for application operation is the first step in enabling data interoperability. After discovering a data source containing data that semantically matches the needed data, then the schematic differences between the needed data and the database data must be resolved. This resolution problem is known as *schematic heterogeneity.* A subtle point regarding the scope of schematic heterogeneity is that it necessarily also includes the problem of identifying schematically similar objects that are semantically unrelated, because this problem can produce other confounding problems similar to those produced by schematically dissimilar yet semantically related objects.

Schematic heterogeneity is a result of several factors. One of the most pervasive and universal complications is that different data engineers, programmers, and users think about data in different ways. As a result, schema implementations, which tend to capture the idiosyncrasies of individual thought processes, have a high chance of exhibiting corresponding idiosyncratic differences. As one researcher states:

> "Schematic heterogeneity arises frequently since names for schema constructs (labels within schemas) often capture some intuitive semantic information. …Even within the relational model it is more the rule than the exception to find data represented in schema constructs. Within semantic or object-based data models it is even more common." [Miller, 1998]

Most often, this phenomenon occurs during the design phase of a database schema. However, its effects can linger to confound schema and data integration efforts long after the design phase is over.

Another factor is data model heterogeneity. For example, in an Entity-Relationship (ER) model, a generalization hierarchy may be represented using *is-a* relationships, while in an extended ER model, the same construct might be modeled using generalization relationships, and in the relational model, there is no specific construct for modeling such abstractions. [Hall, 1995] Numerous other data models have also been introduced into the literature as well, each with its own constructs for representing data and data relationships. Some researchers have suggested solving instances of schematic heterogeneity due to data model heterogeneity by converting all data into the terms of one data model before trying to reconcile any differences; however, this approach has the potential to violate individual schema constraints as well as component schema autonomy. Data model heterogeneity remains a significant challenge to solving schematic heterogeneity problems.

Yet another major cause of schematic heterogeneity is incompatible design specifications between data sources. It is not often the case that two databases share the same exact purpose even though they may be related. As a result there is no reason for the two databases to necessarily have compatible design specifications. Even if two databases do share the same purpose, it is not necessarily the case that designers choose designs that are compatible with each other, since in most cases there is a virtually unlimited number of ways to create a design. For example, a travel agency reservation system could be supported by two databases. One database imposes a cardinality constraint between travelers and bookings, such that each customer can have only one booking at a time, while the other allows customers to have several reservations at once. These design specifications could be completely justified in each database's individual case, yet their incompatibilities will frustrate interoperability efforts. [Batini, 1986; Hammer and McLeod, 1993]

1.5 Types of Schematic Heterogeneity

Schematic heterogeneity conflicts fall roughly into one of the following categories: naming conflicts, structural conflicts, or interface conflicts. [Sciore, Siegel, and Rosenthal, 1994; Sheth and Kashyap, 1992; Cardiff, Catarci, and Santucci, 1997]

1.5.1 Naming Conflicts

Schemas for data models include names for the various entities, attributes, and relationships. Data engineers naturally use their own terminology when designing schemas if there are no requirements to do otherwise. Of course, there is no guarantee that people from different organizations will happen to use the same conventions. Naming conflicts refer to the redundancies and inconsistencies resulting from people incorporating their own names, terminology, and conventions into their schemas. Naming conflicts include the following:

1.5.2 Homonym Conflicts

Schemas may use the same name to describe two different concepts, resulting in inconsistency between the models. For example, one schema could use the word "tank" to refer to a container used to hold a certain amount of liquid, whereas another schema may use the same word to refer to a large, armored war vehicle.

1.5.3 Synonym Conflicts

Schemas may use different names to describe the same concept, resulting in redundant names. For example, one schema could describe a four-wheeled passenger vehicle as a "car," whereas another could describe the same object as an "automobile."

1.5.4 Precision Conflicts

Schemas have different conventions for the number of decimal places used to represent numerical values. For example, one schema could call require all numerical values to be exact to 10 places, whereas another could require all numerical values to be rounded to the nearest integer.

1.5.5 Scaling Conflicts

Schemas may attach different units of measure to numerical values. For instance, one schema could describe distances in terms of feet, whereas another could describe it in meters.

1.5.6 Structural Conflicts

Just as data engineers create different names when not restricted to particular requirements, they also choose different modeling constructs. Data engineers from different organizations can and often do think about the same data in different ways. The schemas that result from the heterogeneous methodologies may exhibit differences in modeling constructs. These differences are referred to as structural conflicts, and include the following:

1.5.7 Type Conflicts

Schemas may utilize different modeling constructs to represent the same concept. For instance, one schema could use an entity to represent a tank while another could use an attribute.

1.5.8 Key Conflicts

Schemas may assign different keys to the same concept. For example, the keys pilot-name and pilot-number could both be conceivably used to identify pilot records.

1.5.9 Cardinality Ratio Conflicts

Schemas may relate a group of concepts among themselves with different cardinality ratios. For example, one schema may call for a one-to-one ratio between "ship" and "captain," with the rationale that each ship has only one captain, whereas another schema may call for a one-to-many ratio for the same concepts, accounting for all of the officers who have ever captained that ship.

1.5.10 Interface Conflicts

Databases can be and often are designed to work with specific applications, and vice versa. When systems are designed in this way, they can be highly optimized to do the things they were specifically designed for. Unfortunately, with no regard for other potential consumers, this methodology also results in data that is difficult, if not impossible, to use with additional applications.

Single-purpose data systems continue to exist today. In the past, this type of system was easier to justify since response-time requirements and relatively slow database performance left no choice but to optimize based on a single purpose. However, such systems have much less justification today, since the underlying technology is now able to give better database performance and most enterprises have a requirement to share data at least within their internal organization, if not with external organizations as well. [Bruce, 1992]

Even if a database is not built with a specific application in mind, it often requires an application to use a specific interface to access the data it contains. Interface conflicts refer to the problems that occur when an application cannot interoperate with a database because the application does not use that database's specific interface.

1.6 Thesis Organization

This chapter discussed the problem of data interoperability, described several of the associated sub-problems of data interoperability, and provided motivations for solutions. The next chapter will provide a description of the related work that has been done towards achieving solutions to data interoperability. Several chapters afterwards will discuss the proposed ADI architecture, specification, advantages, construction, and metrics. After a thorough background is given on the ADI, a comparison of the ADI with other data interoperability approaches is provided. The final chapter ends with conclusions and suggestions for future research.

2 Approaches to Achieving Data Interoperability

The previous section discussed various types of data interoperability problems. The next section explores some of the approaches that have been researched or tried towards achieving better data interoperability. These approaches include human intervention, point-to-point interfaces, standardization, federation, ontologies, and mediation.

2.1 Human Intervention

One approach that solves data interoperability problems is human intervention. In this approach, two or more systems experiencing difficulty in exchanging data can resolve differences through communication by the *owners* of the systems—people who design, maintain or otherwise have a responsibility for the system. Such communication can be in a face to face meeting, through an instant messaging system, or by way of POTS (Plain Old Telephone Service).

Such an approach has the advantage that it is fast and inexpensive, at least in some cases. This approach does not require hardware beyond that which is already typically available in a computing environment. Also, if the owners are sufficiently knowledgeable about their systems or if the data conflict is of a type such that it can be easily resolved, then the human intervention approach can be quicker to implement and yield results than other, more automated, solutions. However, this solution is not effective when scaled for large environments that may potentially encounter a large number of data conflicts, because the human intervention will be needed for every instance of conflict. Therefore, this thesis will focus on more automated approaches, which will yield more scalable results. Nevertheless, it is important to include the human intervention approach because there are cases where it will be an effective solution.

2.2 Point to Point Interfaces

Another way to enable data interoperability is through the use of point-to-point interfaces. To connect any two systems, a piece of special software or interface can be written to negotiate data and facilitate communication between the two systems. Such software is known as a *point-to-point interface*. The advantage of the point-to-point interface is that it is almost always the simplest, fastest, and cheapest way to connect two systems in the short run. However, the benefits of this approach are outweighed by disadvantages when considering a large environment and long term effects. [Renner, 1999]

Building a point-to-point interface may or may not be easy depending on the application and database involved, but in the general case it should be considered as a non-trivial task. [Hammer et al., 1997] However, situations exist where this approach is the most cost-effective one. For example, if it is known that there will be only two systems that need to exchange and interpret data, then a point-to-point interface is almost certainly the best candidate to solve the interoperability need.

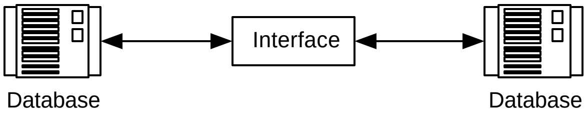

Figure 2.1. Point-to-Point Interface, Two Databases: Only One Interface to Build

Point-to-point interfaces can work well for an environment composed of a small number of individual systems. However, the number of interfaces does not grow linearly with the number of systems. In an environment consisting of *n* systems, the number of interfaces needed to inter-connect all of the *n* systems requires

$$\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2} = \mathrm{O}(n^2)$$

total interfaces, which grows with the square of the number of systems. Clearly, the point-to-point interface approach becomes infeasible when applied to large environments because of the sheer number of interfaces that must be built. [Renner, 1999]

Although a point-to-point interface can be constructed easily and cheaply to connect two systems, the maintenance of an environment full of interfaces is costly in the long run. Changes, for example, become difficult to deal with. In an environment consisting of *n* interfaces, a change to a single system requires a change in each of the up to *n* - 1 interfaces that connects to that system. (In the following diagram, each line represents an interface.)

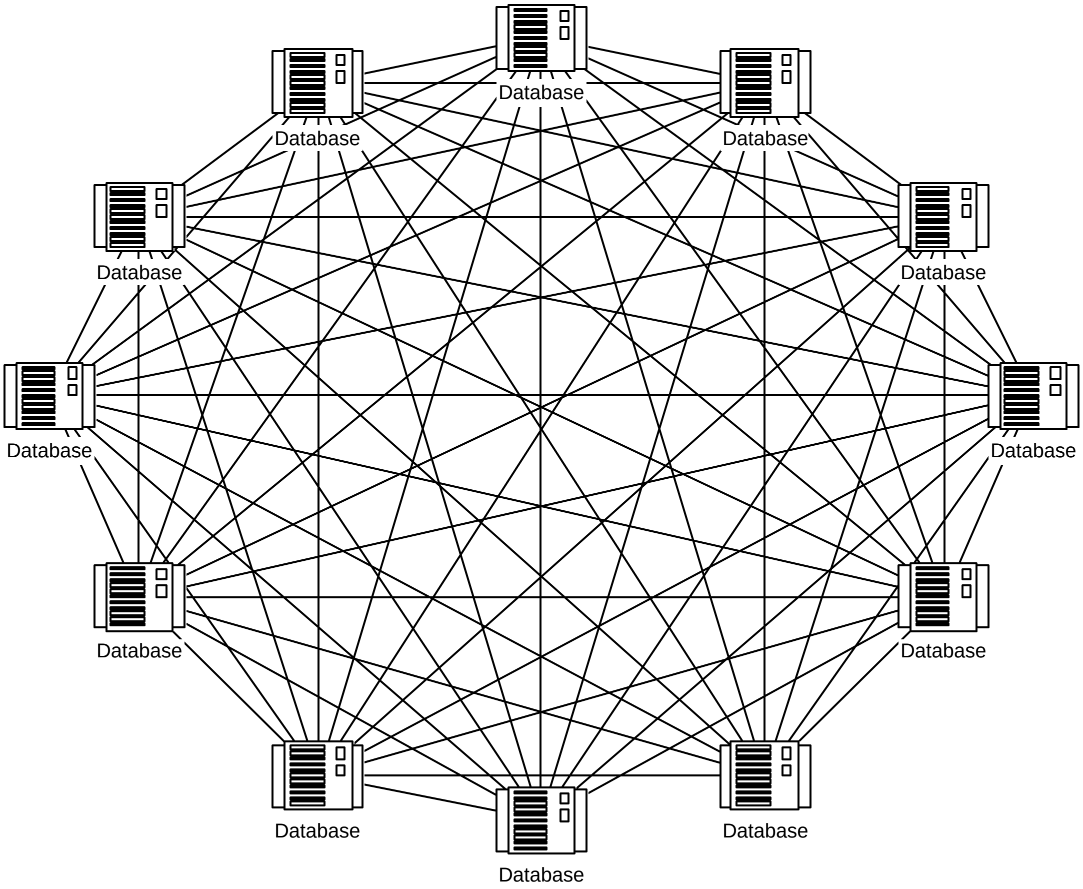

Figure 2.2. Point-to-Point Interface, Many Databases: Even More Interfaces to Build

2.3 Standardization

A way to avoid the data interoperability problem altogether, where feasible, is through standardization. Requiring the set of all intercommunicating systems to use the same data models and structure to represent data assures any system within the set that it will be able to communicate with any other system within the set. Unfortunately, there are potential significant pitfalls with the standardization approach.

Constructing a single, comprehensive data standard is difficult or even impossible. If every system is to be able to conform to a data standard, that standard must include the union of the requirements of the individual systems; if the requirements of any one system are left out, that system will not be able to conform to the standard. The complexity associated with constructing and maintaining a monolithic standard can easily exceed reasonable human limits even in modestly large environments. [Renner, 1999]

Assuming that a standard can be established for some set of systems, maintenance is still a difficult problem in its own right. Because of changing requirements, changing regulations, changing technology, or other reasons, systems always must be prepared to change. Maintaining a standard over a large number of systems implies a high rate of occurrence of changes since the changes propagate from the individual systems into the standard. Assuming that an individual system averages one change every three years, a standard that accommodates 300 individual systems must undergo 100 changes per year, or two changes per week. [Goh, 1994] Unfortunately, standards are by nature resistant to change. Once a standard is established, it is much more difficult to make the standard change around the systems than it is to make the systems conform to the standard. Maintaining a standard over a large number of systems is clearly a difficult problem because of the conflict between inevitable, frequent changes, and the tendency of standards to resist change.

When the standard is changed for whatever reason, compatibility problems can be introduced. To ensure compatibility when a standard is changed, all of the individual systems must also change. However, it is often impossible for all of the individual systems to change simultaneously, especially with a standard that encompasses a large number of systems, giving rise to interoperability problems between systems that have changed and systems that have not. [Renner, 1995]

Standardization is infeasible when all communication is not self-contained. It is rare that systems never encounter outside interaction. Communication with systems outside the standard will necessarily be more difficult than with systems within the standard.

Standardization is not an optimal approach in an environment that consists of many independent systems. Even though systems may be independent there can be a requirement to share data, for example, in an environment where many systems combine vast data stores to enable decision-making. In such an environment, the standard would be seen as intrusive to the individual systems. The special requirements of one particular system may be detrimental to another system, yet the standard must accommodate both systems.

2.4 Federation

As described earlier, the construction of a federation can aid in data interoperability. Individual databases can join in a federation, where each database extends its schema to incorporate subsets of the data held in the other member databases. Federations can be tightly-coupled or loosely-coupled. Both approaches yield a schema against which data consumers can make queries; the former placing the burden of creating and maintaining the schema on administrators, the latter placing that burden on users.

2.4.1 Tightly-Coupled Federation

In the tightly-coupled federation, a shared global schema, which is maintained by an authoritative administration, represents all of the data available in a heterogeneous environment. Tightly-coupled federations require the identification and resolution of data conflicts in advance. Once this task has been accomplished, queries can be made against the resulting global schema. Examples of tightly-coupled federation approaches include Multibase [Smith et al., 1981], PRECI* [Deen et al., 1985], ADDS [Breitbart et al., 1986], and Mermaid [Templeton et al., 1987]; more recent examples include IRO-DB [Gardarin and Sha, 1997], SIM [Motz and Fankhauser, 1998], GIM [Akoka et al., 1999], VHDBS [Wu, 1997], and SIGMA(FDB) [Saake et al., 1997].

The critical task in the creation of a tightly-coupled federation--the creation of a global schema--depends heavily on an ability to determine semantic matches between data structures. In the past, efforts to eliminate semantic ambiguities were mostly empirical, resulting in categorizations of semantic heterogeneity problems. While useful, such research does not address on a formal level what a semantic match is or is not. However, recent trends in the research of tightly-coupled federations have been aimed at formalizing notions of semantic similarity or affinity between different data models. Formal mathematical models or definitions help to better define the logic behind creation of a global schema. [Jannink et al., 1999; Castano and DeAntonellis, 2001; Kiyoki et al., 1995; Hull, 1997]

2.4.2 Loosely-Coupled Federation

In the loosely-coupled federation, the component databases do not share a global schema; rather, they attempt to detect and resolve data conflicts during data exchanges between systems. While this approach avoids the overhead of maintaining a global schema characteristic of tightly-coupled approaches, it places an extra burden of data conflict resolution on the individual data stores and data consumers, a burden that is not present in tightly-coupled approaches. Examples of loosely-coupled approaches in the literature include MRDSM [Litwin and Abdellatif, 1986], FEDDICT [Richardson and Smeaton, 1996], and MASM [Lee and Baik, 1999]. Most of the research pertaining to loosely-coupled federations focuses on the creation or refinement of query languages and query transformations.

2.5 Ontology

An ontology organizes the knowledge about data elements, such as tables and classes, in a data domain at a higher abstraction level into concepts and relationships among concepts. The lower level data elements are often ambiguous or heterogeneous, preventing data consumers from understanding the contents of a data store. Ontologies permit a data consumer to query several distributed databases in a uniform way, overcoming possible heterogeneity. [Castano and DeAntonellis, 1998] Examples of ontology-based approaches include DOME [Cui, 2000], SKAT [Mitra et al., 1999], and Linguistic Dictionaries [Kedad and Metais, 1999]. Much of the research relating to ontologies is closely related to the research on tightly-coupled federations: A common need for data interoperability approaches that rely on ontologies is the need to merge two different, yet possibly overlapping ontologies. Therefore, the need to identify semantic similarities and differences is also an important area of research.

2.6 Mediation

In the traditional mediation approach, an integrated view of data that resides in multiple databases is created and supported. The basic architecture for such an approach utilizes a component known as a mediator that typically resides on a system separate from the individual databases. A schema for the integrated view can be made available from the mediator, and queries can be made against that schema. More generally, a mediator is a system that supports an integrated view over multiple sources. [Masood and Eaglestone, 1998] According to this definition, mediators help enable tightly-coupled federation strategies.

The first approaches to mediation permitted read-only views of the data through the mediator. A natural extension that was developed was to support updates as well as read-only queries from the mediators. In its most general form, the update capability raises the view update problem-maintaining a valid view even after updates to component databases have taken place. However, updates can be supported against the integrated view provided appropriate limitations are utilized. [Keller, 1986] Different systems using the mediation approach feature a variety of techniques to ensure that the integrated view constitutes a valid schema. A mediator may involve data from a variety of information sources, including other mediators, in order to support the proper view. [Masood and Eaglestone, 1998]

Newer architectures utilizing mediators have not always strictly followed the traditional definition of mediator. Such systems have combined ideas from the other previously mentioned architectures. For example, the COIN approach [Goh, 1997] utilizes mediators that rely on data producers and consumers having explicit contexts. This approach combines elements of point-to-point interfaces and ontologies, as well as reducing the data reconciliation efforts of both tightly-coupled and loosely-coupled federation approaches. Other approaches involving mediators include AMOS [Fahl et al., 1993], TSIMMIS [Hammer and McLeod, 1993], DECA [Benslimane et al., 2000], XMLMedia [Gardarin et al., 1999], and DIOM [Liu and Pu, 1997].

2.7 Comparison

For the purposes of comparison to the ADI, several of the approaches to data interoperability will be revisited in a later chapter. The next chapter will introduce the architecture and specifications of the Abstract Data Interface. After a discussion of the ADI, a more detailed discussion of the approaches will be presented along with a comparison to the ADI.

3 Active Data Interface

The previous chapter introduced data interoperability and discussed various approaches to achieving it. This chapter introduces the Active Data Interface (ADI) concept, architecture, and design.

3.1 Active Data Interface Concept

The ADI concept arose from the following observation: In a large-scale distributed environment consisting of physically disparate applications and databases, many different systems utilize different data models and interfaces; but the same functionality is often replicated by interfaces that translate data between the various systems. An analogy in terms of previous approaches is that the point-to-point interfaces in a non-standardized environment often achieve the same purposes, even though the specific implementations are different. The following figure provides an example.

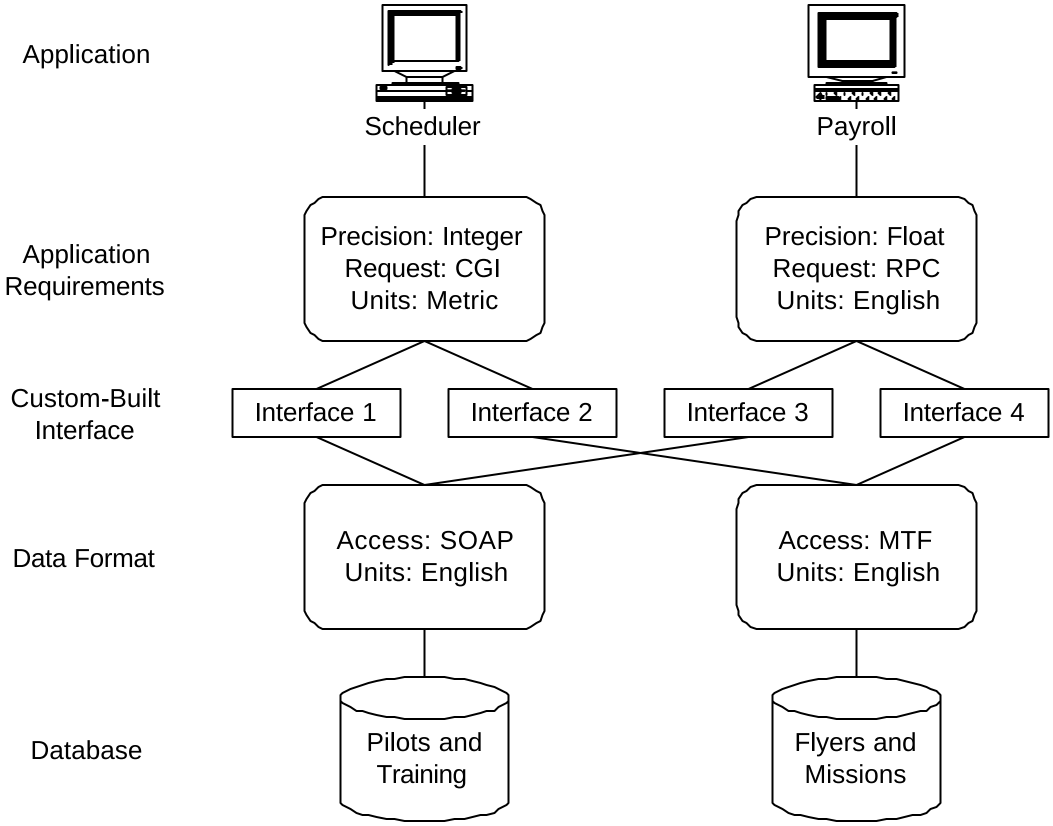

Figure 3.1. Scheduler and Payroll Example

In this example, there are two applications, a Scheduler application and a Payroll application, that need to communicate with two databases in order to produce correct reports. The diagram indicates the application requirements and data format. Application requirements are the assumptions about the data that the application developers make when creating the application. Data format indicates the way in which the relevant data are stored in the databases. Oftentimes the application requirements and the data format are not conceived with each other in mind, and therefore custom built interfaces that allow the application and databases to work together properly must be built and inserted between the database and application layers.

Together, application requirements and data format indicate the functionality required by the four interfaces in the center. Both databases contain raw data in English units. The Pilots and Training Database assumes that it will be queried through a Simple Object Access Protocol (SOAP) access mechanism. The Flyers and Missions Database assumes that it will be queried via requests in Message Text Format (MTF).

The Scheduler application communicates with the Pilots and Training Database via Interface 1, and with the Flyers and Missions Database via Interface 2. The Scheduler expects to communicate to a database through the CGI protocol, and assumes that it will receive data about pilots in metric units, rounded to the nearest integer. Interface 1 needs to translate CGI application requests into SOAP queries to communicate with the Pilots and Training Database, and convert the data from that database from English to metric units and round to the nearest integer. Similarly, Interface 2 needs to convert CGI application requests into MTF queries to communicate with the Flyers and Missions Database, and convert the data from that database from English to metric units, rounded to the nearest integer.

The Payroll application also requires interaction with custom interfaces. It communicates with the Pilots and Training Database via Interface 3, and with the Flyers and Missions Database via Interface 4. The Payroll application expects to communicate with databases through Remote Procedure Calls (RPC), and assumes that it will receive data about pilots in English units, with as much floating-point precision as possible. Therefore, Interface 3 needs to convert RPC application requests into SOAP queries to communicate with the Pilots and Training Database, and interpret data from that database with floating-point precision. Lastly, Interface 4 needs to convert RPC application requests into MTF queries to communicate with the Flyers and Missions Database, and interpret data from that database with floating-point precision. With the Payroll application there happens to be no conflict between systems of units of measurement.

This example demonstrates the repetition inherent in constructing a set of custom interfaces for communication and interaction between these applications and databases. For example, Interfaces 1 and 2 only differ in that they require different ways for the application to access the data in the databases, while the rest of the functionality is the same. As another example, Interfaces 3 and 4 also exhibit different ways to permit application access to data, but the functionality provided by the floating-point conversion is the same in both interfaces. This observation leads one to conclude that it should be possible to construct an interface generator that automatically produces customized interfaces between individual systems in the environment, without duplicating the work usually associated with constructing customized interfaces.

Such an interface generator would only be possible with the proper input. The required input includes what would be analogous to the Application Requirements and Data Format layers in the above example. In simpler language, the interface generator would have to know (1) what is needed by the application; and (2) what is available in the database. These two inputs, in some sense, form the basis of the Abstract Data Interface. Along with these inputs, the interface generator would also have to have the proper mediation tools to perform any necessary translations between application requirements and database data. These three categories of input—application requirements, database content description and mediation tools—are necessary for interface generation.

It should be noted that although these inputs are necessary, they might not be sufficient. Even though an eventual goal of a system that automatically generates these interfaces might be to operate without any human intervention, a system that would overcome such problems as process interoperability might need more inputs. However, a successful implementation of the Abstract Data Interface concept does not require that interfaces be fully automatically generated. Interface programmers would be pleased to have an interface generator that could generate even a portion of an interface, as long as it is able to prevent some amount of work from being duplicated.

3.2 Abstract Data Interface Architecture Specification

The ADI is not intended to act as a stand-alone device. Rather, it is designed to be a part of a framework that could include new applications and databases as well as legacy systems. This section describes the general architecture that includes the ADI, specifies what requirements are needed for each component, and explains how the ADI functions alongside other interoperability components.

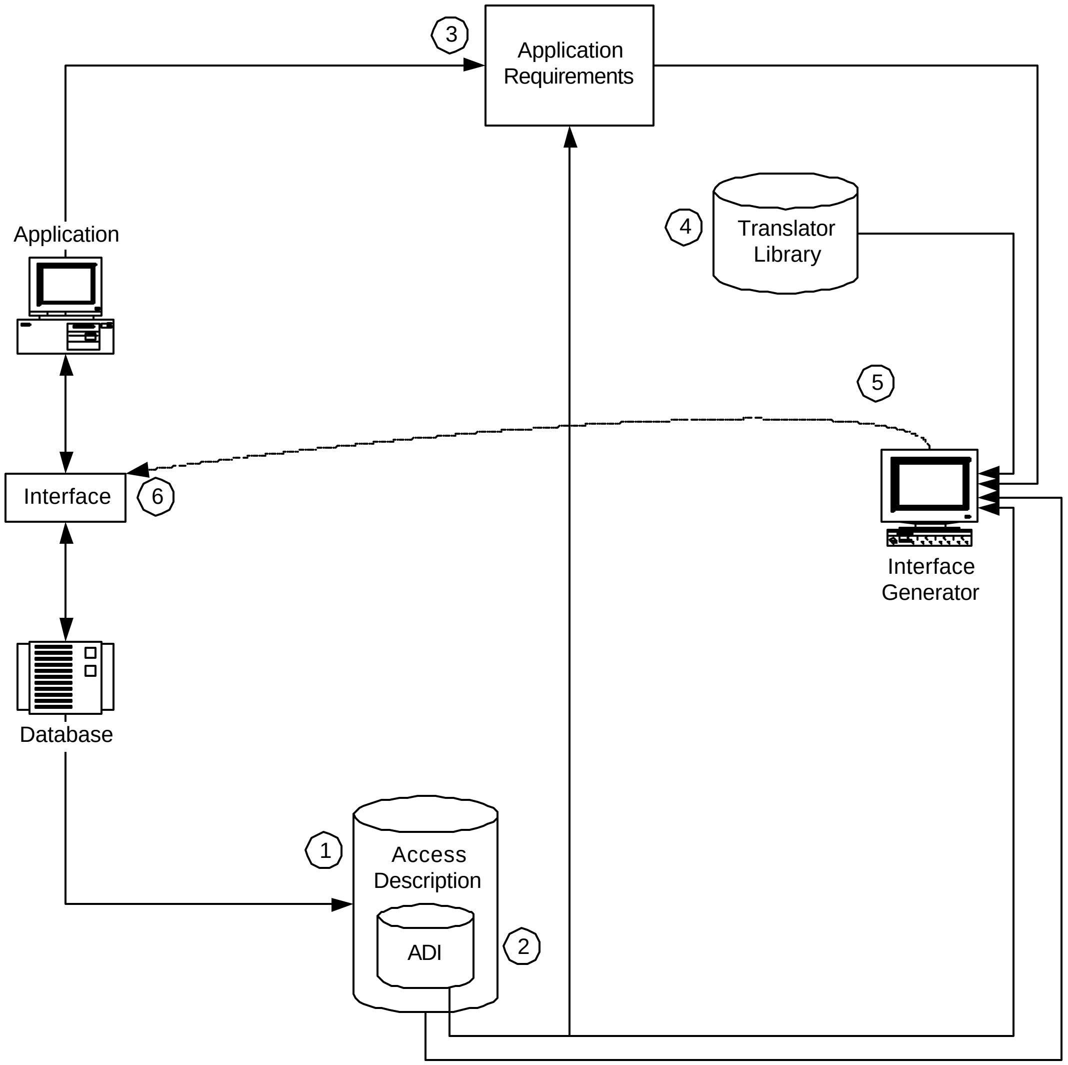

Figure 3.2. Single Application-Single Database ADI Architecture

This diagram shows an interface being generated for a single application-database pair. The generation is accomplished through the following general steps:

1. **Database Contents Publication.** The database contents are published through the ADI.

2. **Database Access Description.** The information and methods needed to access the database are described via a database model or other vehicle.

3. **Application Requirements Description.** The application requirements are selected from the ADI.

4. **Translator Library Preparation.** A collection of translation tools is made available.

5. **Interface Generation.** The interface generator produces an application-to-database interface.

6. **Interface Operation.** The application and database communicate using the newly generated interface.

The following sections will explain each of these requirements in further detail.

3.2.1 Database Contents Publication

Publishing the contents of the database is accomplished through the Abstract Data Interface (ADI). The person who maintains the database—in most cases she will be known as the Database Administrator (DBA)—is the person best equipped to construct the ADI, since she both decides what data should be published or not published, and has the knowledge to assemble such an interface.

The purpose of the ADI is threefold:

1. To allow a DBA to decide what data she wishes to expose, or make available to applications and other data consumers;
2. To allow interface programmers or application developers to easily select the data that is required by an application; and
3. To provide the Interface Generator with information that will allow it to relate the selected data to particular access methods, and access the data needed.

In order for the ADI to achieve the goal of providing an abstract data interface for programmers, the ADI must be easy to read and understand. To this end, the ADI itself will consist of a description of the data, and the relationships between that data, in the database. To the extent possible, supporting data details, that do nothing to describe what the data actually is, are to be omitted from the abstraction. Some details can in general be left out, such as numeric precision, units of measure, and value integrity constraints. In other cases, the DBA will need to make a judgment as to whether a particular attribute should or should not be included in the ADI. In addition, the names of the data fields should be changed to describe the data in as human a language as possible. Data field names that resemble human language, e. g., "Number of Aircraft," lend an intuitive notion of the data and help to better understand what data is available. In contrast, database names for data fields are frequently cryptic, e. g., "2b_noa_acf," and have no meaning except to the database designer or DBA. In the end, the ADI should be descriptive enough to allow an interface programmer or an application developer to select the data objects needed for a particular application, yet simple enough so that the same person does not see any of the underlying data storage implementation details.

To help formalize the notion of an ADI, the following structure, known as a *Data Descriptor Set* or DDS, is defined. A DDS is defined to be a set of tuples:

$$DDS \equiv \{T_1, T_2, T_3, ..., T_n\}$$

for some number of tuples n. Each tuple T is defined by:

$$T_i \equiv \langle d_i, R_i \rangle = \langle d_i, r_{i1}, ..., r_{ij} \rangle$$

for any tuple i and some number of references j. Each d represents some finite piece of data. The corresponding r's represent a reference to the d with which they are grouped. The rationale for the existence of the r's is that in general a single piece of data d cannot be useful on its own. The data must have a frame of reference or context to be useful, e. g. a list of heights is useless unless one knows which individuals each height describes. Also, there may be multiple references r, since users may have many different ways to reference the data.

An ADI is defined to be an instance of a Data Descriptor Set, or *DDS*. In addition, the ADI also has the following constraints for each piece of data d:

$$\forall i.description(d_i) = \min\{\bigcup_b description_b(d_i)\}$$
$$\forall i, j.description(r_{ij}) = \min\{\bigcup_b description_b(r_{ij})\}$$

These constraints capture the concept that the ADI ought to avoid cryptic descriptions. They require that the description of the data and references to be the simplest (minimum) descriptions possible.

3.2.2 Database Access Description

Although it is not important to include database details in an ADI—in fact it is disadvantageous to the ADI to do so—in order for an *interface* to function properly, it will have to have some means of providing the actual data specified in the ADI. In the ADI architecture, the Interface Generator generates the interface, and so the Interface Generator will also require this access. The DBA should be held responsible for the task of constructing the Database Access Description.

In order for the Database Access Description to accomplish the goal of providing a data consumer with any piece of data included in the ADI, the Database Access Description will need to provide access to the contents of the database. That is, for any piece of abstract data specified in the ADI, the description must contain all of the information needed to provide that data. Many or all of the database details that were omitted from the ADI will be needed in the Database Access Description in order to enable this access.

The DBA, having created the ADI, and having intimate knowledge of the workings of the database he maintains, should understand the relation between any piece of information in the ADI and the methods needed in order to access that information. As an example, assume that the attribute, "Person's Height," is a piece of data that could be selected from an ADI. The Database Access Details would include not only the "ht" (height) data field, but also the data-fields "ht-accy" (the accuracy of the height measurement) and "ht-um" (the units of measure for the height).

In more formal language, the Database Access Description (DAD) is defined to be another instance of a *DDS*. The relationship between the DAD and the ADI can be described thus:

$$DDS_{ADI} \subseteq DDS_{DAD}$$

The ADI is a subset of the DAD. This relationship must be true since an ADI by itself cannot contain more than the quantity of information required to retrieve data from the database. After all, the ADI omits database details that must be provided in order to access database data.

This subset relationship also entails the following constraints. First, let *ADI* and *DAD* each be an instance of a *DDS*. Then the number of tuples in *ADI* is less than or equal to the number of tuples in *DAD*, or:

$$|ADI| \leq |DAD|$$

This relationship must be true because the *DAD* may need auxiliary data in order to answer queries. Let *i* be the index of some tuple in *ADI* and *DAD* such that the corresponding *d*'s in each tuple are equal. Then the number of references in the tuple *i* of *ADI* is less than or equal to the number of references in the tuple *i* of *DAD*:

$$T_{i,ADI} = \{d_i, r_{i1}, ..., r_{ij}\}$$

$$T_{i,DAD} = \{d_i, r_{i1}, ..., r_{ik}\}$$

$$j \leq k$$

The comparisons in these definitions are not defined to be strict, as in strict subset or strictly less than. However, in practice the ADI is virtually guaranteed to be a strict subset—not only is it true that the ADI cannot contain more than the quantity of information required to retrieve data from the database, it should in practice contain less because of the omitted data.

In theory, the Database Access Description needs to be responsible for providing access to at least the data that is specified in the ADI. However, providing access to more data than is necessary may or may not be something a DBA will want to do. There are advantages and disadvantages to providing more than the minimum amount of access that will be discussed later.

In summary, the Database Access Description provides a way for an ADI component to access data from the database. Database details need to be known in order to enable proper access. It gives the interface generator the information it needs to create functional interfaces to communicate with the database.

3.2.3 Application Requirements Description

The Application Requirements (ARD) description indicates the data, and the format for that data, that is required by the application. The Scheduler and Payroll example mentioned earlier provides a couple of good examples. In that example, the requirements of the Scheduler application are that it must work with data that is expressed in metric units, rounded to the nearest integer, and returned via CGI. The requirements of the Payroll application are that it must work with data that is expressed in English units, measured as exactly as possible with floating point numbers, and returned via RPC. The Application Requirements description should be able to express the requirements of these two examples in this manner.

The ADI facilitates the construction of the Application Requirements because the application developer or interface programmer should be able to more easily establish a semantic match. That is, he can make a quick and accurate assessment as to whether the data that needed by the application is contained within a particular database. Once a semantic match is established, the Application Requirements dictate how the data is to be transferred between application and database. The interface generator can then use the information contained in this description to create an interface that is tailored to the application's needs.

To define the Application Requirements more formally, an extension of the DDS called a *Data and Format Descriptor Set*, or DFDS, is defined. A DFDS also consists of a set of tuples:

$$DFDS \equiv \{T_1, T_2, T_3, \ldots, T_n\}$$

for some number of tuples n. However, the structure of each tuple T is extended to include a description of format:

$$T_i \equiv \langle d_i, R_i, F_i \rangle = \langle d_i, r_{i1}, \ldots, r_{ij}, f_{i1}, \ldots, f_{ik} \rangle$$

for some number of references j and some number of formats k. The Application Requirements are defined in terms of a DFDS. As before, a piece of data is specified through the use of its associated references R. In addition, an Application Requirements also specifies for each piece of data the way in which an application needs to receive that data, through each data's associated set of formats F.

The definition of an ARD can be refined through the inspection of its associated DDS. Since an ARD is a proper extension of a DDS, the following structure can be defined:

$$DDS_{ARD} \equiv \{T_1', T_2', T_3', \ldots, T_n'\}$$

where the following relationship holds:

$$\forall T_l \in DFDS_{ARD}.T_l = T_l' \circ \{ f_{l1},..., f_{lk} \}$$

for all *l* such that 1 ≤ *l* ≤ *n*, and for the number of factors *k* corresponding to tuple *T* at index *l*. In other words, a DDS for the ARD includes all of the data and references included in the DFDS for that ARD, and does not include any of the factors from the DFDS.

The relationship between the DDS of the Application Requirements and the ADI can then be described thus:

$$DDS_{ARD} \subseteq DDS_{ADI}$$

The DDS of the Application Requirements (the set on the left-hand side) are a subset of the ADI. This relationship must be true since an Application Requirement set can never exceed the data that is allowed to be queried or viewed through the ADI. In other words, an application cannot request data from a database that is not included in that database's ADI. The subset relationship is not a strict-subset relation because it is possible for an application to request the entire set of data.

It should be noted that, in principle, it should be possible to create an actual description or language to represent the Application Requirements description such as the one just described. However, in practice it will probably make more sense to implement the description as a process rather than an actual description. The process might consist of a person working through some sort of graphical user interface (GUI) representing an ADI, to select the desired qualities of the interface. The program containing the GUI can then relay the selections to the appropriate interface generator. Although such a process does not appear to explicitly declare a description of the application requirements, it does implicitly describe the application requirements. Such an implicit description may or may not suffice. This process will be described in further detail in a later section.

3.2.4 Translator Library Preparation

In order to enable the interface generator to generate interfaces, a library of translator tools must be provided. The translator library consists of individual translators—programs that perform some kind of translation from one data format or paradigm into another. These translators can be assembled to produce an interface between some application and some database. The translators represent the building blocks of the interfaces that the interface generator produces.

More formally, a translator is defined to be a function $\boldsymbol{f}$ that imposes a particular format *f* on some piece of data *d*:

$$\boldsymbol{f}(f, d) = f\{d\}$$

where the notation *f{d}* indicates the format *f* imposed on *d*. An example of a format imposed on a piece of data is the following: Assume the existence of the format *f* for meters, and of a piece of data *d* that corresponds to the distances between the bases in a baseball diamond, which are usually measured in feet. However, the format *f* imposed on *d* in this instance yields an exact measurement of those distances in meters instead.

A *translator library*, or TL, is a set of these f functions:

$$TL = \{f_1, ..., f_d\}$$

where d is the number of functions in the Translator Library. A Translator Library may contain any number of individual translators, and should be able to be expanded to include more translators as the need arises.

The Scheduler and Payroll example can be used to demonstrate the translator library concept. In order to implement Interface 1 in the example, a set of specific translators are required. The Scheduler requires that all numeric data be rounded to the nearest integer. The interface will require a translator that rounds the numeric data to the nearest integer. It accesses data via CGI calls, whereas the database interacts with other programs via SOAP calls. The interface will therefore also require translators that convert CGI calls to SOAP calls and vice versa. Finally, the Scheduler requires that quantitative data be presented in metric units, whereas the database stores data in English units. The interface will also require translators that convert various metric units to English units and vice versa, e. g., feet to meters and meters to feet, kilograms to pounds and pounds to kilograms, etc.

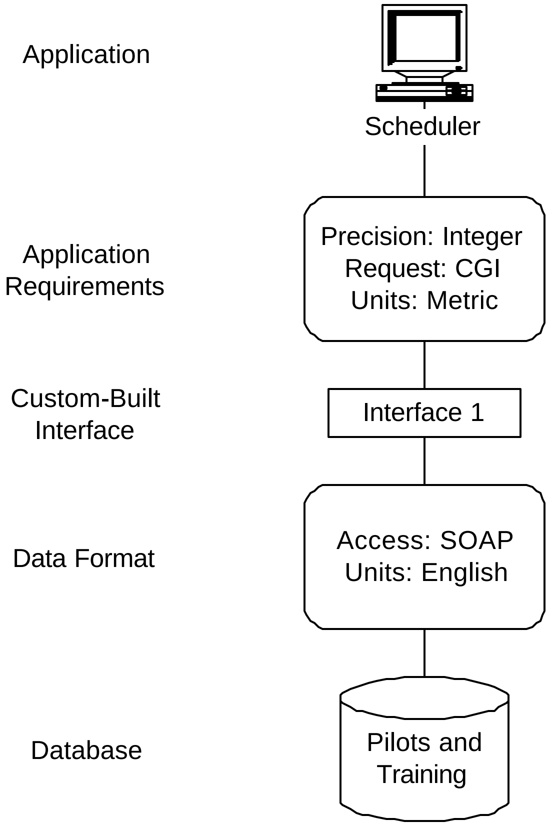

Figure 3.3. Scheduler and Payroll Example, Interface 1

It is easy to see how this example could be expanded to include the required translators for all of the other interfaces in the Scheduler and Payroll example as well.

3.2.5 Interface Generation

The Interface Generator takes in as input the ADI, the Database Access Description of the database, the Application Requirements, and the Translator Library, and produces an interface that will allow the application and database to interoperate with each other. The Database Access Description provides the necessary access to the database, including any required specific metadata information. The ADI facilitates the construction of the

Application Requirements, which in turn dictate how the data is to be exchanged between the application and the database. Finally, the Translator Library provides the tools necessary to perform the various translations between the specific data paradigms of the database and the required formats of the application.

The interface generator is defined as a function G that yields an output *I*, which is also a function. The function G takes in as input an ADI represented by a DSS, a DAD, represented by another DSS, an ARD, represented by a DFDS, and a Translator Library, represented by a set of functions. More formally,

$$G(ADI, DAD, ARD, TL) = I(d)$$

such that

$$\forall T \in ARD, \forall d_i \in T, \forall f_i \in T. I(d_i) \xrightarrow{output} f_{i1} \circ ... \circ f_{ik}\{d_i\}$$

where the input d to the function I is a query for some piece of data. This requirement on G states that for any data in the ARD, the interface I should be able to output the data in the format specified in the ARD.

3.2.6 Interface Operation

An interface created by the Interface Generator allows an application and a database to exchange information with each other. Transactions between the application and database occur through the newly generated interface. Together, that application, database, and interface are capable of interoperation without further intervention from other devices.

4 Goals of the Abstract Data Interface Architecture

The ADI aims to foster improved data interoperability in a scalable, maintainable, and adaptable manner:

1. **Scalability.** The ADI concept should be applicable in large environments consisting of many applications and many databases.
2. **Maintainability.** A system that utilizes the ADI concept should be easy to maintain relative to other solutions.
3. **Adaptability.** Changes to an individual component within the architecture should have minimal impact on other components.

The following sections will discuss the ability of the proposed ADI architecture to meet these goals.

4.1 Scalability

Scalability refers to the ease with which a system can be modified to fit the problem area. The ADI concept extends well beyond one application-database pair. For each database in a set of databases, Detailed Abstractions can be easily created from logical design models. Each database can add its own individual abstract data description to the ADI that encompasses the set of databases. This comprehensive ADI facilitates construction of Application Requirements descriptions as before, but with the added feature that data can be retrieved from any of the individual databases, without necessarily having knowledge of individual databases. This feature is remarkable, since the ability to treat the set of databases as a single logical unit simplifies the tasks and thought processes of the application developer or interface programmer.

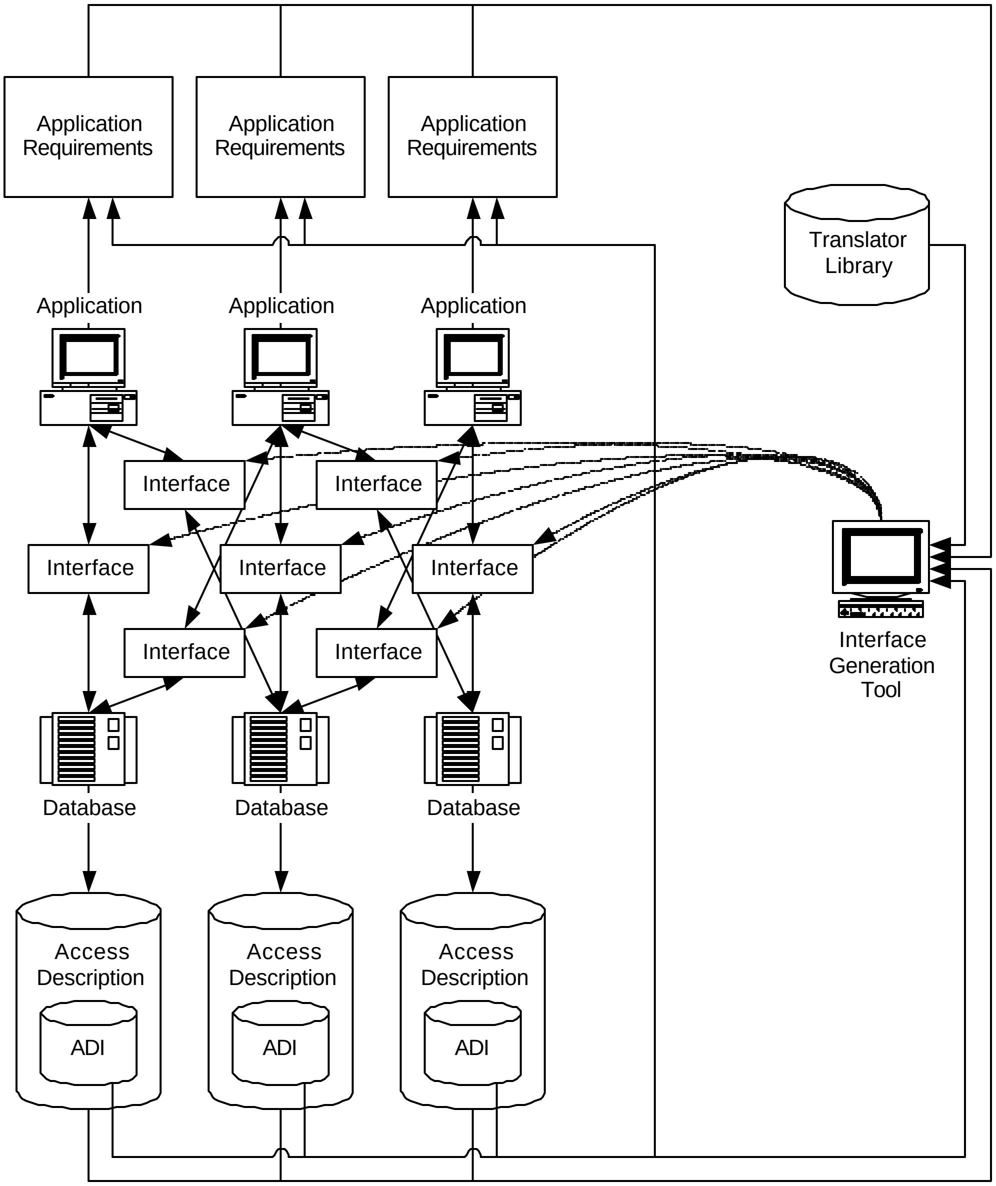

Figure 4.1. ADI Architecture is Scalable

Once the ADI and Database Access Descriptions of each database, the Application Requirements of each application, and the Translator Library with appropriate translators are made available as input to the Interface Generator, interfaces between any application-database pair within the entire system can be automatically generated.

4.2 Maintainability

Maintainability is defined as the ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment. [IEEE, 1990] The ADI concept and architecture allow for a highly maintainable system, since less maintenance work relative to previous approaches is reduced in the event individual databases and applications need to be changed.

The modularity inherent in the ADI architecture allows for significant maintenance ease. The ADI architecture provides for the automated generation of interfaces, which can easily be modified after generation as needed, without impacting the operations between other applications and databases or the rest of the ADI architecture. If faults are discovered in an individual application, database or interface, the components in the ADI architecture are sufficiently modular and separable that they can be modified individually to meet the new need, with minimal impact on the rest of the architecture. While the point-to-point approach described earlier features this kind of maintainability, standardization does not, since standards are more closely coupled with the individual applications and databases that they support.

The ADI architecture requires maintenance attention to a significantly smaller fraction of interfaces as compared to an architecture that utilizes point-to-point interfaces. If maintenance efforts are required, they are aimed at the ADIs and Database Access Descriptions associated with each database, and the Application Requirements. In the case of maintenance efforts that require attention to individual databases, the number of interfaces that must be inspected is in the worst case linear in the number of individual databases and applications. In a point-to-point model, maintenance efforts must be aimed at each individual interface, which in the worst case could require that maintenance work be performed on an exponential number of interfaces compared to the number of databases or applications. For any sizable number of changes, the ADI approach offers a considerable reduction in the amount of work that needs to be done. This reduction in work is possible because, in the ADI model, the interface generator performs the maintenance work for each interface by generating a new interface, whereas in a point-to-point model, each interface must be modified by hand.

In the diagram below, components in marked in bold (applications and databases) are targets of maintenance efforts but all shaded components (interfaces) are generated automatically by the Interface Generator and do not require extra maintenance work. Although it can be seen from the diagram that a significant amount of work is saved, it should be noted that an even more overwhelming amount of work would be saved in an environment with only a few more databases or applications. In such an environment, there could be an exponential number of interfaces, *all* of which would be automatically generated.

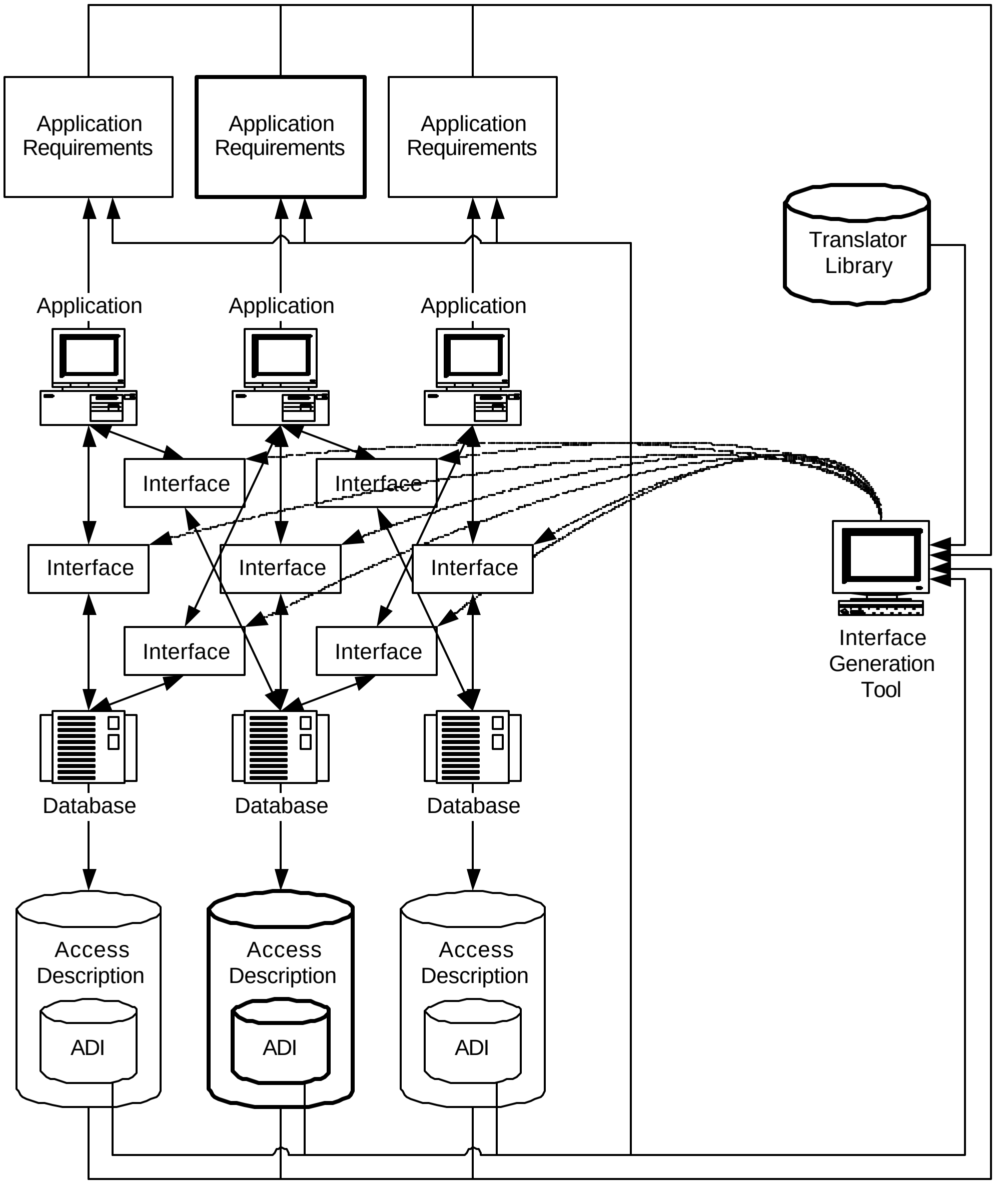

Figure 4.2. ADI Architecture is Maintainable

4.3 Adaptability

Adaptability is defined as the ease with which software satisfies differing system and user constraints. [Evans 87] In this section the ability of the ADI to respond to the challenge of adding new applications or databases is examined. It will be seen that the ADI should be able to handle the challenge effectively.

The ADI architecture can enable a new application to exchange information with the already existing set of databases. In order to enable the exchanges, the application developer or interface programmer uses the ADI database abstractions to create a new set of Application Requirements, which are then passed to the Interface Generator. The Interface Generator, in turn, creates a new interface that could be similar to, but is independent of, any previous interface created.

The ADI architecture can also accommodate the addition of new databases. One who wants to add a new database to the architecture publishes the database using both the ADI abstraction as well as the Detailed Abstraction. The ADI of the new database is added to the Comprehensive ADI, while the Detailed Abstraction is made available to the Interface Generator. Application developers or interface programmers can now use the Comprehensive ADI to create interfaces that will exchange information with a logical database that includes the newly added one. Already-existing interfaces will be affected in one of two ways:

1. **They will not be affected.** This case occurs when the newly added database includes no data that affects the application that connects to the interface.
2. **They will require changes.** This case occurs when the newly added database includes some data that conflicts or otherwise intermingles with previously existing databases.

The first case is easy to deal with. If the new data does not have any relation to the application, the application developer or interface programmer should not need to do anything, and he should instead simply allow the interface to continue to retrieve the same data as before. No further action is necessary to ensure smooth operation.

The second case is somewhat harder to deal with because there is a requirement to expend additional resources to accommodate the change. If the new data serves as a better substitute for the previously used data, or if a combination of the new data and the old data serves the purposes of the application better, then a new Applications Requirements description should be created. The new Applications Requirements description allows the Interface Generator to create a new interface for operation between the application and the new set of databases.

At first glance, the extra work required to make the application work again seems to offer a serious disadvantage. Assuming that there is a requirement to change many of the applications as a result of adding a new database, then work will have to be done on a linear number of applications, in the form of a new Applications Requirements description for each application. A system of point-to-point interfaces under the same assumption, on the other hand, would require programming work to be done on each of the interfaces for each application, which can be the number of applications, *squared*. The Interface Generator in the ADI architecture reduces the programming work necessary.

It would be highly convenient if the ADI architecture were robust enough to accommodate these kinds of changes automatically. However, the application developer or interface programmer must make a choice that dictates the change. The choice can be thought of as a new instance of deciding whether a semantic match exists between the application and the new database. Therefore, the problem of accommodating these kinds of changes is a process problem, and cannot be automated.

5 Sample Component Construction

The previous section discussed the architecture specification, operation, and advantages of the ADI. The next section is concerned with the construction of the individual components of the ADI. The components that need to be considered are:

1. Database
2. Abstract Data Interface
3. Database Access Definition
4. Application
5. Application Requirements
6. Translator Library
7. Interface Generator
8. Interface

In the next section, each of these components will be discussed and demonstrated with examples. The accompanying examples will assume for simplicity that there are two applications and one database.

5.1 Database

According to one definition, a database is “a collection of logically related data stored together in one or more computerized files.” [IEEE, 1990] According to another definition, it is “an electronic repository of information accessible via a query language.” [DoD, 1991]

These definitions characterize at a high level what a database is, but it is one particular characteristic that helps to apply the ADI concept over databases in general: the fact that databases are based on some kind of information model.

Information modeling is a technique used to understand and document the definition and structure of data. A logical information model is a specification of the data structures and business rules needed to support a business area. A physical information model represents a solution to these requirements. Databases utilize information models for a variety of reasons, including education, planning, analysis, design, documentation, standardization, policy, and speed.

Because the ADI concept relies on general information modeling practices that apply to all databases rather than on specific types of databases, virtually all kinds of databases should be eligible for inclusion in the ADI architecture. There exist many kinds of databases, including flat-file text databases, associative flat-file databases, network databases, object databases, hierarchical databases, and relational databases. Relational databases are by far the most useful type commonly available, and so this thesis will for the most part assume a focus on relational databases. However, it should be stressed that the concepts presented here do not depend on any particular type of database, and that any bias in the examples is merely due to the convenience of a simplifying assumption.

The ADI architecture will accommodate any database. Such databases do not need any modification to interact with the ADI architecture. The discussion of ADI architecture construction assumes that the databases it interacts with will be provided by an outside source. The examples in this paper are drawn from a fictional SQL-based database containing mission, aircraft, and pilot information, which will hereafter be referred to as the MAP database, or simply the MAP.

5.2 Abstract Data Interface

As stated previously, the Abstract Data Interface of a database is a view, created by a DBA, which enables the application developer or interface programmer to easily ascertain the contents of a database without being hampered by unnecessary details. It will remove attributes that are not necessary for determining semantic match, and it will rename data to help bring out intuitive meanings.

The DBA can implement an ADI by assembling a modified IDEF1X information model. (See appendix for background information on IDEF1X.) An IDEF1X model is composed of three main building blocks: [Bruce, 1992]

1. *Entities*, which refer to any distinguishable person, place, thing, event, or concept about which information is kept;
2. *Attributes*, which refer to properties of entities; and
3. *Relationships,* which refer to connections between two entities.

An ADI will capture most entities and relationships in a database, and a portion of the attributes—the attributes that are not essential to determining semantic matches are omitted from the ADI. An example relating to the MAP database is given below:

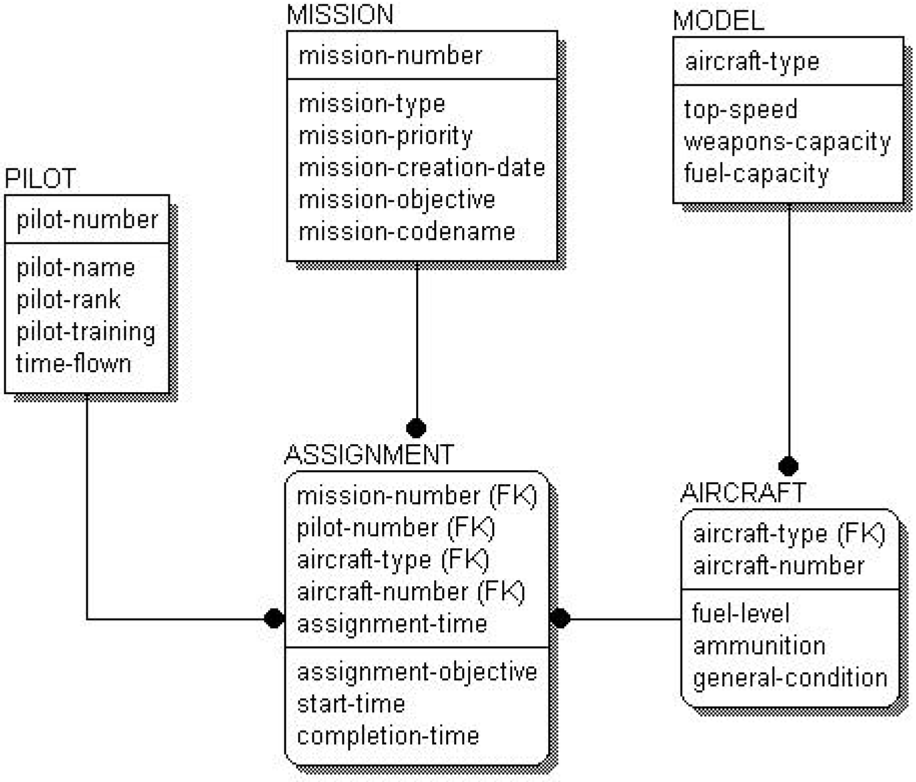

Figure 5.1. ADI for the MAP Database Example

This diagram attempts to convey in as clear a manner as possible the contents of the database that the DBA has decided to expose. Schema details have been omitted, and names have been selected to give an intrinsic, intuitive meaning.

In brief summary, this ADI exposes data about Pilots, Missions, and Planes. Missions are composed of individual Assignments, which individual Pilots are assigned to. An aircraft, which can be one of several models, can be assigned to an Assignment as well. These fields can be indexed by various identifying keys, including unique pilot numbers, aircraft numbers, and mission numbers.

A note on the selection of IDEF1X for the ADI is this case is in order. The choice of IDEF1X is related to the fact that the database that is of the relational variety; IDEF1X is particularly effective at modeling relational databases, although other choices, such as a modified Entity-Relationship (ER) model, do exist. However, the assertion that the ADI is not dependent on a particular type of database still holds. The database literature contains logical data models suited to represent any type of database; a database textbook should provide a good starting point. Any such logical data model should be able to be modified in a similar way that the IDEF1X model was modified in this example to provide a good ADI.

5.3 Database Access Description

The Database Access Description is an access mechanism for the Interface Generator, and ultimately, the individual generated Interfaces. It allows a component to execute commands to actually retrieve data from the database, which cannot be done through the ADI alone.

It was mentioned earlier that the amount of access permitted through the Database Access Description could be variable. There must be at least the minimum amount to access all of the data contained in the ADI. If there were not, then an interface programmer could select some piece of data from the ADI to be incorporated into a generated Interface, but not actually be able to create an interface that correctly accessed and manipulated that data.

If the Access Description were to contain the minimum amount of information available, then the DBA has two advantages. She has not lessened her degree of data security by publishing a document that would permit access to her database in unwanted ways. Also, creating a minimal Access Description would in general imply that she would have to spend less effort in assembling such a description. On the other hand, there are advantages to including more information. If the complete set of information needed for data access is included in the data model, then the Access Description the DBA creates could serve as a complete database model for that database. Such a product could serve as a useful document; after all, in general most database design documents are created to be useful only during the design phase, and then are left to become either lost or obsolete.

In the MAP example, a straightforward method the DBA could use to implement a Database Access Description would be to create a SQL Data Definition Language (DDL) file corresponding to the MAP database. For each entry in the ADI, the DBA will understand how the data is assembled from the database to create that entry. Note that there may not necessarily be a one-to-one correspondence between ADI and Access Description data fields. After all, the ADI is simplified representation of what data is in the database, and a set of more complex pieces of data may combine to yield one simple piece of data.

As an example, an expanded IDEF1X model corresponding to sample SQL DDL for the MAP database is shown below. The DDL is included in the appendices.

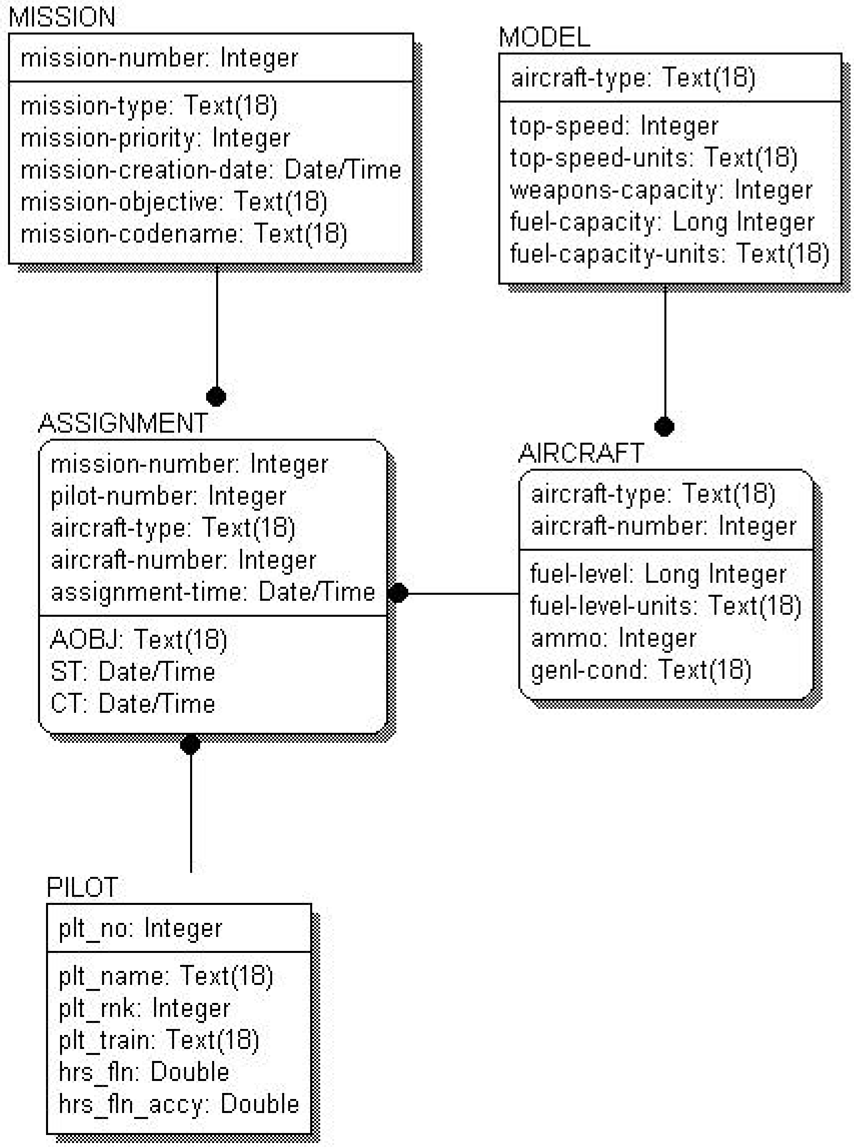

Figure 5.2. Database Access Description for MAP Database Example

As can be seen in the above diagram, there is some extra information in this diagram that was not included in the ADI. The *fuel-level-units* attribute in the *AIRCRAFT* entity, as well as the *top-speed-units* and *fuel-capacity-units* attributes in the *MODEL* entity denote what kind of units of measure are associated with some of the other attributes. Also, the *hrs_fln_accy* attribute in the *PILOT* entity denotes the margin of error when pilot logged his flying time. Further, note that while there are attributes that correspond to the data given in the ADI, some of the names are not the English-like names seen in the ADI. After all, in order to communicate with the database, one must use the database data field names, which may not be as user-friendly as one might like.

As in the ADI example, the choice of SQL DDL is influenced by the fact that this example is based on a relational database. However, other database types have other ways of accessing data, and this access is not dependent on the use of SQL DDL. If other database types are to be used, then a different database access method should be used—it should not be difficult to find one that will work well for any given database type.

5.4 Application

The application is the data consumer in the ADI architecture. Intuitively, the ADI enables an application, or any other data consumer, to specify the data that it needs, and then specify the way in which it needs that data. Consider the following applications:

1. **Real-Time Fueling.**

   This application takes data about vehicles from a variety of sources and determines which vehicles need fueling most urgently. The information will be used by a special team that delivers fuel in the most urgent situations.

   Requires Input: Aircraft, indexed by an identification number, and corresponding fuel levels, measured in liters. Data expected to be in CORBA IDL.

2. **Personnel Assignment.**

   This application takes data from personnel databases and determines whether a person has had enough experience and training to be scheduled on special missions.

   Requires Input: Pilots, identified by their identification number, along with the number of hours they have flown, and the training courses they have taken. Data should be returned in SOAP.

3. **NASA Scheduler.**

   This application takes data about space satellite missions, checks for changes in priority on each probe's mission, and decides the next course of action for that probe, which is one of promote priority, demote priority, or leave priority unchanged.

   Requires Input: Missions, identified by unique identification number, along with probe identification number and probe status. Requires data to be returned via RPC.

5.5 Application Requirements

The application developer or interface programmer must state the application requirements and pass this information to the Interface Generator in order for the proper interface to be generated. The interface programmer accomplishes this task by examining the ADI and deciding what data is needed. Because the ADI contains so little specific information about the data, the interface programmer is not able to think about the data details. He should not have to; after all, schematic differences can be resolved.

The interface programmer begins with the Real-Time Fueling application. He examines the ADI for the data he needs, and finds that *aircraft-number* and *fuel-level* should suit his purposes. For the Personnel Assignment application, he performs another examination, and discovers that *pilot-number*, *pilot-training*, and *time-flown*, should be the correct inputs. Finally, for the NASA Scheduler application, he determines that *mission-number* and *mission-priority* is the data he needs.

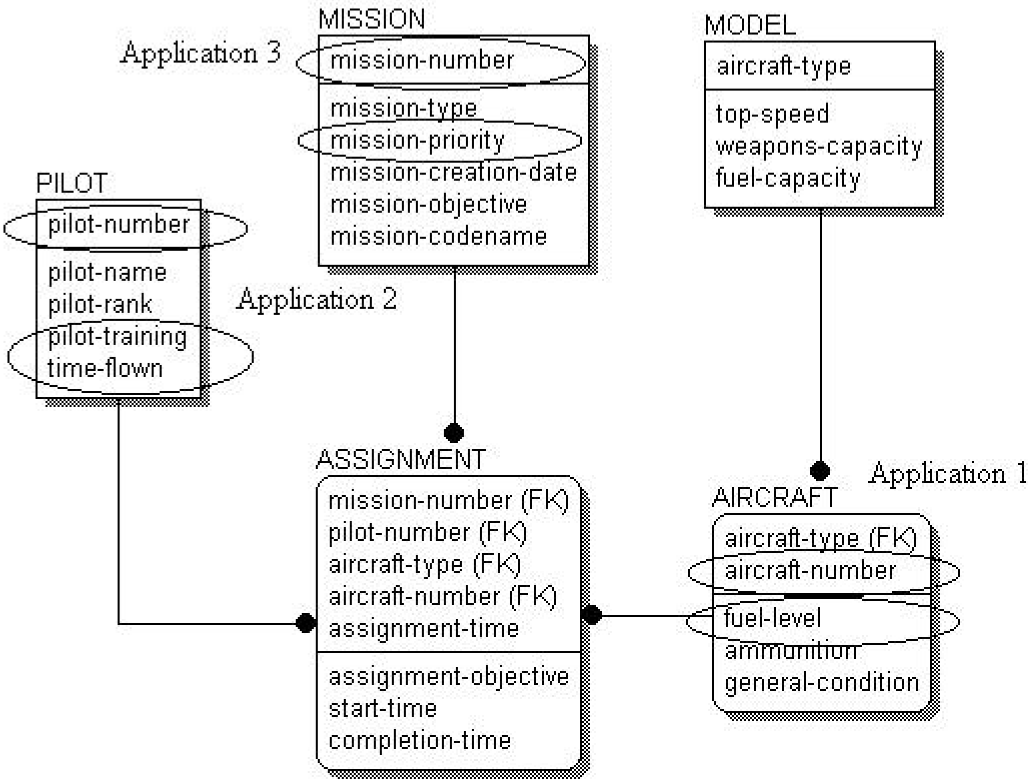

Figure 5.3. Application Requirements Declaration

An important step for the interface programmer now is to verify that the items picked from the ADI do indeed constitute a semantic match. The degree of rigor used in this verification should be tantamount to the task at hand; for mission critical operations, the verification should be very rigorous indeed. For example, if the interface programmer did such verification at this point, she would discover that the data she thought was appropriate for the NASA application was, in fact, no good at all; it is relevant only to *airplane* missions. It is an important point that the ADI architecture treats the semantic heterogeneity problem as a process problem, and so manual verification is necessary.

An implementation of the Application Requirements component can take at least two forms. One form would be that of a specialized language designed for submitting selected values. Most likely, the language, while specialized for selecting items from the ADI, should actually be as neutral as possible so that interactions with the Interface Generator are not affected by the particulars of one specific programming language or another. The Interface Generator needs to be able to generate a wide variety of Interfaces. UML shows promising potential for this application, and other researchers have worked on assembling specialized toolkits or languages for similar purposes. [Hammer et al., 1997]

An even more intuitive Application Requirements implementation calls for a Graphical User Interface (GUI) so that interface programmers or application developers can look at the ADI diagram and work off of it. Of course, such an approach would require that an implementation of the language described in the previous paragraph, or a similar functionality, be available before a GUI can be implemented over the logic.

5.6 Translator Library

The Translator Library serves as a repository for tools that the Interface Generator needs in order to generate Interfaces. In the general case, an application requires a piece of data from a database; the data must be converted or *translated* into a form that the application can use if the information exchange is to be successful. Of course, an information exchange can, and often will, involve the exchange of numerous pieces of data; each individual piece may require one or more translations.

The Real-Time Fueling and Personnel Assignment application examples will demonstrate the vital function of the translator library. The Real-Time Fueling application requires information about specific aircraft and their respective fuel levels, in meters. According the ADI for the MAP database, the fuel levels for aircraft is available. However, if the fuel levels are measured in gallons in the database, then a *translator* is required to convert the units of measure from gallons to liters. Also, the MAP database returns answers as a SQL database; however, the application requires that data be returned via CORBA IDL. Therefore another translator is necessary to return the answer to the query through an IDL stub.

The Personnel Assignment application requires information about Pilots and their respective flying experience and combat training. The application requires that the flying experience be measured by hours flown, but fortunately the MAP database includes the time in hours already. Note that although the database provides an attribute denoting the precision or accuracy of the figure for hours flown, an Interface generated by the Interface Generator is not required to use this figure because the application does not require it. If the situation were reversed, that is, the application required a similar kind of metadata but the database did not include it, then there is no solution; this situation is tantamount to a semantic mismatch, since the data consumer is seeking something that the data producer simply does not have. In the end, the only translation needed for the Personnel Assignment application is the translation of the *access mechanism*—the way in which the application retrieves the information. In this case, the application requires interaction via SOAP.

If the required translators just described are provided to the Interface Generator, an Interface should be able to be produced. For the purposes of reuse and efficiency, the ADI architecture utilizes a Translator Library to store translators that have already been created. Thus, if a translator has been written once, another Interface requiring the same translator can make use of a previously created instance of the translator rather than requiring that a new translator be written every time the same instance of translation occurs. Further, translators should be written in a modular way so as to promote their reuse.

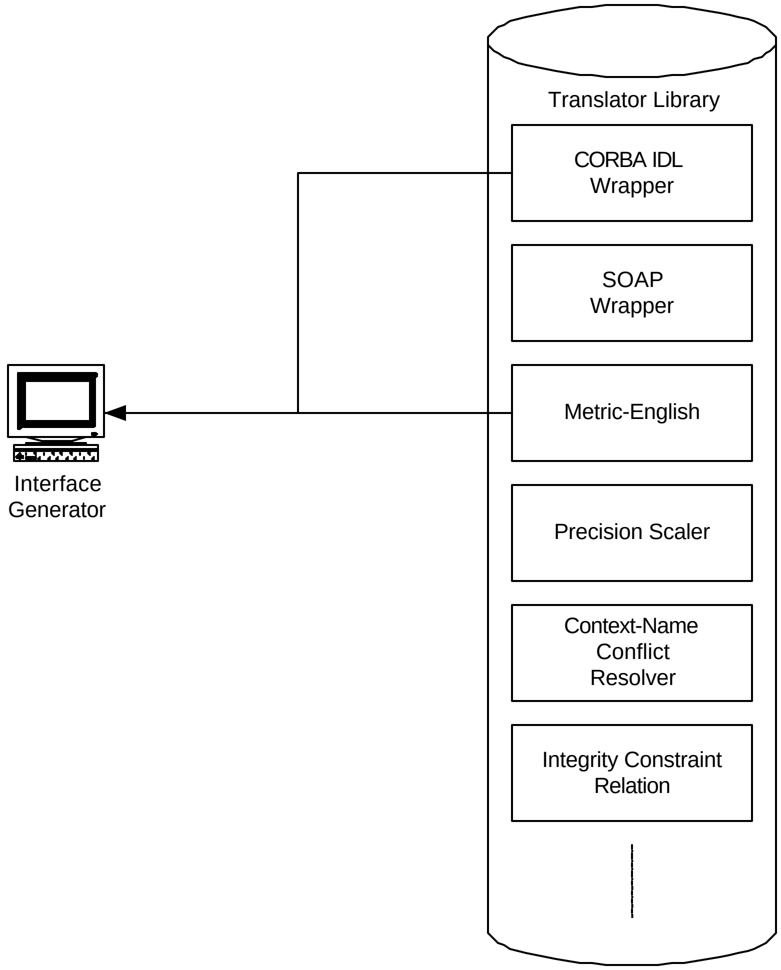

Figure 5.4. Translator Library Provides Translators to Interface Generator

A full description of the implementation of the Translator Library would be well beyond the scope of this thesis; however, a brief specification is provided: The Library must have a way to interact with the Interface Generator in such a way so that the correct translators are retrieved when needed. The Library should store the translators in a neutral form so that the Interface Generator, which needs to generate Interfaces in arbitrary languages and formats, can make use of the translator in a code generator. Again, UML is a strong candidate for this purpose, as COTS UML code generators already exist for other applications. [Microtool, 2001] Finally, the Library should also make use of efficient sorting and search algorithms in the literature [Cormen, Leiserson, and Rivest, 1990], since such a device could grow to a considerable size if many translators are created and retained.

5.7 Interface Generator

The Interface Generator assembles the four inputs, Abstract Data Interface, Database Access Description, Application Requirements, and Translator Library, into an Interface. The Abstract Data Interface declares what kinds of data are available. The Database Access Description allows it to actually access the data in the database. The Application Requirements state what data, out of the available data, is needed, and how that data should be presented. The Translator Library provides the tools that convert the data that is needed into a form usable by the application. The result is an Interface that is capable of enabling interoperable communication between an application and database.

This task can be implemented via code generators. An interface generator essentially packages the translators that the first three inputs determined were required in an interface. After selection of the appropriate translators, the assembly process should be able to be automated via UML code generation techniques. The final step is to automate the assembly of the actual interface that will be used to translate between the application and database.

5.8 Interface

The Interface is the output of the Interface Generator, when presented with inputs of Database Access Description, Abstract Data Interface, Application Requirements, and Translator Library. Once the Interface is generated and enabling effective communication between application and database, then the rest of the ADI Architecture components no longer need to interfere. The only time the ADI would come into play again would be if the application requirements, the application itself, or the database underwent changes. In this case, the Interface Generation process should start anew to generate a new Interface that meets the new constraints.

6 Metrics

In this section, metrics that could be used to determine the success of the ADI architecture are discussed. These metrics should be measured using qualitative as well as quantitative means, since in many cases, there is no standard to test the hypotheses against.

6.1 Time and Labor Expenditure

One hypothesis of the ADI architecture is that use of the ADI architecture and methods will provide significant savings in terms of time and labor spent in enabling and maintaining an interoperable system. Therefore the ADI approach should be evaluated in terms of the number of man-hours required to accomplish a similar level of interoperability as in other approaches. Several areas where effort may need to be applied are identified, and their evaluation in terms of this metric is discussed.

6.1.1 ADI and Database Access Description Construction

An assumption of the ADI architecture is that DBAs will be responsible for the construction of two vital parts, the ADI and the Database Access Description. The rationale behind this assumption is that the DBA is the individual responsible for deciding how access to the database should be granted in most organizations; further, he should be capable of accomplishing this task easily. However, there exist situations that have the potential to undermine the utility of this assumption. For example, an organization that does not recognize the need for, or have the funds to afford, a DBA may assign the task of constructing the ADI and DAD to others who do not have the required expertise. Excessive effort spent in constructing these components will undermine the utility of the ADI, and therefore such efforts should be accounted for as a metric.

6.1.2 Application Requirements Construction

The effort required to construct an Application Requirements Description should be minimal, because the ADI is intended to make the contents of the database as clear as possible. However, actual implementations of the tools used to create an ARD can complicate matters for the interface or maintenance programmer, who would be the individual charged with the task of constructing an ARD. For instance, one suggested way to facilitate ARD construction is a scripting language that allows one to specify the data and format that is needed by some application. However, poorly implemented or overly complicated scripting languages could make it difficult for the interface programmer to construct the ARD. Effort is expended to construct this component, and so that effort should be measured.

6.1.3 Translator Library Maintenance

At first glance it may seem that a lot of effort is spent to maintain the Translator Library. Every time the Interface Generator encounters a data translation that it is unable to accommodate, a new translator must be constructed and added to the Translator Library. However, it should be emphasized that such effort results in components that can be reused. Therefore, such effort can optionally be included in the metric. The argument for including it is that the metric should account for all effort expended in constructing an interface. The argument against is that the metric should account for only the efforts that must be undertaken every time an interface is created; in other words, the metric should account only for recurring costs. However, the effort spent writing a translator can be treated as a sunk cost because the translator can be reused. The organization evaluating its data interoperability initiatives must make a decision between the two that will depend on its particular environment and situation.

6.1.4 Interface Modification

Once an interface is created, there may or may not be extra effort required to ensure seamless operation. Most likely interface or maintenance programmers would be responsible for this task if proves to be necessary. If modifications must be made before the interface can be deployed as an autonomous unit, then the effort spent to achieve those modifications must be accounted for.

6.2 Interface Speed

Ideally, interfaces generated by the ADI architecture's Interface Generator will not only enable interoperability, but also operates quickly at runtime. However, it might be expected that a generated interface would be slower than an interface custom coded by hand. The automatically generated one depends on automated construction that must apply equally well to construction of different interfaces, whereas a handcrafted interface has the possibility of being optimized based on details of a particular database-application data exchange. The operating time between the two should be compared for the purposes of evaluating the ADI architecture.

The question of interface speed will depend heavily on the relative value of interoperable data. The organization concerned with these issues will need to make an engineering decision to determine the tradeoff between having interfaces that operate quickly and data that works with many applications at low cost (in terms of time and effort spent). An organization that depends on a small set of independent data consumers and producers, knows with a high degree of certainty that the data environment is stable, and has no desire to work with outside parties, may decide that it is more cost-effective to custom-code all of the necessary interfaces. Many organizations, however, are not of this type, and so it is expected that the value of having interoperable data at low cost will outweigh the disadvantage of having interfaces that do not operate as quickly.

An alternative to evaluating a strict tradeoff between interface operating speed and interoperable data is to evaluate the performance of an interface that, after automatic generation, is tweaked by hand for better speed. In this case, however, the effort necessary to incorporate the optimizations must also be accounted for.

7 Related Work

In this section, several of recent approaches from the data interoperability literature introduced in the earlier chapter on data interoperability approaches are discussed, compared, and contrasted with the ADI approach.

7.1 Human Intervention

Some aspects of human intervention are present in the ADI, and it is important to recognize them because typically there is a high cost associated with human labor relative to automated computer work. A DBA or equivalent person must construct the ADI and Database Access Description. Interface programmers or application developers must construct the Application Requirements. Together, these people are also responsible for maintaining the Translator Library. Finally, the interface programmer may be called upon yet again to make adjustments or tweaks to the generated interface.

As described, the ADI architecture still contains opportunities to further minimize the amount of human intervention necessary. It was suggested that the ADI be created purely from the DBA's concepts of what should and should not be accessed. However, a potentially labor-saving alternative is to employ a method of generating a database model from the database itself. One such method can discover conceptual object models from instances of relational databases. [Shen et al., 1999] Another potential savings could result from careful accounting and reuse practices for the Translator Library, to ensure the highest possible chances of reusability of translators contributed to the Library.

7.2 Tightly-Coupled Federation

Although the ADI bears more resemblance to a loosely-coupled federation rather than a tightly-coupled federation, a discussion of some tightly-coupled federations is provided here to describe the characteristics of such an approach, as well as to convey recent research trends in this area.

7.2.1 SIM

SIM, or Schema Integration Methodology, accomplishes the integration of schemas by resolving a set of equivalence correspondences between arbitrarily complex local sub-schemas. From such a set of correspondences, SIM semi-automatically derives schema transformations, termed schema augmentations, from each local schema to the integrated one. The transformation is conducted in such a way that corresponding data among the local databases is mapped to the same structure in the integrated database. The generated schema augmentations enhance the schemas with classes and paths, resulting in an integrated non-redundant schema. [Fankhauser, 1997]

Efforts have been made to enhance SIM with the capability of resolving some semantic heterogeneity problems as well, through the incremental integration of schemas. During incremental integration, SIM admits the declaration of new equivalence correspondences between sub-schemas, but only in cases that do not lead to ambiguity or inconsistency with respect to previous integration steps. SIM's augmentation constraints allow it to identify inconsistent correspondences within the schema structure, as sub-schemas are incorporated, and reject them. [Motz and Fankhauser, 1998]

7.2.2 VHDBS

VHDBS is a federated database system based on a client/server architecture. In this approach, federated data is stored in an object-oriented data model, which is an extension of the ODMG-93 object model. Objects are organized into repositories and unified by a schema that includes the types of all objects, their inheritances, and the repository schemas. A federated view is achieved either by viewing the repositories, or manipulating the data types such that only the desired methods or attributes of these types are seen.

7.2.3 ADI and Tightly-Coupled Federations

Compared to the ADI, tightly-coupled federations have the advantage of a central focus for the purposes of resolving conflicts between data models. If there are overlaps between the data models of two different data sources, then resolution efforts will be aimed towards the integrated schema. In contrast, the ADI approach leaves the data consumer or application with the problem of resolving such conflicts. If the data consumer needs to query many overlapping sources, resolution efforts will have to be aimed over all of the overlaps. For a large number of overlaps, the ADI approach will have a more complex overlap resolution process.

The ADI approach permits more resilience to changes than the tightly-coupled federation approach. In the federated database system, the ability to exchange data between data producers and consumers hinges on the integrity of the integrated schema. If one sub-schema in the integrated schema changes, the effects could be propagated through the entire environment. On the other hand, the ADI facilitates data exchanges between data consumer-data producer pairs. In the ADI approach, the response to a change in a single component within the environment is to generate a new interface for that pair. The impact on the data interoperability of the environment as a whole is minimal.

7.3 Mediated and Loosely-Coupled Federations

The ADI is architecturally more similar to a mediation or a loosely-coupled federation approach. The next section investigates some of these approaches found in the literature in more detail and again compares them to the ADI approach.

7.3.1 COIN

In the Context Interchange approach, semantic interoperation is accomplished by making use of declarative definitions that correspond to source and receiver contexts. Each party in the data exchange expresses is own constraints, choices, and preferences for representing and interpreting data. A context mediator can then automatically identify and resolve any potential semantic conflicts. [Bressan and Goh, 1998]

The major difference between the COIN and ADI approaches is the choice of target for automation. Under the COIN approach, schematic details particular to data producers and consumers are declared, and then the semantic differences are automatically resolved by the mediator. However, COIN has been shown to exhibit some failures in determining proper semantic correspondences. [Ouksel and Ahmed, 1999] In contrast, in the ADI approach, semantic matches are determined first, and then schematic differences are automatically resolved by the interface generated.

7.3.2 TSIMMIS

The Stanford-IBM Manager of Multiple Information Sources, otherwise known as the TSIMMIS System, is an approach to data interoperability developed by researchers at Stanford University and IBM. The TSIMMIS system enables access to multiple heterogeneous information sources by translating source information into a common self-describing object model known as the Object Exchange Model, or OEM. Source specific wrappers and "intelligent" modules known as mediators provide the integrated access to the heterogeneous sources. The wrappers convert queries over information in the common OEM model that into requests that the source can execute, and the data returned by the source is converted back into the common model. The mediators collect information from one or more sources, process and combine that information, and export the resulting information to the end user or application program. Users or applications can choose to interact either directly with the translators or indirectly via one or more mediators. [Hammer et al., 1995]

Later work on the TSIMMIS project resulted in the development of a wrapper implementation toolkit for quickly building wrappers. The work was motivated by the fact that building a wrapper is a task that requires a lot of effort and time, thus diminishing the usefulness and applicability of writing wrappers in situations where it is important or desirable to gain access to new data sources quickly. The work was based on an observation that only a relatively small part of the code deals with the specific access details of the source. [Hammer et al., 1997] Indeed, the integration wrapper implementation toolkit developed by the researchers at Stanford University is analogous in some sense to the Translator Library in the ADI. The difference between the two is that in the base case, the TSIMMIS wrapper implementation toolkit allows many data sources to be made to conform to a single application's native query, whereas the ADI provides interfaces for many applications to interoperate with a single database.

7.3.3 ODMG-OQL and SQL Query Mediation

An approach based on two query languages, ODMG-OQL and SQL, has been proposed to allow query transformation as a mediator for data interoperability. [Huang et al., 2000] In this approach, the mediation architecture provides a way of accessing underlying heterogeneous databases without using an integrated model. Automated query transformations permit data consumers to use native query languages to access heterogeneous databases without acquiring or adapting to the target schema and syntax.

The ODMG-OQL/SQL query mediation architecture is also a complement of the ADI architecture. This approach permits one data consumer to query a set of heterogeneous data sources in the consumer's native language and constructs, whereas the ADI architecture permits many applications to exchange data with a single database, through an interface that provides the interoperability.

7.3.4 SIMS

The SIMS information mediator [Arens et al., 1996], also complements the ADI in the same way as the TSIMMIS and Query Mediation approaches mentioned above. However, SIMS offers optimizations in its mediators in that they are able to take domain-level queries and dynamically select only the appropriate information sources based on content and availability. Then the mediator generates a query access plan that specifies the operations and their order for processing the data. Semantic query optimizations are performed to minimize the overall execution time.

The SIMS method offers some features that differentiate it from other architectures. First, the SIMS architecture provides optimizations to the queries as they are being formulated. Second, the mediator must contain a model of its own domain of expertise, which provides the terminology for interacting with the mediator as well as the models of all the individual data stores available to it. These components have their analogues in the ADI architecture: the former is analogous in some sense to the Translator Library, whereas the latter is analogous to the Database Access Descriptions of the individual databases. However, in the ADI architecture there is no restriction that a mediator deal only with a "domain of expertise," since modular components can always be added if a greater breadth of functionality is desired. Further, because the mediator contains a model of its own domain of expertise, it is expected that changes to the environment might be hard to maintain. When individual systems are removed from or added to the SIMS environment, the mediator model must change along with it. On the other hand, the ADI approach would call for the addition or deletion of an individual interface that does not interact with any other interface, and thus such a change would have minimal impact.

7.3.5 YAT and TranScm

YAT is a data model that consists of named ordered labeled trees that may be assembled to form a pattern. A set of patterns forms a model that is used to represent real-world data. The interesting characteristic of YAT models is that they can be mapped from one format into another, while retaining the equivalent data model. Furthermore, the YAT language, or YATL, can be used to customize the data model to a specific need. [Abiteboul et al., 1999]

YAT can be seen as a model or even a candidate for implementation of the Interface Generation portion of the ADI. A useful feature of YAT data models is that a significant portion of YAT model translations can be generated automatically using the TranScm system. In the context of the ADI, the YAT model is analogous to the data description inputs to the Interface Generator (but is not analogous to the Translator Library). The TranScm system should be capable of doing most of the work that the Interface Generator will do. Any extra work that needs to be done by interface programmers can be accomplished through YATL.

Having discussed the architecture of the ADI as well as comparisons to approaches found in the literature, this thesis closes with conclusions and future work in the next section.

8 Conclusion

8.1 Epilogue

Chapters 1 and 2 introduced the data interoperability problem, motivations, history, and approaches. Chapters 3 through 6 discussed the proposed ADI architecture, specification, advantages, examples, and metrics. Chapter 7 compared and contrasted the ADI approach with some of the previous approaches in detail. All of the background presented suggests that the ADI approach has the potential to be a viable solution to data interoperability problems.

The ADI has the most applicability in large distributed environments where a high value is placed on the ability of the data interoperability solution to adapt to changes and maintain itself. In general, larger environments will derive more value from the ADI than smaller ones since there are a higher potential number of interfaces that would have to be built in its stead. Environments that need to change frequently yet gracefully will also derive more value from the ADI than environments that are more stable. The environment undergoing significant changes can take advantage of the fact that the ADI can simply discard old interfaces that have been made obsolete due to changes. Organizations that depend on distributed environments will also appreciate the fact that ADI allows maintenance efforts to be directed at a small set of targets while preserving autonomy of the data producers and consumers.

8.2 Future Work

In this section, future work that builds on the research presented in this thesis is described. Possible future research projects include implementation, alternate implementations, relaxation of simplifying assumptions, and a converse application focus.

8.2.1 Implementation

While the architecture described in this thesis is based on sound principles and the most recent research in the field of data interoperability, a concrete implementation and demonstration would provide a more convincing argument for the feasibility of the ADI and its architecture. The chapter regarding component construction provides a good start, and performance of such a demonstration could be measured according to the chapter regarding metrics.

8.2.2 Alternate Implementations

Although the chapter regarding component construction provides one way to implement the ADI architecture, the methods described therein are not the only way to implement it. As an example, one possible variation is to reverse the order of construction of the ADI and the Database Access Description. In some cases, a data source will have up to date documentation or methodology for access to all of its data within. In such a case, it might be advantageous for the DBA to create the ADI based on the Database Access Description. Rather than first deciding what data will be exposed, and then trying to reverse engineer methods to provide that data, she could examine a database model that determines the comprehensive set of what data *could* be provided. Then the task of producing an ADI is reduced to a problem of selecting a subset of data from that model.

8.2.3 Relaxation of Simplifying Assumptions

The examples in this thesis made simplifying assumptions so that the concepts could be clearly explained. One such assumption is that of semantic heterogeneity as a process problem. Although a great deal of focus has been on schematic heterogeneity in the past, semantic heterogeneity is beginning to emerge in the minds of researchers as a problem with equal, if not more, significance. A future task is to incorporate methods of solving semantic heterogeneity to help automate the generation of Application Requirements within the ADI architecture.

A second simplifying assumption made in this thesis was an environment consisting of multiple applications and one database. The ADI concept theoretically extends to many applications and many databases. However, there are even more complicated semantic heterogeneity issues that need to be considered when attempting to allow an application to access and work with data from two different data sources, such as determining overlaps between domains of data, and then resolving and unifying the overlapping instances. Another future task is to apply the ADI architecture to environments with multiple data sources and consumers.

A third simplifying assumption was the assumption that the data source was a relational database. This assumption was made for the purposes of the examples, because of the popularity and convenience of the relational model. Some of the examples appear the way that they do because of the choice of this assumption. Nevertheless, it should be stressed that the examples do not depend on the assumption of a relational model. Instead, different database models will require slightly different but analogous ADI components. A future task is to explicitly specify what models and constructs are analogous to the examples presented in this thesis, and then to implement those models and constructs.

8.2.4 Application Focus

The ADI differs from some of the approaches discussed in the previous chapter including the TSIMMIS System, the SIMS information mediator, and the Query Mediation techniques in that it focuses on allowing multiple applications to access a single data source. These approaches, on the other hand, focus on allowing an application to access multiple data sources. Future work should investigate the possibility of an Abstract Application Interface, an approach that would attempt the converse of the focus taken with the ADI, as in these other approaches. Afterwards, a study could be conducted to see which cases of data interoperability problems were solved more efficiently when using either focus. It is conceivable that such a study could yield a hybrid approach.

8.3 Concluding Remarks

Achieving data interoperability is well recognized as a problem that currently has no ideal solution: tradeoffs must always be made between quality and cost of information. The ADI architecture represents a step towards improved data interoperability that should be implemented and evaluated in a real-world environment. Although much research has been conducted, and many advances made, researchers need to continue to work on this problem, lest the growing amount of digital information completely overwhelm the data systems of the future.

9 Appendix A. Glossary of Acronyms

Acronyms can be convenient to use, and the computer world is brimming with them. However, they can also befuddle readers who are not aware of their meaning. This table of the various acronyms used in this thesis is provided as an aid to the reader.

ACM	Association for Computing Machinery
ADI	Abstract Data Interface
ADDS	Amoco Distributed Database System
ARD	Application Requirements Description
CGI	Common Gateway Interface
COIN	COntext INterchange
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off-the-Shelf
DAD	Database Access Description
DBA	DataBase Administrator
DBMS	Data Base Management System
DFDS	Data and Format Descriptor Set
DDL	Data Definition Language
DDS	Data Descriptor Set
DIOM	Distributed Interoperable Object Model
DoD	Department of Defense
DOME	Domain Ontology Management Environment

ER	Entity Relationship
FAM	Fully Attributed Model
GIM	Generic Integration Model
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
ICAM	Integrated Computer Aided Manufacturing
IDEF1X	ICAM DEFinition 1 eXtended
IDL	Interface Definition Language
IEEE	Institute of Electrical and Electronics Engineers
IRO-DB	Interoperable Relational & Object-oriented DataBases
MAP	Mission-Aircraft-Pilot database
MASM	Multi-Aspect Semantic Model
MRDSM	Multics Relational Data Store Multidatabase
MTF	Message Text Format
OQL	Object Query Language
ODMG	Object Data Management Group
POTS	Plain Old Telephone Service
PRECI*	Prototype of a RElational Canonical Interface
RPC	Remote Procedure Call
SIGMA[FDB]	Schema Integration & Global Integrity Maintenance Approach for Federated Databases
SIM	Schema Integration Methodology
SKAT	Semantic Knowledge Articulation Tool

SOAP	Simple Object Access Protocol
SQL	Structured Query Language
TSIMMIS	The Stanford-IBM Manager of Multiple Information Sources
TL	Translator Library
VHDBS	*Verteiltes Heterogenes DatenBankSystem*
XML	eXtensible Markup Language
XQL	XML Query Language
XSL	eXtensible Stylesheet Language

10 Appendix B. Sample DDL for the MAP Example

This appendix includes the sample DDL for the MAP example. This DDL was generated using ERwin version 3.52. ERwin is a database design tool that permits the creation of a *visual blueprint* or *data model*. Among its many features are the ability to draw various data models including IDEF1X, IE, and ER models; and reverse and forward engineer between data models and DBMS code. [CA, 2000]

map.sql

```
'  Starting Access Basic DAO Session...

Dim ERwinWorkspace As Workspace
Dim ERwinDatabase As Database
Dim ERwinTableDef As TableDef
Dim ERwinQueryDef As QueryDef
Dim ERwinIndex As Index
Dim ERwinField As Field
Dim ERwinRelation As Relation

Set ERwinWorkspace = DBEngine.WorkSpaces(0)

Set ERwinDatabase =
ERwinWorkspace.OpenDatabase(sERwinDatabase)

'  CREATE TABLE "AIRCRAFT"

Set ERwinTableDef = ERwinDatabase.CreateTableDef("AIRCRAFT")
Set ERwinField = ERwinTableDef.CreateField("aircraft-type",
DB_TEXT, 18)
ERwinField.Required = True
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("aircraft-number",
DB_INTEGER)
ERwinField.Required = True
ERwinTableDef.Fields.Append ERwinField
```

```
Set ERwinField = ERwinTableDef.CreateField("fuel-level",
DB_LONG)
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("fuel-level-units",
DB_TEXT, 18)
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("ammo", DB_INTEGER)
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("genl-cond",
DB_TEXT, 18)
ERwinTableDef.Fields.Append ERwinField
ERwinDatabase.TableDefs.Append ERwinTableDef
Set ERwinField = ERwinTableDef.Fields("ammo")
SetFieldProp (ERwinField, "Caption", DB_TEXT, "ammunition:")
Set ERwinField = ERwinTableDef.Fields("genl-cond")
SetFieldProp (ERwinField, "Caption", DB_TEXT, "general-
condition:")

'  CREATE INDEX "PrimaryKey"

Set ERwinTableDef = ERwinDatabase.TableDefs("AIRCRAFT")
Set ERwinIndex = ERwinTableDef.CreateIndex("PrimaryKey")
Set ERwinField = ERwinIndex.CreateField("aircraft-type")
ERwinIndex.Fields.Append ERwinField
Set ERwinField = ERwinIndex.CreateField("aircraft-number")
ERwinIndex.Fields.Append ERwinField
ERwinIndex.Primary = True
ERwinIndex.Clustered = True
ERwinTableDef.Indexes.Append ERwinIndex

'  CREATE TABLE "ASSIGNMENT"

Set ERwinTableDef = ERwinDatabase.CreateTableDef("ASSIGNMENT")
Set ERwinField = ERwinTableDef.CreateField("mission-number",
DB_INTEGER)
ERwinField.Required = True
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("pilot-number",
DB_INTEGER)
ERwinField.Required = True
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("aircraft-type",
DB_TEXT, 18)
ERwinField.Required = True
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("aircraft-number",
DB_INTEGER)
```

```
ERwinField.Required = True
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("assignment-time",
DB_DATETIME)
ERwinField.Required = True
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("AOBJ", DB_TEXT,
18)
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("ST", DB_DATETIME)
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("CT", DB_DATETIME)
ERwinTableDef.Fields.Append ERwinField
ERwinDatabase.TableDefs.Append ERwinTableDef
Set ERwinField = ERwinTableDef.Fields("AOBJ")
SetFieldProp (ERwinField, "Caption", DB_TEXT, "assignment-
objective:")
Set ERwinField = ERwinTableDef.Fields("ST")
SetFieldProp (ERwinField, "Caption", DB_TEXT, "start-time:")
Set ERwinField = ERwinTableDef.Fields("CT")
SetFieldProp (ERwinField, "Caption", DB_TEXT, "completion-
time:")


'  CREATE INDEX "PrimaryKey"

Set ERwinTableDef = ERwinDatabase.TableDefs("ASSIGNMENT")
Set ERwinIndex = ERwinTableDef.CreateIndex("PrimaryKey")
Set ERwinField = ERwinIndex.CreateField("mission-number")
ERwinIndex.Fields.Append ERwinField
Set ERwinField = ERwinIndex.CreateField("pilot-number")
ERwinIndex.Fields.Append ERwinField
Set ERwinField = ERwinIndex.CreateField("aircraft-type")
ERwinIndex.Fields.Append ERwinField
Set ERwinField = ERwinIndex.CreateField("aircraft-number")
ERwinIndex.Fields.Append ERwinField
Set ERwinField = ERwinIndex.CreateField("assignment-time")
ERwinIndex.Fields.Append ERwinField
ERwinIndex.Primary = True
ERwinIndex.Clustered = True
ERwinTableDef.Indexes.Append ERwinIndex

'  CREATE TABLE "MISSION"

Set ERwinTableDef = ERwinDatabase.CreateTableDef("MISSION")
Set ERwinField = ERwinTableDef.CreateField("mission-number",
DB_INTEGER)
```

```
ERwinField.Required = True
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("mission-type",
DB_TEXT, 18)
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("mission-priority",
DB_INTEGER)
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("mission-creation-
date", DB_DATETIME)
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("mission-
objective", DB_TEXT, 18)
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("mission-codename",
DB_TEXT, 18)
ERwinTableDef.Fields.Append ERwinField
ERwinDatabase.TableDefs.Append ERwinTableDef

'  CREATE INDEX "PrimaryKey"

Set ERwinTableDef = ERwinDatabase.TableDefs("MISSION")
Set ERwinIndex = ERwinTableDef.CreateIndex("PrimaryKey")
Set ERwinField = ERwinIndex.CreateField("mission-number")
ERwinIndex.Fields.Append ERwinField
ERwinIndex.Primary = True
ERwinIndex.Clustered = True
ERwinTableDef.Indexes.Append ERwinIndex

'  CREATE TABLE "MODEL"

Set ERwinTableDef = ERwinDatabase.CreateTableDef("MODEL")
Set ERwinField = ERwinTableDef.CreateField("aircraft-type",
DB_TEXT, 18)
ERwinField.Required = True
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("top-speed",
DB_INTEGER)
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("top-speed-units",
DB_TEXT, 18)
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("weapons-capacity",
DB_INTEGER)
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("fuel-capacity",
DB_LONG)
```

```
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("fuel-capacity-
units", DB_TEXT, 18)
ERwinTableDef.Fields.Append ERwinField
ERwinDatabase.TableDefs.Append ERwinTableDef

'  CREATE INDEX "PrimaryKey"

Set ERwinTableDef = ERwinDatabase.TableDefs("MODEL")
Set ERwinIndex = ERwinTableDef.CreateIndex("PrimaryKey")
Set ERwinField = ERwinIndex.CreateField("aircraft-type")
ERwinIndex.Fields.Append ERwinField
ERwinIndex.Primary = True
ERwinIndex.Clustered = True
ERwinTableDef.Indexes.Append ERwinIndex

'  CREATE TABLE "PILOT"

Set ERwinTableDef = ERwinDatabase.CreateTableDef("PILOT")
Set ERwinField = ERwinTableDef.CreateField("plt_no",
DB_INTEGER)
ERwinField.Required = True
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("plt_name",
DB_TEXT, 18)
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("plt_rnk",
DB_INTEGER)
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("plt_train",
DB_TEXT, 18)
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("hrs_fln",
DB_DOUBLE)
ERwinTableDef.Fields.Append ERwinField
Set ERwinField = ERwinTableDef.CreateField("hrs_fln_accy",
DB_DOUBLE)
ERwinTableDef.Fields.Append ERwinField
ERwinDatabase.TableDefs.Append ERwinTableDef
Set ERwinField = ERwinTableDef.Fields("plt_no")
SetFieldProp (ERwinField, "Caption", DB_TEXT, "pilot-number:")
Set ERwinField = ERwinTableDef.Fields("plt_name")
SetFieldProp (ERwinField, "Caption", DB_TEXT, "pilot-name:")
Set ERwinField = ERwinTableDef.Fields("plt_rnk")
SetFieldProp (ERwinField, "Caption", DB_TEXT, "pilot-rank:")
Set ERwinField = ERwinTableDef.Fields("plt_train")
```

```
SetFieldProp (ERwinField, "Caption", DB_TEXT, "pilot-
training:")
Set ERwinField = ERwinTableDef.Fields("hrs_fln")
SetFieldProp (ERwinField, "Caption", DB_TEXT, "hours-flown:")

'  CREATE INDEX "PrimaryKey"

Set ERwinTableDef = ERwinDatabase.TableDefs("PILOT")
Set ERwinIndex = ERwinTableDef.CreateIndex("PrimaryKey")
Set ERwinField = ERwinIndex.CreateField("plt_no")
ERwinIndex.Fields.Append ERwinField
ERwinIndex.Primary = True
ERwinIndex.Clustered = True
ERwinTableDef.Indexes.Append ERwinIndex

'  CREATE RELATIONSHIP "R/4"

Set ERwinRelation = ERwinDatabase.CreateRelation("R/4",
"MODEL", "AIRCRAFT")
Set ERwinField = ERwinRelation.CreateField("aircraft-type")
ERwinField.ForeignName = "aircraft-type"
ERwinRelation.Fields.Append ERwinField
ERwinDatabase.Relations.Append ERwinRelation

'  CREATE RELATIONSHIP "R/6"

Set ERwinRelation = ERwinDatabase.CreateRelation("R/6",
"AIRCRAFT", "ASSIGNMENT")
Set ERwinField = ERwinRelation.CreateField("aircraft-type")
ERwinField.ForeignName = "aircraft-type"
ERwinRelation.Fields.Append ERwinField
Set ERwinField = ERwinRelation.CreateField("aircraft-number")
ERwinField.ForeignName = "aircraft-number"
ERwinRelation.Fields.Append ERwinField
ERwinDatabase.Relations.Append ERwinRelation

'  CREATE RELATIONSHIP "R/3"

Set ERwinRelation = ERwinDatabase.CreateRelation("R/3",
"MISSION", "ASSIGNMENT")
Set ERwinField = ERwinRelation.CreateField("mission-number")
ERwinField.ForeignName = "mission-number"
ERwinRelation.Fields.Append ERwinField
ERwinDatabase.Relations.Append ERwinRelation

'  CREATE RELATIONSHIP "R/2"
```

```
Set ERwinRelation = ERwinDatabase.CreateRelation("R/2",
"PILOT", "ASSIGNMENT")
Set ERwinField = ERwinRelation.CreateField("plt_no")
ERwinField.ForeignName = "pilot-number"
ERwinRelation.Fields.Append ERwinField
ERwinDatabase.Relations.Append ERwinRelation

ERwinDatabase.Close
ERwinWorkspace.Close
'  Terminating Access Basic DAO Session...
```

11 Appendix C. IDEF1X Background

Robert G. Brown originally conceived of IDEF1X in 1979 while working as a consultant at Lockheed. It was based on evolving relational database theory, as well as work by early database researchers such as Chen, Codd, Smith, and others. The following year, Brown brought his ideas to the Bank of America, which was at that time, struggling with delivery and database applications. A need for information modeling techniques and data-centered design concepts was identified at the bank, and Brown's ideas received some healthy support. Those ideas eventually were named ADAM internally within the bank, and Data Modeling Technique, or DMT, outside the bank.

Other organizations began to recognize a need for data modeling standards around that time. The U. S. Air Force conducted studies known as Integrated Computer Aided Manufacturing, which identified a set of three graphic methods for defining the functions, data structures, and dynamics of manufacturing businesses. These three methods became to be known as the IDEF methods, short for *I*CAM *DEF*inition. The function method was dubbed IDEF0; the data method, IDEF1; and the dynamics method, IDEF2.

Brown retained the rights to ADAM after his departure from the bank in 1985 through his company, the Data Base Design Group. ADAM became commercially available through an arrangement with the D. Appleton Company, otherwise known as DACOM. Later that same year, DACOM approached the U. S. Air Force, which by then was a major user of the IDEF methods, to propose extensions to IDEF1 by adding some capabilities available in ADAM. Instead of extending IDEF1 with ADAM, the Air Force elected to replace IDEF1 completely with ADAM, and the new model came to be known as IDEF1X—the proper pronunciation of IDEF1X is “eye deaf one ecks,” and the X stands for eXtended. IDEF1X was accepted as an Air Force standard shortly after. [Bruce, 1992]

12 Appendix D. IDEF1X Notation Conventions

As a convenience to the reader, the relevant IDEF1X notation explanations are provided here to aid in the interpretation of the figures that use them. [Adapted from Bruce, 1992] Specifically, the figures that utilize IDEF1X notation are Figures 5.1, 5.2, and 5.3 in Chapter 5.

IDEF1X has powerful data model expression capabilities. However, a comprehensive description of all of the abilities of IDEF1X is beyond the scope of this thesis. Readers are urged to consult [Bruce, 1992] for more information.

12.1 Entity Notation

An entity is said to be an *independent entity* when it depends on no other entities for its identification. Such an entity is denoted thus:

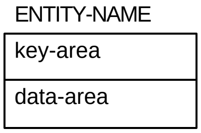

Figure 12.1. Independent Entity

An entity is said to be a *dependent entity* when it does depend on others for its identification. Such an entity is denoted thus:

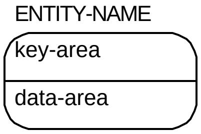

Figure 12.2. Dependent Entity

12.2 Attribute Notation

An attribute denoted thus:

$$attribute(FK)$$

has a foreign key associated with it. A primary key of another entity is contributed via a relationship.

12.3 Relationship Notation

The following figure denotes a *one-to-many identifying* relationship: one parent, to zero or more children:

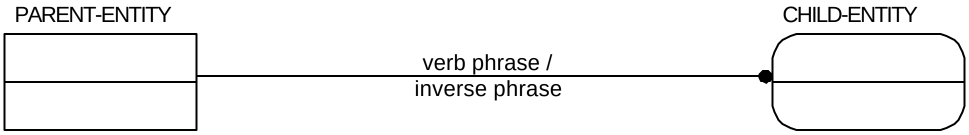

Figure 12.3. One-to-Many Relationship

In such a relationship, the primary key attributes of the parent entity become the primary key attributes of the child entity.

13 Bibliography

Abiteboul, S., Cluet, S., Milo, T., Mogilevsky, P., Simeon, J., Zohar, S. "Tools for Data Translation and Integration." *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering.* 1999.

Arens, Y., Knoblock, C. A., and Hsu, C. "Query Processing in the SIMS Information Mediator." *Advanced Planning Technology*. AAAI Press, Menlo Park, CA, 1996.

Batini, C., Lenzerini, M., and Navathe, S. "A comparative analysis of methodology for database schema integration." *ACM Computing Surveys*, Vol. 18, No. 4, 1986.

Benslimane, D., Yetongnon, K., Chraibi, S., Leclercq, E., Abdelwahed, E. H. "DECA: A Framework for Cooperative Information Systems." *Technique et Science Informatiques*, Vol. 19, No. 7. September, 2000.

Bishr, Y. A., Pundt, H., Ruther, C. "Proceeding on the road of semantic interoperability—design of a semantic mapper based on a case study from transportation." Interoperating Geographic Information Systems. *Proceedings from the 2nd International Conference, INTEROP'99.* 1999.

Bouguettaya, A. Ontologies and Databases. Kluwer Academic Publishers, 1999.

Breitbart, Y., Olson, P. L., Thompson, G. R. "Database Integration in a Distributed Heterogeneous Database System." *IEEE Conference on Data Engineering.* February, 1986.

Bressan, S., Goh, C. H. "Answering Queries in Context." *Datalogiske Skrifter*, No. 78, 1998.

Bruce, T. A. Designing Quality Databases with IDEF1X Information Models. Dorset House Publishing, 1992.

Cardiff, J., Catarci, T., and Santucci, G. "Exploitation of Interschema Knowledge in a Multidatabase System." In *Proceedings of the 4th KRDB Workshop*, Athens, Greece, 1997.

Castano, S., and DeAntonellis, V. "Global Viewing of Heterogeneous Data Sources." *IEEE Transactions on Knowledge and Data Engineering*, Vol. 13, No. 2, March-April, 2001.

Codd, E. F. "A Relational Model of Data for Large Shared Data Banks." *Communications of the ACM*, Vol. 13, No. 6. June, 1970.

Cohen, W. "Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity." In *Proceedings of ACM SIGMOD-98*, Seattle, WA, 1998.

Committee on Innovations in Computing and Communications (CICC): Lessons from History, National Research Council. "Funding a Revolution: Government Support for Computing Research." 1999.

Computer Associates (CA). "ERwin: Features Guide." Computer Associates International, Inc., Islandia, NY, 2000.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. Introduction to Algorithms. MIT Press, 1990.

Coulomb, R. M. "Impact of Semantic Heterogeneity on Federating Databases." *Computer Journal,* Vol. 40, No. 5. Oxford University Press, 1997.

Cui, Z., O'Brien, P. "Domain Ontology Management Environment." *Proceedings of the 33rd Annual Hawaii International Conference on System Sciences.* IEEE Computer Society, 2000.

Deen, S. M., Amin, R. R., Ofori-Dwumfuo, G. O., and Taylor, M. C. "The Architecture of a Generalised Distributed Database System—PRECI*." *The Computer Journal*, Vol. 28, No. 3. 1985.

Department of Defense (DoD). *Software Technology Strategy*. Decenber, 1991.

Essmayr, W., Kastner, F., Pernul, G., and Tjoa, A. M. "The Security Architecture of IRO-DB." In *Proceedings of the Twelfth IFIP Conference on Information Security,* Island of Samos, Greece, May, 1996.

Evans, M. W., and Marciniak, J. Software Quality Assurance and Management. John Wiley & Sons, Inc., New York, NY, 1987.

Fankhauser, P. "Methodology for Knowledge-Based Schema Integration." Ph. D. Thesis, University of Vienna, Austria, December, 1997.

Fahl, G., Risch, T., and Skold, M. "AMOS—An Architecture for Active Mediators." *Proceedings of the International Workshop on Next Generation Information Technologies and Systems.* Haifa, Israel, June, 1993.

Gardarin, G., Sha, F. "Using Conceptual Modeling and Intelligent Agents to Integrate Semi-Structured Documents in Federated Databases." *Conceptual Modeling: Current Issues and Future Directions.* 1999.

Gardarin, G., Sha, F., Ngoc, T. D. "XML-Based Components for Federating Multiple Heterogeneous Data Sources." *Proceedings of the 18th International Conference on Conceptual Modeling.* Paris, France, 1999.

Goodchild, M. F., Egenhofer, M. J., and Fegeas, R. "Interoperating GISs." *Report of a Specialist Meeting Held Under the Auspices of the Varenius Project Panel on Computational Implementations of Geographic Concepts*, December 1997.

Goh, C., Bressan, S., Madnick, S., and Siegel, M. "Context Interchange: New Features and Formalisms for the Intelligent Integration of Information." *ACM Transactions on Information Systems*, Vol. 17, No. 3, July 1999.

Goh, C., Madnick, S., and Siegel, M. "Context Interchange: Overcoming the Challenges of Large-Scale Interoperable Database Systems in a Dynamic Environment." In *Proceedings of the Third International Conference on Information and Knowledge Management*, 1994.

Gravano, L., Garcia-Molina, H., Tomasic, A. "The Effectiveness of GlOSS for the Text Database Discovery Problem." In *ACM SIGMOD Record*, 1994.

Gupta, A. Integration of Information Systems: Bridging Heterogeneous Databases. IEEE Press, 1989.

Hall, G. "Negotiation in Database Schema Integration." Presented at *The Inaugural Association for Information Systems Americas Conference*, Pittsburgh, PA, August, 1995.

Hammer, J., Garcia-Molina, H., Ireland, K., Papakonstantinou, Y., Ullman, J., Widom, J. "Information Translation, Mediation, and Mosaic-Based Browsing in the TSIMMIS System." In *Proceedings of ACM SIGMOD*, San Jose, CA, 1995.

Hammer, J., and McLeod, D. "An Approach to Resolving Semantic Heterogeneity in a Federation of Autonomous, Heterogeneous Database Systems." In *International Journal of Intelligent & Cooperative Information Systems*, World Scientific, Vol. 2, No. 1, 1993.

Hammer, J., Breunig, M., Garcia-Molina, H., Nestorov, S., Vassalos, V., Yerneni, R. "Template-Based Wrappers in the TSIMMIS System." In *Proceedings of the Twenty-Sixth SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 12-15, 1997.

Hasselbring, W. "Federated Integration of Replicated Information Within Hospitals." *International Journal on Digital Libraries*, Vol. 1, No. 3. November, 1997.

Huang, H. C., Kerridge, J., Chen, S. L. "A Query Mediation Approach to Interoperability of Heterogeneous Databases." *Proceedings of the 11th Australasian Database Conference*. IEEE Computer Society, 1999.

Hull, R. "Managing Semantic Heterogeneity in Databases: A Theoretical Perspective." *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM, 1997.

Institute of Electrical and Electronics Engineers (IEEE). IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY, 1990.

Jannink, J., Mitra, P., Neuhold, E., Pichai, S., Studer, R., Wiederhold, G. "An Algebra for Semantic Interoperation of Semistructured Data." *Proceedings of the 1999 Workshop on Knowledge and Data Engineering Exchange*. IEEE Computer Society, 2000.

Jones, S. B., Franklin, J. "Integration of Heterogeneous Biotechnology Databases." *Proceedings of the 1998 International Chemical Information Conference*. Infonortics. Tetbury, UK, 1998.

Kashyap, V., and Sheth, A. "Semantic Heterogeneity in Global Information Systems: The Role of Metadata, Context and Ontologies." In *Cooperative Information Systems: Current Trends and Directions*, Papazoglou, M., and Schlageter, G., eds. 1996.

Kedad, Z., Metais, E. "Dealing with Semantic Heterogeneity During Data Integration." *Proceedings of the 18th International Conference on Conceptual Modeling*. 1999.

Keller, A. M. "The Role of Semantics in Translating View Updates." *IEEE Computer*. January, 1986.

Kitakami, H., Mori, Y., Airkawa, M., Sato, A. "Integration Method for Biological Taxonomy Databases in the Presence of Semantic Heterogeneity." *Transactions of the Institute of Electronics & Communication Engineers of Japan*, Part D. Vol. J82D-I, No. 1. January, 1999.

Kiyoki, Y., Kitagawa, T., Hitomi, Y. "A Fundamental Framework for Realizing Semantic Interoperability in a Multidatabase Environment." *Integrated Computer-Aided Engineering*, Vol. 2, No. 1. Wiley, USA, 1995.

Leclercq, E., Benslimane, D., Yetongnon, K. "HORUS: A Semantic Broker for GIS Interoperability." *First International Workshop on Telegeoprocessing*. Lyon, France, 1999.

Lee, J. O., Baik, D. K. "SemQL: A Semantic Query Language for Multidatabase Systems." *Proceedings of the Eighth International Conference on Information Knowledge Management*. ACM, 1999.

Lee, J. O., Baik, D. K. "Semantic Integration of Information Based on the Multi Aspect Semantic Model." *Joint Conference on Intelligent Systems.* 1999.

Litwin, W., and Abdellatif, A. "Multidatabase Interoperability." *IEEE Computer.* December, 1986.

Liu, L., and Pu, C. "An Adaptive Object Oriented Approach to Integration and Access of Heterogeneous Information Sources." *Distributed and Parallel Databases,* Vol. 5, No. 2. April, 1997.

Masood, N., and Eaglestone, B. "Semantics Based Schema Analysis." Database and Expert Systems Applications. *Proceedings of the 9th International Conference, DEXA '98.* 1998.

Microtool GmbH. "OOP by Scripting—Customized Code Generation." Microtool GmbH, Berlin, 2001.

Miller, R. J. "Using Semantically Heterogeneous Structures." In *ACM SIGMOD '98.* Seattle, WA, USA, 1998.

Miller, R. J., Ioannidis, Y. E., Ramakrishnan, R. "The Use of Information Capacity in Schema Integration and Translation." *Proceedings of the 19th VLDB Conference.* Dublin, Ireland, 1993.

Mitra, P., Wiederhold, G., Jannink, J. "Semi-automatic Integration of Knowledge Sources." *Proceedings of the Second International Conference on Information Fusion.* Mountain View, CA, USA, 1999.

Motz, R., Fankhauser, P. "Propagation of Semantic Modifications to an Integrated Schema." *Proceedings of the 3rd IFCIS International conference on Cooperative Information Systems.* IEEE Computer Society, 1998.

Ouksel, A. M., and Ahmed, I. "Ontologies are not the Panacea in Data Integration: A Flexible Coordinator to Mediate Context Construction." *Distributed and Parallel Databases.* Kluwer Academic Publishers, Netherlands, 1999.

Phoha, S. "Information Quality Control for Network Centric Ship Maintenance." *Proceedings of the 1999 American Control Conference.* IEEE, 1999.

Renner, Scott A., and Scarano, James G. "Data Interoperability: Standardization or Mediation." *DOD Database Colloquium '95,* August 1995.

Renner, Scott. "Data Issues." *AF Architecture Workshop*, November 1999.

Richardson, R., Smeaton, A. F. "An Information Retrieval Approach to Locating Information in Large Scale Federated Database Systems." Applications of Natural Language to Information Systems. *Proceedings of the Second International Workshop*. Amsterdam, Netherlands, 1996.

Roth, M. T., Arya, M., Haas, L., Carey, M., Cody, W., Fagin, R., Schwarz, P., Thomas, J., and Wimmers, E. "The Garlic Project." In *ACM SIGMOD*, Montreal, Canada, 1996.

Saake, G., Christiansen, A., Conrad, S., Hoding, M., Schmitt, I., Turker, C. "The Federation of Heterogeneous Database Systems and Local Data Components for Ensuring System-Wide Integrity—a Short Introduction to the SIGMA[FDB] Project." *Databases in Office, Technology, and Science.* 1997.

Sciore, E., Siegel, M., and Rosenthal, A. "Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems." In *ACM Transactions on Database Systems*, Vol. 19, No. 2, June 1994.

Shen, W. M., Zhang, W., Wang, X., and Arens, Y. "Discovering and constructing conceptual object model from large instances of relational databases." *International Journal on Data Mining and Knowledge Discovery*. January, 1999.

Sheth, A., and Kashyap, V. "So Far (Schematically) yet So Near (Semantically)." In *IFTP TC2/WG2.6 Conference on Semantics of Interoperable Database Systems*. Elsevier Scientific Publisher B. V., November 1992.

Sheth, A. P., and Larson, J. A. "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases." *ACM Computing Surveys*, Vol. 22, No. 3, 1990.

Stern, D. "New Search and Navigation Techniques in the Digital Library." *Source and Technology Libraries*, Vol. 17, No. 3-4. 1999.

Templeton, M., Brill, D., Dao, S. K., Lund, E., Ward, P., Chen, A. L. P., and MacGregor, R. "Mermaid—A Front-End to Distributed Heterogeneous Databases." *Proceedings of the IEEE*, Vol. 75, No. 5. May, 1987.

Vermeer, M. W. W., and Apers, P. M. G. "On the Applicability of Schema Integration Techniques to Database Interoperation." In *Proceedings of the Fifteenth International Conference on Conceptual Modelling*, Cottbus, Germany. Springer-Verlag, Berlin, 1996.

Vidal, V. M. P. Loscio, B. F. "Solving the Problem of Semantic Heterogeneity in Defining Mediator Update Translators." *Proceedings of the 18th International Conference on Conceptual Modeling.* 1999.

Winters, Melanie, and Wilczynski, Brian. "Data Interoperability: Foundation of Information Superiority." *CHIPS*, July 2000.

Wood, J. "What's in a Link?" In *Readings in Knowledge Representation*. Morgan Kaufmann, 1985.

Wu, X. "A CORBA-Based Architecture for Integrating Distributed and Heterogeneous Databases." *Proceedings of the 5th IEEE International Conference on Engineering of Complex Computer Systems*. IEEE Computer Society, 1999.

Wu, X. "Integrating Heterogeneous Database Systems to an Object Oriented Client/Server Architecture." Data Mining, Data Warehousing, and Client/Server Databases. *Proceedings of the 8th International Database Workshop*. 1997.

Zisman, A., Kramer, J. "An Architecture to Support Interoperability of Autonomous Database Systems." In *2nd International Baltic Workshop on DB and IS*. Estonia-Tallin, June 1996.