UNIVERSITY of MANITOBA

Automatic Conflict Resolution to Integrate Relational Schema

A dissertation submitted in partial satisfaction of the
requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Ramon Lawrence

May 2001

Automatic Conflict Resolution to Integrate Relational Schema

Acknowledgements

I want to thank my family for being supportive over all these years, especially my beautiful wife, Carri, and my mother, Gloria. I would also like to recognize the tireless efforts of my supervisor, Dr. Ken Barker, and the committee for evaluating this work. Finally, I am very appreciative to funding agencies such as NSERC, TRLabs, and the University of Manitoba whose financial support made this thesis possible.

Abstract

Automatic Conflict Resolution to Integrate Relational Schema

by

Ramon Lawrence

With the constantly increasing reliance on database systems to store, process, and display data comes the additional problem of ensuring interoperability between these systems. On a wider scale, the World-Wide Web (WWW) provides users with the ability to access a vast number of data sources distributed across the planet. However, a fundamental problem with distributed data access is the determination of semantically equivalent data. Ideally, users should be able to extract data from multiple sites and have it automatically combined and presented to them in a usable form. No system has been able to accomplish these goals due to limitations in expressing and capturing data semantics.

Schema integration is required to provide database interoperability and involves the resolution of naming, structural, and semantic conflicts. To this point, automatic schema integration has not been possible. This thesis demonstrates that integration may be increasingly automated by capturing data semantics using a standard dictionary.

This thesis proposes an architecture for automatically constructing an integrated view by combining local views that are defined by independently expressing database semantics in XML documents (X-Specs) using only a pre-defined dictionary as a binding between integration sites. The dictionary eliminates naming conflicts and reduces semantic conflicts. Structural conflicts are resolved at query-time by translating from the semantic integrated view to structural queries. The system provides both logical and physical access transparency by mapping user queries on high-level concepts to schema elements in the underlying data sources. The architecture automatically integrates relational databases, and its application of standardization to the integration problem is unique.

The architecture may be deployed in a centralized or distributed fashion, and preserves full database autonomy while allowing transparent access to all databases participating in a global federation without the user's knowledge of the underlying data sources, their location, and their structures. Thus, the contribution is a system which provides

system transparency to users, while preserving autonomy for all systems. A distributed deployment allows integration using a web browser, and would have a major impact on how the Web is used and delivered.

The integration software, Unity, is the bridge between concept and implementation. Unity is a complete software package for the construction and modification of standard dictionaries, parsing of database schema and metadata to construct X-Specs, combining X-Specs into an integrated view, and for transparent querying. Integration results obtained using Unity illustrate the usefulness of the approach.

Contents

List of Figures

Chapter I

Introduction

I.1 Motivation

Interoperability of database systems is becoming increasingly important as organizations increase their number of operational systems and add new decision-support systems. The construction, operation, and maintenance of these systems is complicated and time-consuming and the complexity grows quickly as the number of systems increases. Integrating diverse data sources is required for interoperability. Schema integration is involved in constructing both a multidatabase system (MDBS) and a data warehouse (DW). Both of these architectures are finding more applications because they allow high-level, transparent access to data across multiple sites and provide a uniform, encompassing view of the data in an organization. Thus, a method for simplifying schema integration would be of great theoretical and practical importance.

On a wider-scale, the World-Wide Web is a massive source of information distributed across the globe. This information is exchanged and displayed using the standardized protocol TCP/IP and the language HTML. However, the information in these web pages is far from standardized. Web users are still responsible for finding, and more importantly, deciphering the vast quantities of data presented to them. By presenting a framework for capturing data semantics, it is more likely that databases that were never intended to work together can be made to interoperate. This is especially important on the WWW where users want to access data from multiple, unrelated sources.

Although algorithms have been proposed for schema integration, none are sufficiently automated for real-world implementations as they either require database designers

to explicitly integrate knowledge between all data sources or are not sufficiently automated to integrate many, diverse data sources in a timely fashion. Manual integration processes do not scale well when the number and size of the databases increases. Industry standards have been proposed as a bridge measure to ensure systems can at least communicate knowledge. These systems achieve increased automation by acceptance of restrictive standardization. This work combines these two ideas by using standardization to increase automation but still allows high flexibility by utilizing intelligent integration algorithms.

All applications require a notion of data semantics. Although it is not immediately apparent, capturing the meaning of data is fundamental in any computer system. Computers simply store, process, and display information. Computer systems have been developed in a user-centric way which assumes the user or designer is responsible and able to define and appreciate the data semantics inherent in a web page, data warehouse, or database table. As our systems grow larger and interoperate across networks, it is becoming increasingly difficult for users and programmers to fully understand the semantics of the data and applications they use and develop, and the vast numbers of other data sources that they interact with daily.

This thesis shows how to describe database semantics in a system-centric way which allows the computer system to determine the semantics of the data it processes, and thus free the user from this mundane task. By using data semantics, a schema integration methodology is proposed which is a foundation component in an overall architecture to support database interoperability.

I.2 Thesis

The goal is to define a system capable of automatically integrating relational database schemas and resolving naming, structural, and semantic conflicts. The system preserves full autonomy of the underlying databases which have no direct knowledge of their participation in a global system. The architecture allows transparent querying of all data sources participating in a global federation without the user's knowledge of the underlying data sources, their location, and their structures. Thus, a key contribution is a system which provides system transparency to users, while preserving autonomy for all systems.

The integration architecture automatically resolves conflicts inherent in the schema integration problem. The model integrates database systems into a multidatabase using

XML [100] and contains four main components: a standard term dictionary, an XML-based metadata specification language (X-Specs), an integration algorithm for combining X-Specs into an integrated view, and a query processor for resolving structural conflicts at query-time and integrating results. A software package, Unity, is developed which demonstrates the model's functionality. The architecture contribution is the merging of industry standards with new research algorithms to create an automatic schema integration methodology used for the construction of integrated schemas for multidatabases and integrating data sources on the World-Wide Web.

I.3 Overview of Thesis

This thesis describes the integration architecture and its conflict resolution capabilities. Background information on schema integration and a taxonomy classifying integration techniques is in Chapter 2. Chapter 3 details the integration architecture, which utilizes a standard dictionary for identifying similar concepts, a specification language (X-specs) used to exchange metadata on systems, an integration algorithm for combining the metadata specifications together into a unified schema, and a query processor for query execution and result presentation. The implementation issues for the architecture components in Unity are discussed in Chapter 4. A discussion of the architecture including its conflict resolution capabilities, applications, and comparisons with other systems is presented in Chapter 5. Integration examples and results using Unity are in Chapter 6, and the thesis closes with future work and conclusions in Chapter 7.

I.4 Example Databases

Two database schemas are referenced throughout this thesis to illustrate key ideas. The standard Northwind database provided with Microsoft Access© is given in Figure I.1. The other sample database, called Southstorm, is a fictional order database which is poorly designed and non-normalized. The Southstorm schema is shown in Figure I.2. As a convention used throughout this thesis, field and table names appear in text as *table_name* or *field_name*. Later, as we introduce semantic names for database elements and describe the implementation, C++ class names and semantic names are in the form: `semantic_name` and `C++_class_name`. The semantic name mappings are provided in Section VI.3.

Tables	Fields
Categories	CategoryID, CategoryName, Description, Picture
Customers	CustomerID, CompanyName, ContactName, ContactTitle, Address, City, Region, PostalCode, Country, Phone, Fax
Employees	EmployeeID, LastName, FirstName, Title, TitleofCourtesy, BirthDate, HireDate, Address, City, Region, PostalCode, Country, HomePhone, Extension, Photo, Notes, ReportsTo
OrderDetails	OrderID, ProductID, UnitPrice, Quantity, Discount
Orders	OrderID, CustomerID, EmployeeID, OrderDate, RequiredDate, ShippedDate, Shipvia, Freight, ShipName, ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry
Products	ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit, UnitPrice, UnitsInStock, UnitsOnOrder, Reorderlevel, Discontinued
Shippers	ShipperID, CompanyName, Phone
Suppliers	SupplierID, CompanyName, ContactName, ContactTitle, Address, City, Region, PostalCode, Country, Phone, Fax, HomePage

Figure I.1: Northwind Database Schema

Tables	Fields
Orders_tb	Order_num, Cust_name, Cust_address, Cust_city, Cust_pc, Cust_country, Emp_name, Item1_id, Item1_qty, Item1_price, Item2_id, Item2_qty, Item2_price

Figure I.2: Southstorm Database Schema

Chapter II

Background

II.1 Database Interoperability and Schema Integration

The interoperability problem involves combining two or more database systems into a coherent, integrated system. At first glance, it appears that the interoperability problem is a straightforward extension of distributed database systems. Unfortunately, providing interoperability between systems is considerably more challenging than building a distributed database system. The primary difference in these two architectures is *autonomy*.

In a distributed database system, a single database is designed to be distributed across a network. There is typically only one implementation of the database software which is distributed along with the data. Thus, a distributed system does not typically handle heterogeneity in the database software and its underlying protocols. More importantly, a common assumption in distributed database design is the database software at each network node has knowledge of others which cooperate in processing transactions. This implies that query scheduling and transaction management is often a cooperative effort between the distributed sites. Another fundamental difference is the data is organized and agreed upon between the distributed sites. These systems distribute the data mostly for performance. They assume that the database schema is designed to facilitate this distribution, and the data is transacted in such a way that consistency is guaranteed.

The interoperability problem applies to a multidatabase system. A **multidatabase** (MDBS) is a collection of autonomous databases participating in a federation to exchange data. A multidatabase is different than a distributed database because of database system autonomy. **Autonomy** is the independent operation of a database system even though it

may be participating in a global federation. Autonomy implies that no changes are allowed to the implementation, operation, or structure of the database to enable its participation in the MDBS. A more thorough description of autonomy issues is provided by Barker [7].

The basic multidatabase system (MDBS) architecture (see Figure II.1) consists of a global transaction manager (GTM) which handles the execution of global transactions (GTs) and is responsible for dividing them into subtransactions (STs) for submission to the local database systems (LDBSs) participating in the MDBS. Each LDBS in the system has an associated global transaction server (GTS) which converts the subtransactions issued by the GTM into a form usable by the LDBS. Certain algorithms also rely on the GTS for concurrency control and simulating features, such as visible two-phase commit, that the LDBS does not provide. Each LDBS is autonomous so modifications are not allowed. Many proposed algorithms assume the LDBS exhibits certain properties which may violate the principle of full autonomy. Finally, local transactions, not under the control of the GTM, are allowed at each LDBS as if the LDBS was not participating in the MDBS.

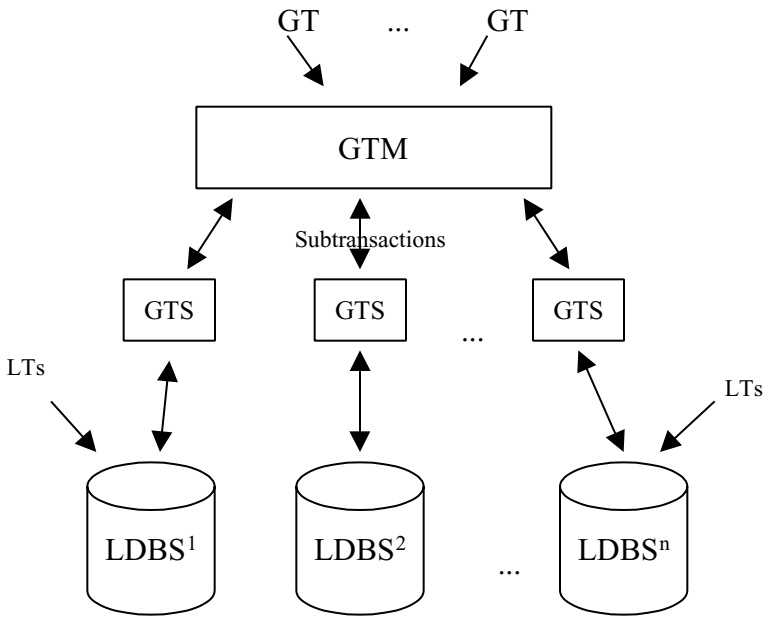

Figure II.1: MDBS Architecture

Global transactions are posed using the global view constructed by integrating the local views provided by each LDBS. A local view may be a subset of the entire local view for the LDBS. Schema integration involves taking the local views provided by the LDBSs

and combining them into a global view by resolving any conflicts between them.

Since databases participating in a multidatabase are automonous, this introduces several new issues not present in a distributed architecture. First, protocols must handle database heterogeneity. Autonomous databases are not assumed to be implemented on the same operating system or have similar architectures, transaction management protocols, or even data models. Second, since each database is still required to accept local transactions related to its primary function, in addition to global transactions originating from its participation in the global federation, the issue of transaction management becomes more complex. Lastly, the knowledge contained in each database must be integrated to provide an integrated view of the data across all databases. This integration of schemas and data is especially complicated. There are surveys [92, 69] of the issues involved in the construction and operation of multidatabase systems and federated systems available.

Providing database interoperability using a multidatabase architecture can be divided into two largely orthogonal problems:

- *Schema and data integration* - the process of combining multiple database schema into an integrated view (schema integration) and then matching the individual data instances and types accordingly (data integration)
- *Transaction management* - the definition of protocols to allow transactions to be efficiently executed on multiple database systems without modifying the existing systems (preserving full autonomy)

Constructing an integrated view is difficult because data sources will store different types of data, in varying formats, with different meanings, and reference it using different names. Subsequently, the construction of the integrated view must handle the different mechanisms for storing data (structural conflicts), for referencing data (naming conflicts), and for attributing meaning to the data (semantic conflicts). Data in individual data sources must be integrated at both the schema level (the description of the data) and the data level (individual data instances). **Schema integration** is the process of combining database schemas into a coherent, integrated view. **Data integration** integrates information at the data item level. This involves such tasks as comparing keys to ensure that they represent the same entity in different databases, handling spatial or temporal data, or combining similar data items stored in different formats.

Transaction execution in a multidatabase must maintain consistency in the underlying data sources. Local and global transactions simultaneously query and update databases. Since the databases have no knowledge of their participation in the MDBS, transaction management software at the global level is required to preserve consistency. Global transaction managers maximize concurrency and performance while maintaining consistency.

Work on transaction management in multidatabase architectures includes Barker's algorithm [6] and serialization using tickets [38]. We have simulated several such algorithms in previous work [61] and have determined via these simulations that Barker's algorithm is more efficient in practice. The simulation results are surprising because Barker's algorithm is a more serial approach than the Ticket method. Massive parallelism of global transactions causes escalating conflicts and delays which reduces overall system performance compared to more regulated, serial-like transaction execution.

Systems which do not allow updates at the global level do not require global level transaction management to co-ordinate updates across systems. The architecture proposed in this thesis is a query-only system. Thus, global transaction management is an issue that will not be further discussed.

II.1.1 Schema Integration

Schema integration is often required between applications or databases with widely differing views of the data and its organization. Thus, integration is hard because conflicts at both the structural and semantic level must be addressed. Further complicating the problem is that most systems do not explicitly capture semantic information. This forces designers performing the integration to impose assumptions on the data and manually integrate various data sources based on those assumptions. Therefore, to perform integration, some specification of data semantics is required to identify related data. Since names and structure in a schema do not always provide a good indication of data meaning, it often falls on the designer to determine when data sources store related or equivalent data.

Schema integration challenges arise due to differences in how the data is represented and stored. The task of integrating systems with different views is difficult, and

there are many factors that cause schema diversity [8, 36]:

- Different user or view perspectives.
- Equivalence among model constructs. For example, a designer uses an attribute in one schema and an entity in another.
- Incompatible design specifications (eg. cardinality constraints, bad name choices).
- Common concepts may be represented differently, although they may be identical, equivalent, compatible, or incompatible.
- Concepts may be related by some semantic property arising only by combining the schemas (interschema properties).

Integration conflicts can be classified into three general categories:

- **Naming conflicts** - occur when systems do not agree on names for concepts.
  - **Homonym naming problem** - the same name is used for two different concepts.
  - **Synonym naming problem** - the identical concept is referenced using two or more different names.
- **Structural conflicts** - arise when data is represented using different models, model constructs, or integrity constraints.
  - **Model diversity problem** - data represented using different models (hierarchical, relational, object-oriented).
  - **Type conflicts** - using different model constructs to represent equivalent data.
  - **Dependency conflicts** - contrasting concept interrelationships in schemas (eg. relationship cardinality, participation)
  - **Key conflicts** - different keys for the same entity.
  - **Behavioral conflicts** - different insertion/deletion/update policies for the identical concept.
  - **Interschema properties** - schema properties that only arise when schemas are combined.
- **Semantic conflicts** - occur when the meaning of the data is not consistent. This often arises due to differing "world views".

A survey by Kim [50] of structural conflicts in relational SQL is available. In this work, structural conflicts are categorized into schema and data conflicts. Schema conflicts include representational conflicts such as handling the situation when a data item

is represented as a table in one schema and as an attribute in another. Schema conflicts also include naming, constraint, and cardinality conflicts. Data conflicts include missing or incorrect data and different data representations (format, units, *etc.*).

Integrating the Northwind and Southstorm databases illustrates these potential conflicts. Type conflicts are the most common. For example, the notion of an ordered item is represented as an attribute of a relation in the Southstorm database and a record-instance of a relationship in the Northwind database. This representational difference causes a dependency conflict as the Northwind database can store any number of items per order, while the Southstorm database can store only two. Both the order and item ids are only unique within each database which causes both a key conflict (the order tables do not have the same key) and an interschema conflict as the order id is no longer unique when the two databases are combined into an integrated system. Finally, a customer can be added to the Northwind database without association with an order whereas a customer can only be added to the Southstorm database with an order and may be added redundantly. This behavior is the result of the poor normalization of the Southstorm database and represents a behavioral conflict between the two systems.

Due to the wide-ranging types of conflicts, it has been argued [26] that automatic schema integration is not feasible.

II.1.2 Data Integration

Data integration is the process of combining data at the entity-level and is challenging because similar data entities may not have the same key. Combining data instances involves entity identification (determining which instances are equivalent), resolving attribute value conflicts (two different values for the same attribute), and handling data conflicts related to field types, sizes, precision, and scaling factors. Entity identification [66] determines the correspondence between object instances from more than one database. Common methods for determining entity equivalence can be grouped into four areas:

- **Key equivalence techniques** - assumes a common key between records. This is especially useful when records have a globally accepted key such as a social security number for a person or a product SKU number for a retail item.

- **User specified equivalence** - has a user performing the record matching algorithm. This manual solution only works with small record sets.
- **Probalistic approaches** - based on some probability function, match equivalent records. The integration accuracy depends on the matching algorithm which may only use portions of the key (probabilistic key equivalence) or use all common attributes (probabilistic attribute equivalence). Both approaches are subject to incorrect and undetected matches.
- **Knowledge-base approaches** - use heuristic rules, prior matching information, or other metadata to increase the probability of correct matchings.

Data integration is further complicated because attribute values may be range values. For example, parametric data is spatial as values vary from one point in space to another. Temporal data is a special case of parametric data. Integrating such data may be extremely complex. Work on data integration includes Lim's work on entity identification [66], using evidential reasoning to resolve attribute conflicts [67], and combining data in parametric databases [35].

Automated data integration using a variant of SQL [90] is possible by attaching context information to simple data values. This context information is stored using defined names and values, so that comparisons between data values under different contexts is possible. A set of conversion functions is defined to convert from one context to another. For example, context information on a stock price includes currency value and scaling factor, and a conversion function can convert a currency between U.S. dollars and Canadian dollars. Using a form of SQL called Context-SQL (C-SQL), these conversions are automatically applied transparently within a query or as requested by the user within the query. This data integration work can be incorporated into any algorithm which implements schema integration.

II.2 Schema Integration Taxonomy

Three taxonomies are developed in this research for categorizing schema integration methodologies. The fundamental taxonomy (see Figure II.2) describes the integration algorithm, while two secondary taxons categorize the inputs to the integration algorithm (see Figure II.3) and its outputs (see Figure II.4).

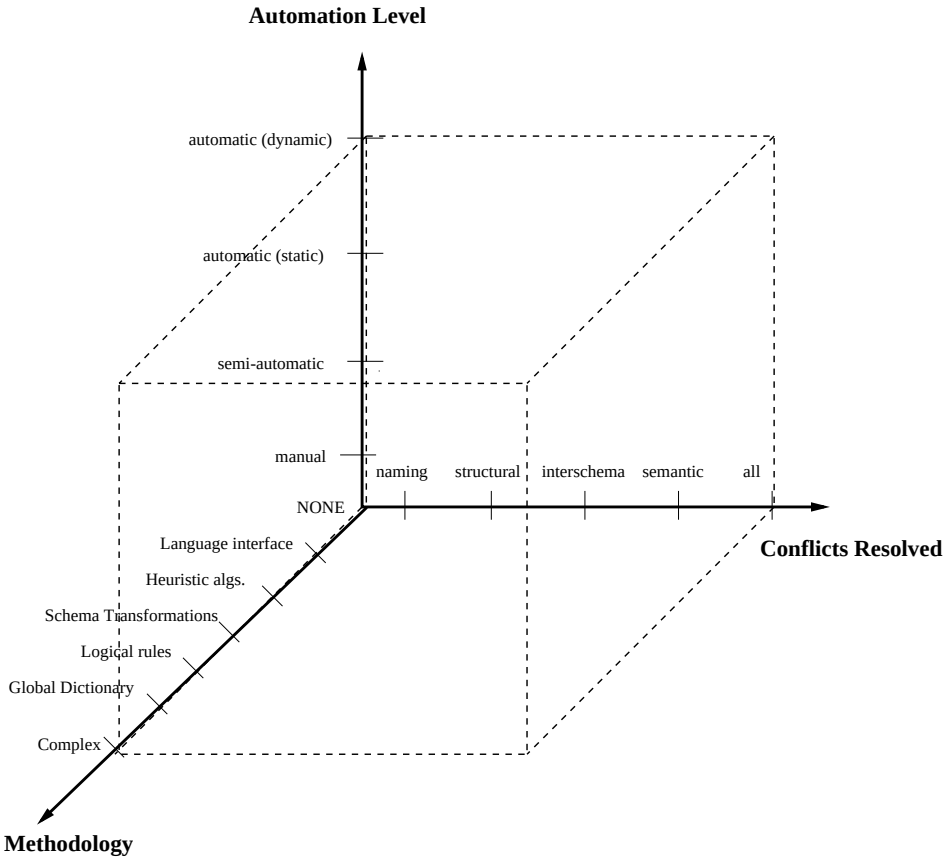

Figure II.2: Fundamental Integration Taxonomy

The fundamental taxonomy has three categorizing axes: resolved conflicts, methodology, and automation level. The ability of an integration methodology to resolve conflicts is fundamental as schemas will be developed under different data models and ideologies. This results in both structural and semantic conflicts between schemas. Ideally, a schema integration method resolves all conflicts including naming, structural, and semantic conflicts. The power of the integration method is measured by the types of conflicts that it resolves and is a measure of system applicability. Each method resolves or assumes away these conflicts.

The methodology used to perform the integration is the second classifier. Various methodologies such as heuristic algorithms, language-level integration, wrappers and mediators, AI approaches, and shared dictionary algorithms are proposed. The integration

methodology is intimately connected with the other classification categories. For example, if the integration methodology is a language interface, this naturally implies that the automation level is manual and that the types of conflicts resolved are dependent on the programmer. Further, the system will demonstrate no transparency.

A historical classification category [8] which divides algorithms based on how the schemas are combined (eg. one at a time, binary, all at once, *etc.*) is not included because it has limited distinguishing properties except for early heuristic algorithms. Most recent techniques attempt to combine schemas on an as needed basis and not all at once. Thus, this classifier is a very special case of the methodology classifier.

The automation level measures the ability to automate the integration procedure. Automation level is a strong distinguishing characteristic because many algorithms are not suitable for large-scale integrations. Without automation, a human designer is required to resolve conflicts which is costly and error prone. Automation involves minimal user interaction and hides integration complexity from users and designers whenever possible. Thus, it is a form of transparency to the integrator.

The three characteristics of resolved conflicts, methodology, and automation level form a fundamental integration taxonomy. However, integration algorithms are further categorized by the inputs they assume or process and their output result after integration is completed. Although the integration methodology commonly affects the inputs and outputs, different algorithms employing similar integration techniques may vary dramatically in the types of inputs they use and the results they obtain.

The input taxonomy (see Figure II.3) characterizes methods based on the inputs to integration including the types of databases combined and their assumed construction. The integration sources axis distinguishes what types of data representations the integration algorithm handles including legacy applications, relational databases, object-oriented databases, or a diverse mixture of data sources. Ideally, an algorithm should handle a diverse mixture of data sources, but some algorithms only target a particular type of data source and then rely on schema translation to convert other data sources into a standard form. The two other classification categories are metadata content and metadata structure. Metadata content is the type of metadata used by the algorithm including structural and

operational metadata. The metadata structure categorizes how this metadata is stored and represented in the system. Metadata may be in the form of logical rules, relations, or complex objects. Metadata source, referring to how metadata is gathered by the system, is a possible axis, but current methods are mostly manual so it is a poor classifier.

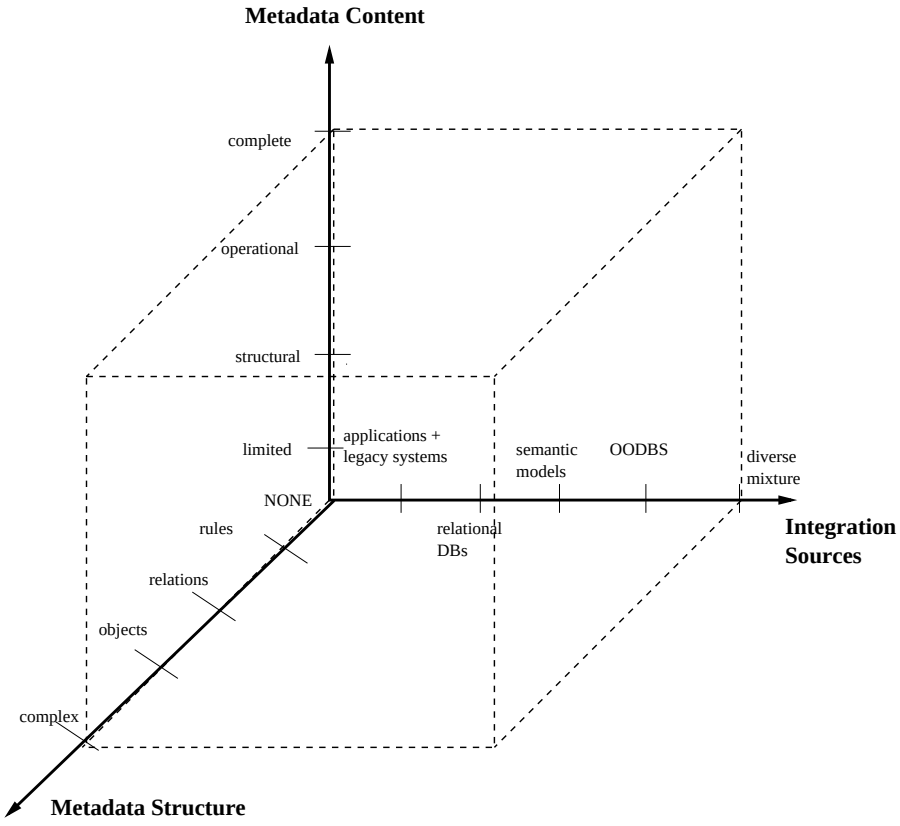

Figure II.3: Integration Inputs Taxonomy

The integration product taxonomy (see Figure II.4) classifies the end result after integration is performed and how this result is used. Most integration methodologies are designed to provide a consistent, global view, but others may also provide transaction management. End-user transparency is a key distinguishing property of schema integration techniques. Transparency measures how well the integration result hides the system complexity from the user and application. Automation in the fundamental taxonomy measures the integration transparency, or the user's involvement during integration, which differs from end-user transparency that measures how isolated a user querying the integrated systems is

from their implementation and distribution characteristics after integration. Systems which do not produce a global view generally have poor transparency.

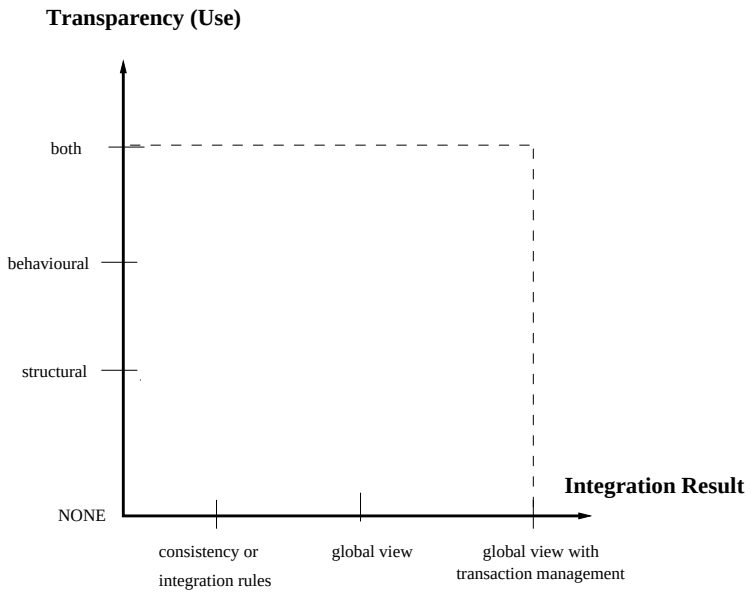

Figure II.4: Integration Product Taxonomy

There are two key forms of transparency: structural transparency and behavioral transparency. A system which provides structural transparency hides the integration, distribution, and organization characteristics of the MDBS. These systems hide the data structure from the user which manipulates data at a higher conceptual level. Most centralized databases provide good structural transparency as the user is abstracted away from most data organization characteristics. In a MDBS, the user must also be hidden from distribution and integration conflicts to provide true structural transparency. Behavioral transparency is a measure of the system's ability to hide database operational characteristics from the user. In a centralized database, the scheduling of transactions is hidden from the user and is a form of behavioral transparency. In a MDBS, behavioral transparency also includes masking distributed updates, handling replication and migration, and updating component database systems using global transactions. Behavioral transparency is challenging to achieve as it requires a system executing global transaction updates while propagating those updates to the local databases as required.

Consistency and integration rules are less desirable than a global view for transparency because they are often context-dependent and integration specific. Moreover, these rules often relate local database sources to each other, instead of relating local database sources to a global view. Without a mapping to some form of global view, the rules become dependent not only on changes to the local database, but to other databases referenced by the rules. These rules do not scale well when data sources are added or removed from the system and do not provide the application with a complete, coherent view. Rather, the "global view" must be extracted by using the rules to achieve mappings between the diverse data sources. For example, in combining the Southstorm and Northwind databases, a logical rule could associate the *Orders_tb* table with the *Orders* table. Although this has the same logical affect as building an integrated view with both tables, the system must process this rule during query execution to obtain the combination of the two tables. As the number of rules increases, the complexity of processing these rules increases dramatically, especially if the user must be aware of their existence during query submission.

In summary, the fundamental integration taxonomy, along with special taxons categorizing integration inputs and outputs, classifies work on schema integration. Some of the axes are interrelated by nature. For instance, the integration methodology often dictates the automation level and the end-user transparency. The goal of all integration algorithms is to obtain high automation characteristics with high end-user transparency regardless of how these results are achieved.

II.3 Integration Methodologies

There are various techniques for performing schema integration. This section categorizes previous work and discusses any shortcomings. The variety of approaches includes heuristic algorithms, rules and knowledge-base approaches, using lexical semantics, and the definition of integration languages. Industrial-level work defines standard languages, wrappers, mediators, and protocols to aid the integration process. An early survey by Batini *et al.* [8] is one of the few surveys in this area. A more recent survey [83] characterizes the problems and issues in database integration with limited focus on categorizing and ex-

plaining the various solutions. The sample databases and classification taxonomy are used in these following sections to review the proposed solutions categorized by methodology as follows:

- **Early Algorithms and Heuristics** - including relational, functional, and semantic canonical models and early heuristic algorithms.
- **Schema Transformations** - including schema re-engineering (using transformations to convert schemas) and object-oriented transformations.
- **Language-Level Integration** - using query languages and programming interfaces to achieve integration.
- **Logical Rules and AI approaches** - using logical rules, knowledge bases, or ontologies to maintain consistency and store sufficient metadata for integration.
- **Global Dictionaries and Lexical Semantics** - using a standard dictionary of terms to describe and integrate knowledge. Related to lexical semantics and expressing semantics in free-form language.
- **Industrial Systems and Standards** - real-world implementations and standards designed for communication and integration including XML [100].
- **Wrappers, Mediators, and Interoperability Architectures** - provide query access to distributed databases and web information sources although the focus is more on query processing than integration.

II.3.1 Early Algorithms and Heuristics

The earliest and most common methods of schema integration are based on semantic models that capture database information. The semantic models are then manipulated and interpreted, often with extensive user intervention, to perform the integration.

An early survey [8] of these model-based methods remains one of the few surveys in the area despite continuing research. These methods define procedures and heuristic algorithms for schema integration in view integration (producing a global view in one database) and database integration (producing a global view from distributed databases). Their goal is to "produce an integrated schema starting from several application views that have been produced independently." [8]

Early pioneers in the integration area were quick to enumerate the potential difficulties in their task. They recognized the problems of varying user perspectives, using

different model constructs, and determining equivalent structures representing an identical concept. The methodology typically consisted of four steps:

- **Pre-integration** - analyzes schemas before integration to determine the integration technique, order of integration, and to collect additional information.
- **Schema comparison** - compares concepts and searches for conflicts and interschema properties.
- **Conforming the schemas** - resolves schema conflicts.
- **Merging and restructuring** - merges and restructures the schemas so they conform to certain criteria.

Numerous algorithms followed these basic steps to produce integrated schemas that were judged on the basis of completeness, correctness, minimality, and understandability. Unfortunately, most of these algorithms exhibit minimal automation and often rely on the user to manually find and resolve schema conflicts. The inputs to the system consisted of the enterprise view, some database assertions (constraints), processing requirements, and mapping rules from component schemas to an integrated schema.

Early integration techniques can be classified in one of two ways: relational models or semantic models. In relational models, integrators made the *Universal Relation Assumption* which allowed them to ignore naming conflicts. The Universal Relation Assumption [71] says that a single relation can be defined that contains all attributes from all relations. This implies that every attribute in the database has a unique name, which allows algorithms to use the relational model's set properties. Semantic models dealt more with conflicts and did not assume certain naming characteristics or design perspectives as in the relational models. Thus, the task of integration using semantic models is more difficult but more realistic.

The diversity of models overviewed [8] is extensive, but one important fact summarized their effectiveness: "...among the methodologies surveyed, none 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." In short, early work on schema integration clearly defined the problems and proposed some heuristic algorithms that assisted designers in their attempts to integrate diverse schemas.

Semantic models are surveyed in depth by Hull and King [44]. The ER-model is probably the best known semantic model. Semantic models have rich techniques for data representation, but are best utilized by database designers and are not specifically designed to automate database integration. Semantic models are better than relational models because they achieve better separation between the conceptual and physical views of the data, have decreased semantic overloading of relationship types, and have more convenient abstraction mechanisms. Although semantic models are suited for data description, they are inadequate for integration because they are not designed for the task.

Model-based methods use a canonical model to represent the global view. Database schemas are mapped into the canonical model and integrated. The suitability of data models as canonical models is discussed by Saltor *et al.* [88] who present a taxonomy of desirable features required by a canonical model. Saltor *et al.* [88] argue that object-oriented models are better canonical models than ER-models (semantic models) because of their expressiveness and ability to capture additional semantic information not in component schemas. The issue of the best canonical model remains an open question.

Other modeling techniques considered for the canonical model include DAPLEX [93] and SDM [42]. DAPLEX is a data definition and query language based on the functional data model. The model uses functions to describe the database. Queries are formulated by combining functions defined on the database that return the requested results. DAPLEX allows very general functions to be defined including aggregation functions, functions representing subtype/supertype relationships, and functions that return derived data and conceptual abstractions. Such mechanisms allow the system to support different user views and to manipulate metadata. This generality could allow DAPLEX to be used as a database front-end and function as a universal query language. Unfortunately, using DAPLEX for integration is limited because naming and structural conflicts cannot be readily handled. Thus, specialized functions are required to map between data sources. Although this mapping is achievable using DAPLEX, it is a manual process highly susceptible to schema changes and must be applied between all databases being integrated.

The Semantic Data Model (SDM) [42] defines a database as a collection of entities organized into classes. A SDM class is a meaningful collection of entities and attributes.

Database classes are logically related by interclass connections which are similar to relationships in other models. The ability to define complex interclass connections and associate a semantic meaning with their definition gives SDM its descriptive power. Interclass connections may be defined as subclass connections relating two classes by a subclass/superclass relationship or by grouping connections which relate classes based on grouping conditions that may be based on attribute values or entities present in other classes. Attributes in different classes are related by defining attribute interrelationships and are accessed from different base classes using mappings based on these interrelationships. Thus, a SDM schema consists of base classes of "real-world" entities augmented with derived classes using interclass connections. This structure allows SDM to capture more semantic data about the database during its construction.

The major weakness of SDM is the difficulty in reconciling name differences. SDM stores data semantics intrinsically by the definition of the derived classes. However, the derived classes often have very complex names which are difficult to reconcile. SDM's ability to explicitly capture the relationship between classes, attribute semantics, and interclass connections provides a solid foundation for an integration model, but lacks a model formulation which is more suitable for automatic integration.

The work on semantic and canonical models has not produced automatic integration algorithms. Although these models allow for manual integration and heuristic algorithms, they are insufficient for automating the integration process.

In terms of the taxonomy (Figure II.2), heuristic algorithms based on semantic models offer low automation although they resolve most structural conflicts. Behavioral conflicts are typically not considered. Any structural or semantic conflicts that arise must be detected and resolved by the designer. Hence, these algorithms offer low automation during the design phase. However, once the global view has been created, the operational system demonstrates high transparency as queries are mapped through the global view. The low automation characteristics make these systems undesirable for large, complex integration tasks. More recent work on schema analysis and models [18] provides more semi-automatic heuristics, algorithms, and techniques for schema analysis and classification but still rely on designer interaction to achieve integration.

II.3.2 Schema Transformations

A logical extension of semantic modeling is schema re-engineering. In schema re-engineering, schemas are mapped into one canonical model and then compared by performing semantic preserving transformations until the schemas are identical or are seen to contain common concepts. By automating the mapping process and providing a set of suitable transformations, diverse schemas are compared and integrated.

There have been numerous approaches based on schema transformations. One approach [48] constructed a generic mapping tool to map schemas into a meta model capable of describing all data models. The schema is subsequently mapped from the meta model to the target model. A mapping from a relational schema to an ER-diagram with minimal user input is possible. Miller *et al.* [79] use a graph structure to represent a database and its constraints and define a set of equivalence preserving transformations. However, the transformations are not applied automatically.

Schema integration transformations for the ER-model are studied by McBrien *et al.* [72]. In this work, a set of primitive transformations is defined which are used to describe common schema transformations and create new composite transformations. Further, they demonstrate variants of schema equivalence such as transformational, mapping, and behavioral equivalence. The work is expanded [73] to produce a general framework to support schema integration based on a hypergraph common data model and a complete set of reversible primitive and composite transformations which allow for transforming schemas to determine their equivalence. A prototype tool is developed to allow designers to apply the transformations to merge, conform, and compare schemas.

The major problem in the transformation approach is that the decision of what transformations to apply to determine schema equivalence is the responsibility of the designer. Although manually applying transformations allows schemas to be compared and integrated, the designer is still performing the four basic steps of schema integration defined by the earlier algorithms. Even when aided by software tools, this approach offers low automation and significant complexity for large schemas.

The object-oriented model is popular because of its ability to represent complicated

concepts in an organized manner. Using encapsulation, methods, and inheritance, it is possible to scalably define any type of data representation. Consequently, the object-oriented model (OO-model) has been used [89] as a canonical model as it has very high expressiveness and is able to represent all common modeling techniques present in other data models. There has been work on defining logical semantics for object-oriented databases [75]. It is also possible to automatically generate an OODBS schema from an abstract specification in a high-level language [11].

An in-depth study [97] of schema transformations in an OODBS included renaming, aggregation, and objectify. From the many possible transformations and equivalent schemas, the target schema is chosen based on the relatedness of the object methods in the schema. Various measures quantified the object methods' correspondence with object attributes and are used to define a heuristic algorithm.

Although this work is a promising first step toward integration using an object-oriented model, there is still no definition of equivalence. Without a definition of equivalence, it is impossible to integrate schemas, regardless of the power of the schema transformations. The ability to generate an OODBS schema from a high-level language is promising because it may be possible to define equivalence at the language level instead of at the schema level.

Using an object-oriented model as a canonical representation for integration is reasonable. However, the object-oriented model is very general which makes determining equivalence between schemas quite complex. Current techniques based on object-oriented models are in their infancy and tend to rely on heuristic algorithms similar to semantic models. Thus, most conflict resolution is manual which results in low transparency and low automation.

Overall, schema re-engineering and mapping methods yield good results when mapping from one model to another and provide techniques for mapping diverse schemas into the same model. For the relational and ER models, transformations are sufficiently powerful to conform and compare schemas, whereas determining equivalence in the OO-model is challenging because of the many possible resulting schemas obtainable after applying transformations. Further, the set of equivalence preserving transformations is not sufficient to determine if schemas are identical. Without the ability to define equivalence, mapping

to a canonical model does not solve the integration problem, but rather transforms it into the complicated, and no simpler, problem of determining schema equivalence.

Schema re-engineering techniques suffer from the same problems as the heuristic, semantic model algorithms on which they are based. Mapping between schema representation models is not a solution to the schema integration problem by itself. The models must map into a schema representation model which allows equivalence comparisons. The basic problem with schema re-engineering is that models which define sufficient transformations to determine equivalence must be manually performed, and algorithms which apply automatic transformations are unable to determine equivalence because of the complexity of selecting the appropriate transformations.

II.3.3 Language-Level Integration

In language-level integration, applications are responsible for performing the integration. This eliminates schema integration at the database level and gives the application more freedom. However, applications are responsible for resolving conflicts and are not hidden from the complexities of data organization and distribution. Thus, although language-level integration may have benefits in certain situations, it is not a generally applicable methodology.

In language-based systems, applications are developed using a language which masks some of the complexities of the MDBS environment. Many languages are based on SQL. How conflicts are resolved by the language is very important in determining its usefulness. In Bright's survey of MDBS issues [13], schema level conflicts are differentiated from language level conflicts. Schema level conflicts arise from constructing a coherent, global view from many data sources. As the number of data sources increases, resolving all semantic conflicts may prove impossible. To combat this scalability issue, language level integration is proposed where a language library helps the user and local DBA perform the integration. Although this approach may be more scalable, it suffers from poor automation and transparency.

One multidatabase query language is IDL (Interoperable Database Language) [53] which has interoperability features that subsume those of MSQL [68]. The language allows

the user to define higher order queries and views by allowing database variables to range over metadata in addition to regular data. Metadata includes database names, relational names, and attribute names. This ability allows queries across database systems in addition to regular relational expressions.

Integration is achieved by defining higher order views which resolve some of the naming and structural conflicts between databases. A view is defined using rule-based expressions that range over data and metadata. MDBS updates are defined using rule-like expressions called update programs. Although defining views over a language provides some integration capabilities, these views are constructed manually by the schema administrator and are subject to change. As more databases are added, constructing these views is more difficult, and there is limited automation as the schema administrator must be aware of the data semantics at all integration sites. Thus, although the language provides facilities for an administrator to integrate MDBS data sources, the amount of manual work and lack of transparency and automation makes this method only slightly more desirable than early heuristic algorithms.

Specifying Interdatabase Dependencies

Enforcing consistency across multiple autonomous databases is very difficult and costly especially when all updates and transactions are atomic. An approach recommended by Sheth [86, 91] involves specifying interdatabase dependencies and enforcing them periodically. Instead of providing immediate consistency, consistency enforcement is time-delayed or is performed as required by the application and users. Consistency is enforced based on data characteristics such as updates and accesses.

Interdatabase dependencies are specified using data dependency descriptors (D^3) which are five-tuples defined as: $D^3 =< S, U, P, C, A >$ where S is a set of source data objects, U is a target data object, P is a Boolean-valued interdatabase dependency predicate defining the relationship between S and U, C is a Boolean-valued mutual consistency predicate that specifies consistency requirements and defines when P must be satisfied, and A is a collection of consistency restoration procedures specifying actions to be taken to restore consistency and to ensure that P is satisfied.

Dependency systems at each local site store and execute procedures implied by these rules. Thus, each dependency system is aware of all transactions submitted to its local database as it may have to enforce consistency based on local transaction execution.

Although the ability to relax consistency constraints has numerous applications in industry, specifying interdatabase dependencies has several problems. First, specifying the dependencies is complex and must be performed at the database object level. Each dependency between all objects effecting a given object must be specified. Thus, the number of mappings grows tremendously as the number of databases increases because mappings are specified between local databases and not directed through any form of global view. Update procedures are consistency programs which are undesirable because they are complex, time-consuming to construct, and subject to underlying database changes. Further, it is a challenging task for the dependency system to intercept all transactions to a database. Many older legacy applications do not have clear transaction management rules and the dependency system may have restricted access to the internal workings of these systems.

A more fundamental problem is that relaxing consistency introduces a new level of semantic conflicts. With relaxed consistency, an application is only assured consistency as provided by the rules. However, users expect current information. Further, resolving conflicts among consistency requirements between applications using the same data may be complex. One application may expect an object be current for the last 24 hours, while another may expect the object to always be current. Resolving these types of semantic timing conflicts is challenging especially when the number of applications grows.

Despite its formal treatment, specifying interdatabase dependencies is essentially a formalized method for coding integration patches between systems. Although relaxed consistency may be beneficial to certain applications, others may exhibit poor integration properties. When this approach is mapped to our taxonomy, the automation level and transparency are low because the programmer or designer is responsible for manually detecting and coding integration rules to resolve conflicts. No global view is produced, and the system does not specify a mechanism for executing global queries and updates. In total, specifying interdatabase dependencies is good for enforcing relaxed consistency but is essentially the same integration methodology as custom, application integration programming.

II.3.4 Logical Rules and Artificial Intelligence Approaches

Reasoning with incomplete and inconsistent data is a primary focus of artificial intelligence research. When schemas with limited data semantics are combined, their union contains both incomplete and inconsistent data. Thus it is natural that AI techniques have been applied to the integration problem.

The major assumption of AI techniques is that the fundamental database model is insufficient when dealing with diverse information sources. In the Pegasus project [49] at HP, databases are combined into "spheres of knowledge". A sphere of knowledge is data, which may be spread across different databases, that is coherent, correct, and integrated. Thus, differences in data representation or data values only occur when accessing different spheres of knowledge.

Siegel *et al.* [94] stores metadata as a set of rules on attributes. A rule captures the semantics of an attribute both at the schema level and the instance level. An application uses these rules to determine schema equivalence. This approach captures data semantics in the form of machine-processable rules. Unfortunately, the rules must be constructed by the system designer and the actual integration of the rules is performed by the application at run-time. Thus, applications are not isolated from changes in the semantics or data organization. The rules provide a limited form of transparency as the application uses the previously defined rules to simplify integration.

Another AI approach consists of storing data and knowledge in packets [101]. A global thesaurus maintains a common dictionary of terms and actively works with the user to formulate queries. Each LDBS has its structural and operational semantics captured in an OO domain model and is a knowledge source. Users access information in LDBSs (knowledge sources) by posing a query to the global thesaurus. Query results are posted on a flexible, opportunistic, blackboard architecture from which all knowledge sources access, send and receive query information and results. This blackboard architecture shows promise because it also tackles the operational (transaction management) issue in MDBSs. The difficulty with the system is the complexity of its construction and defining how the global thesaurus cooperates with the user to formulate queries.

An integration system based on the Cyc knowledge base was developed for the Carnot project [25]. In this system, each component system is integrated into a global schema defined using the Cyc knowledge base. This knowledge base contains 50,000 items including entities, relationships, and knowledge of data models. Thus, any information integrated into the global schema maps to an existing entry in the knowledge base or is added to it. Resource integration is achieved by separate mappings between each information resource and the global schema. The system uses schema knowledge (data structure, integrity constraints), resource knowledge (designer comments, support systems), and organization knowledge (corporate rules governing use of resource) during the integration process.

During the mapping, both a syntactic and a semantic translation is performed. The syntactic translation consists of converting the local language to the global context language (GCL). The global schema contains entries about data models (relational, object-oriented, *etc.*) which allow the designer to map the local data model into the global context. Semantic translation occurs between two equivalent expressions in GCL using a set of logical equivalences called articulation axioms. An *articulation axiom* is a statement of equivalence between components of two theories. An object in one view can be converted to an object in another view (and vice versa) by mapping through the axiom.
The integration procedure proceeds in three phases:

- *schema representation* - represents the local schema in global context language.
- *concept matching* - map Cyc concepts to concepts in global schema.
- *axiom construction* - an axiom is constructed for each match found.

This integration method has several desirable characteristics including:

- **Individual mappings** - information sources are included in the global schema one at a time independent of other sources.
- **Global view constructed** - the system maps to a global view consistent across all sites which provides query transparency.
- **Handles schema conflicts** - using mappings to GCL, representational (structural) conflicts are resolved by using a common integration language.

The Carnot project has many desirable features as can be seen when it is mapped onto the taxonomy. The transparency of the system is high as it only requires the designer to

map an information source to the global view once. Then, all query translation and mapping is handled through the global schema and GCL. This results in a high automation level as resources are integrated independently, quickly, and are relatively free from changes in other resources. Further, most conflict types are resolved. Structural conflicts are resolved by mapping data model concepts into the GCL. Behavioral conflicts are not studied, although a global transaction model is proposed [102].

The Carnot project clearly has much to offer our understanding of the integration problem. The key problem is that knowledge base systems, especially one this large, require significant computational resources and complexity in determining related concepts. Further, it is unclear how all semantic conflicts are resolved. Mapping a local data model into GCL can be used to address structural conflicts, but conflicts within the GCL are not discussed. For example, one designer could integrate a concept into the global schema using an entity while another could use a relationship. In this case, the concepts are identical but the integration problem is now at the global level in GCL. Depending on the frequency of these types of conflicts, this may be a serious problem.

A knowledge base is also used by Embury *et al.* [28] to capture metadata and database semantics. The knowledge base is organized as a semantic network, and new terms are integrated by determining and merging into the knowledge base the best subnetwork that spans the concepts common to the existing knowledge base and the newly added schema. An important contribution is the knowledge base is organized into 4 layers: concept layer, view layer, metadata layer, and the database layer. The authors correctly realize that integrating databases occurs at higher levels (conceptual, view) than metadata and structural organizations. The knowledge base is organized to capture and link information at all layers.

Like other knowledge base approaches, the fundamental problem with the technique is imprecision. Although common concepts are often well integrated by using spanning subnetworks, this is not guaranteed. Further, the resulting knowledge base does not provide a "unified" global view sufficient for SQL-type querying as it only supports imprecise queries or is used as an integration tool.

Other related projects include the work by Gruber in defining ontologies for knowl-

edge sharing [40, 41], and the definition of KQML [32]. Although the focus of these projects is on the communication of knowledge between knowledge bases and intelligent agents, knowledge communication and languages are related to database integration. More theoretical treatment of integrating logical views is performed by Ullman [98] in his discussion of Datalog programs, conjunctive query containment, and related integration architectures.

II.3.5 Global Dictionaries and Lexical Semantics

Lexical semantics [27] is the study of the meaning of words. Lexical semantics is a very complicated area because much of human speech is ambiguous, open to interpretation, and context-dependent. Nevertheless, lexical semantics may have a role to play in schema integration.

Lexical semantics plays a prominent role in the Summary Schemas Model (SSM) proposed by Bright *et al.* [14]. The Summary Schemas Model is a combination of adding metadata and intelligent querying based on the meaning of words. The SSM provides automated support for identification of semantically similar entities and processes imprecise user queries by constructing an abstract view of the data using a hierarchy of summary schemas. Summary schemas at higher levels of the hierarchy are more abstract, while those at the leaf nodes represent more specific concepts. The target environment is a MDBS language level system which provides the integration. Thus, the SSM is a user interface method for dealing with diverse data sources rather than a method for integrating them. The actual integration is still performed at the MDBS language level. However, the SSM could be used on top of another integration architecture as its support for imprecise queries is easily extendible.

The heart of the SSM is the linguistic knowledge stored in an on-line taxonomy, the 1965 version of Roget's Thesaurus. The taxonomy contains an entry for each disambiguated definition of each form from a general lexicon of the English language. Each entry has a precise definition and semantic links to related terms. Links include synonymy links (similar terms) and hypernymy/hyponymy links (related hierarchically).

Using this taxonomy, a semantic distance between words is constructed. The semantic distance allows the system to translate user-specified words (imprecise terms) into

database terms using the hierarchy of summary schemas. A leaf node in the hierarchy contains all the terms defined by a DBA for a local database. Local summary schemas are then combined and abstracted into higher-level summary schemas using hierarchical relationships between words. The system allows the user to specify a query using their own key words and the system translates the query into the best fit *semantically* from the information provided in summary schemas. However, no schema integration is performed. The summary schemas do not represent integrated schemas but rather an overview of the data summarized into English-language categories and words.

Related to knowledge bases and ontologies, semantic dictionaries and concept hierarchies allow database integrators to capture system knowledge and metadata in a more manageable form. Castano [17] defined concept hierarchies from a set of conceptual (ER) schemas from Italian government databases. Concepts at the top of the hierarchy are more general than lower level concepts. Concepts are connected in the hierarchy by instance-of-links (if an entity is an instance of a given concept), kind-of links (links concepts based on super-concept/sub-concept relationships), and part-of links (links concepts related by aggregation relationships). Concept hierarchies are defined using either an integration based approach, where the integrator maps data sources to a pre-existing concept hierarchy, or using a discovery-based approach, where concept hierarchies can be incrementally defined one schema at a time by integrating with known concepts.

Using a concept hierarchy to define similar concepts is a useful idea. By also defining formulas to determine semantic distance between concepts in the hierarchy, it is possible to estimate semantic equivalence across systems. The authors also present a mechanism for querying the MDBS based on the concept hierarchy and concept properties. Semantic distance calculations are used to evolve and create the hierarchy by adding new concepts or grouping concepts appropriately as they are integrated.

The major problem with this approach is that it does not produce a concept hierarchy which can easily be queried. The authors propose querying the MDBS for entities which are “similar” by virtue of having related structural and behavioral properties. Although this may be sufficient in some cases, the majority of applications require more precise querying using a variant of SQL. Another problem is that constructing the concept hierarchy in a

discovery-based approach requires human intervention and decision making. The hierarchy may be more useful if the structure and behavior of the concepts are removed and it solely focuses on identifying similar concepts.

Other work involving global dictionaries includes the definition of SemQL [62] which allows semantic querying of databases using a version of SQL and semantic words from the WordNet [78] online dictionary. By choosing appropriate terms from the WordNet dictionary and querying on these words, transparent querying of a multidatabase is possible.

II.3.6 Industrial Systems and Standards

Industrial Systems

Despite considerable research effort on schema integration, current techniques do not apply well to industrial problems. Many of the approaches are not scalable or require too much human intervention. Custom-coding of translations or wrappers is used in industry because it is simple, despite being time-consuming and capital intensive.

Although the Interbase system [16] was developed at Purdue University, the architecture has a framework which is similar to many industrial systems. These systems focus more on transaction management and distributed data access than the conflict resolution procedures involved in schema integration. The Interbase system is a framework for integrating diverse data sources based on a distributed flexible transaction manager which guarantees global consistency. The system codes interfaces to individual data sources and provides a high-level programming language and graphical interface for access to the data sources. The global transaction manager guarantees resource ordering and most of the integration is performed manually. Nevertheless the Interbase system represents a workable architecture for industrial applications. There are other similar projects [4, 84] but all rely primarily on human coding and manual resolution of integration conflicts.

XML and Integration Standards

Internet and industrial standards organizations take a more pragmatic approach to the integration problem by standardizing the definition, organization, and exchange mecha-

nisms for data communications. Instead of integrating entire database systems, their goal is to standardize the exchange of data between systems which is a more straightforward problem because it is widely accepted that communications occur using standardized protocols and structures. Communication standards require capturing data semantics for only the data required in the communication and formatting it using a standard during transmission.

Work on capturing metadata information in industry results in the formation of standardization bodies for exchanging data. An early data exchange standard is Electronic Data Interchange (EDI) which provides a mechanism for communicating business information such as orders and shipments between companies. As the web has become more prevalent, standards have been proposed for exchanging data using Extensible Markup Language (XML) and standardized XML schemas based on it. Microsoft leads a consortium promoting BizTalk [77, 76] which allows data to be exchanged between systems using standardized XML schemas. There is also a metadata consortium involving many companies in the database and software communities whose goal is to standardize ways of capturing metadata so that it may be exchanged between systems. The consortium defined the Metadata Interchange Specification (MDIS) version 1.1 [23] as an emerging standard for specifying and exchanging metadata. Other standardization efforts include the Standard Interchange Language (SIL) [45].

Rather than attempting to define global dictionaries to capture data semantics within a database schema, industrial approaches define data schemas for data exchange. EDI, SIL [45], and Microsoft BizTalk [77] use standardized formats and structure for representing data communications. The standardized dictionaries are typically defined by industry as each type of business has slightly different terms for data elements. Further, the type and size of data elements are also standardized which results in less flexibility. However, the standardization of dictionaries allows data transfer between differing systems without any semantic confusion because the dictionaries are standardized schema with well-defined structure, types, and sizes. There is no schema integration in these systems *per se*, rather data that previously existed in a database is mapped into a standardized schema document and transmitted to another database which decodes the document using the standard schema. Although communication is achieved between systems, no integration occurred.

XML

Extensible Markup Language (XML) [100, 43] is a standard for web communications. XML is a subset of Standard Generalized Markup Language (SGML). A description of XML given by W3C [100] is:

> XML documents are made up of storage units called entities which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure.

The power of XML as a description language is its ability to associate markup terms with data elements. These markup terms serve as metadata allowing formalized description of the content and structure of the accompanying data. XML appears destined to subsume HTML as the communication language for the Internet, although graphical display properties of an XML document are still being defined. By associating metadata terms with data elements, XML documents can be communicated between organizations and their semantics completely understood both by human and machine agents.

Although XML is a powerful language for exchanging semantics between systems, XML is not the total solution to standardized business communications and system interoperability. XML does not have a standard set of markup terms. For data and metadata to be automatically combined and parsed, organizations must standardize on a set of markup terms to capture concept semantics. Various standards organizations and private companies are actively defining XML term dictionaries and web sites for exchanging knowledge and transacting business via standard dictionaries and commerce portals. Defining a general, powerful set of XML tags would allow for the integration of knowledge between systems. Unfortunately, most industrial efforts focus on subsets of the problem as it applies to individual domains.

Industrial vs. Research Approaches

There are fundamental differences between the database research approach and the industrial approach. The research approach defines mechanisms for combining entire

database systems so that they interoperate as if they were one large database. Approaching the problem in this manner is beneficial because you can argue about integration problems at a conceptual level and design database protocols abstractly. The pragmatic approach in industry results in the definition of standards for communication and software translation allowing systems to communicate. The important distinction is that the communication between systems is standardized not the actual data contained in the systems. These systems do not generally attempt schema integration across databases, and their standardized dictionaries are limited to the data involved in exchanges. This implies that they are unsuitable for integration of entire systems because they do not capture the entire database semantics. Rather, they merely exchange data after the semantics are completely understood.

Integration of the two approaches may be possible. By utilizing standardized dictionaries and languages developed in industry, a multidatabase architecture could be built allowing SQL-type queries and updates of the underlying, autonomous database systems by using the algorithms developed in the research community.

II.3.7 Wrappers, Mediators, and Interoperability Architectures

Related to communication standards and industrial projects are architectures which rely on wrappers or mediators to achieve integration. The basic idea of these systems is that additional wrapper software is installed above each data source to be integrated so that it can interact with the global database federation. The wrapper software acts as a mediator between the global transaction requests and the implementation of the data source. The mediator provides transaction management and data access methods for global transactions in a standard form, even though many of the legacy systems may not have the necessary features. These systems focus on answering queries across a wide-range of data sources including unstructured and semi-structured data sources such as web pages and text documents.

Although wrappers provide general, standard facilities for querying data sources, automatic schema integration is not a primary focus. Mediator and wrapper based systems such as Information Manifold [64], Infomaster [37], TSIMMIS [65], and others [5, 25, 12] do not tackle the schema integration problem directly. Some of these systems [5] do not attempt

to provide an integrated view. Rather, information is integrated based on best-match algorithms or estimated relevance. Systems such as TSIMMIS [65] and Infomaster [37] that do provide integrated views typically construct these integrated views using designer-based approaches. That is, mediator systems either assume an integrated view of the data sources is constructed *a priori* by designers or do not construct an integrated view. This integrated view is then mapped to the local views of the mediators by logical rules or query expressions specified by the designer. Thus, these systems achieve database interoperability by providing an integrated view and its associated mappings to local systems, then automatically process a query specified on the integrated view into queries on the individual data sources. Global queries are mapped using a query language or logical rules into views or queries on the individual data sources. The definition of the integrated views and the resolution of conflicts between local and global views are manually resolved by the designers. Once all integration conflicts are resolved and an integrated global view and corresponding mappings to source views are logically encoded, wrapper systems are systematically able to query diverse data sources.

Wrapper and mediator systems provide interoperability by explicitly defining mappings from a global view to individual source views. However, they do not define algorithms for resolving conflicts, or producing the global view and its associated mappings from data source information. The global view is a mediated view constructed manually. A list of wrapper and industrial systems compiled from various surveys [47, 34] is given in Figure II.5, and short overviews of some of these systems follows.

System	**Institution**	**Data Model**	**Global View/ Updates**	**Target Data Sources**
Multibase [30]	Computer Corp.	Functional + generalization	Yes/Yes	Structured databases
MRDSM [69]	INRIA, France	Relational	No/Yes	Relational databases
TSIMMIS [65]	Stanford	Object Exchange Model (OEM)	No/No	Structured/Unstructured
HERMES [1]	U. of Maryland	Two-Stage Entity Relationship	Yes/No	Knowledge bases
Information Manifold [64]	AT&T Bell Labs	Relational + class hierarchies	Yes/No	Web-based data sources
Metadatabase [22]	Rensselaer	None (metadata in ER model)	No/No	Enterprise databases
COntext INterchange [39]	MIT	Description Logic (axioms)	No/No	Structured
Informia [5]	Ubilab IT Lab	ODMG model	No/No	Structured/Unstructured
Infomaster [37]	Stanford	Translation Rules	Yes/No	Structured/Web Sites

Figure II.5: Selected Wrapper, Mediator and Interoperability Systems

Multibase [30] is a multidatabase system based on the functional data model designed by Computer Corporation of America. The user queries on a single global view called the superview which hides the details of the individual databases. The superview is constructed using the DAPLEX language in a two-step process. First, DBAs translate source database schema into the functional model and record metadata information. Then, a global administrator combines these schemas into a global view. The system is a tightly-coupled federated database system because it produces a static, global view in advance before query submission. Thus, this system is related to multidatabase systems which are surveyed by Bright *et al.* [13].

MRDSM [69] is a loosely-coupled federated database system because dynamic integration is performed by end-users using a data manipulation language. This is an example of a language-level integration system, such as IDL [53], which was previously mentioned.

TSIMMIS [65] is a mediator system developed at Stanford for simplifying the extraction and integration of data from unstructured or semi-structured data sources. The approach defines the Object Exchange Model (OEM) which allows self-describing data items. That is, instead of using a schema, object structure is encoded with every data item. Each OEM object has the structure: **(Object-ID, Label, Type, Value)**. OEM supports object nesting. Queries are posed using OEM-QL which is similar to SQL and supports wild-card matching. Another contribution is the Mediator Specification Language (MSL). MSL is a declarative language which uses rules and functions for translating objects and integrating data sources.

HERMES [1] is a mediator system for integrating knowledge bases and domains into a single reasoning system. Individual systems are combined into the global semantic model called the Generalized Annotated Program (GAP) Framework. This system also defines a rule-based mediator language and supports uncertainty. The Carnot system [25] is similar and uses the Cyc knowledge base.

Information Manifold [64] is a logic system for integrating web-based data sources. Queries are specified declaratively against a static view. There is a single global view for all users called the World View expressed using the relational data model and class hierarchies.

Integration is achieved by specifying the attributes that are accessible in the world view using content records and binding, and selection criteria using capability records. Capability records are necessary because web data sources often have limited query functionality and may not be able to return attributes in all possible combinations.

Metadatabase [22] is a dynamic integration system which does not construct a global view. Instead, Metadatabase allows the user access to metadata used to formulate global queries. Queries are constructed by graphically traversing data source metadata and generating a Metadatabase Query Language (MQL) query for retrieving the results.

The COntext INterchange (COIN) system [39] performs data integration based on logical axioms. COIN is designed to automatically combine structured and semi-structured data sources. Data source information is encoded into elevation axioms (for mapping values), context axioms (for representing context semantics), and conversion functions for converting from one context to another. Rather than building an integrated view, a context mediator is used for querying to reconcile potential conflicts between the data source information expressed as axioms.

Informia [5] is a mediator system for web integration that combines information using estimated relevance calculations. It uses an object model and object query language similar to Garlic [85] which was developed at IBM Alamaden Research center.

Infomaster [37] combines data sources using wrappers and translation rules. Translation rules are logical rules that map from a global view to individual data sources. Structural and semantic conflicts are resolved by the definition of these translation rules and provide a form of global view. Infomaster allows the definition of a reference schema which is essentially a standardized, global view of all sources. Each user can subsequently modify the reference schema for their own personal use using additional translation rules.

Related work includes web source querying such as WebSQL [74], WebOQL [3], STRUDEL [31], and others [70]. These systems focus on modeling and querying the web and integrating query results for the user. Since web sources are typically semi-structured with no schema information, these systems must deal with limited metadata information, data inconsistencies, and unstructured information. The query languages are modeled after SQL and use link and document information for querying. Thus, the focus of web integration

systems is the transparent querying of web information sources and the integration of results for the user. This is a different problem than schema integration. Wrapper and mediator systems are especially suited to the Web environment. Florescu *et al.* [34] survey web integration techniques. Other work on wrappers and mediators includes determining the computing capabilities of mediators [103]. There is also work [33, 20, 9, 82, 63, 46] on query optimization and formulation in wrapper systems.

II.4 Integration and Semantics

A fundamental requirement for automatic schema integration is the ability to capture data semantics. Computer systems have been developed in a user-centric way. Describing data semantics in a system-centric way allows the computer system to determine the semantics of the data it processes, thus freeing the user from this mundane task. By using data semantics, the following section overviews a general methodology for automatic schema integration.

II.4.1 General Semantic Integration Methodology

A general integration algorithm (see Figure II.6) based on a global dictionary has the following basic steps:

1. Determine integration scope.
2. Enumerate global concepts with appropriate names.
3. Transform application data into the integration language.
4. Combine application specifications into a global view.
5. Refine integrated schema.

The first step common to any integration methodology is determining the integration scope. This process defines what data sources are integrated and which data contained in the sources is used and summarized in the global schema. Further, this process defines a high-level mechanism for resolving data conflicts. Basically, this step produces a list of

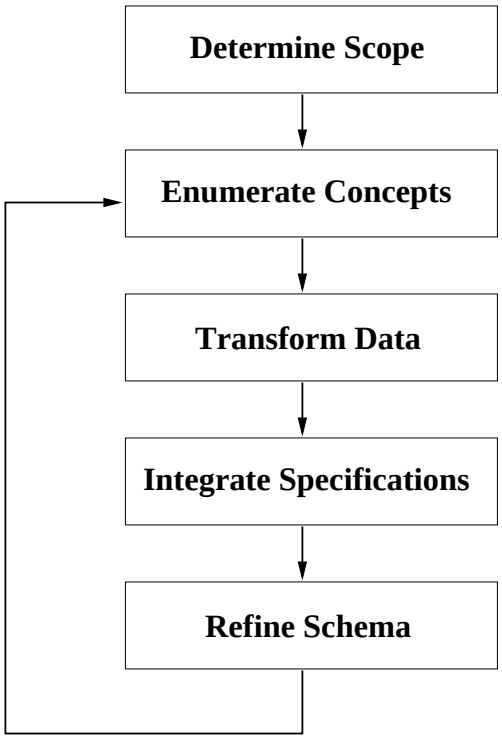

Figure II.6: General Integration Methodology

global concepts that are represented in the global view, and a set of abstract mechanisms for handling conflicts. This is the organizational view of the data.

Enumerating global concepts is the second step. In this step, global concepts and entities defined in the preliminary step are given meaningful names to be used throughout the integration process. Unique names are required for all entities, attributes, and possibly relationships. These unique names are given to system integrators to represent application data in the integration language. By defining the global names before integration, the metadata gathered from the applications should be free of naming conflicts.

The first two steps are performed at a high level of abstraction by management and information technology personnel working together to decide what data is stored in the global view. Once the data in the global view is agreed upon, the subsequent mapping of application data to the global view is started.

Given the set of global concepts and entities with their standardized names, application integrators work independently on mapping application data into the integration language. Although this process may be performed semi-automatically by extracting schema information using schema re-engineering techniques, much of the additional metadata required for the integration language is inserted manually. At the minimum, the integrator has to convert names used at the local level into the globally defined names. When this representation procedure is complete, there is a mapping to and from the integration language for each application.

For some applications, automatically extracting information is impossible. Such applications generally have a poor mechanism for schema definition or may not have a schema. For database systems, schema information can be used to fill in fields in the integration language including local names and sizes, relationships, entities, and some foreign key information. This information is readily extracted using schema re-engineering techniques discussed previously.

Most of the metadata on the schema is inserted by the designer but it only applies to the local application. The only connection an integrator has with the rest of the integrators is the standardized global names used in the mapping. Manually extracted data includes operational behavior, data semantics, and relationship semantics. Although this is a complicated task, it is easier to capture semantics when examining one application at a time rather then when examining them all at once.

After mapping into the integration language, automatic integration of data using schema equivalence rules is possible. Many of the conflicts previously resolved manually are now handled automatically by using global, standardized names for concepts. Further, the integration language is structured to automatically detect equivalent semantic representations. After automatic integration is completed, a uniform, global view representing the concepts outlined in the preliminary phases is produced.

During integration, mappings are generated from the global view to the individual, local application views. The application views are stored in the integration language which itself is a mapping of the integration data. Thus, two levels of mapping are needed to convert from the original application view to the global view.

The final phase of integration is refining the integration. In this step, designers use the application specifications in the integration language to discover new data which could be represented in the global view, add or change concepts in the global view, and tune the system for performance.

The integration architecture is based on this general methodology. The goal is to define a system capable of automatically integrating diverse data sources to provide interoperability between systems which preserves full autonomy of the underlying data sources that have no direct knowledge of their participation in a global system. The contribution of the architecture is the merging of industry standards with research algorithms to create an automatic schema integration methodology.

II.5 Remaining Challenges and Future Directions

Despite extensive research on schema integration, it is not a solved problem. Over 20 years of research has enhanced our understanding of integration conflicts and challenges, interoperability constraints, and methodologies for manually resolving conflicts. Industrial systems and standardized languages and protocols demonstrate that interoperability is possible and semantic exchange is feasible.

The time has come to revisit the schema integration issue. It is no longer sufficient to accept that schema integration is exclusively the sophisticated domain of database administrators and system integrators. Automatic schema integration **is possible** with a systematic method for capturing data semantics.

What new approaches are required to sufficiently express database semantics as metadata? The answer may require shared ontologies, dictionaries, logical rules, or knowledge bases. Do we have to accept standardization to achieve automation? Unquestionably, the greatest advances in interoperability and networking all have their foundations in standards such as TCP/IP, HTML, XML, ODBC, and others. Is it time for database design to be standardized instead of treated more as an art-form? Maybe the most important achievement will be the realization that some standardization is required, but to what extent and in what role may be debatable. Regardless, as a society we have accepted standardization

in order to effectively communicate and interoperate, and the benefits of standardization will eventually be felt in the database community.

In summary, the integration architecture is built to answer these questions. The goal is to construct an architecture combining standards for semantic exchange with research algorithms capable of automatic schema integration. The result is a system with unique properties and exciting potential and challenges.

Chapter III

Integration Architecture

III.1 Integration Architecture Overview

The integration architecture (Figure III.1) is based on the standard multidatabase architecture. At the bottom level are automonous local databases which continue to process local transactions. At the top level is a multidatabase layer with which global users interact to query the database federation.

Within the multidatabase layer, the architecture consists of four components: a standard term dictionary, a metadata specification language (X-Specs) for capturing data semantics, an integration algorithm for combining metadata specifications into an integrated view, and a query processor for resolving conflicts at query-time. The dictionary provides a set of terms for describing schema elements and avoiding naming conflicts. The integration algorithm matches concepts to produce an integrated view, and the query processor translates a semantic query on the integrated view to structural query expressions.

The architecture utilizes these components in a three phase process to construct an integrated view of data source information:

- **Capture Process:** A capture process is independently performed at each data source to extract and format database schema information and metadata into an XML document called an X-Spec.
- **Integration Process:** The integration process combines X-Specs into a structurally-neutral hierarchy of database concepts called an integrated context view.

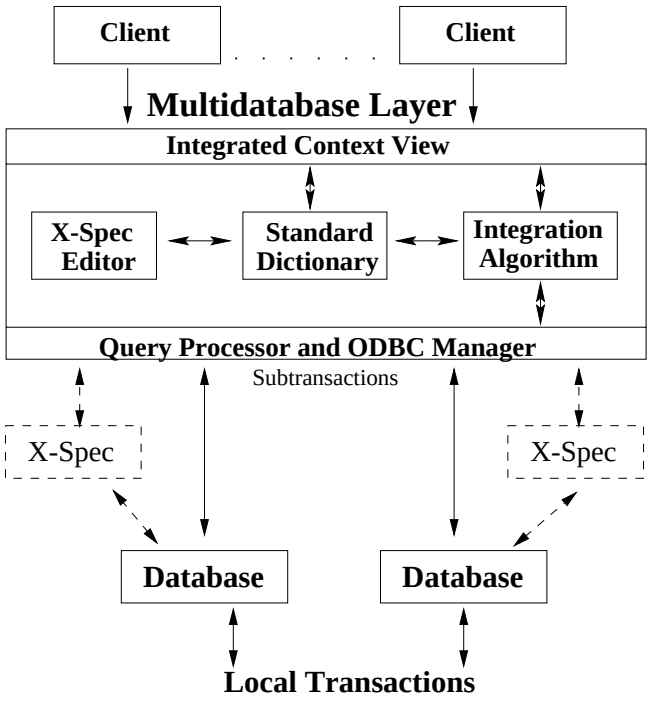

Figure III.1: Integration Architecture

- **Query Process:** The user formulates queries on the integrated view that are mapped by the query processor to structural queries, and the results are integrated.

There are two main target environments for deploying the architecture:

- **Centralized System:** In a centralized deployment, the multidatabase layer resides on a single server. The central server pre-combines all X-Specs and provides integrated querying for all users in a central location.
- **Distributed System:** Distributing the system involves implementing the multidatabase layer across systems such as in the web browser of all database clients. The clients then independently connect to the database systems, download the X-Specs, and produce an integrated view for their own local use.

The distinguishing feature of the architecture is that wrapper or mediator software is not required. A database schema is encoded into an X-Spec independently of other data sources and the integrated view itself. Once an X-Spec has been created for the data source, the multidatabase layer uses this information to produce an integrated view, and queries are mapped to SQL and communicated directly with the database.

In the capture process, the X-Spec for a given data source is constructed, and the semantics of the schema elements are mapped to semantic names consisting of terms from the standardized dictionary. This capture process is performed independent of the capture processes occurring at other data sources. The only "binding" between individual capture processes at different data sources is via the dictionary that provides standard terms for referencing data. The specification editor tool is used to extract database schema, add the semantic names and additional metadata, and store the result in an X-Spec.

The key benefit of the three phase process is the isolation of the capture process from the integration process. This allows multiple capture processes to be performed concurrently and without knowledge of each other. The capture process at one data source is not affected by the capture process at any other data source. Thus, the capture process is performed only once per source, regardless of how many data sources may actually be integrated. This is a significant advantage because the semantics of the database can be captured at design time. If the database is then integrated with other systems, its semantic description is already available as provided by its creator.

The multidatabase layer takes the X-Specs of the individual data sources and executes the integration algorithm to produce an integrated view. This integrated view is provided to clients allowing them to issue SQL-type transactions against it. The integrated view displays a context hierarchy of semantic name terms. A SQL query is then constructed by the user using the semantic names. The system performs the necessary mapping from semantic names to system names and divides the query into subqueries against the data sources. Once results are returned from the individual data sources they are integrated based on the unified view and subsequently returned to the user.

The architecture does not require translational or wrapper software at individual data sources. Once the X-Spec has been provided for the data source and integrated in the multidatabase layer, the software communicates directly with the data source. All translation, integration, and global transaction management is handled at the multidatabase layer. This allows **full autonomy** of the underlying participating databases as the multidatabase layer appears as just another client submitting transactions to the database.

III.2 Standard Dictionary

To provide a framework for exchanging knowledge there must be a common language in which to describe the knowledge. During ordinary conversation, people use words and their definitions to exchange knowledge. Knowledge transfer in conversation arises from the definitions of the words used and the structure in which they are presented. Since a computer has no built-in mechanism for associating semantics to words and symbols, an on-line dictionary is required for the computer to determine semantically equivalent expressions.

The foundation of the architecture is the acceptance of a standard term dictionary which provides terms to represent concept semantics that are agreed upon across systems. Thus, the architecture operates under the assumption that naming conflicts are prevented by utilizing standard terms to exchange semantics.

The standard dictionary is organized as a hierarchy of concept terms. Concept terms are related using 'IS-A' relationships for modeling generalization and specialization and 'HAS-A' relationships to construct component relationships. The dictionary is represented using XML, which guarantees systems can communicate provided they conform to the XML standard, and is used to construct XML tags that represent data semantics. Creating a standard dictionary has many desirable features:

- *Standardized set of base concepts* - the dictionary provides a standard set of terms for representing common database concepts.
- *Flexibility* - Unlike industrial approaches, the standard dictionary provides a set of common terms used to describe existing database schema, not a standard schema. The dictionary can be expanded to include new terms.
- *Hierarchical organization* - The dictionary terms are organized in a concept hierarchy to model how terms are related to each other.

A term dictionary has been constructed (see Appendix G) starting from the top-level ontological categories proposed by Sowa [95]. Note that the exact terms and their placement is irrelevant. The dictionary is treated as a standard whether within an organization or for the whole Internet community. Individual organizations may modify the dictionary, but successful integration within a domain is only guaranteed with total standard acceptance. Thus, we will not argue about the correctness of the dictionary organization,

but simply assert that by conforming to any standard term dictionary allows the architecture to function properly. Ultimately, dictionary evolution could be directed by some standardization organization to ensure that new concepts are integrated properly over time.

The exact terms and organization of the standard dictionary is irrelevant. Although this may seem surprising, any language is simply a standard for expressing semantics. For example, there is no intrinsic reason why the word "apple" should describe an apple. Conceivably, the word "orange" could be used or "X123" as long as that was the accepted terminology to represent the concept of an apple. Similarly, the exact organization of the concept hierarchy and the terms used to represent concepts is irrelevant as long as they are agreed upon. For example, the Standard Interchange Language (SIL) uses standardized field names like F01, F02, and F03 to represent data elements. However, the goal of XML is to be human readable, so the dictionary terms should be recognized English words for their concepts and the base hierarchy should be evolved in a way that models current standardization efforts and real-world organizations. Thus, we will not debate the exact definition of the standardized dictionary because it does not affect the correctness of the approach. Any standardized dictionary can be used as long as it is formatted correctly and has the necessary terms to capture the semantics of every schema element to be integrated from the corresponding data sources.

The problem in defining a standard dictionary is the complexity of determining semantically equivalent words and phrases. The English language is very large with many equivalent words for specifying equivalent concepts. Thus, simply using an on-line English dictionary for the computer to consult is not practical. Not only is the size of the database a problem but it is complicated for the computer to determine when two words represent semantically equivalent data.

By using English words instead of abstract field names, the metadata specification is more readable and is easier for the designer to assign correct terms to represent data semantics. However, a word may have a slightly different semantic connotation to different people which may affect their choice of a "correct" term. For the purpose of this discussion, we will assume that a designer correctly associates the proper dictionary term to represent the semantics of an element. For example, consider the case where two databases contain

home phone numbers. One designer correctly uses the term "Home Phone#" from the dictionary where another uses the more general term "Phone#". Although integration would still occur correctly, the second designer lost some semantics in his choice of term and technically should have chosen the other term. We assume that these mis-naming problems are handled using an external error-checking mechanism.

In summary, the standard dictionary is a tree of concepts related by either IS-A or HAS-A links and stored in XML format. We have defined a basic standard dictionary but allow organizations to add terms as required. Unlike a BizTalk schema or simple set of XML tags, the dictionary terms are not used independently to define data semantics. Since the actual organization of the data (schema) is not defined in the dictionary, it is impossible to assume that a given dictionary term has the same meaning every time it is used. Therefore, the context of the term as it is used in the metadata specification describing the database determines its meaning.

By analogy, the dictionary is like an English dictionary as it defines the semantics of accepted words used to convey knowledge. However, overall semantics are communicated by organizing words into a structure such as sentences. The structure for semantic communication is a *semantic name* whose simplified structure is easily parsed.

III.2.1 Top-level Standard Dictionary Terms

The top-level terms for the standard dictionary are those proposed by Sowa [95]. Sowa proposes the top-level categories in Figure III.2 by combining previous work by philosophers and computer scientists.

These categories serve as the root of our standard dictionary tree and all new concepts are subconcepts of these categories. The root of the tree, T, "is a neutral representation for the ultimate"[95](679). T is everything and nothing at the same time. Below the root is the first level of categorization which separates physical matter and information. "Information is pure structure whose nature does not depend on the objects it describes or the physical medium used to record it."[95](680) Thus, the second level of the dictionary divides concepts into physical (tangible) concepts and information (intangible) concepts.

The third level consists of a ternary division of concepts under both the physical

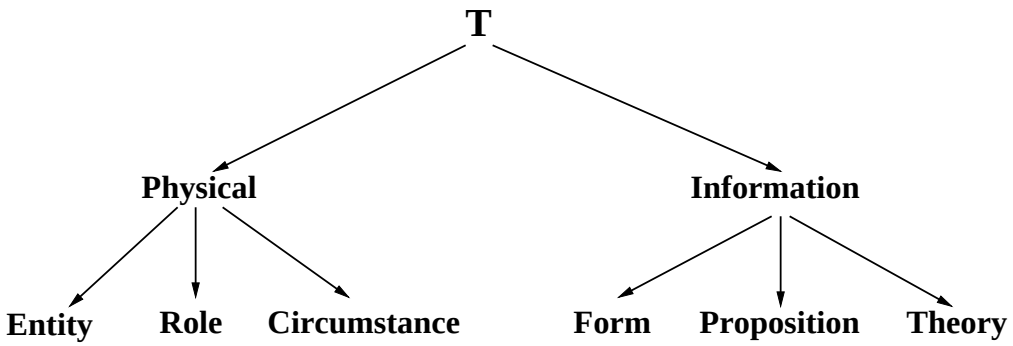

Figure III.2: Top-level Dictionary Terms

and information branches. This ternary division is the result of dividing concepts into first, second, and third level concepts. The notion of firstness, secondness, and thirdness is as follows:

> First is the conception of being or existing independent of anything else. Second is the conception of being relative to, the conception of reaction with, something else. Third is the conception of mediation, whereby a first and a second are brought into relation. [95](677)

Placing database concepts into the categories of firstness, secondness, and thirdness is non-trivial. For example, the concept of "Woman" is a first level concept because it represents a person existing independent of anything else. A concept at the second level has some additional semantic notion attached to it. The notion of a "Wife" is a second level concept as a "Woman" becomes a "Wife" by marriage to a "Husband." The "Husband" is the concept that adds the additional semantics to "Woman" to make her a "Wife". Finally, "marriage" is the thirdness, or mediating circumstance, relating the first level concept "Woman" to the second-level concept "Wife".

Formalizing the concepts of firstness (F), secondness (S), and thirdness (T), we arrive at a functional specification:

$f(T_1, A_1, A_2, ..., A_N) => R$, where T_1 *in* F, $A_1..A_N$ *in* F *or* S, R *in* S, *and* f *in* T.

Using the previous example, $f(woman, husband) => wife$, where $f = marriage$. For simplicity, we often abbreviate the function specification to omit the concepts which react with the first level concept to produce a second level concept. The function then becomes:

$f(T_1) => R$. In our example, $f(woman) => wife$, $f = marriage$, and we omit the husband in the functional specification as it is often not required.

Given this top-level ontology, there are some general rules and procedures for adding new terms to the dictionary:

- Terms should be added on an as-needed basis to keep the size of the dictionary small.
- A term should not be added if its semantics can be accurately represented by the combination of existing terms in the dictionary.
- A term should be placed in the hierarchy as close as possible to terms with similar semantics.
- More general concepts appear at the top of the hierarchy so specific terms should appear near the leaf nodes.
- Always add the more general definition of a term that captures the necessary concept semantics. For example, add the term `Id` to represent an id for a book table not `Book Id`.

Once it has been determined that a term represents a new concept that must be added to the dictionary, there are several decisions that must be made before adding the term:

- Choose the most general English term to represent the concept.
- Determine if the concept is physical (tangible) in nature or information (intangible) in nature. This corresponds to choosing the correct branch at the second level of the hierarchy.
- Determine if the item is a first, second, or third level concept. Typically, database concepts are first or second order concepts. The easiest way to determine if a concept is a first order concept is to determine if its semantics require a relationship with another concept. If the element is a base level term with no relationship semantics it is most likely a first level concept. If the term requires a relationship with another term to define its existence, it is probably a second level concept.

A standard dictionary constructed as a product of test integrations is listed in its entirety in Appendix G. A demonstration of how the dictionary evolves as terms are added is given in Section VI.3, which illustrates how the Northwind database concepts are added to the base hierarchy.

III.2.2 Constructing Semantic Names

A *semantic name* captures system-independent semantics of a relational schema element including contextual information by combining one or more dictionary terms. A semantic name is a **context** if it is associated with a table and a **concept** if it is associated with a field. A context contains no data itself and is described using one or more concepts. Similar to a field in the relational model, a semantic name which is a concept represents atomic or lowest-level semantics.

A semantic name consists of a context and concept portion. The **context portion** is one or more terms from the dictionary which describe the context of the schema element. Adjacent context terms are related by either IS-A (represented using a ",") or HAS-A (represented using a ";") relationships. The **concept portion** is a single dictionary term called a concept name and is only present if the semantic name is a concept (maps to a field). Thus, a semantic name for a table never has a concept name because it contains only context information. The formal specification of a semantic name is as follows:

$$semantic\_name ::= [CT\_Type] \mid [CT\_Type]\ CN$$

$$CT\_Type ::= CT \mid CT\ ;CT\_Type \mid CT\ ,CT\_Type$$

$$CT ::= <\ context\ term\ >,\ CN ::= <\ concept\ name\ >$$

A *dictionary term* is a single, unambiguous word or word phrase present in the standard term dictionary. Each term represents a unique semantic connotation of a given word phrase, so words with multiple definitions are represented as multiple terms in the dictionary. A *context term* is a dictionary term used in a semantic name which describes the context of the schema element associated with the semantic name. A *concept term* is a single dictionary term used in a semantic name which provides the lowest level semantic description of a database field. For example, the semantic name `[Category] Id` is a concept because it maps to the database field *CategoryID*. The semantic name `[Category]` is a context because it maps to the database table *Categories*. The concept portion of `[Category] Id` is `Id`, and the context portion is `[Category]`.

Formally, a *semantic name* S_i consists of an ordered set of dictionary terms $T = \{T_1, T_2, ..., T_N\}$ where $N \geq 1$ which uniquely describe the semantic connotation of a schema

element. If $N = 1$, then T_1 is a context term. The last term T_N is a concept name if S_i has a concept name, otherwise it is the most specific context of S_i. When integrating semantic names into a context view, it is necessary to match semantic names based on their associated terms. For this purpose, it is useful to define the *context closure* of a semantic name.

Definition. The *context closure* of a semantic name S_i denoted S_i^* is the set of semantic names produced by extracting and combining ordered subsets of the set of terms $T = \{T_1, T_2, ...T_N\}$ of S_i starting from T_1. ■

Example 1. Given a semantic name $S_i = [A; B; C]D$ the context closure is:

$$S_i^* = \{[A], [A; B], [A; B; C], [A; B; C]D\}.$$

The algorithm for properly constructing semantic names is represented by a flow chart in Figure III.3. The first decision to be made when assigning a semantic name is to determine if the semantic name is for a field (concept) or a table (context). If the schema element is a table, then its semantic name will only have context terms. However, if the table is dependent on another table either due to a dependency relationship or a generalization relationship, then typically its context terms are related to the context terms of the table to which it is related. For example, *OrderDetails* (`[Order;Product]`) is dependent on *Orders* (`[Order]`). Thus, the context `[Order;Product]` of *OrderDetails* is a subcontext of `[Order]`.

Assigning a semantic name to a field is slightly more complex. First, if the field is a foreign key to another table, then its semantic name must be the context portion of the semantic name of the field's parent table plus the semantic name of the primary key of the table related by the foreign key. For example, *CustomerID* (`[Order;Customer] Id`) in *Orders* (`[Order]`) consists of the `[Order]` context combined with the semantic name of *CustomerID* (`[Customer] Id`) from *Customers*. The reason for defining the semantic names in this way is to ensure that when contexts are merged and queried in later steps there is an implicit connection based on the semantic name. By assigning the semantic name `[Order;Customer] Id`, it is obvious to both the user and the system that this field is a foreign key linking field which combines the `Order` context in the *Orders* table with the `Customer` context in the *Customers* table using the *CustomerID* field.

Assuming a field is not a foreign key, then it must be determined if the field is an

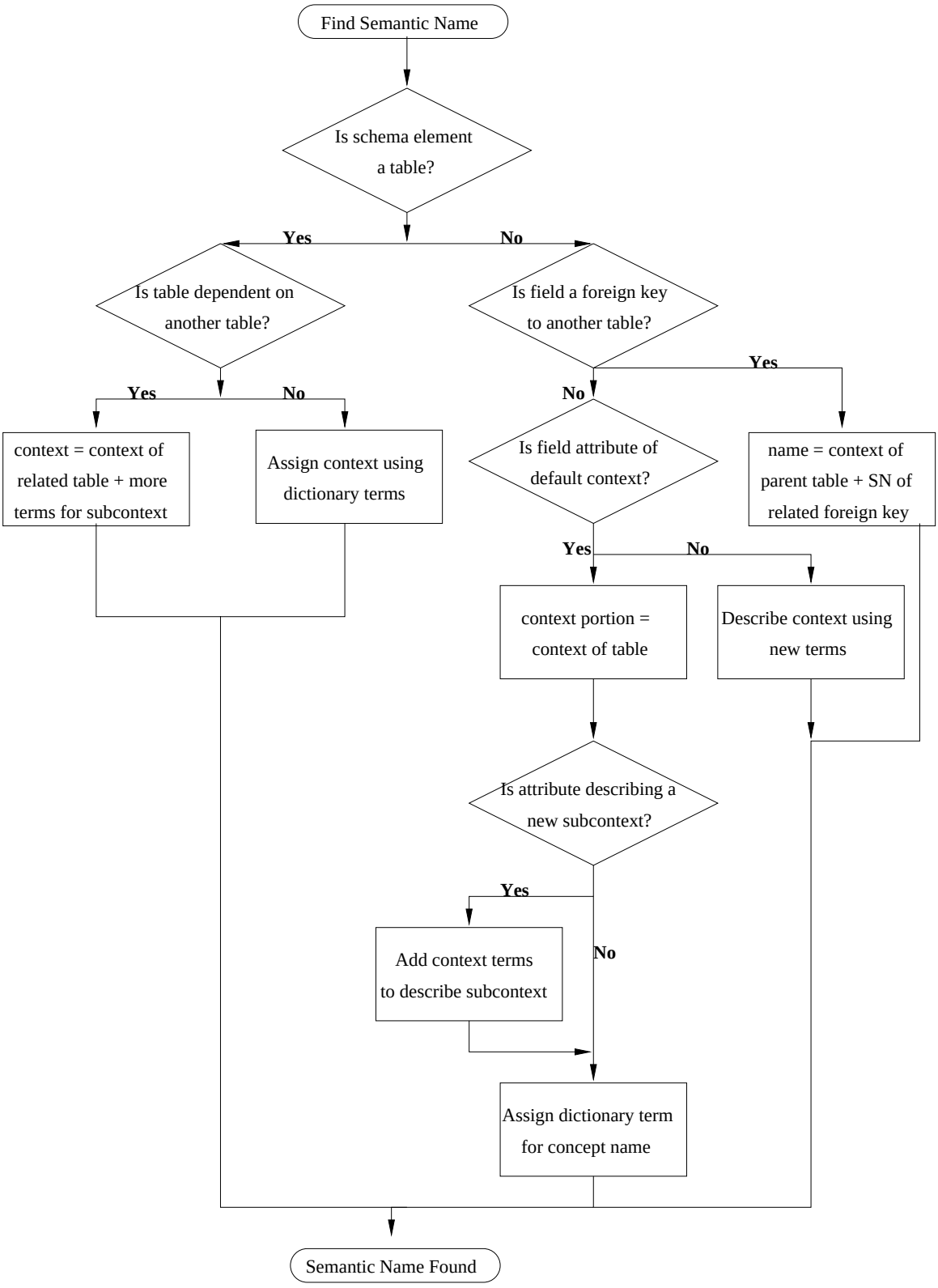

Figure III.3: Constructing a Semantic Name

attribute of the default context. The default context of the table is the semantic name of the table. Thus, the default context of *Orders* is `[Order]` indicating that fields (attributes) of this table are describing an order context. However, it is possible that a field in a table does not relate or describe the default context. In this case, the context portion of the semantic name would be different than that of the table, and the designer has to use judgement to determine the proper context. Presumably, this should be an extremely rare occurrence because a field which is not modifying the default context in a table must be out of place as it contains orthogonal information that probably should not be stored in the table. The only reasonable exception to this rule is key fields in a dependent table. For example, since *OrderDetails* (`[Order;Product]`) is dependent on *Orders* (`[Order]`), the key field *OrderID* (`[Order] Id`) is present in *OrderDetails* as part of its key. However, the context of *OrderID* does not match the context of its parent table `[Order;Product]`.

The most common case is that the field is describing the default context of the table. Thus, the first context terms in the field's semantic name are identical to those of the table. However, if the concept being described by the attribute is sufficiently general to warrant its own subcontext, then additional terms may be added to fully describe this new subcontext. For example, in the *Customers* table, address information is stored in the field *Address*. The semantic name assigned to this field is `[Customer;Address] Address Line 1` because although the field is describing information about a customer, it is more specifically describing information about a recognizable subcontext (an address). The additional context term `Address` is used to identify this subcontext and effectively group attributes more explicitly by subcontext.

Finally, in all cases, a field is assigned a single dictionary term as a concept name to describe its lowest level semantics. Typical concept names include dictionary terms `Id` for keys, `Name` for people and company names, and address information.

There are several properties of semantic names which are important to note:

- Every semantic name always has at least one context term.
- A table does not have a concept portion as part of its semantic name, while a semantic name for a field does have a concept portion.
- The context portion for primary key fields of a table must be identical to the context

portion of the table unless the primary key results from a dependency relationship. (e.g. *OrderID* = `[Order] Id` and *Orders* = `[Order]`)

- Semantic names for foreign key fields are always assigned such that they are the combination of the context portion of the field's parent table plus the semantic name of the key field in the other table related by the foreign key.
- The context portion of a field F_1 in a table T_1 should be identical or a subcontext of T_1 unless F_1 is not describing the default context of T_1.

The definition of a semantic name for a given database element is not a straightforward mapping to a single dictionary term because a dictionary term provides only a name and definition for a given concept without providing the necessary context to frame the concept. In BizTalk [77] schemas, the required context is assumed because a standard schema only applies to a very limited communication domain. For example, there are separate schemas for a purchase order, a shipment notification, and order receipt confirmations. Hence, by defining separate schemas, forcing the communication software to choose a specific schema, and defining separate terms for each schema, a single dictionary term provides both context and concept information. For our system to utilize a single dictionary, it is necessary to combine dictionary terms to provide both context and concept information.

III.3 X-Spec - A Metadata Specification Language

The definition of a standard dictionary is not sufficient to achieve integration because it does not define a standard schema for communication. It only defines the terms used to represent data concepts. These data concepts can be represented in vastly different ways in various data sources, so it is unreasonable to assume a standardized representation and organization for a given data concept. Thus, a system for describing the schema of a data source using dictionary terms and additional metadata must be defined. Our integration language uses a structure called an X-Spec to store semantic metadata on a data source. The X-Spec is essentially a database schema encoded in XML format and is organized in relational form with tables and fields as basic elements.

An X-Spec stores a relational database schema including keys, relationships, joins, and field semantics. Further, each table and field in the X-Spec has an associated semantic

name as previously discussed. Information on joins including their cardinality, join fields, and connecting tables is stored so that the query processor may identify which joins to apply during query formulation. Similarly, field relational dependencies are also stored so that query-time result normalization is possible.

An X-Spec is constructed using a specification editor during the capture process, where the semantics of schema elements are mapped to semantic names. Capture processes are performed independently because the only "binding" between individual capture processes is the use of the dictionary to provide standard terms for referencing data.

X-Specs are constructed using XML because XML is an emerging semantic exchange standard. However, X-Specs may also be represented as formatted text files or structured binary files. XML is used to support interoperability with emerging standards.

Describing a database using an X-Spec is very similar to standard schema development in BizTalk. We attempt to follow emerging industry standards in the description of schemas using XML and model an X-Spec schema description after BizTalk schemas. The important distinction between an X-Spec schema and a BizTalk schema is that an entire BizTalk schema is standardized whereas an X-Spec schema only uses standardized terms from the global dictionary. An X-Spec describes a database dependent schema rather than conform to one. As an X-Spec is intended to capture as much metadata as possible about a database schema, additional XML tags are used in its specification that are not in BizTalk schemas.

The specification editor allows the user to modify the X-Spec to include information that may not be electronically stored such as relationships, foreign key constraints, categorization fields and values, and other undocumented data relationships. More importantly, the specification editor requires the user to build a semantic name for each field and table name in a relational schema. The semantic name captures the semantics of the schema element using standard terms. During integration, the integration algorithm parses the semantic name to determine how to combine the element into the unified view. Once in the unified view, the semantic name acts as a user's portal to the database; the semantic name is used in SQL-queries, and the architecture maps the semantic names back to system names before the actual execution of a transaction.

```
<?xml version="1.0" ?>
<Schema
        name = "Southstorm_xspec.xml"
        xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:dt="urn:schemas-microsoft-com:datatypes">

<ElementType name="[Order]" sys_name = "Orders_tb" sys_type="Table">
        <element type = "[Order] Id" sys_name = "Order_num" sys_type = "Field"/>
        <element type = "[Order] Total Amount" sys_name = "Order_total" sys_type = "Field"/>
        <element type = "[Order;Customer] Name" sys_name = "Cust_name" sys_type = "Field"/>
        <element type = "[Order;Customer;Address] Address Line 1" sys_name="Cust_address"
            sys_type="Field"/>
        <element type = "[Order;Customer;Address] City" sys_name = "Cust_city" sys_type = "Field"/>
        <element type = "[Order;Customer;Address] Postal Code" sys_name="Cust_pc" sys_type="Field"/>
        <element type = "[Order;Customer;Address] Country" sys_name="Cust_country" sys_type="Field"/>
        <element type = "[Order;Product] Id" sys_name = "Item1_id" sys_type = "Field"/>
        <element type = "[Order;Product] Quantity" sys_name = "Item1_quantity" sys_type = "Field"/>
        <element type = "[Order;Product] Price" sys_name = "Item1_price" sys_type = "Field"/>
        <element type = "[Order;Product] Id" sys_name = "Item2_id" sys_type = "Field"/>
        <element type = "[Order;Product] Quantity" sys_name = "Item2_quantity" sys_type = "Field"/>
        <element type = "[Order;Product] Price" sys_name = "Item2_price" sys_type = "Field"/>
</ElementType>
</Schema>
```

Figure III.4: Southstorm X-Spec

Consider designing an X-Spec for the Southstorm database. The semantic name for the *Orders_tb* table is `[Order]` because its records represent an order context. A semantic name for the field *Order_num* is `[Order] Id` since the order number is the identifying attribute of an order. The context term is `[Order]` and the concept name `Id` represent the semantics of the id attribute for the context. Similarly, the field *Item1_id* is mapped to the semantic name `[Order;Product] Id`, as the `Id` concept applies to a `Product` which is part of an `Order`. The complete X-Spec (types and field sizes omitted) is in Figure III.4.

The X-Spec is a simple XML schema storing the Southstorm database schema. The additional tags `sys_name` and `sys_type` define the system name for the database element and indicate whether it is a table or a field. The names assigned to the individual elements are semantic names consisting of multiple terms from the standardized dictionary. A complete X-Spec is augmented with additional metadata about the field types and sizes.

In summary, an X-Spec is a database schema and metadata encoded in XML that stores semantic names to describe schema elements and is exchanged between systems. Although XML is used for transmission, an X-Spec is more than XML because it uses XML tags to capture metadata. An X-Spec is different than a BizTalk schema because it is not

intended as a standardized schema for data communication. It is a document describing an existing database schema which uses terms out of the standard dictionary to capture data semantics. As such, there will be a different X-Spec for each data source.

III.4 Integration Algorithm

Integration algorithms for industrial systems are straightforward because there is total uniformity. To participate in EDI and BizTalk communications; one must rigorously follow the standardized schema. Since the standardized schema dictates the exact structure, organization, and types of all fields, an EDI or BizTalk parser must only match fields based on their location or name. There is no room for interpretation; either an EDI/BizTalk document conforms to the schema or it does not. This standardization makes it relatively straightforward to define the necessary software to perform the integration task and communicate data between systems. The drawback to such rigid standardization is that it is inflexible. Transforming data into the correct standard format may require considerable effort. Further, definitions of the standard schemas will increase in complexity in accordance with the legacy system's complexity.

Integration algorithms proposed in the database research community have been incapable of achieving automation because they do not accept standardization. Theoretical research tends to be wide ranging so some constraints must be placed to ensure only tractable contexts are considered. A standard dictionary is required as a framework for communication but integration of database schema is possible without rigid conformity to an exact standard schema.

An X-Spec describing a database is not guaranteed to exactly match the X-Spec describing an almost identical database. There are also no guarantees that two X-Specs created by two different designers describing the exact same database will be identical, although they should be very similar. The reason for this is that the X-Spec does not format data according to a standardized schema, rather it describes the current structure of the database and uses terms out of the standard dictionary to express semantics.

Thus, integration is achieved not by the exact parsing of a document but by match-

ing standard terms to relate concepts. The same term used in two different X-Specs is known to represent the identical concept regardless of its representation. Since one X-Spec may represent a concept as a field while another represents it as a table, the integration algorithm must handle this diversity and organize the resulting integrated schema accordingly.

The integration algorithm processes one or more X-Specs describing database schema. It then uses the semantic names present in the X-Specs to match related concepts. The integration algorithm is a straightforward term matching algorithm. An overview of the integration algorithm is provided in Figure III.5, and the C++ implementation is given in Appendix D. The product of the integration algorithm is a structurally-neutral, hierarchical representation of database concepts called an integrated, context view (CV).

Definition. A *context view (CV)* is defined as follows:

- If a semantic name S_i is in CV, then for any S_j in S_i^*, S_j is also in CV.
- For each semantic name S_i in CV, there exists a set of zero or more mappings M_i which associate a schema element E_j with S_i.
- A semantic name S_i can only occur in the CV once.

That is, for every semantic name that exists in the context view, all its associated semantic names, formed by taking a subset of its terms, are also in the context view. Each semantic name in the view can be mapped to physical fields and tables by the set of mappings provided by the system.

The integration architecture combines schema elements into the context view by merging their associated semantic names with the semantic names currently present in the CV. Matching proceeds term-wise until a complete match is found or until no further matches are found. Thus, the CV is a tree of nodes $N = \{N_1, N_2, ..., N_n\}$, where each node N_i has a full semantic name S_i consisting of one or more dictionary terms $\{T_1, T_2, ..., T_m\}$. When a node is added, each of its corresponding terms are recursively added starting at the root.

Consider the Southstorm database example. Start with an empty integrated view V. The semantic name `[Order]` is automatically added to V as it is empty. When adding `[Order;Product]`, the first term `[Order]` matches with the order term already in V. The

algorithm searches for the term `[Product]` in V under `[Order]` which is not present so `[Product]` is added to V under `[Order]`. Similarly, when `[Order] Id` is integrated into V, `Id` gets added under `[Order]` in V. Repetition of the process produces the integrated view in Figure III.6.

```
Procedure Integrate(X as X_Spec, ByRef V as View)
    SN as Semantic_name
    T_array as Array of Term
    E as ElementType

    For each E in X                                                          (1)
        SN = get_semantic_name(E) // Extract SN from XML definition of E     (2)
        get_terms(SN,T_array) // Parse SN to extract its component terms     (3)
        match_sname(SN,T_array,1,X,V,V.get_root())                           (4)
    Next                                                                     (5)
End Procedure

Procedure match_sname(SN as Semantic_name, T_array as Array of Term, term# as Int,
                      X as X_Spec, ByRef V as View, cur_node as View_node)

    If term# == T_array.count() Then                                         (1)
        add_DB_map(SN,X,V) // Store DB mapping                               (2)
        Return // Matched all terms                                          (3)
    End if                                                                   (4)

    If cur_node == NULL Then                                                 (5)
        Return                                                               (6)

    // Otherwise, match all children of cur_node with the current term
    For each child_node of cur_node                                          (7)
        If child_node.sem_name.term[term#] == T_array[term#] Then             (8)
        // Match at this level, so recursively match at next level
            Return match_sname(SN,T_array,term#+1,X,V,child_node)            (9)
        End If                                                               (10)
    Next                                                                     (11)

    // There is no further match - add all remaining terms (including this one)
    For term_num to T_array.count                                            (12)
        add_node(SN,X,V)              // Add term to V and record mapping to X (13)
        If term_num == T_array.count Then                                    (14)
            add_DB_map(SN,X,V) // Store DB mapping                           (15)
        End if                                                               (16)
    Next                                                                     (17)
End Procedure
```

Figure III.5: Integration Algorithm

Given this integrated view, a user constructs queries by choosing which fields should be displayed in the final result. The required joins between the tables are automatically inserted by the query processor. This example illustrates the integration of an X-Spec with an empty integrated view but it is no more complex to now integrate with another

Integrated View Term	Data Source Mappings (not visible to user)
V (view root)	**N/A**
- [Order]	SS.Orders_tb
- Id	SS.Orders_tb.Order_num
- Total_amount	SS.Orders_tb.Order_total
- [Customer]	
- Name	SS.Orders_tb.Cust_name
- [Address]	
- Address Line 1	SS.Orders_tb.Cust_address
- City	SS.Orders_tb.Cust_city
- Postal Code	SS.Orders_tb.Cust_pc
- Country	SS.Orders_tb.Cust_country
- [Product]	
- Id	SS.Orders_tb.Item1_id, SS.Orders_tb.Item2_id
- Quantity	SS.Orders_tb.Item1_quantity, SS.Orders_tb.Item2_quantity
- Price	SS.Orders_tb.Item1_price, SS.Orders_tb.Item2_price

Figure III.6: Southstorm Integrated View

X-Spec describing a different database. Actually, the integration is identical because during this example the X-Spec was integrated into the view with itself. That is, as each new term was added to the view, the following term was integrated with all the terms that were already added. Integrating another X-Spec with the same semantic names would yield the same result. The order in which X-Specs are integrated is irrelevant and the same X-Specs can be integrated several times with no change. As more X-Specs are integrated, the number of fields and concepts would grow, but assuming the semantic names are properly assigned, the integration effectiveness is unchanged.

Once all semantic names of an X-Spec are integrated, a second optional integration phase is performed. The second step uses metadata in the individual data sources to refine the integrated view. The metadata phase of the integration algorithm performs the following actions:

- Uses relationship information (1-1, 1-N, M-N) in the specification to validate matchings and re-organize the integrated view.
- Uses field sizes and types to validate initial matchings.
- Creates mechanisms for calculating totals and functions to derive attributes in one data source which are not explicitly stored in others.

III.4.1 Concept Promotion

Since there are only two modeling constructs in the context view, a context and a concept, the only possible structural conflict in the integrated view occurs when two semantic names have the same terms and one is a context and the other is a concept. In this case, the context term is more general than the concept term which implies that in one database the idea is more detailed than in another. For example, if one database stores an employee's name as a single field with semantic name `[Employee] Name` and in a second database, an employee's name is further subdivided into first and last names `[Employee;Name] Last Name, [Employee;Name] First Name`, during integration, the semantic names `[Employee] Name` and `[Employee;Name]` will be combined. However, one is a concept and the other is a context.

These conflicts at the integrated view level are caused by conflicts in the relational model such as the table versus attribute conflict or representing a concept using different numbers of attributes. The solution to this problem is *promotion*. The more general context term `[Employee;Name]` is added to the integrated view. The concept term is promoted to a concept of this new context. The semantic name becomes `[Employee;Name] Name`. Abstractly, promotion is a mechanism for handling generalization related conflicts.

III.4.2 The Context View as a Universal Relation

A fundamental database model is the Universal Relational Model which provides logical and physical query transparency by modeling an entire database as a single relation. We will demonstrate the similarity of the context view with the Universal Relation Model [71], and thus argue that our system also provides logical and physical query transparency. There has been substantial work presented on querying in a Universal Relation environment [15], and more generally in the theory of joins [2] and querying [87, 52].

There is an underlying similarity between a context view and a Universal Relation. A Universal Relation (UR) contains all the attributes of the database where each attribute has a unique name and semantic connotation. Although further extending assumptions on the Universal Relation related to access paths and attribute interrelationships are proposed

by Maier *et al.* [71], the fundamental feature of the UR is that all attributes are uniquely named with a unique connotation.

Lemma. *A context view (CV) is a valid Universal Relation if each semantic name is considered an attribute.*

Proof. For a given data source, each field is assigned a semantic name. The semantic name defines a unique semantic connotation for the field. To violate the Universal Relation assumption, a given semantic name must either occur more than once in the CV (non-unique attribute names) or two or more semantic names must have identical connotations (non-unique semantic connotations). A semantic name can only occur once in a CV by definition. Hence, each semantic (attribute) name is unique. The construction of a semantic name by combining terms defines its semantics such that two different semantic names cannot have the same semantic connotation. Thus, a context view is a valid Universal Relation. ■

Essentially, the automatic construction of a context view by combining semantic names builds a Universal Relation describing a data source. However, the context view is a hierarchy of contexts and concepts which serve to semantically subdivide the attributes of the relation.

Although a given semantic name occurs only once in a CV, it is entirely possible that there is more than one mapping to physical fields in even a single data source. For example, consider the Northwind database with tables *Orders* and *OrderDetails*. The field *OrderID* in both tables is assigned the exact same semantic name `[Order] Id`. This makes sense because each field has the same semantic connotation and is only represented as two fields due to the normalization of the tables. When these two tables are combined into a UR, only one instance is retained. However, the query system must decide on the correct and more efficient mapping when generating query access plans.

A context view examined as a Universal Relation addresses several of the problems of the UR model. First, the context view is automatically created by the system when the database semantics are systematically described by the DBA. In the construction of the semantic names, the DBA uniquely defines the semantics and name for each field. The system then uses the supplied semantics, schema and join information to automatically build the context view. As we will demonstrate, this process can be applied in reverse

to extract query results from normalized database tables given a query expressed on the context view.

The context view also resolves the issue of large and complex Universal Relations. Since the context view is organized hierarchically by context, there is an explicit division of the context view into semantically grouped topics as opposed to one, flat relation containing all attributes. This reduces the semantic burden on the user when selecting query fields.

The context view is more than a Universal Relation. It is a hierarchically organized, integrated view of database knowledge in one or more systems. It is designed for easy integration of databases by capturing the semantics of their schema elements. Unlike a strict Universal Relation implementation, the context view is never physically constructed. Rather, like a view, it is an amalgamation of data stored in other structures which is built as needed. Queries posed through the context view can be physically realized by an automatic algorithm which maps from semantics to structure and produces SQL expressions on the underlying databases to extract the relevant data.

III.5 Query Processor

The integrated view of concepts is not a structural view consisting of relations and attributes. Rather, the *context view* is a hierarchy of concepts and contexts which map to tables and fields in the underlying databases. Thus, querying the integrated view is different than existing systems, and implementing the query processor results in new challenges.

The system implements querying by context that allows the user to formulate queries by manipulating elements in the context view. As discussed, the context view provides physical and logical access transparency similar to the Universal Relation [71]. Users generate queries by manipulating semantic names. The user is not responsible for determining schema element mappings, joins between tables in a given data source, or joins across data sources which are all determined by the system.

The query processor performs two basic steps. The first step is **dynamic view creation** which extracts the relevant query data from each database. The second step **automatically integrates the views** constructed for each database. The query processor executes the following procedures:

- **Dynamic View Construction** - Given the user's query and semantic names selected, determine data mappings at each database to generate an SQL query. The query result produces a dynamic view of relevant information.
  - **Enumerate Semantic Names** - required by the user.
  - **Determine Relevant Fields and Tables** - For each data source, determine the best field mapping(s) for each semantic name and their associated tables.
  - **Determine Join Conditions** - Given a set of fields and tables to access in a data source, determine which joins to insert to connect database tables.
  - **Generate and Execute SQL Queries** - created in the previous steps.
- **Automatic View Integration** - involves retrieving the individual local views from the databases and reconciling any remaining conflicts including result normalization, global field orderings, global joins and keys, and formatting to resolve data conflicts.
  - **Result normalization** - is required when the local views are not in the same normalization state.
  - **Global Keys and Joins** - may be automatically applied if databases have keys which transcend their local database scope and can be compared across systems.
  - **Data Integration** - resolves lowest level field value conflicts resulting from different data representations, scaling factors, precision, and data types.

III.6 Dynamic View Construction

III.6.1 Enumerating Semantic Names

In some cases, a real-world concept may be represented multiple times in the database and possibly with different semantic names. The first task for the user is to determine the correct semantic name for querying. For example, in Northwind there are two semantic names that map to the concept of a "Shipper ID": *ShipperID* (`[Shipper] Id`) in *Shippers* and *Shipvia* (`[Order;Shipper] Id`) in *Orders*.

The choice of semantic name depends on the query requirements. Selecting `[Shipper] Id` will select all shipper ids whether or not they have transported an order, whereas `[Order;Shipper] Id` only returns shippers who have sent orders. Notice that the difference between the semantic names is based on the additional constraint that one is dependent on order information while the other is not.

When there is only a single database integrated into the context view, choosing the correct semantic name is not a problem. The user will be able to select the correct instance based on their query requirements. However, when such semantic constraints differ across databases, the system becomes responsible for choosing the correct semantic name for each database. Even more generally, the query system must be able to support the notion of joins between contexts which may exhibit slightly different semantic properties.

There is no concept of a "join" in the integrated view because it does not specify a structure for data representation. However, an operation equivalent to a join in the integrated view is a context merger. A *context merger* is the combination of two contexts by applying a relationship condition. A relationship condition may be a join between tables if a join exists between the contexts or a cross-product if it does not.

The user never directly specifies the relationship condition between two contexts in the integrated view, so the system must determine how to relate the two contexts in each data source. Since the user is able to query on any semantic name in the integrated view, the semantic concepts chosen may be related by being in the same table, by joins between tables, or not related at all. For example, if the user requires the names of all customers that have placed orders (`[Order;Customer] Name`), the relationship between the `[Order]`

and `[Customer]` contexts must be determined. In Southstorm, no join or cross-product is applied because the field is directly present in the *Orders_tb* table. For Northwind, the query processor must insert a join from *Orders* to *Customers* to retrieve the same information.

Context merging is also applied to discover mappings which result from inter-schema relationships. Using the customer name example, there are actually two terms for customer names: `[Customer] Name` (Northwind) and `[Order;Customer] Name` (Southstorm). Although related, these terms are not identical because in Southstorm, unlike Northwind, a customer name cannot exist without an order. The extra context `[Order]` in Southstorm implicitly captures this structural constraint by indicating that a `[Customer]` context is only valid as part of an order. However, if the user requires the names of all customers (`[Customer] Name`), the implicit assumption in this query is that customers in Southstorm should be displayed even though they exist only as order information. Thus, since Southstorm has no direct mapping for `[Customer] Name`, the system searches for a semantic name in the Southstorm X-Spec containing `[Customer] Name` and finds `[Order;Customer] Name` which is used in the query mapping.

III.6.2 Determining Relevant Database Fields and Tables

The query system determines which physical tables and fields to access in the data source based on the semantic names chosen by the user. In most cases, a semantic name in the integrated view has only one mapping to a physical field. However, in special cases, especially when considering key fields, a semantic name may map to several physical fields. Since the choice of field (and its parent table) may affect the semantics of the query, the query system must have well-defined rules which are logical and easily conveyed to the user.

Semantic names are selected by the user for display in the final result (projection) or for specifying selection criteria. Regardless, if the field is being used in a selection or projection operation, all fields are treated uniformly by the query system.

Determining the correct field instance to select if a given semantic name can be mapped to multiple fields in the underlying database is complex. Fortunately, it is unlikely that a semantic name has multiple field mappings when the database is normalized if the field is not a key field. However, the choice of a key field with multiple mappings is especially

important as it affects the join semantics. Depending on the field mapping chosen, different tables are joined together. For example, the semantic name `[Order] Id` maps to two physical fields: *OrderID* in the *Orders* table and *OrderID* in the *OrderDetails* table. In both cases, the field has the same semantics. However, depending on which of the two mappings is selected, a new join may be introduced into the query if the table is not currently in the query. The general heuristic is to choose the primary key instance (*Orders*) unless the user selects attributes from the *OrderDetails* table.

There is one other special case when a semantic name may have multiple field mappings. When a database is not normalized, multiple fields in a single table may map to a semantic name. For example in Southstorm an order has only two items (*Item1_id, Item2_id*) stored in the *Orders_tb* table, and the semantic name `[Order;Product] Id` has two mappings in the table. The semantically correct query should accept both field mappings, and then automatically normalize the data by splitting one order record into two normalized records when the results are presented to the user.

To handle multiple mappings, the query system first selects a field which is currently present in the tables already in the query. Otherwise, it chooses the field whose parent table context matches the field context. This is done to identify the most logical semantic choice for the field. Presumably, this identifies the most common occurrences of the field and often is the primary key of the parent table. Finally, the system takes the first field mapping encountered if no other heuristic applies. The algorithm presented (see Figure III.7) constructs a set of fields (F) and tables (T) which best map to the set of query nodes (semantic names) $Q = \{Q_1, Q_2, ..., Q_n\}$ given by the user.

III.6.3 Determining Join Conditions

Given a set of fields and tables to access in the query, the query system must determine a set of join conditions between the tables. It is important to isolate the user from join construction while choosing appropriate joins to preserve the query semantics.

We define a *join graph* as an undirected graph where each node corresponds to a table in the database, and there is a link from node N_i to node N_j if there is a join between the corresponding two tables. For this discussion, we ignore multiple joins between two

```
Procedure find_field_mappings()
    For each term Q_i in Q                                                              (1)
        SN_i = semantic name of Q_i                                                     (2)
        search_XSpec(X_j,SN_i,num,R) // Search X-Spec for SN. Return results in R.      (3)

        If num = 1 // Only one occurrence of semantic name                              (4)
            Add field R_k to F                                                          (5)
            Add parent table of R_k to T                                                (6)
        Else                                                                            (7)
            If multiple occurrences but only in one table Then                          (8)
                For each result R_k in R                                                (9)
                    Add field R_k to F                                                 (10)
                Next                                                                   (11)
                Add parent table of R_1 to T                                           (12)
            End if                                                                     (13)
        End if                                                                         (14)
    Next                                                                               (15)

    // Second pass to resolve multiple occurrences
    For each term Q_i in Q                                                             (16)
        SN_i = semantic name of Q_i                                                    (17)

        If Q_i has not been mapped                                                     (18)
            search_XSpec(X_j,SN_i,num,R) // Search X-Spec for SN. Return results in R. (19)

            If there exists any mapping R_k of R with parent table T_j already in T Then (20)
                Add field R_k to F                                                     (21)
                Add parent table of R_k to T                                           (22)
            ElseIf Find parent table T_j of R_k with context portion = context portion of Q_i Then (23)
                Add field R_k to F                                                     (24)
                Add parent table of R_k to T                                           (25)
            Else                                                                       (26)
                Add field R_1 to F // Otherwise, add first mapping                     (27)
                Add parent table of R_1 to T                                           (28)
            End if                                                                     (29)
        End if                                                                         (30)
    Next                                                                               (31)
End Procedure
```

Figure III.7: Field Selection Algorithm

tables on different keys. A *join path* is a sequence of one or more joins interconnecting two nodes (tables) in the graph, and a *join tree* is a set of one or more joins interconnecting two or more nodes. Assume without loss of generality[1] that the join graph is connected. The join graph for the Northwind database is illustrated in Figure III.8.

Lemma 1. *If a join graph is acyclic, there exists only one join path between any two nodes.*

Proof. Proof by contradiction. Assume that two join paths exist between node N_i and node N_j where $i \neq j$. Then, we could take the first path from N_i to N_j, and return on the second path from N_j to N_i. This implies that the graph has a cycle. ■

[1]Otherwise, we apply the algorithm to each connected subset and connect them using a cross-product.

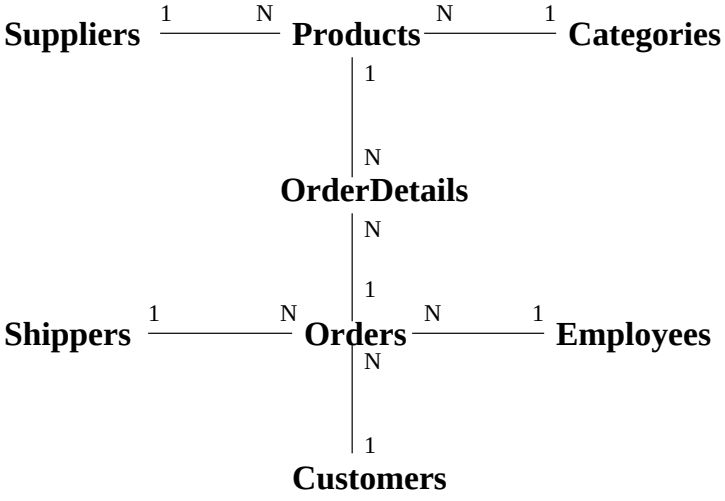

Figure III.8: Join Graph for Northwind Database

Lemma 2. *If a join graph is acyclic, there exists only one join tree between any subset of its nodes.*

Proof. Proof by induction. The statement is true for two nodes as per Lemma 1. Given a subset of m nodes with only one join tree connecting them in the join graph, add another node N to the set. Assume that by adding N there exists more than one join tree in the new subset of $m+1$ nodes. Since, there was only one join tree for the previous m nodes, this implies that N must be connected to more than one node in the subset. If N is connected to two nodes N_i and N_j in the m nodes where $i \neq j$, then there must be a path from N to N_i, N_i to N_j, and N_j to N by Lemma 1. This produces a cycle. Thus, the statement holds for $m + 1$ nodes, and the result follows by induction. ■

The consequences of Lemma 2 are important. If the join graph for a database is acyclic, there exists only one possible join tree for any of its tables. This implies that the query system does not have any decisions involving which joins to apply. We must only identify which joins are needed to connect the required tables by constructing the join tree. The actual order in which the joins are applied is a join optimization problem which has been actively studied but will not be discussed here.

From this result, it is possible to construct an algorithm which builds a matrix M where entry $M[N_i, N_j]$ is the shortest join path between any pair of nodes N_i and N_j.

By combining join paths, the query system can identify all the joins required to combine database tables by constructing the only possible join tree. The general algorithm is presented later in the section. Theorem 1 proves how the join tree can be constructed for any tables in the join graph using the shortest join path matrix M. Determining the join tree allows the query system to insert the correct join conditions during SQL generation.

Theorem 1. *Given a matrix M, which stores shortest join paths for an acyclic join graph, and a set of tables T to join, a join tree can be constructed by choosing any table T_i from T and combining the join paths in $M[N_i, N_1]$, $M[N_i, N_2], ..., M[N_i, N_n]$ where $N_1, N_2, ..., N_n$ are the nodes corresponding to the tables in T.*

Proof. Proof by contradiction. Since the graph is connected, the matrix entries $M[N_i, N_1]$, $M[N_i, N_2], ..., M[N_i, N_n]$ represent join paths from N_i to all other nodes in the subset. Assume a join tree is not constructed. Thus, there is no path between some two nodes N_j and N_k where $j \neq k$. However, there is a path from node N_i to N_j and from node N_i to N_k. Combining these paths results in a path from N_j to N_k. Thus all nodes are connected with the join tree, and it is the only possible join tree as per Lemma 2. ■

Normalized databases often have acyclic join graphs. However, the general case of a cyclic join graph must be considered. A cyclic join graph arises when redundant joins are present in the database or when tables serve multiple semantic roles in a database. A given table can assume multiple semantic roles in several ways. One way is by acting as a lookup table for several other tables. For example, assume the Northwind database (see Figure III.9) also stored information on the employee who entered each order product in addition to the employee who entered the overall order. In this case, *Orders* and *OrderDetails* have foreign keys to the *Employees* table storing the employee who entered the record. This produces a cycle between *Orders*, *OrderDetails*, and *Employees*. Notice that the join path chosen between the tables represent different semantic queries. The join path *Orders* → *Employees; Orders* → *OrderDetails* represents the orders entered by employee with their products; *OrderDetails* → *Orders;OrderDetails* → *Employees* represents the orders with their products grouped by the employee entering the product; and *Employees* → *Orders;Employees* → *OrderDetails* represents which employees entered both an order product and its overall accompanying order.

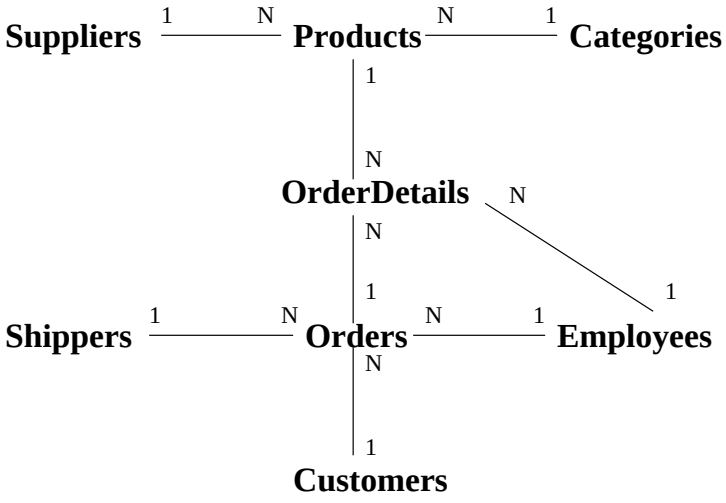

Figure III.9: Cyclic Join Graph for Northwind Database

A second instance of cyclic join graphs appears when a table stores a generalized concept which may have multiple subconcepts. Using Northwind as an example, assume the *Shippers* and *Suppliers* table are combined into one table called *Companies*. The *Companies* table effectively stores the general notion of a company and the specific types of companies: shippers and suppliers. Cycles occur when tables join to the different semantic instances (shipper,supplier) in the *Companies* table.

Finally, cycles occur when redundant joins are added to the database. For example, the *CategoryID* field could be added to *OrderDetails* for a direct link to *Categories* instead of joining through *Products*. This results in a cycle involving *OrderDetails*, *Products*, and *Categories*. Note that joins of this nature may be lossy when used in combination with other, valid lossless joins. An invalid lossy sequence of joins contains a join with an N-1 cardinality followed by a join with an 1-N cardinality. There may be other joins between these two joins. The result is a lossy join because it results in an M-N cardinality relationship between the merged tables. Effectively, this leads to invalid information being created by using these joins. Also, a join of cardinality M-N between two tables is always lossy because invalid information is introduced because of the join. Thus, the algorithm first attempts to find join paths without using these types of lossy joins.

Consider the case of a join cycle between the tables *Categories*, *Products*, and

OrderDetails. The cycle is caused by a redundant join between *OrderDetails* and *Categories* on *CategoryID*. Now, consider that *OrderDetails* has one record, {*(OD,P1,C1)*}, *Categories* has one record, {*(C1)*}, and *Products* has two records, {*(P1,C1), (P2,C1)*}. If the system joins these tables on *CategoryID* from *OrderDetails* to *Categories* and from *OrderDetails* to *Products*, the result is {*(OD,P1,C1),(OD,P2,C1)*}. Although this may be a valid result in some cases, incorrect information (the second record) was introduced because of this sequence of joins. Hence, the query system will try to avoid such lossy joins and only apply them if a cross-product is the only other alternative. A similar example can be constructed for the "employee and order join" example mentioned previously where the *Employees* table had multiple joins because it had two different semantic roles.

To handle cycles, the query system must make a determination of the best join paths between nodes. The query system uses join semantics, path length, and join properties (total participation, lossless vs. lossy) to determine best join paths. The breadth-first search based algorithm (Figure III.10) constructs the matrix M of best join paths. It works for both cyclic and acyclic join graphs. The algorithm selects the shortest join paths with no lossy joins. Equal length join paths may be differentiated based on total participation or other join properties. Lossy joins are only used if there exists no other path between nodes (i.e. when a cross-product would be necessary).

It would be ideal if the algorithm in Figure III.10 and Theorem 1 would produce a single correct join tree for a cyclic join graph. However, if the graph is cyclic, there will be multiple join trees possible depending on the choice of starting node. These join trees are each semantically valid depending on the semantics of the query. The system cannot differentiate for the user without more knowledge about the intended query semantics. Although it may be possible to define heuristic algorithms to choose the correct join tree based on the attributes chosen for the query, it is more desirable to have a precise mechanism for the user to exploit. Thus, we define extensions to the query model that allow the user to more precisely define the semantics of the query such that the system can uniquely determine the join tree required.

```
Procedure calc_join_paths(ByRef M as matrix, G as graph)
    // M is an N x N matrix where N is the number of nodes in the graph
    // NQ is a FIFO queue structure

    count as Integer
    F, N, LTN as Node
    L as Link
    accept_lossy as Boolean
    join_type as Integer // Type of join by cardinality: 1-1,1-N,N-1,M-N

    For each node F in G                                                          (1)
        M[F,F] = Null // Empty join path to itself                                (2)
        count = 0                                                                 (3)
        accept_lossy = false                                                      (4)

    repeat_label:                                                                 (5)
        add F to NQ                                                               (6)

        While NQ is not empty                                                     (7)
            remove first node N from NQ                                           (8)
            For each outgoing link L of N                                         (9)
                LTN = destination node of link L from N                           (10)
                join_type = cardinality of join for L (from N to LTN)             (11)
                If LTN is not visited and (accept_lossy
                    or not((M[F,N] has a N-1 join and join_type=1-N)) or join_type=M-N) Then   (12)
                    add LTN to NQ                                                 (13)
                    mark LTN as visited                                           (14)
                    M[F,LTN] = M[F,N] + LTN                                       (15)
                    count++                                                       (16)
                ElseIf accept_lossy or not ((M[F,N] has a N-1 join and join_type = 1-N))
                    or join_type = M-N) Then                                      (17)
                        // May want to replace a join path already constructed (M[F,LTN]) if
                        // - new join path is the same length as current one and
                        // - new join path has better properties (eg. total participation)
                Endif                                                             (18)
            Next                                                                  (19)
        End while                                                                 (20)

        clear_flags() // Clear all visited flags for all nodes in G               (21)

        If count ≤ # of nodes in G and accept_lossy = false Then                  (22)
            accept_lossy = true                                                   (23)
            Goto repeat_label // Repeat algorithm accepting all joins (even lossy)   (24)
        Endif                                                                     (25)
    Next                                                                          (26)

    // Note: For any matrix entries not assigned no join path exists and a cross-product is required
```

Figure III.10: Algorithm to Calculate Join Paths

III.6.4 Query Extensions

To enable the user to more precisely define the semantics of the query, extensions to the integrated view are possible. The extensions allow the user to more accurately convey the semantics of the query or to override the system default semantics. The first extension is to allow the user to pick the root join table. Essentially, this gives the user the ability to choose which row in the join matrix to use. Semantically, the root join table chosen by the user is the starting point of all join paths. This allows the system to unambiguously construct a join tree which matches the users' intended query semantics. This extension is required because there are multiple join trees in a database with a cyclic join graph, each of which has different query semantics.

The second optimization is an enhancement of the integrated view presentation. Currently, the integrated view consists of a hierarchy of contexts and concepts. The join conditions relating the individual concepts are largely hidden to the user. However, these join conditions are automatically inserted by the query system as required. To make these interrelationships more apparent, the system can automatically display them. If a given semantic name (node) in the integrated view is actually a foreign key to another concept (table) then when the user clicks on this concept, the attributes of the linked concept are displayed.

For example, in Northwind the field *EmployeeID* in *Orders* has a semantic name `[Order;Employee] Id` corresponding to the foreign key from *Orders* to *Employees*. When the user clicks on this semantic name, the system automatically performs the join to the *Employees* table and displays to the user the fields of *Employees* (*EmployeeID*, *LastName*, *FirstName*) which can be added to the query.

This approach has several benefits. First, it reduces the semantic burden on the user by automatically displaying concept interrelationships. More importantly, it reduces the query generation complexity for the system. By explicitly displaying the join information and associated fields, the system now has an unambiguous reference from the user on which fields to use, from what tables, and the corresponding join condition (attribute) to use to relate the two different contexts.

For example, if the user selects the *LastName* field for inclusion into the query directly from the *Employees* table, it may be ambiguous how to join *Orders* and *Employees* if there are multiple join trees. (This is not a problem in the original version of Northwind but could be an issue in other cases as discussed.) However, if the user selects the mapping to *LastName* which results from expanding the join through `[Order;Employee] Id`, the system and the user now know the exact join path.

III.6.5 Generation and Execution of SQL Queries

Given the set of database fields and tables to access and a set of joins to apply, it is straightforward to construct an SQL select-project-join type query. The field mappings are used to form the **Select** portion of the SQL statement. The parent tables of the fields are included in the **From** section. The **Where** section contains the join conditions determined by constructing the appropriate join tree for the query. In cases where there are multiple join trees, the user's input when selecting semantic names allows the system to uniquely determine the join tree and use the shortest join paths in the join path matrix M to determine the correct joins to apply. The **Where** clause also includes any query conditions selected by the user. Ordering information is added to the **OrderBy** clause.

After building an SQL query string, the query is transmitted to the database for execution. Although there are multiple open standards and proprietary protocols for each database and environment, the architecture is designed to utilize the ubiquity of the ODBC standard to access all major database systems. Results returned from each ODBC query are then processed by the client to perform global level conflict resolution and formatting.

III.7 Automatic View Integration

III.7.1 Global Keys and Joins across Databases

Since X-Spec information is produced independently for each data source, the query processor does not have join information available to merge contexts across databases. To provide joins across databases, the architecture requires a common key to relate contexts. Even though key fields may be identical semantically, unless the scope of the key transcends the database itself, that key cannot be used to join across systems. The *scope* of a key is the context in which it is valid. We define two scope hierarchies:

- **Geographical hierarchy:** International, National, and Regional.
- **Organizational hierarchy:** Organization, Group, and Database.

The hierarchies are not mutually exclusive, although in practice they tend to be used in that manner. For example, a key in a book database may be ISBN. An ISBN is an internationally recognized key, so the scope of the key is International. Obviously, the international scope of ISBN subsumes the Organization scope.

At each level of the scope hierarchy, scope instances are defined. For example, under the National scope there are country names such as Canada, United States, *etc.* Under the organization scope, there are organization names such as Northwind, Southstorm, *etc.* Similarly, under the Database scope there are database names. Thus, a key scope consists of a hierarchy level and a scope instance. For example, a social security number (SSN) is a common government key. However, the scope of a Canadian database with a SSN key is National and "Canada", whereas the same key in an American database is National and "United States". Although both fields store the concept of SSN, this key will not be integrated across the databases because of their different scope.

The overall database scope is also important and is stored in an X-Spec. For instance, the database context information for Northwind is: National = "United States", Regional = "Texas", Organization = "Northwind", and Database = "Northwind.mdb". The database context for Southstorm is: National = "Canada", Regional = "Manitoba", Organization = "Southstorm", and Database = "Southstorm.mdb".

Given these two database contexts, consider the order id key field. In both cases, the order is only guaranteed unique within the organization. Thus, the scope of *Order_num* in Southstorm is Organization = "Southstorm", and the scope of *OrderID* in Northwind is Organization = "Northwind". If order records are queried from both databases, no integration of results is performed because the scopes of the order keys are different. However, if the product keys are internationally recognized SKU numbers, then the key scope of *ProductID* and *Item1_id/Item2_id* is International. When products are queried, the system matches product ids because their key scopes are identical.

Although this system is simplistic, the scope hierarchy can easily be tailored to meet specific application needs as required. The important benefits are:

- The context information of a database is easily captured.
- The scope of a key is stored for the system to automatically determine if information across databases should be joined or unioned together.
- The user can filter databases accessed at query time by specifying particular database contexts.

III.7.2 Result Normalization at Query-Time

The integrated view has no implied structural representation and organizes concepts hierarchically. The underlying databases may have similar data represented in various structural organizations. The previous sections demonstrated how structural and naming conflicts are resolved by mapping through the integrated view to produced dynamic views of each data source. However, the normalization states of each local database view may be different. For example, the Northwind database is in third normal form (3NF), but the Southstorm database is not normalized. Thus, the system is responsible for "normalizing" results returned at query-time to reconcile the different database normalization states.

The task of the view integration system is two-fold. First, there must be a mechanism for specifying dependencies between fields. Second, the system must process these dependencies, reconcile them with the integrated view, and normalize the query results.

Functional dependency information is stored in X-Specs. The system restricts the specification of functional dependencies such that a field may only specify one field which

functionally determines its value even if there are several, and a functional dependency is only specified for a field if its value is not functionally determined by the primary key of the table. The system builds dependency trees for each database table requiring normalization.

Definition. Define a *dependency tree*, $DT = (N, D)$, of functional dependencies for a database table such that:

- N is the set of nodes of the tree and D is the set of dependency links
- The root of the tree (R) represents the relation (table).
- There is a link from R to field F_1 if there is a field F_2 where $F_1 \rightarrow F_2$ and there does not exist a field F_3 where $F_3 \rightarrow F_1$.
- If $F_1 \rightarrow F_2$, then there is a link from field F_1 to field F_2.

Thus for each table T, there is a set of attributes N which require normalization and are present in the dependency tree. The remaining attributes are uniquely determined by the primary key and thus require no normalization. For example, in Southstorm, $Item1\_id \rightarrow Item1\_price$ and $Item1\_id \rightarrow Item1\_qty$ because given the item id, the quantity and price fields can be uniquely determined. The dependency tree of the *Orders_tb* table is given in Figure III.11. The Northwind database does not require any query-time normalization.

A dependency tree may contain several different semantic concepts. A *normalized dependency tree* is a dependency tree where at each level of the tree all fields have the same semantic name. Normalized dependency trees for Southstorm are in Figure III.12.

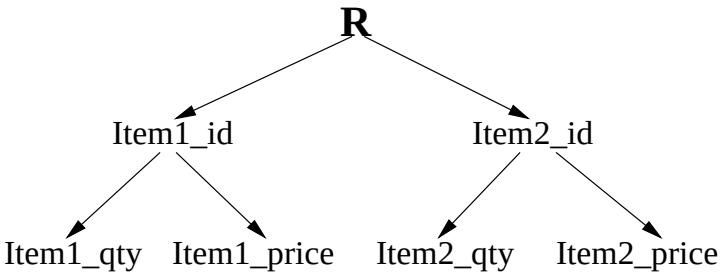

Figure III.11: Dependency Tree for Orders_tb in Southstorm Database

Given a set of normalized dependency trees, the resulting combinations of fields to produce normalized rows are easily constructed. For a single normalized dependency

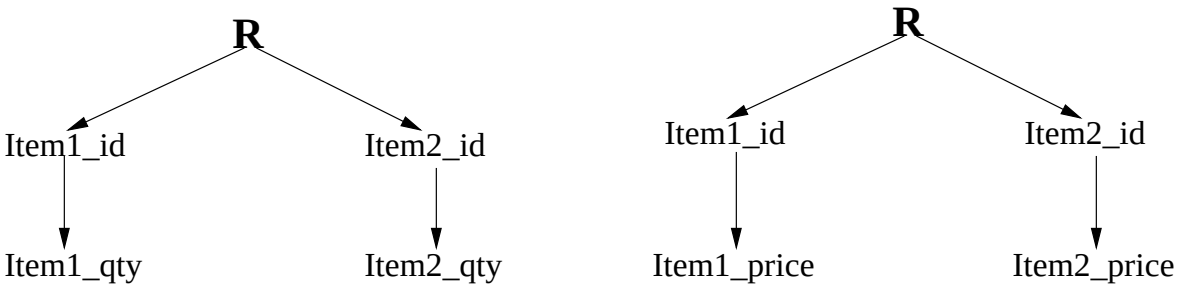

Figure III.12: Normalized Dependency Trees for Orders_tb in Southstorm Database

tree, the set of fields for each normalized row is produced by constructing the set of all depth-first search paths from the root node to any child node. Integrating result sets from multiple normalized dependency trees requires performing a cross-product between the result sets. During the cross-product procedure, filtering is performed because not all results from the cross-product are valid. A result is valid if and only if for each semantic name in the result there is a unique field mapping. For example, in Southstorm the semantic name `[Order;Product] Id` has mappings to *Item1_id* and *Item2_id*. If a row produced by forming the cross-product has `[Order;Product] Id` map to both of these fields then the row result is invalid and is discarded.

Given a result set which contains a mapping from semantic names to system field names for each row, the client-side processor iterates through the result set for each row returned by the query extracting the appropriate fields and inserting normalized rows in the final result. A normalized row consists of all attributes not requiring normalization plus an instance of the attributes requiring normalization. For example, normalizing `[Order] Id` and `[Order;Product] Id` for Southstorm, results in the normalized row sets of:

{(Order_num, Item1_id), (Order_num, Item2_id)}.

III.7.3 Query Examples

This section gives four example queries on the integrated view of the Northwind and Southstorm databases, and the SQL statements generated by the query processor. Construction of the integration view used in these examples is explained in Section VI.3, and the entire integrated view is given in Figures VI.10 and VI.11.

Example 1. The user requires the order id (`[Order] Id`) and order date (`[Order] Date`) for all available orders and selects the semantic names from the context view.

Southstorm	**Northwind**
Select Order_num	**Select** OrderID, OrderDate
From Orders_tb	**From** Orders

Execution 1. Notice that no date field is available in the Southstorm database. This field is left blank when a row result is displayed. The query system receives the output of both queries and displays them to the user. No join is applied on the order ids because the order keys are only valid within each database.

Example 2a. The user requires all customer names of orders (`[Order;Customer] Name`).

Southstorm	**Northwind**
Select Cust_name	**Select** CompanyName
From Orders_tb	**From** Customers, Orders
	Where Customers.CustomerID = Orders.CustomerID

Execution 2a. In this example, a structural conflict is inherently resolved by mapping through the context view. The customer name is retrieved from the *Orders_tb* table for Southstorm and from the *Customers* table for Northwind. The query processor performs a context merger by combining the `[Order]` context with the `[Customer]` context. A join is required between *Orders* and *Customers* in Northwind to merge the two contexts.

Example 2b. The user requires all customer names (`[Customer] Name`).

Southstorm	**Northwind**
Select Cust_name	**Select** CompanyName
From Orders_tb	**From** Customers

Execution 2b. This query is almost identical to the previous query except that it requests customers whether or not they have an order. In Northwind, this query can be answered without accessing the *Orders* table. However, due to the structural constraint in Southstorm, this information is embedded in order information. Since Southstorm has no direct mapping, the query system searches for related mappings with more specific contexts and finds `[Order;Customer] Name` and its associated system field *Cust_name*.

Example 3. The user requires all product ids (`[Product] Id`).

Southstorm	**Northwind**
Select Item1_id, Item2_id	**Select** ProductID
From Orders_tb	**From** Products

Execution 3. This example requires a context merger in Southstorm because the product ids are only specified within an `[Order]` context. Second, assuming the product ids are internationally recognized keys such as a product SKU number, they can be integrated across databases. Finally, Southstorm requires result normalization because multiple products ids appear in a single result row.

Example 4. Query for orders (`[Order] Id`) and products (`[Order;Product] Id`).

Southstorm	**Northwind**
Select Order_num, Item1_id, Item2_id	**Select** OrderID, ProductID
From Orders_tb	**From** OrderDetails

Execution 4. The system does not match record instances across the two databases because although the product id has international scope, the order id key is only valid within each database. Southstorm requires result normalization.

As shown in these simple examples, physical and logical query transparency is provided to the user who queries the system by semantic name. The system handles the necessary mapping from semantics to a structural query and inserts join conditions as required. The query processor discovers and constructs the query in a dynamic fashion based on the supplied mappings.

III.7.4 Comparison with SQL

The overwhelming acceptance of the SQL standard [29] has curtailed continuing research work in relational database query languages and processing. Since all commercial relational database systems conform with the SQL standard, there is little motivation for developing new query languages.

Despite its benefits and wide-spread acceptance, SQL is not a perfect query language. Complex database schemas challenge even experienced database programmers during query formulation. As increasing numbers of less sophisticated users access numerous data sources within an organization or across the Internet, their ability to accurately construct queries with the appropriate structure and semantics diminishes. SQL can be hard to use as it provides only physical access transparency not logical transparency. That is, a user is responsible for mapping the semantics of the query to the semantics and structure of

the database. Although graphical tools for query construction and high-level programming languages mask some of the complexity, the notion of querying by structure is intrinsic to most forms of data access.

Despite dramatic changes in database size, complexity, and interoperability, SQL has remained fundamentally unchanged. The wide-variety of applications, users, and independent systems accessing databases rely on Structured Query Language (SQL) [29] to retrieve the required information. Although the complexity of SQL generation has been partially hidden by graphical design tools and more powerful programming languages, the fundamental challenges of SQL remain.

The basic problem of SQL is also one of its greatest benefits. SQL allows a database to be queried by a clearly defined language which is a vast improvement over hierarchical methods and direct access technologies that require explicit navigation between records. Unfortunately, an SQL user is responsible for understanding the structure of a database schema, the names associated with schematic elements, and the semantics of the data stored. Query formulation involves mapping query semantics into the semantics of the database and then realizing those semantics by combining the appropriate database structures.

The SQL standard allows users to query different database platforms using one language. This provides a degree of interoperability between systems and prevents users from learning query languages for each database platform. SQL provides an efficient and structured way for accessing relational data. However, specifying complex SQL queries with numerous join conditions and subqueries is too complex for most users [10]. Further, developing SQL queries requires knowledge of both the structure and semantics of the database. Unfortunately, database semantics are not always immediately apparent from the database schema, and mapping the required query semantics into an SQL query on database structure is often complex.

There are graphical query tools [21, 19, 96] to aid in the formulation of SQL queries and proposed extensions to the SQL language exist [99]. Many commercial databases use similar tools to aid the user in query construction. However, at the lowest level, a user is still responsible for mapping the semantics of the query into a structural representation suitable for the database. Unfortunately, this semantics-to-structural mapping is non-trivial,

especially in large databases.

SQL is unsuitable for querying multidatabases or federated databases. These systems are a collection of two or more databases operating to share data. Extensions of SQL such as MSQL [68] and its successor IDL [53] provide features for multidatabase querying as discussed in Section II.3.2. The fundamental weakness in multidatabase query languages is the reliance on the user's knowledge of the database structure and semantics to construct queries. Further, data organization is optimized for efficiency not understanding. Understanding the structure and semantics of one data source is complicated in itself and the in-depth knowledge required to formulate queries on multiple databases is extremely rare. Although multidatabase query languages allow for the construction of multidatabase queries, they do not reduce the need for the user to thoroughly understand the semantics.

To simplify querying, systems [81, 24, 51] have been developed which allow users to query by word phrases. These systems are not powerful enough for a general multidatabase environment because they do not allow the user to precisely define the exact data returned. Unlike an SQL query which is deterministic and precise, "query by word" systems which simplify query formulation by ignoring structure, sacrifice query precision. Other systems which augment a relational database with logical rules or knowledge [54, 80] or change or add to the database in some manner to enable advanced queries to be posed violate database autonomy and thus are not desirable.

In a general environment, a query system must isolate the user from structure and system details and at the same time provide a query language powerful enough to produce precise, formatted results. SemQL [62] attempts to support semantic querying using semantic networks and synonym sets from WordNet [78]. Although their approach is similar to ours, using a large online dictionary such as WordNet increases the complexity of matching word semantics. Also, since no integrated view is produced, it is not clear to the user which concepts are present in the databases to be queried. Our approach improves on SemQL by providing a condensed term dictionary, an integrated view to convey database semantics to the user, and a systematic method for SQL generation.

Chapter IV

Unity - The Architecture Implementation

IV.1 Unity Overview

Our overall integration architecture has four components: a standard term dictionary, a metadata specification for capturing data semantics, an integration algorithm for combining metadata specifications into an integrated view, and a query system for generating and executing multidatabase queries. Unity implements all four components in one software package. The following sections describe each component in detail.

Unity is written in Visual C++ 6 and is intended for deployment on a Windows NT platform. The entire functionality of the integration architecture is built into Unity which displays the concepts in a graphical form. The system is constructed following the Microsoft Document/View architecture, and hence is highly modular.

Although Unity runs on a Windows platform, this does not limit the software from integrating databases run under different operating systems and database managers. All Unity requires is a text file description of a database schema or the ability to automatically connect to a database to retrieve its schema via ODBC, so it can begin the initial integration steps. The integration algorithm itself is C++ code that can be easily ported to different environments.

Unity is considered a multiple-document interface (MDI) application which supports multiple document types. The four document types Unity supports are: a global dictionary (GD) document, an X-Spec document, an integrated view document, and a query document. Any combination of these four types of documents can be open for manipulation in Unity at the same time.

IV.2 The Global Dictionary Editor

The global dictionary editor component of Unity allows a user to create and maintain the standard dictionary of terms. This component of Unity has been used to construct our base dictionary and new terms are continually added as required. For this discussion, the terms standard dictionary and global dictionary are used interchangeably.

Global dictionary (GD) information is stored in a document derived from the Microsoft Foundation Classes (MFC) document class. This document, called `CGDDoc`, is used for serialization, event handling, and view representation. The GD data itself is contained in the class `CGD`, an instance of which is an attribute of `CGDDoc`. The data is stored and retrieved from disk in binary form and is implemented using MFC serialization routines for each class.

In memory, the `CGD` structure is functionally a tree, although it is implemented as a linked-list of node pointers. The `CGD` has a defined root and allows various mechanisms for adding, renaming, and deleting nodes. Other methods include an iterator, which allows the node list to be traversed sequentially, and routines for performing breadth-first and depth-first searches. Display routines graphically display the structure, and search routines allow the GD to be searched by name for a given term or closely matching terms.

Each term in the global dictionary is represented by a single node in the `CGD` structure. The node class, `CGD_node`, has a unique key allowing the node to be identified. The unique key is constructed by combining the semantic name of the node with its definition number in the form: $key = sem\_name + \text{“}-\text{”} + str(def\_num)$. The semantic name of a node is an English word or phrase. The definition number is added by the system to ensure that two instances of the same English word in the dictionary can be distinguished.

For example, the word "order" could appear in two different places in the dictionary. One definition of "order" may be "a request to supply something", whereas a second definition of "order" may be "an instruction or authorization". Since the word "order" has (at least these) two different semantic connotations, it would be represented in the dictionary as two nodes. The first node would have a key of `Order-0` as its definition number would be `0`, and the second node would have a key `Order-1` as its definition number would be `1`. The assignment of a definition number to a given term definition is arbitrary when the term is first added to the global dictionary. However, once a definition number is assigned, the user of the dictionary must choose the correct definition number when using the dictionary term. Not only does the use of a definition number allow the system to uniquely determine a node, it also allows the system to uniquely determine its semantics. If a single node is used for the term "order", it may not be completely obvious what connotation of "order" should be used. This approach eliminates confusion. Each definition or connotation of a word is given its own node and key.

A node also contains a link to its parent node and a list of pointers to its child nodes. It is important to note that the entire GD structure is virtually created by the linking of pointers. All `CGD` nodes are created at run-time in memory. Once a node is created, it is first added to the list of all nodes in the `CGD` structure. The node is then added to the list of child pointers for its parent node and the node's parent link is updated accordingly. This structure gives very good flexibility and allows excellent performance in both searching the tree by traversing tree links or sequentially scanning the entire tree. Finally, a `CGD_node` contains information on its display characteristics, a synonym list of terms which have a similar meaning, and provides methods for adding, deleting, and finding its child nodes. The link between two nodes is defined using a `CGD_link` structure which contains information on the link type (IS-A or HAS-A), and other attributes such as a link weighting which are not currently used. The C++ implementation of these classes can be found in Appendix A.

Updating the global dictionary is done through a GUI (see Figure IV.1). A single `CGDDoc` is displayed in Unity using two different views: a Windows-Explorer, tree-style view on the left-hand side (`CLeftView`), and a graphical representation of the tree structure

on the right-hand side (`CGDView`). These views represent a user's window into the global dictionary structure and manipulation routines.

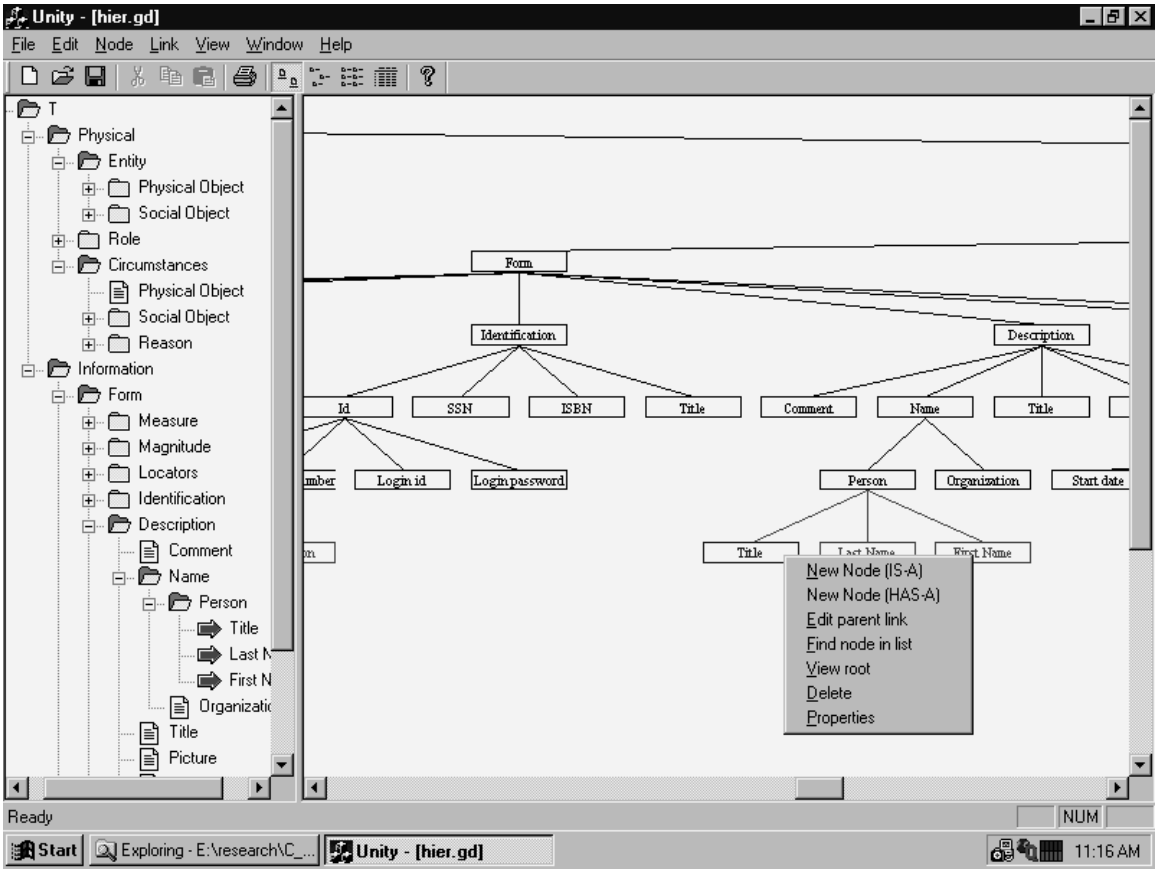

Figure IV.1: Editing a Global Dictionary in Unity

The `CLeftView` class is derived from the `CTreeView` class which represents a tree-structure of items similar to how Windows Explorer displays a directory list. Instead of directories, `CLeftView` displays the nodes of the global dictionary according to their organization in the tree. Open and closed folder icons are used to represent non-child nodes and items related by HAS-A and IS-A links have separate icons as well. The text for each tree item is the node's key. `CLeftView` allows in-place renaming and drag-and-drop movement of nodes. Although some of the functions to add and remove nodes are in the

menu for the main frame, most of the functionality of Unity is accessed by pop-up menus. Pop-up menus appear when a user right-clicks on a given element in the view and are *context-sensitive* to the element that is clicked.

For `CGDDoc`, a pop-up menu is displayed if the user right-clicks on a tree node in `CLeftView` or on a link or node in the graphical view of `CGDView`. These pop-up menus allow the user to perform functions such as adding nodes, editing node/link properties, and finding nodes in both views. Right-clicking on empty space in `CLeftView` displays a pop-up menu to add a new root only if the tree is empty. Right-clicking on empty space in `CGDView` shows a menu to adjust the display properties of the graphical representation of the tree.

The `CGDView` class displays the same global dictionary information as `CLeftView` in a graphical form. The nodes of the GD are displayed as a tree structure on a large, scrollable canvas. The `CGD` class has routines for calculating how to display the tree. The user is unable to move the display nodes, though this is seldom necessary as the tree is displayed in a very organized manner. The user selects nodes and links by left or right clicking on them and they become hilighted in blue. A right click on a node or link displays a pop-up menu. Right-clicking on a link brings up the link pop-up menu which allows a user to view the link properties. The node pop-up menu allows the user to add a new node, edit the parent link properties, find the node in the `CLeftView`, delete the node, or view the node properties. Right-clicking on an open area of the canvas allows the user to change the display properties including the scaling factor of the display, the maximum depth to display, and the type of links to display. Further, a user can have the display show a certain subset of the tree by right-clicking on the node and selecting `View as Root`. This makes the selected node the root of the display. Selecting `View All` from the display's pop-up menu re-displays the entire tree from the normal root.

A selected node in one view can be found in another view by selecting the appropriate pop-up menu item. Other display classes are also required to allow the user to edit a node's properties (`CNodeDialog`), a link's properties (`CLinkDlg`), and the graphical view properties (`CViewProp`).

In summary, the standard dictionary is a tree of concepts related by either IS-A or HAS-A links. We define a base dictionary but new terms can be added as required. Unity

contains a global dictionary editor which allows the user to add, remove, and change terms. Each dictionary term consists of a standard name, definition number, and text definition along with synonym and system name information.

IV.3 The X-Spec Editor

Unity allows a user to quickly and easily construct an X-Spec for a given database. Unity contains a specification editor that parses existing relational schema and formats the schema information into an X-Spec. The software allows the user to modify the X-Spec to include information that may not be electronically stored such as relationships, foreign key constraints, categorization fields and values, and other undocumented data relationships. More importantly, the specification editor requires the user to match each field and table name in a relational schema to one or more terms in the standard dictionary to form a semantic name for the element. The semantic name is intended to capture the semantics of the schema element using the standardized terms.

Before describing how X-Specs are created and modified in Unity, it is necessary to describe the data structure that they represent. A relational schema is parsed and translated into a metadata document (`CMDDoc`) which contains the metadata information (`CMDSource`) that describes the relational schema and its component metadata. Although a metadata document is not one of the four major documents handled by Unity, source metadata information can be loaded, viewed, and saved in the process of creating an X-Spec to describe the data source. Classes are used to display field properties (`CMDFldPropDlg`), table properties (`CMDTblPropDlg`), join properties (`CMDJoinDlg`), keys (`CMDKeysDlg`), and overall source properties (`CMDSrcPropDlg`). The C++ definitions for these data classes are in Appendix B.

Similar to the construction of global dictionary documents, X-Specs are contained in specification documents (`CSpecDoc`) which contain the specification class (`CSpec`). Specification documents are loaded and saved using MFC serialization routines. An X-Spec is represented by a `CSpec` class which contains all the information of the metadata class plus additional information such as semantic names for the elements. A `CSpecDoc` has two

views (see Figure IV.2). A tree view on the left-hand size of the screen (`CSpecLView`) and a metadata document view (`CSpecMDView`) on the right-hand side of the screen.

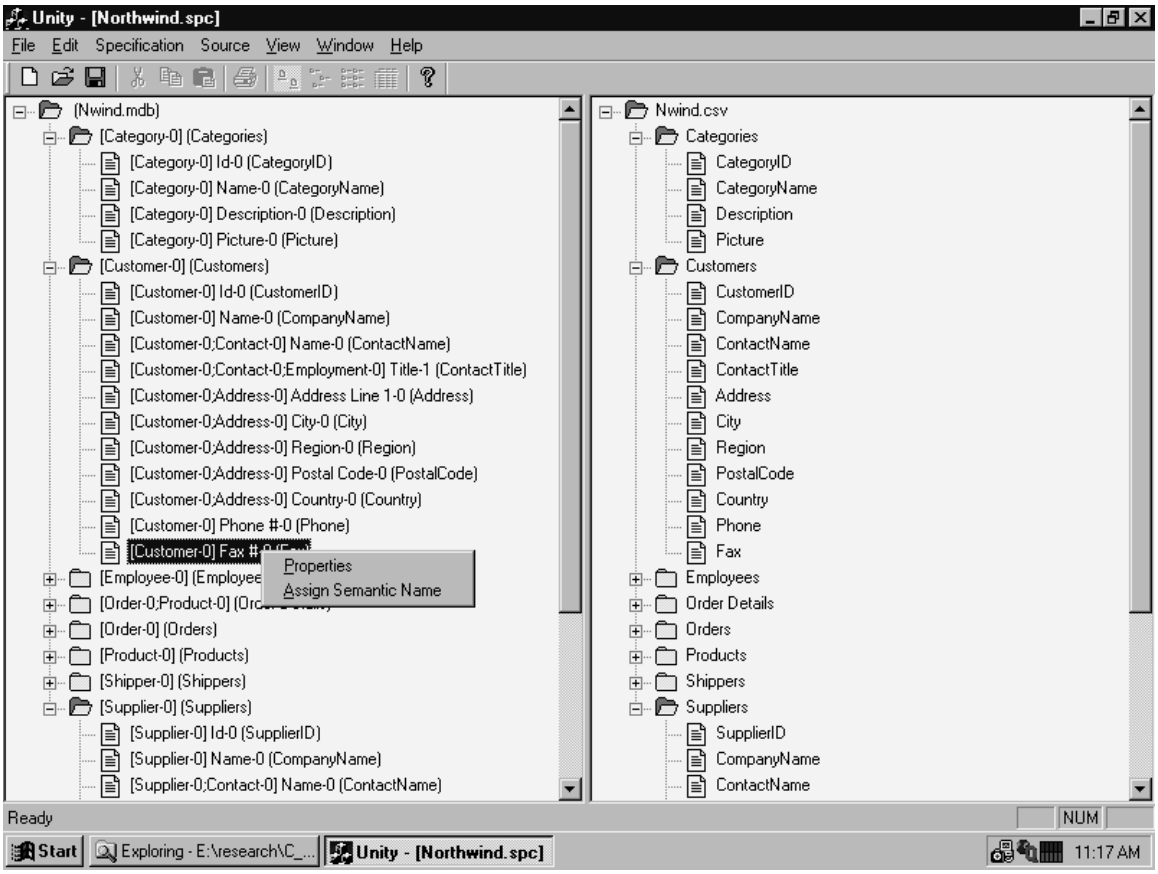

Figure IV.2: Editing an X-Spec in Unity

The `CSpecMDView` is used to load, save, and view metadata sources in a tree form. This view manages a dynamic connection to a `CMDDoc` document. Using the menu item, `Add all to Spec.`, a user is able to add the entire metadata source in the `CSpecMDView` to the current X-Spec. The system copies over the corresponding metadata information and constructs the appropriate specification structures.

Similar to the global dictionary tree view, `CSpecLView` provides a tree-view of the current X-Spec. Since an X-Spec describes a data source there will be typically only 3

"levels" to this view. The root node is the root of the specification and contains information on the entire specification. The direct children of the root node are specification tables (`CSpecTable`) which provide metadata information on data source tables. Nodes at depth 3 are children of a `CSpecTable` and represent specification fields (`CSpecField`) which map to data source fields. These classes are presented in Appendix C.

The main function of the specification editor is to allow the user to assign semantic names to fields and tables in the X-Spec. The first step is to associate a standardized dictionary with the X-Spec by selecting the `Set GD` menu item and loading a saved global dictionary file from disk. Once an X-Spec is associated with a GD, this GD is loaded into memory every time the specification is loaded and allows the user to look-up terms from the dictionary in the construction of a semantic name.

The construction of a semantic name is performed by the class `CSnameDlg`. This dialog class allows the user to search for a dictionary term and to combine dictionary terms into a semantic name. As mentioned previously, a semantic name consists of one or more dictionary terms, and typically consists of context terms and possibly a concept name. A context term simply provides a context for describing knowledge. A semantic name with only context terms is used to describe a table in an X-Spec. A concept name is a single term typically used in describing a field of a table. It is considered an end point. A concept name is a "final context" and cannot be broken down any further. A semantic name for a field consists of the context, which is generally the context of its corresponding table (although not always), and a concept name for describing the semantics of the field itself. Thus, the specification editor makes a distinction between assigning a semantic name to a field and to a table. Once semantic names are assigned to all fields and tables in an X-Spec, sufficient information is present for the X-Spec to be integrated by the integration algorithm.

The X-Spec component contains dialog classes to edit the database scope and global properties (`CSPPropDlg`), field (`CSPFldPropDlg`) and table (`CSPTblPropDlg`) properties, keys (`CSpecKeysDlg`), and joins (`CSpecJoinDlg`).

In summary, an X-Spec contains database schema and metadata information on a given data source. Unity allows a user to automatically extract metadata information from a data source and insert it into an X-Spec. Unity also provides an easy mechanism

for associating a semantic name with a database element by combining terms from a global dictionary, and for specifying key and join information.

IV.4 The Integration Algorithm

The integration algorithm is a straightforward term matching algorithm, and its C++ implementation is provided in Appendix D. The integration result of combining one or more X-Specs is stored in a schema integration document (`CSchDoc`). The `CSchDoc` contains an instance of the `CSchema` class which actually contains the schema data. Similar to the `CSpecDoc`, the schema document has a left-hand and right-hand view (see Figure IV.3). The left-hand view (`CSchLView`) displays the current schema document in tree-form. The right-hand view (`CSchSPView`) is used to load, save, and integrate previously constructed X-Specs. When an X-Spec is loaded into `CSchSPView`, it can be integrated into the current schema by selecting the `Add all to Schema` menu item. This will perform the integration algorithm previously described and update the schema view on the left-hand side. Subsequently, the schema may be queried by the user to generate transactions against the individual systems.

The actual structure of `CSchema` is more related to the global dictionary tree-structure than the X-Spec structure. This is because `CSchema` is organized by contexts which can be of arbitrary depth since there can be an arbitrary number of context terms in a semantic name. Correspondingly, an X-Spec only has a depth of 3 (source, tables, and fields), although the knowledge contained in these structures may have deeply nested contexts. The `CSchema` classes are defined in Appendix E.

The mappings for a semantic name in the schema are displayed by right-clicking on an item in `CSchLView` and selecting `Properties` from the pop-up menu. In the properties dialog (`CSchnPropDlg`), default display and query properties can also be set. Further, the menu item `Schema Contexts` under the `Schema` menu displays the database names and organizations associated with all X-Specs integrated into the schema.

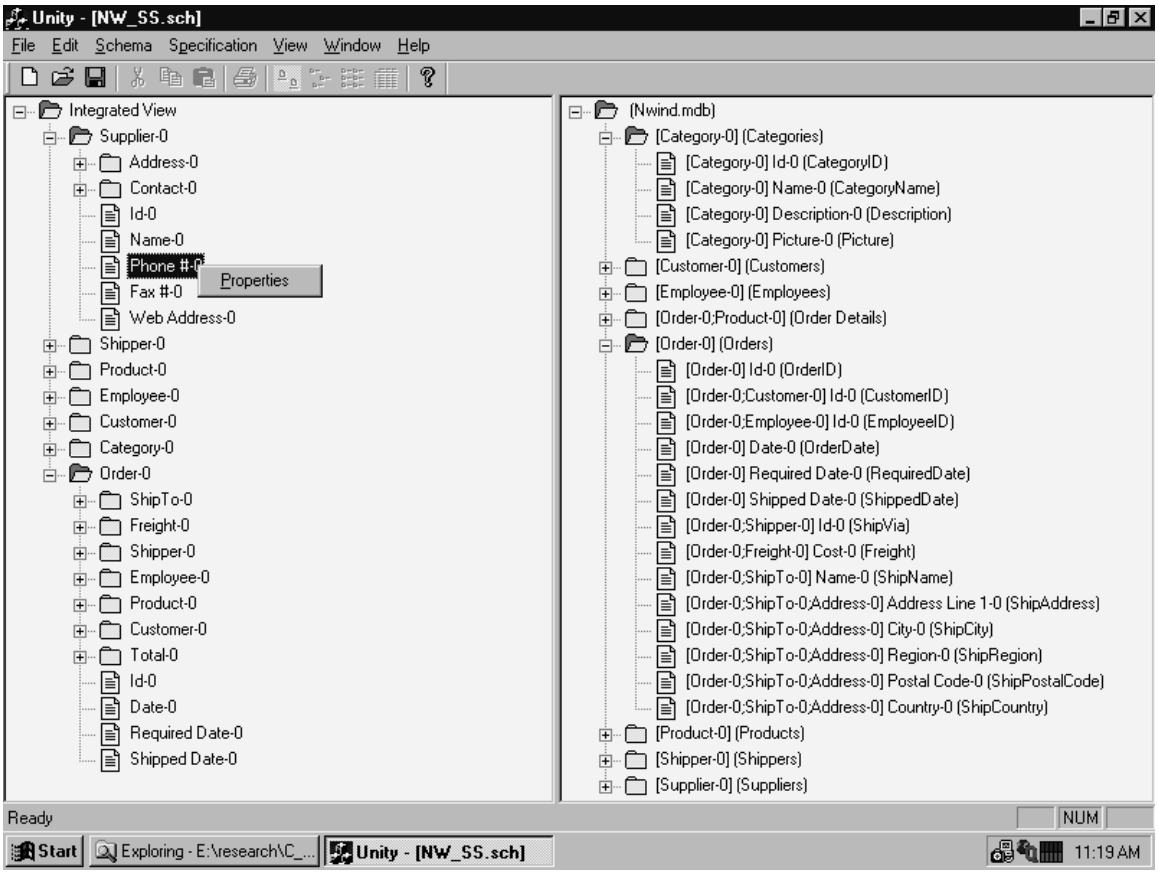

Figure IV.3: Integrating a Schema in Unity

IV.5 The Query Processor

Integrating concepts into an integrated view has little benefit without an associated query system to query the database using that view. We have developed algorithms [56] for querying the integrated view and have implemented them in Unity.

The query system performs a mapping from semantic names in the integrated view to structural names in the underlying data sources. Given a set of semantic names to access, the query system generates a SQL query with the appropriate join conditions. The information required to select the appropriate fields and tables is present in X-Specs. X-Specs also contain information on the keys (`CKey`) and joins (`CJoin`) for the tables. This

information allows the query processor to select appropriate joins as required. Joins within a database are calculated by constructing a path matrix of all joins in the database and constructing an appropriate join tree to combine the required tables. The C++ definitions for these and other related query classes are in Appendix F.

A query is stored in a query document class (`CQryDoc`) and displayed in a frame (`CQryFrame`). The actual data is stored in the `CQuery` class. The frame is divided into two sides (see Figure IV.4). The left-hand side (`CQryLView`) contains semantic names selected by the user for inclusion in the query. The right-hand side (`CQrySchView`) displays the integrated view on which the query is posed.

When a query is first created, the integrated view it uses must be defined using the `Load` menu item under the `Schema` menu. After a schema is loaded into the right-hand side, the user can add or remove semantic names for the query. Since a query is structured as a tree, there are 3 add/remove operations: add/remove a single semantic name (`Add/Remove Item`), add/remove a branch (subtree) of the tree (`Add/Remove Branch`), or add/remove the entire tree (`Add/Remove Tree`). These functions can either be accessed by using the menu items under the `Query` and `Schema` menus, or by selecting the appropriate tree node, right-clicking, and choosing the desired function out of the pop-up menu. The system ensures that a semantic name cannot be added to the query twice. Using the pop-up menu, a user can edit the query properties (`CQryNProp`) for each semantic name including the display width and format.

To filter the databases accessed by a query, the user selects `Set Scope`, under the `Query` menu and sets their target locations (`CQryCtx`). Result ordering information is modified using the dialog `CQryOrderBy` opened with the `Ordering` menu item. Selection criteria is created in the `CQryCriteria` dialog using the `Set Criteria` menu item.

Once the user has selected the appropriate semantic names, the SQL queries used to access the individual data sources can be displayed by selecting the `Show SQL` menu item under the `Query` menu. This displays a pop-up dialog box (`CQrySQLDlg`) which displays the generated SQL for each data source.

To execute a query, the user selects the `Execute` menu item under the `Query` menu. The query processor then generates the SQL query, executes it for each data source using

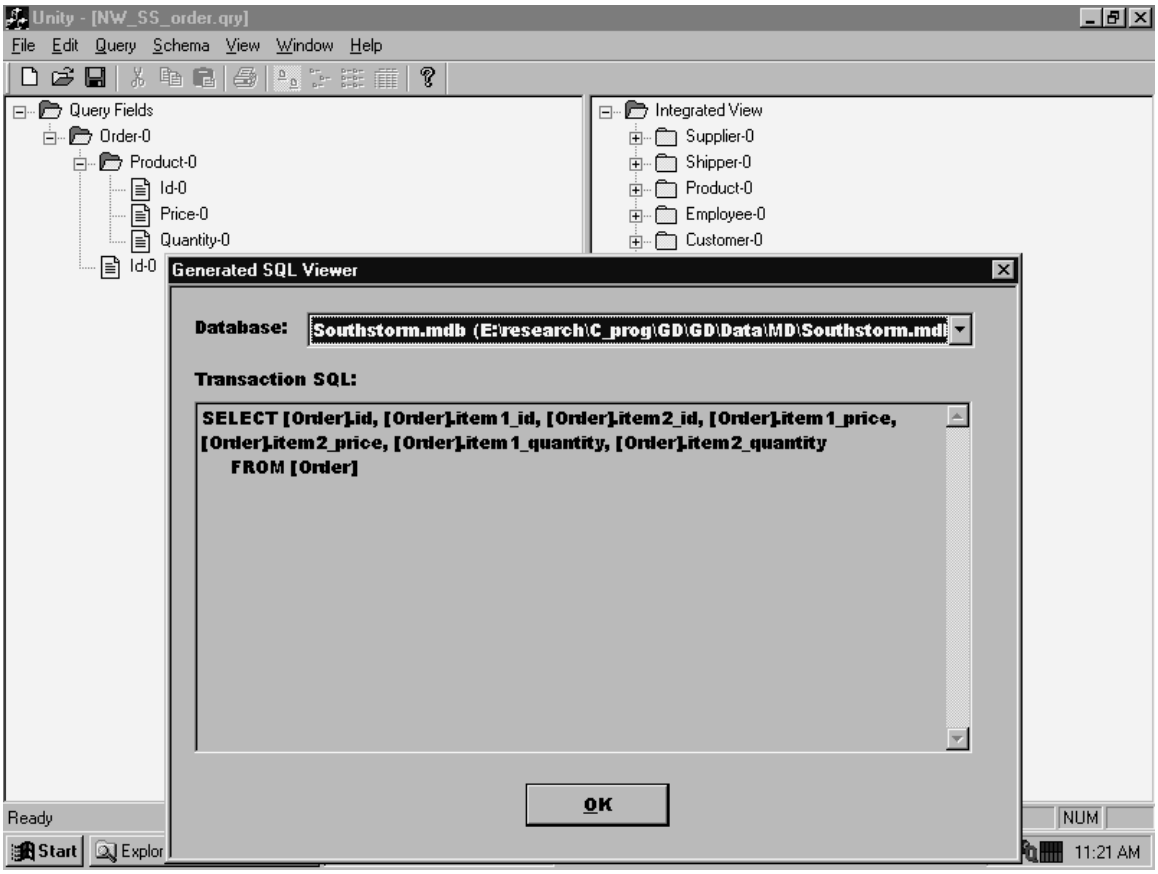

Figure IV.4: Querying in Unity

ODBC, and normalizes and integrates the results as required. Results for each subquery are stored in a record set structure (`CRecSet`) as they are returned via ODBC. Integration of subquery results are stored in a second record set structure (`CResultSet`). Finally, the results are then displayed to the user in a result dialog (`CQryResDlg`).

The algorithms for generating the SQL queries and performing normalization and integration are implemented in the `CSpec` and `CQuery` classes. Implementation of the join path matrix and its associated algorithms are relatively straightforward. However, implementing query-time normalization is more complex. Query-time normalization is achieved by constructing dependency trees (`CDepTree`) for each table of a data source that requires

normalization. These dependency trees are built using information in X-Specs. Dependency tree construction currently occurs before query execution, but may be pre-compiled in the future to avoid repeating the procedure for every query execution.

Using normalized dependency trees, the system is able to construct normalized result sets. These result sets contain a subset of the fields retrieved by the original query. That is, one row returned from the database (in `CRecSet`) becomes multiple normalized rows in the query result (in `CResultSet`).

Overall, the query system is continually evolving and is an active research area. The system currently supports union of results across databases and automatic SQL generation of semantic queries. The implementation and efficiency of Unity will be enhanced as the query algorithms are refined. An implementation of the algorithms for joins across databases and multidatabase conflict resolution are ongoing improvements being implemented in Unity.

Chapter V

Architecture Contributions and Discussion

V.1 Architecture Contributions

The integration architecture contributes several new ideas:

- The definition of a hierarchical, standard term dictionary that can be used across industries and organizations.
- A system for representing, in XML, entire database schemas using the standard dictionary and metadata about the data sources (X-Specs).
- An integration algorithm that combines X-Specs into a structurally-neutral integrated view that hides the structural, organizational, and distribution characteristics of the integrated systems.
- A query processor capable of translating from high-level semantic queries generated by the user to SQL queries extracting relevant information (views) from each data source. Further, generated local views are automatically combined by resolving structural conflicts, normalizing results, and applying joins across databases.

The importance of the work is the unification of two different approaches to a similar problem. By combining the work done in defining industry standards for data exchange with protocols for integrating database systems, we have constructed an integration architecture that utilizes a standard protocol (XML) to exchange data semantics on database systems and thereby allows them to interoperate. The integration architecture goes beyond

simple data communications and captures the semantics of the data source itself. This allows future data sources to be added into an organization with minimal integration effort. It also allows databases connected on the WWW to interoperate and provides exciting new opportunities for the integration of knowledge across systems.

V.2 Implementation Applications

V.2.1 Multidatabases and Data Warehouses

Multidatabases and data warehouses provide an integrated view of data. These architectures combine and summarize data from numerous data sources into a single, organizational view. Thus, creation of a multidatabase and a data warehouse requires that the data in the underlying sources be understood, interpreted, and manipulated. The approach of constructing X-Specs to capture data semantics greatly simplifies this task by providing a formalized method for capturing data semantics.

An X-Spec is used to capture the semantics of each data source. Then, when the data from this source is integrated into the data warehouse, its exact semantics are known, and it can be more readily combined with other data sources in the system. As the number of data sources grows, the data semantics present in the X-Specs provide the data warehouse designer with an easily accessible specification of all organizational data, and they can be integrated using the integration algorithm to display the data available.

V.2.2 World-Wide Web

Integrating data sources automatically would have a major impact on how the World-Wide Web is used. The major limitation in the use of the WWW, besides the limited bandwidth, is in the inability to find and integrate the extensive databases of information that exist. When a user accesses the Web searching for something, they are often required to access many different web sites and systems, and manually pull together the information presented to them. The task of finding, filtering, and integrating data consumes the majority of the time, when all the user really requires is the information. For example, when the user wishes to purchase a product on-line and wants the best price, they must visit the

appropriate web sites and "comparison shop". It would be useful if the user's web browser could do the comparison shopping for them.

Our architecture supports these types of queries. To achieve this, each web site would specify their database using an X-spec. The client's browser would connect to the integration site to pose queries on data sources that it has integrated. A portal like Yahoo could combine data sources together as a central site for the user to gather information from. Even more exciting would be a distributed version of the architecture, where there is no central site and the user's browser is responsible for the necessary translation and management. When the user wishes to purchase an item, the browser downloads the X-Specs from the on-line stores, integrates them using the standardized dictionary, and then allows the user to query all databases at once through the "global view of web sites" that is constructed. Obviously, the integration itself is complex, but a system which achieves automatic integration of data sources would have a major impact on how the Web is used and delivered.

V.3 Integration Validity

The validity of the integration result depends directly on our use and construction of the standard dictionary. By defining a standard term dictionary and basing our integration architecture on the matching of those terms, we are effectively assuming that no naming conflicts are present in the system. If naming conflicts do occur, caused by the incorrect construction of semantic names, the integration procedure yields less desirable results. Since identical concepts are identified solely by semantic name, incorrect or missed matchings can only occur if the semantic names are not properly assigned.

Standardization of structure and naming conventions are used to provide interoperability and communication between systems in industry. Thus, assuming away naming conflicts has been applied in other architectures. Our assertion is that naming conflicts must be assumed away in some fashion for interoperability to be successful. Without a standard set of terms or names to communicate knowledge, knowledge cannot be integrated or exchanged because its semantics are not known. Names represent a standard for exchanging

semantics and are required in some form for any communication to be effective. Thus, by accepting a standard dictionary, schema, or set of XML tags, a system assumes away the naming problem by accepting a lexical semantic framework for the expression of data semantics. Whether this framework is defined based on tags or names as in our approach and XML, or structure and position of data elements as in EDI, a language framework is required to exchange knowledge between systems, similar to our human acceptance of spoken languages to facilitate communication.

Although our architecture assumes away the naming problem, it does not impose a structural organization on the concepts represented by the names. XML tags, BizTalk schemas, and EDI documents in addition to standardizing concept names also standardize their structure, organization, and relationships. Thus, these architectures also assume away all structural conflicts as well. This results in inflexible standards which have limited applicability across domains.

Accepting a linguistic framework to prevent naming conflicts is required for integration, but imposing a structural organization to concepts is unnecessary. Our architecture identifies similar concepts by name regardless of their physical or logical representations in the individual data sources. Thus, concept knowledge from data sources is combined even though the actual representation of the data may be very different.

The integration result is a hierarchy of contexts and concepts which implies no particular physical representation. Users access data sources through semantic names which consist of a hierarchical organization of those contexts and concepts. The physical representation of the concepts is irrelevant to the user. Thus, by not imposing structural constraints on concept representation, knowledge from systems may be combined regardless of data representation characteristics, and the user is provided with only the relevant information.

In summary, our integration is valid because it correctly combines database schemas into an integrated view given the assumption of no naming conflicts. The architecture avoids naming conflicts by developing and using a standard dictionary of terms and combining them appropriately into context and concept information to express schema element semantics. Since the semantic names constructed are known to represent the same concept

if their names match, integration of concepts across schemas is possible simply by matching the semantic names, regardless of their implementation or physical structure. The automatic construction of the integrated view isolates the user from the complexities of data distribution, organization, structure, and from local naming conventions.

V.4 Automatic Conflict Resolution

The integration architecture performs automatic conflict resolution. The capture process avoids naming conflicts by using a standard term dictionary. Physical and logical access transparency is provided to the user by transforming schema element semantics into semantic names which are combined into a structurally-neutral concept hierarchy. The query processor then uses X-Spec information to map from the context view to structural queries, and in the process of this mapping, structural differences in schema and data representation are resolved. Based on previous work [50] classifying conflicts present in the relational model, the types of conflicts resolved by the architecture are summarized in Figure V.1. Conflicts resolved by the architecture can be attributed to four basic features:

- Using a standard dictionary to build semantic names.
- Constructing a structurally-neutral integrated view from semantic names and mapping semantic queries to structural query expressions.
- Concept promotion and manipulation in the integrated view.
- Explicitly specifying data contexts and mappings and executing conversion functions to exchange data between different contexts.

The standard dictionary foundation of the architecture resolves the most basic, yet hardest to solve, naming conflicts. Using a standard dictionary and building semantic names to capture schema element semantics solves the table naming conflict and the attribute naming conflict because contexts (tables) and concepts (attributes) will not be integrated unless they have the same semantics. Thus, it also implicitly resolves semantic conflicts related to naming. Fundamentally, the most basic semantic conflict of what a schema element represents is resolved by agreeing on the proper names to represent concept semantics.

Schema Conflicts		
Conflict Type	**Description**	**Resolution**
Table Name Conflicts	Using different names for equivalent tables or the same name for different tables	Using standard dictionary to build semantic names
Table Structure Conflicts	One table contains more attributes than another table with equivalent concepts	Mapping from structurally-neutral integrated view to structural queries
Table Constraint Conflicts	Incompatible key and update constraints	Update procedures not currently considered.
Multiple Table Conflicts	Using different numbers of tables to store information	Mapping from structurally-neutral integrated view to structural queries
Attribute Name Conflicts	Using different names for equivalent attributes or the same name for different attributes	Using standard dictionary to build semantic names
Multiple Field Conflicts	Representing a concept using more fields in one database than another	Mapping to structurally-neutral integrated view and promoting concepts to contexts
Table versus Attribute Conflict	Representing a concept as a table in one database and as a field in another	Mapping to structurally-neutral integrated view and promoting concepts to contexts
Data Conflicts		
Conflict Type	**Description**	**Resolution**
Different Data Representations	Using different words, strings, and codes for same data	Explicitly specifying mapping of concept to data representation in X-Spec and applying conversion functions.
Data Type Inconsistencies	Fields with different data types (integer, string, *etc.*)	For query purposes only, fields are converted to strings for display. Updates not considered.
Different Units and Precisions	Numerical data represented using different units, precision, or scaling factors	Conversion functions applied during result integration

Figure V.1: Conflicts Resolved by Architecture

Structural conflicts such as tables having different numbers of attributes or multiple tables representing a concept are resolved by constructing the structurally-neutral integrated view. If a table in one database does not have all the identical fields as a table in another database with which it is integrated, the query processor only queries and extracts the fields which are present. Multiple tables storing fields about the same context are seemlessly combined into the integrated view by virtue of related semantic names. If two tables have fields describing the same context, then they will have identical context portions of the semantic name. These context portions are then matched in the integrated view, and it appears to the user that the fields describing the context are not actually structurally distributed between tables. At query-time, the query processor extracts the fields using mappings and applies joins to connect the tables into a single context.

As discussed in Section III.4.1, conflicts are possible within the integrated view only when two semantic names have the same terms and one is a context and the other is a concept. In this case, the context term is more general than the concept term which implies that in one database the idea is more detailed than in another. These conflicts at the integrated view level often arise because of conflicts in the relational model. For example, a concept may be represented as a field in one database and as multiple fields in another. This typically results in semantic names with identical terms but one is a concept and the other is a context. The solution to this problem is *promotion*, where a new context is created which contains all the relevant attributes.

The final layer of conflict resolution is at the data-item layer. Data level conflicts include differences in types, sizes, precision, units, and scaling factors. These conflicts are resolved by defining functions which convert from one context to another and by formally expressing context semantics. For example, if one database uses “F” for Female and “M” for Male, this mapping is stored in the X-Spec. If another database, uses “1” and “0”, the data is converted to “F” and “M” by applying a transformation function. Similar conversion functions have been previously proposed [39, 90].

V.5 Architecture Discussion

The integration architecture is a combination of standardization and intelligent conflict resolution algorithms. We believe that some level of standardization is required to achieve more automatic integration. Specifically, by accepting a standard term dictionary for describing schema element semantics and thus avoiding naming conflicts, structural conflicts are automatically resolved. We approach the integration problem from a different perspective than mediator systems. Our goal is to separate the specification of database semantics from the integration procedure, and then apply automatic integration procedures to combine semantic specifications and resolve conflicts. The combination of industrial standards such as XML and standardized dictionaries with research algorithms in application to the schema integration problem is unique.

The key benefit of the architecture is that the integration of data sources is automatic once the capture processes are completed. This is a substantial improvement over systems [69] which require the user to query all databases by structure. Our work is unique because it automatically produces an integrated view from data source specifications developed independently of other data sources and the global view itself. The architecture uses standardization to achieve more automatic integration but does not force a structural representation on the data which allows for greater flexibility and for existing systems to be unmodified during integration. Thus, the system preserves full autonomy of all data sources and no translational or wrapper software is required.

The major challenge inherent in the architecture is the definition of the standard dictionary. Although defining terms to represent concepts is challenging, it is not without precedent. Industrial systems such as EDI, XML, and BizTalk all rely on the acceptance of standardized formats. Our architecture is even less restrictive as only names are standardized not structure and organization. Further, common ideas such as customers, orders, names, keys, identifiers, and addresses are well understood and easily mapped into a standard dictionary. In addition, even if a total standard is not achievable across the whole Internet, it is still possible to define localized standards. The architecture allows an organization to define its own dictionary. As long as the standard dictionary is conformed to

within that domain, integration is possible. This allows easy integration of data sources within an organization. However, the ultimate goal is the definition of a standard dictionary applicable across all domains not just certain industries and environments as targeted by EDI, BizTalk, and E-commerce portals. Acceptance of standardization is a benefit, but it is not a common practice in the database community as it is difficult to acknowledge that some problems may require standardization to be solved.

Since the integrated view is constructed as needed with no designer input, challenges arise in ensuring correct integrations. First, the architecture has no built-in mechanism for validating the assignment of semantic names. If a semantic name does not correctly capture the semantics of the schema element, it may be poorly integrated into the integrated view. Naming problems result from either poor conformance to the standard or inadequate construction of the X-Spec. Either problem can be resolved by re-examining and updating information in the X-Spec and re-integrating.

Chapter VI

Integration Examples

In this section, we briefly describe three simple examples and show how the architecture achieves the necessary integration. These examples provide a good overview of how Unity is used in practice but many details are omitted to ensure a clear presentation of the key concepts.

VI.1 Combining Two Order Databases

Consider ABC Company which stores an order database consisting of orders,

Order(Id,Customer,Total_amount), and items,

Order_item(Order_num,Item_id,Quantity,Price,Amount), tables.

The first step is to construct an X-Spec describing the database using the specification editor. First, we extract the metadata information from the database schema including table and field names, types, sizes, keys, and relationships. This information is added to the X-Spec. Now, the X-Spec designer uses terms from the standard dictionary to describe each table and field. The *Order* table has a semantic name of `[Order]`, and its id field is `[Order] Id`. Similarly, the *Order_item* table has a name `[Order;Product]` as it contains items (or products) for orders. After each field and table is assigned a semantic name, an X-Spec is produced as shown in Figure VI.1.

The X-Spec is then passed through the integration algorithm to produce the output shown in Figure VI.2. Notice that although only one data source is integrated, we already

```
<?xml version="1.0" ?>
<Schema
        name = "order_xspec.xml"
        xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:dt="urn:schemas-microsoft-com:datatypes">

<ElementType name="[Order]" sys_name = "Order" sys_type="Table">
        <element type = "[Order] Id" sys_name = "Id" sys_type = "Field"/>
        <element type = "[Order] Customer" sys_name = "Customer" sys_type = "Field"/>
        <element type = "[Order] Total Amount" sys_name = "Total_amount" sys_type = "Field"/>
</ElementType>

<ElementType name="[Order;Product]" sys_name = "Order_item" sys_type = "Table">
        <element type = "[Order] Id" sys_name = "Order_num" sys_type = "Field"/>
        <element type = "[Order;Product] Id" sys_name = "Item_id" sys_type = "Field"/>
        <element type = "[Order;Product] Quantity" sys_name = "Quantity" sys_type = "Field/>
        <element type = "[Order;Product] Price" sys_name = "Price" sys_type = "Field"/>
        <element type = "[Order;Product] Amount" sys_name = "Amount" sys_type = "Field"/>
</ElementType>
</Schema>
```

Figure VI.1: Order X-Spec

Integrated View Term	Data Source Mappings (not visible to user)
V (view root)	**N/A**
- [Order]	ABC.Order
- Id	ABC.Order.Id, ABC.Order_item.Order_num
- Customer	ABC.Order.Customer
- Total Amount	ABC.Order.Total_amount
- [Product]	ABC.Order_item
- Id	ABC.Order_item.Item_id
- Quantity	ABC.Order_item.Quantity
- Price	ABC.Order_item.Price
- Amount	ABC.Order_item.Amount

Figure VI.2: Integrated View

achieve some advantages. First, the user is no longer responsible for generating SQL using system names and joins. The structure and location of the database is hidden as the user formulates queries by picking which fields they wish to see in the result. The system is responsible for generating the SQL and join conditions, mapping from semantic to system names, and communicating with the database. Second, full autonomy of the database is preserved as the system appears to be just another client issuing database transactions.

Now, if ABC Company buys XYZ company, a similar business, and we assume that XYZ company has a similar database for storing their orders, it would be useful if the two could be integrated easily. Initially, we do not want to change the database at either

company. We would prefer both company databases to function unchanged, but we need to enable management to see a global-view of all orders at both companies as required.

We already have an X-Spec for the ABC database which we do not have to change. We must generate a new X-Spec for the XYZ database. Since both databases store almost identical data, the X-Spec will be very similar for the XYZ database except the system table and field names may be different. Integrating the XYZ X-Spec with the ABC X-Spec produces the exact same integrated view. However, in this case when a user issues queries against the integrated view, they are actually issuing queries against both databases. The query processor at the integration site divides the user query into two separate transactions at the component databases and then integrates the results that they return. From the user's perspective, they cannot tell that they were accessing two different databases, each of which is operating as if they were not participating in a MDBS.

Obviously, this is a very simple example but the algorithm scales to any number of concepts and databases. By proper assignment of semantic names, the users are isolated from the database details and full database autonomy is preserved. Integration is performed on an as-needed basis and does not have to be re-done when a new database is added to the integrated view.

VI.2 Comparison Shopping on the WWW

The ultimate goal is a distributed version of the architecture where a user's browser is responsible for performing the integration. The browser would contain the standard dictionary and the integration algorithm and receive from web sites X-Specs to integrate and present to the user. Such a system allows knowledge to be combined across web sites even though they were not originally intended to work together and simplifies the problem of searching the web for the required information.

As Internet commerce, or E-Commerce, becomes more prevalent, an outstanding issue is comparison shopping. As in the off-line world, users would like to visit different "stores" to compare prices and options. Currently, the user is either forced to go to each site individually to search for the required item or attempt to go through dedicated comparison

shopping sites which may or may not contain all the stores they are interested in and present the information in the desired form. Our architecture allows a user to comparison shop only the sites that they require and query the information in a form with which they are comfortable.

The integration of web sites begins at each individual web site. For this discussion, consider two web sites that sell books. The first site, called **Books-for-Less**, stores its book catalog using the structure: $Book(ISBN, Title, Author, Publisher, Price)$. The second site, called **Cheap Books**, stores its book database with the structure:

$Book(ISBN, Author\_id, Publisher\_id, Title, Price, Description)$,

$Author(Id, Name)$, and $Publisher(Id, Name)$.

Each web site independently creates an X-Spec to describe their data without any knowledge of the X-Specs at other sites. Unity is used to parse the schema at a data source and assign semantic names to the schema elements.

Constructing the X-Spec for the **Books-for-Less** database is the easier of the two. First, the X-Spec designer imports the database schema into the X-Spec to get the system names, types, and sizes. The next step is to assign semantic names to the table and fields of the database. The semantic name for the table name *Book* will be `[Book]` as it is describing a book context. The name for the field *ISBN* is `[Book] ISBN` as it describes the concept of an ISBN for a book. Similarly the fields, *Title* and *Price* have semantic names `[Book] Title` and `[Book] Price`. The name for the *Author* field is not quite as straightforward. Although technically it would be correct to give it the name `[Book] Author`, what this name really assumes is that the concept of *Author* is defaulting to the author's name. Thus, it makes more sense to represent this explicitly as `[Book;Author] Name`. Likewise, the *Publisher* field is called `[Book;Publisher] Name`. The final X-Spec is in Figure VI.3.

The database schema for **Cheap Books** is more complex because the database is more normalized and uses relationships and joins to combine the information appropriately. However, since the key notion of the database is about books, the fundamental concept is `[Book]`, which is the semantic name of the *Book* table. The semantic names of the rest of the fields in the *Book* table are straightforward except for *Author_id* and *Publisher_id* which have semantic names of `[Book;Author] Id` and `[Book;Publisher] Id`, respectively. We

```
<?xml version="1.0" ?>
<Schema
        name = "Books-for-Less.xml"
        xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:dt="urn:schemas-microsoft-com:datatypes">

<ElementType name="[Book]" sys_name = "Book" sys_type="Table">
        <element type = "[Book] ISBN" sys_name = "ISBN" sys_type = "Field"/>
        <element type = "[Book] Title" sys_name = "Title" sys_type = "Field"/>
        <element type = "[Book] Price" sys_name = "Price" sys_type = "Field"/>
        <element type = "[Book;Author] Name" sys_name = "Author" sys_type = "Field"/>
        <element type = "[Book;Publisher] Name" sys_name = "Publisher" sys_type = "Field"/>
</ElementType>
</Schema>
```

Figure VI.3: Books-for-Less X-Spec

have a choice when assigning the semantic name of the *Author* table. Technically, the name `[Author]` is correct because it describes authors. However, it may be more beneficial to assign a semantic name of `[Book;Author]` meaning that the context is authors of books. The difference between the two semantic name possibilities for *Author* relate to how the data is queried. If author information is always presented as related to books, then `[Book;Author]` is the better semantic name as it makes this relationship more apparent to the user. However, in general, the semantic name `[Author]` would be used as author information can exist without relation to book information and may be queried separately. The fields *Id* and *Name* in the *Author* table have names `[Book;Author] Id`, and `[Book;Author] Name`. Similarly the *Publisher* table has a name `[Book;Publisher]`, and fields `[Book;Publisher] Id` and `[Book;Publisher] Name`. The final X-Spec is shown in Figure VI.4.

The reason for representing the *Author* and *Publisher* tables with the semantic names of `[Book;Author]` and `[Book;Publisher]` becomes obvious when performing the integration. At this point, each web site has independently expressed their database using an X-Spec. When the user wants to query the databases for the best book price, the user's browser connects to the web site, logs in, and downloads the X-Specs from each site. Combining the X-Specs produces the integrated view in Figure VI.5.

The information from both databases has been seemlessly combined even though the data representation was quite different. The binding element was the choice of semantic names which allow the integration algorithm to relate concepts despite their representation. Selecting the semantic names `[Author]` and `[Publisher]` for the *Author* and *Publisher*

```
<?xml version="1.0" ?>
<Schema
        name = "Cheap_Books.xml"
        xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:dt="urn:schemas-microsoft-com:datatypes">

<ElementType name="[Book]" sys_name = "Book" sys_type="Table">
        <element type = "[Book] ISBN" sys_name = "ISBN" sys_type = "Field"/>
        <element type = "[Book] Title" sys_name = "Title" sys_type = "Field"/>
        <element type = "[Book] Price" sys_name = "Price" sys_type = "Field"/>
        <element type = "[Book] Description" sys_name = "Description" sys_type = "Field"/>
        <element type = "[Book;Author] Id" sys_name = "Author_id" sys_type = "Field"/>
        <element type = "[Book;Publisher] Id" sys_name = "Publisher_id" sys_type = "Field"/>
</ElementType>

<ElementType name="[Book;Author]" sys_name = "Author" sys_type="Table">
        <element type = "[Book;Author] Id  sys_name = "Id" sys_type = "Field""/>
        <element type = "[Book;Author] Name"  sys_name = "Name" sys_type = "Field"/>
</ElementType>

<ElementType name="[Book;Publisher]" sys_name = "Publisher" sys_type="Table">
        <element type = "[Book;Publisher] Id"  sys_name = "Id" sys_type = "Field"/>
        <element type = "[Book;Publisher] Name"  sys_name = "Name" sys_type = "Field"/>
</ElementType>
</Schema>
```

Figure VI.4: Cheap Books X-Spec

Global View Term	Data Source Mappings (not visible to user)
V (view root)	**N/A**
- [Book]	CB.Book, BfL.Book
- ISBN	CB.Book.ISBN, BfL.Book.ISBN
- Title	CB.Book.Title, BfL.Book.Title
- Price	CB.Book.Price, BfL.Book.Price
- Description	CB.Book.Description
- [Author]	CB.Author
- Id	CB.Book.Author_id, CB.Author.Id
- Name	CB.Author.Name, BfL.Book.Author
- [Publisher]	CB.Publisher
- Id	CB.Book.Publisher_id, CB.Publisher.Id
- Name	CB.Publisher.Name, BfL.Book.Publisher

Figure VI.5: Integrated View of Books Databases

Integrated View Term	Data Source Mappings (not visible to user)
V (view root)	**N/A**
- [Book]	CB.Book, BfL.Book
- ISBN	CB.Book.ISBN, BfL.Book.ISBN
- Title	CB.Book.Title, BfL.Book.Title
- Price	CB.Book.Price, BfL.Book.Price
- Description	CB.Book.Description
- [Author]	CB.Author
- Id	CB.Book.Author_id
- Name	BfL.Book.Author
- [Publisher]	CB.Publisher
- Id	CB.Book.Publisher_id
- Name	BfL.Book.Publisher
- [Author]	CB.Author
- Id	CB.Author.Id
- Name	CB.Author.Name
- [Publisher]	CB.Publisher
- Id	CB.Publisher.Id
- Name	CB.Publisher.Name

Figure VI.6: Integrated View of Books Databases with Different X-Specs

tables produces the integrated view given in Figure VI.6.

This integrated view is as correct as the previous one and the user can issue the same queries as before. However, the relationship between the author and publisher information is not as explicit to the user as in the previous view. The system still combines the information using the appropriate joins as there is just a slightly different conceptual view of the data. The choice of integrated view is application dependent and there is nothing preventing the integration algorithm from rearranging the second view to look like the first view if that is the user's preference.

Issuing queries on the integrated view is as simple as choosing which fields to display in the query result. The query generator uses the X-Spec information to generate the proper joins and mappings from semantic to system names. A "global key" such as ISBN is important in query generation as it is guaranteed unique across databases. Such keys allow the system to perform joins across databases.

VI.3 Integrating the Northwind and Southstorm Databases

The last (and largest) example integrates the Northwind and Southstorm databases. This example also details how the standard dictionary is evolved by adding terms needed to represent concepts in the Northwind database. As we build the Northwind X-Spec, we simultaneously describe how the concept terms are added to the dictionary.

VI.3.1 Creating an X-Spec for the Northwind Database

In this example, terms are added to the dictionary as required to represent the semantics of the Northwind database. Starting with a base-level dictionary, the user parses the metadata information of the Northwind database into an X-Spec using Unity and starts associating semantic names to tables and fields of the database. The first table encountered is *Categories* which stores product categories. An initial thought is to add the term `Product Category` to the dictionary. However, the more general and correct term to add is `Category` representing the general notion of a category regardless of what type of category. A category is information and is a concept that exists independent of other concepts. Thus, `Category` belongs under `Form` on the `Information` side of the hierarchy. There are many types of categorizers, so the first term added is `Categorizers` under `Form.` Then, the term `Category` is added under `Categorizers`. With the term `Category` now in the dictionary, the user can assign the *Categories* table the semantic name of `[Category]`.

The first field in the *Category* table is *CategoryID*. The notion of an id or identifier is the most common concept encountered in databases. By nature, an "id" is information which is independent of other information. Thus, the notion of `Id` belongs on the `Information` side under `Form`. We add the term `Identifiers` under `Form`, and then `Id` under `Identifiers`. Thus, the semantic name for the *CategoryID* field is assigned as `[Category] Id`. Naming is another common concept which is encountered in the *CategoryName* field. A name is an information descriptor, so we add the term `Descriptors` under `Form`, and then `Name` under `Descriptors`. This allows us to assign the semantic name `[Category] Name` to the *CategoryName* field. Another descriptor is a regular description field, like *Description*, so we add the term `Description` under `Descriptors` and assign the

Description field the name `[Category] Description`. Finally, the *Picture* field is a visual form of descriptor, so we added the term `Picture` under `Descriptors` and assign the field the semantic name `[Category] Picture`. At this point, we have added sufficient terms to the dictionary to assign semantic names to the *Category* table and all its fields.

The next table is the *Customers* table. The concept of `Customer` is a new one which must be added to the dictionary. What is a customer? A customer is definitely on the physical side because it is a physical entity. A human is a first level entity on the physical side so customer is not a first level concept. Actually, a customer must be a second level concept because a customer is a human which becomes a customer by engaging in a transaction ($f(human) => customer$, where $f = transaction$). Thus, the concept of a customer belongs under `Role`. Since we anticipate that a human may have numerous roles, we first add the term `Discretionary` under `Role` indicating that these roles are undertaken at a person's discretion. Then, we add the term `Customer` under `Discretionary` and assign the *Customer* table the semantic name `[Customer]`.

The semantic name for *CustomerID* is `[Customer] Id`. Since both of these terms are already in the dictionary, no terms are added. Notice that the term `Id` initially added for *CategoryId* is re-used. An id applies to any context (or table) such as customer, product, and category. Thus, we re-use the `Id` term for any table id just by assigning the proper contexts terms to which it applies. The *CompanyName* field stores the name for the customer which can be represented by `[Customer] Name` without adding new terms. Similar to the `Id` term, the `Name` term is a general term which can be re-used across different contexts. The *ContactName* field requires the concept of `Contact` to be added to the dictionary. Similar to being a customer, being a contact is a discretionary role of a human, thus we place the term `Contact` under `Discretionary`, and assign the name `[Customer;Contact] Name` to the field. This is the first semantic name with more than one context. The first context, `Customer`, implies the information is about a customer. The next context is `Contact` which means that a customer has a contact and the field is storing the contact's name. The semantics of the *ContactTitle* field are ambiguous when examining only the field name. By examining the data values, it is obvious that this field actually stores the contact's employment or job title. First, the dictionary has no notion of a `Title`

which is a descriptor, so we add the term `Title` under `Descriptors`. Also, there is no notion of employment. Placing the concept of employment is considerably harder than previous concepts because it can have several possible semantic connotations. In this case, consider employment as the mediating circumstance that causes a human to become an employee. Thus, employment is a third-level concept on the physical side so the term `Employment` is added under `Circumstance`. This allows us to assign the semantic name of `[Customer;Contact;Employment] Title` to the field.

The next fields in the customer table are very common database concepts such as addresses and phone numbers. An address is basic information so it belongs under `Form`. We insert the general term `Locators` under `Form`, then add the terms `Physical Address` and `Virtual Address` to subdivide the types of addresses. The term `Address` is placed under `Physical Address` to represent a typical mailing address. A mailing address consists of several common components such as city, state, and postal code. Thus, under `Address`, we add HAS-A links to the concepts `Address Line 1`, `City`, `Region`, `Postal Code`, and `Country`. Assign the semantic names `[Customer;Address] Address Line 1`, `[Customer;Address] City`, `[Customer;Address] Region`, `[Customer;Address] Country`, and `[Customer;Address] Postal Code` to the fields *Address*, *City*, *Region*, *Country*, and *Postal Code*, respectively.

Phone and fax numbers are virtual addresses, so we add `Phone #` under `Virtual Address`, and add `Fax #` under `Phone #`, as it is a special type of phone number. Then, the fields *Phone* and *Fax* are assigned the semantic names of `[Customer] Phone #` and `[Customer] Fax #`. This completes assigning semantic names to the *Customer* table.

The Northwind database has an *Employee* table storing information on its own employees. The `Employee` concept is a second level concept. Under `Role` we add the term `Work`, then place `Employee` under `Work`. The semantic name for the *Employee* table then becomes `[Employee]`. The *EmployeeId* is assigned a semantic name of `[Employee] Id` without any dictionary additions. The *LastName* and *FirstName* fields require the concept of a `Person Name` under `Name`. Then, HAS-A links are created under `Person Name` to the concepts `Last Name` and `First Name`. The semantic names for these fields become `[Employee;Name] Last Name` and `[Employee;Name] First Name`. Similar to the

Customer table, the *TitleOf* an employee has a semantic name of `[Employee;Employment] Title`. The *TitleofCourtesy* field (`[Employee] Title`) requires a HAS-A link to add the concept of `Title` as part of a person's name.

An employee's *birthdate* is the first instance of the very common date concept. The general notion of a date is simply information, so under `Form` we add a `Temporal` term, then place `Date` under `Temporal` to represent the concept of a generic date. A birthdate is a date with the additional semantics of representing a time of birth. Thus, it is a second order information concept. We add the term `Temporal` under `Proposition`, then `Date` under `Temporal`, and finally `Birthdate` under `Date`. The *Birthdate* field is then assigned a semantic name of `[Employee] Birthdate`. The *HireDate* field is given the semantic name `[Employee;Employment] Date` without adding new terms. Similarly, all the address related fields are assigned names without adding additional terms. A `Home Phone #` term is added under `Phone #` for use in the name `[Employee] Home Phone #` for the *HomePhone* field. A HAS-A link is added under `Phone #` to put the `Extension` term into the dictionary, and the term `Note` is added under `Description` for describing the *Notes* field. Finally, the term `Supervisor` is added under `Work`, so that the *ReportsTo* field can be assigned the semantic name `[Employee;Supervisor] Name`.

The *OrderDetails* table stores information on products that have been ordered. This introduces two new concepts: product and order. An order is a type of transaction. A transaction can be considered a mediating circumstance which causes an item to become a product. Thus, we add `Commerce` under `Circumstance`, `Transaction` under `Commerce`, and `Order` under `Transaction`. A `Product` term is added under `Role`. The semantic name for the table becomes `[Order;Product]`. The *OrderID* field is assigned the semantic name `[Order] Id`. This is different than previous fields because the context of the field does not include the context of its parent table. The *ProductID* field has a name `[Order;Product] Id`. To assign a name to the *UnitPrice* field, we need a notion of `Price`. A price is information with semantic context implying monetary value. Thus, we attach the term `Monetary` under `Proposition`, and `Price` under `Monetary`. The semantic name for the field is then `[Order;Product] Price`. The *Quantity* field is given the name `[Order;Product] Quantity` by adding the term `Numeric` under `Form`, and `Quantity` under `Numeric`. Finally,

the term *Discount%* is appended with a HAS-A link under `Price` for its use in the name `[Order;Product] Discount%` for the *Discount* field.

At this point, most of the terms needed to capture the semantics of the *Order* table have already been added to the dictionary. The table is assigned as semantic name of `[Order]`, and the *OrderId* field is referred to as `[Order] Id`. The *CustomerId*, *EmployeeId*, and *OrderDate* fields are given names `[Order;Customer] Id`, `[Order;Employee] Id`, and `[Order] Date`, respectively. The *RequiredDate* and *ShippedDate* fields have names `[Order] Required Date` and `[Order] Shipped Date` as the terms `Required Date` and `Shipped Date` are placed under `Date` in the `Proposition` branch. The *ShipVia* field stores the company name of the shipper company. We must add the concept of company to the dictionary by adding `Company` under `Role`, `Transportation Company` under `Company`, and `Shipper` under `Transportation Company`. Then, the field has a name of `[Order;Shipper] Name`. The *Freight* field stores the cost of shipping the goods. To construct the name `[Order;Freight] Cost`, we add `Cost` under `Monetary` and `Freight` under `Product`. The rest of the fields store information on a ship-to company address, so we add the term `ShipTo` under company. The rest of the address fields are easily assigned semantic names by using the previously defined address terms.

The *Products* table is given a name `[Product]`, and its *ProductId* and *ProductName* fields have the names `[Product] Id` and `[Product] Name`. To assign `[Product;Supplier] Id` to *SupplierId*, we insert the term `Supplier` under `Company`. The *CategoryId* and *UnitPrice* fields have names `[Product;Category] Id` and `[Product] Price`. HAS-A links for the new terms `Quantity-per-Unit` and `Discontinued` under `Product` are added to construct `[Product] Quantity-per-Unit` for the *QuantityPerUnit* field and `[Product] Discontinued` for the *Discontinued* field. The rest of the fields concern the concept of inventory which is a second-level information concept. Thus, we add `Inventory` under `Proposition`, and under `Inventory` insert HAS-A links for the terms `On-Order` and `Reorder Level`.

The *Shippers* table and *Suppliers* table require only one new term, `Web Address`, to be appended under `Virtual Address` to store the semantics of the *HomePage* field.

We have now created a full X-Spec for the Northwind database and defined the

necessary dictionary terms to capture its semantics (see Figures VI.8 and VI.9). The procedure is not complicated. Often the most difficult situation is determining if a concept is a first, second, or third level concept. There is an enormous re-use of dictionary terms, so the dictionary grows rather slowly. The dictionary constructed in this example is listed at the end of Appendix G.

VI.3.2 An X-Spec for the Southstorm Database

The creation of the X-Spec for the Southstorm database was discussed previously in Section III.3, so it is not repeated here. The final X-Spec is given in Figure VI.7.

```
<?xml version="1.0" ?>
<Schema
        name = "Southstorm_xspec.xml"
        xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:dt="urn:schemas-microsoft-com:datatypes">

<ElementType name="[Order]" sys_name = "Orders_tb" sys_type="Table">
        <element type = "[Order] Id" sys_name = "Order_num" sys_type = "Field"/>
        <element type = "[Order] Total Amount" sys_name = "Order_total" sys_type = "Field"/>
        <element type = "[Order;Customer] Name" sys_name = "Cust_name" sys_type = "Field"/>
        <element type = "[Order;Customer;Address] Address Line 1" sys_name="Cust_address"
             sys_type="Field"/>
        <element type = "[Order;Customer;Address] City" sys_name = "Cust_city" sys_type = "Field"/>
        <element type = "[Order;Customer;Address] Postal Code" sys_name="Cust_pc" sys_type="Field"/>
        <element type = "[Order;Customer;Address] Country" sys_name="Cust_country" sys_type="Field"/>
        <element type = "[Order;Product] Id" sys_name = "Item1_id" sys_type = "Field"/>
        <element type = "[Order;Product] Quantity" sys_name = "Item1_quantity" sys_type = "Field"/>
        <element type = "[Order;Product] Price" sys_name = "Item1_price" sys_type = "Field"/>
        <element type = "[Order;Product] Id" sys_name = "Item2_id" sys_type = "Field"/>
        <element type = "[Order;Product] Quantity" sys_name = "Item2_quantity" sys_type = "Field"/>
        <element type = "[Order;Product] Price" sys_name = "Item2_price" sys_type = "Field"/>
</ElementType>
</Schema>
```

Figure VI.7: Southstorm X-Spec

Notice that constructing the X-Spec requires no new terms in the dictionary. The basic terms `Order`, `Customer`, `Product`, and `Id` were all added for the construction of the Northwind X-Spec and are re-used.

VI.3.3 Integrating the Southstorm and Northwind Databases

Once X-Specs for both databases are constructed the integration algorithm is executed and produces the integrated view shown in Figures VI.10 and VI.11.

```
<?xml version="1.0" ?>
<Schema
        name = "Northwind_xspec.xml"
        xmlns="urn:schemas-microsoft-com:xml-data"
        xmlns:dt="urn:schemas-microsoft-com:datatypes">

<ElementType name="[Category]" sys_name = "Categories" sys_type="Table">
        <element type = "[Category] Id" sys_name = "CategoryID" sys_type = "Field"/>
        <element type = "[Category] Name" sys_name = "CategoryName" sys_type = "Field"/>
        <element type = "[Category] Description" sys_name = "Description" sys_type = "Field"/>
        <element type = "[Category] Picture" sys_name = "Picture" sys_type = "Field"/>
</ElementType>

<ElementType name="[Customer]" sys_name = "Customers" sys_type="Table">
        <element type = "[Customer] Id" sys_name = "CustomerID" sys_type = "Field"/>
        <element type = "[Customer] Name" sys_name = "CompanyName" sys_type = "Field"/>
        <element type = "[Customer;Contact] Name" sys_name = "ContactName" sys_type = "Field"/>
        <element type = "[Customer;Contact;Employment] Title" sys_name="ContactTitle"
             sys_type="Field"/>
        <element type = "[Customer;Address] Address Line 1" sys_name = "Address" sys_type = "Field"/>
        <element type = "[Customer;Address] City" sys_name = "City" sys_type = "Field"/>
        <element type = "[Customer;Address] Region" sys_name = "Region" sys_type = "Field"/>
        <element type = "[Customer;Address] Postal Code" sys_name = "PostalCode" sys_type = "Field"/>
        <element type = "[Customer;Address] Country" sys_name = "Country" sys_type = "Field"/>
        <element type = "[Customer] Phone #" sys_name = "Phone" sys_type = "Field"/>
        <element type = "[Customer] Fax #" sys_name = "Fax" sys_type = "Field"/>
</ElementType>

<ElementType name="[Employee]" sys_name = "Employees" sys_type="Table">
        <element type = "[Employee] Id" sys_name = "EmployeeID" sys_type = "Field"/>
        <element type = "[Employee;Name] Last Name" sys_name = "LastName" sys_type = "Field"/>
        <element type = "[Employee;Name] First Name" sys_name = "FirstName" sys_type = "Field"/>
        <element type = "[Employee;Employment] Title" sys_name = "Title" sys_type = "Field"/>
        <element type = "[Employee] Title" sys_name = "TitleofCourtesy" sys_type = "Field"/>
        <element type = "[Employee] Birthdate" sys_name = "BirthDate" sys_type = "Field"/>
        <element type = "[Employee;Employment] Date" sys_name = "HireDate" sys_type = "Field"/>
        <element type = "[Employee;Address] Address Line 1" sys_name = "Address" sys_type = "Field"/>
        <element type = "[Employee;Address] City" sys_name = "City" sys_type = "Field"/>
        <element type = "[Employee;Address] Region" sys_name = "Region" sys_type = "Field"/>
        <element type = "[Employee;Address] Postal Code" sys_name = "PostalCode" sys_type = "Field"/>
        <element type = "[Employee;Address] Country" sys_name = "Country" sys_type = "Field"/>
        <element type = "[Employee] Home Phone #" sys_name = "HomePhone" sys_type = "Field"/>
        <element type = "[Employee] Extension" sys_name = "Extension" sys_type = "Field"/>
        <element type = "[Employee] Picture" sys_name = "Photo" sys_type = "Field"/>
        <element type = "[Employee] Note" sys_name = "Notes" sys_type = "Field"/>
        <element type = "[Employee;Supervisor] Name" sys_name = "ReportsTo" sys_type = "Field"/>
</ElementType>

<ElementType name="[Order;Product]" sys_name = "OrderDetails" sys_type="Table">
        <element type = "[Order] Id" sys_name = "OrderID" sys_type = "Field"/>
        <element type = "[Order;Product] Id" sys_name = "ProductID" sys_type = "Field"/>
        <element type = "[Order;Product] Price" sys_name = "UnitPrice" sys_type = "Field"/>
        <element type = "[Order;Product] Quantity" sys_name = "Quantity" sys_type = "Field"/>
        <element type = "[Order;Product] Discount%" sys_name = "Discount" sys_type = "Field"/>
</ElementType>
```

Figure VI.8: Northwind X-Spec Part 1

```
<ElementType name="[Order]" sys_name = "Orders" sys_type="Table">
        <element type = "[Order] Id" sys_name = "OrderID" sys_type = "Field"/>
        <element type = "[Order;Customer] Id" sys_name = "CustomerID" sys_type = "Field"/>
        <element type = "[Order;Employee] Id" sys_name = "EmployeeID" sys_type = "Field"/>
        <element type = "[Order] Date" sys_name = "OrderDate" sys_type = "Field"/>
        <element type = "[Order] Required Date" sys_name = "RequiredDate" sys_type = "Field"/>
        <element type = "[Order] Shipped Date" sys_name = "ShippedDate" sys_type = "Field"/>
        <element type = "[Order;Shipper] Id" sys_name = "Shipvia" sys_type = "Field"/>
        <element type = "[Order;Freight] Cost" sys_name = "Freight" sys_type = "Field"/>
        <element type = "[Order;Shipto] Name" sys_name = "ShipName" sys_type = "Field"/>
        <element type = "[Order;Shipto;Address] Address Line 1" sys_name="ShipAddress"
             sys_type="Field"/>
        <element type = "[Order;Shipto;Address] City" sys_name="ShipCity" sys_type="Field"/>
        <element type = "[Order;Shipto;Address] Region" sys_name="ShipRegion" sys_type="Field"/>
        <element type = "[Order;Shipto;Address] Postal Code" sys_name="ShipPostalCode"
             sys_type="Field"/>
        <element type = "[Order;Shipto;Address] Country" sys_name="ShipCountry" sys_type="Field"/>
</ElementType>

<ElementType name="[Product]" sys_name = "Products" sys_type="Table">
        <element type = "[Product] Id" sys_name = "ProductID" sys_type = "Field"/>
        <element type = "[Product] Name" sys_name = "ProductName" sys_type = "Field"/>
        <element type = "[Product;Supplier] Id" sys_name = "SupplierID" sys_type = "Field"/>
        <element type = "[Product;Category] Id" sys_name = "CategoryID" sys_type = "Field"/>
        <element type = "[Product] Quantity-per-Unit" sys_name="QuantityPerUnit" sys_type="Field"/>
        <element type = "[Product] Price" sys_name = "UnitPrice" sys_type = "Field"/>
        <element type = "[Product] Inventory" sys_name = "UnitsInStock" sys_type = "Field"/>
        <element type = "[Product] On-order" sys_name = "UnitsOnOrder" sys_type = "Field"/>
        <element type = "[Product] Re-order level" sys_name = "Reorderlevel" sys_type = "Field"/>
        <element type = "[Product] Discontinued" sys_name = "Discontinued" sys_type = "Field"/>
</ElementType>

<ElementType name="[Shipper]" sys_name = "Shippers" sys_type="Table">
        <element type = "[Shipper] Id" sys_name = "ShipperID" sys_type = "Field"/>
        <element type = "[Shipper] Name" sys_name = "CompanyName" sys_type = "Field"/>
        <element type = "[Shipper] Phone #" sys_name = "Phone" sys_type = "Field"/>
</ElementType>

<ElementType name="[Supplier]" sys_name = "Suppliers" sys_type="Table">
        <element type = "[Supplier] Id" sys_name = "SupplierID" sys_type = "Field"/>
        <element type = "[Supplier] Name" sys_name = "CompanyName" sys_type = "Field"/>
        <element type = "[Supplier;Contact] Name" sys_name = "ContactName" sys_type = "Field"/>
        <element type = "[Supplier;Contact;Employment] Title" sys_name="ContactTitle"
             sys_type="Field"/>
        <element type = "[Supplier;Address] Address Line 1" sys_name="Address" sys_type="Field"/>
        <element type = "[Supplier;Address] City" sys_name = "City" sys_type = "Field"/>
        <element type = "[Supplier;Address] Region" sys_name = "Region" sys_type = "Field"/>
        <element type = "[Supplier;Address] Postal Code" sys_name="PostalCode" sys_type="Field"/>
        <element type = "[Supplier;Address] Country" sys_name = "Country" sys_type = "Field"/>
        <element type = "[Supplier] Phone #" sys_name = "Phone" sys_type = "Field"/>
        <element type = "[Supplier] Fax #" sys_name = "Fax" sys_type = "Field"/>
        <element type = "[Supplier] Web Address" sys_name = "HomePage" sys_type = "Field"/>
</ElementType>
</Schema>
```

Figure VI.9: Northwind X-Spec Part 2

Global View Term	Data Source Mappings (not visible to user)
V (view root)	**N/A**
- [Customer]	NW.Customers
- [Address]	
- Address Line 1	NW.Customers.Address
- City	NW.Customers.City
- Region	NW.Customers.Region
- Postal Code	NW.Customers.PostalCode
- [Contact]	
- [Employment]	
- Title	NW.Customers.ContactTitle
- Name	NW.Customers.ContactName
- Id	NW.Customers.CustomerID
- Name	NW.Customers.CompanyName
- Phone #	NW.Customers.Phone
- Fax #	NW.Customers.Fax
- [Category]	NW.Categories
- Id	NW.Categories.CategoryID
- Name	NW.Categories.CategoryName
- Description	NW.Categories.Description
- Picture	NW.Categories.Picture
- [Employee]	NW.Employees
- [Supervisor]	
- Name	NW.Employees.ReportsTo
- [Address]	
- Address Line 1	NW.Employees.Address
- City	NW.Employees.City
- Region	NW.Employees.Region
- Postal Code	NW.Employees.PostalCode
- Country	NW.Employees.Country
- [Employment]	
- Title	NW.Employees.Title
- Date	NW.Employees.HireDate
- Id	NW.Employees.EmployeeID
- [Name]	
- Last Name	NW.Employees.LastName
- First Name	NW.Employees.FirstName
- Title	NW.Employees.TitleofCourtesy
- Birthdate	NW.Employees.Birthdate
- Home Phone #	NW.Employees.HomePhone
- Extension	NW.Employees.Extension
- Picture	NW.Employees.Picture
- Note	NW.Employees.Note
- [Product]	NW.Products
- [Category]	
- Id	NW.Products.CategoryID
- [Supplier]	
- Id	NW.Products.SupplierID
- Id	NW.Products.ProductID
- Name	NW.Products.ProductName
- Quantity-per-Unit	NW.Products.QuantityPerUnit
- Price	NW.Products.UnitPrice
- Inventory	NW.Products.UnitsInStock
- On-order	NW.Products.UnitsOnOrder
- Re-order level	NW.Products.Reorderlevel
- Discontinued	NW.Products.Discontinued

Figure VI.10: Northwind/Southstorm Integrated View Part 1

Global View Term	Data Source Mappings (not visible to user)
V (view root)	**N/A**
- [Order]	NW.Orders
- [Shipto]	
- [Address]	
- Address Line 1	NW.Orders.ShipAddress
- City	NW.Orders.ShipCity
- Region	NW.Orders.ShipRegion
- Postal Code	NW.Orders.ShipPostalCode
- Country	NW.Orders.ShipCountry
- Name	NW.Orders.ShipName
- [Freight]	
- Cost	NW.Orders.Freight
- [Shipper]	
- Id	NW.Orders.Shipvia
- [Employee]	
- Id	NW.Orders.EmployeeID
- [Customer]	
- Id	NW.Orders.CustomerID
- Name	SS.Orders_tb.Cust_name
- [Address]	
- Address Line 1	SS.Orders_tb.Cust_address
- City	SS.Orders_tb.Cust_city
- Region	SS.Orders_tb.Cust_region
- Postal Code	SS.Orders_tb.Cust_pc
- Country	SS.Orders_tb.Cust_country
- [Product]	NW.Orders.OrderDetails
- Id	NW.Orders.OrderDetails.ProductID, SS.Orders_tb.Item1_id, SS.Orders_tb.Item2_id
- Price	NW.Orders.OrderDetails.UnitPrice, SS.Orders_tb.Item1_price, SS.Orders_tb.Item2_price
- Quantity	NW.Orders.OrderDetails.Quantity, SS.Orders_tb.Item1_quantity, SS.Orders_tb.Item2_quantity
- Discount%	NW.Orders.OrderDetails.Discount
- Id	NW.Orders.OrderDetails.OrderID, NW.Orders.Orders.OrderID
- Date	NW.Orders.OrderDate
- Required Date	NW.Orders.RequiredDate
- Shipped Date	NW.Orders.ShippedDate
- Total Amount	SS.Orders_tb.Order_total
- [Supplier]	NW.Suppliers
- [Address]	NW.Suppliers
- Address Line 1	NW.Suppliers.Address
- City	NW.Suppliers.City
- Region	NW.Suppliers.Region
- Postal Code	NW.Suppliers.PostalCode
- Country	NW.Suppliers.Country
- [Contact]	
- [Employment]	
- Title	NW.Suppliers.ContactTitle
- Name	NW.Suppliers.ContactName
- Id	NW.Suppliers.SupplierID
- Name	NW.Suppliers.CompanyName
- Phone #	NW.Suppliers.Phone
- Fax #	NW.Suppliers.Fax
- Web address	NW.Suppliers.HomePage
- [Shipper]	NW.Shippers
- Id	NW.Shippers.ShipperID
- Name	NW.Shippers.CompanyName
- Phone #	NW.Shippers.Phone

Figure VI.11: Northwind/Southstorm Integrated View Part 2

Chapter VII

Conclusions and Future Work

VII.1 Contributions and Conclusions

In this thesis, we have detailed how a standard global dictionary, a formalized method for capturing data semantics (X-Specs), an integration algorithm, and a query processor are combined into an integration architecture that integrates relational databases. Data sources are transparently queried by semantic names, and the multidatabase layer acts as an intermediary providing the necessary integration of concepts and translation between semantic and system names. Integrating entire data sources is a major step forward from industry standards which are inflexible and only integrate a small subset of the data actually involved in communications. Applications of the approach include integration of web databases and integration of organization database systems allowing easier deployment of advanced technologies such as data warehouses and decision support systems.

The standard dictionary foundation of the architecture is its major contribution. The dictionary functions as a shared ontology providing terms and definitions for the description of database semantics. Unlike knowledge bases which suffer from imprecision and complexity, the dictionary is a simplified standard for the expression of data semantics to describe schema elements. The construction, modification, and use of the dictionary is considerably easier than other approaches. It is designed from its inception to be deployable in industrial environments. Industry standards and approaches have clearly demonstrated

that despite the often conflicting interests between companies, consumers, and technology users, standardization is achievable in almost any domain. Further, when powerful standards are developed and implemented, the cost to use, deploy, and enhance the technology decreases dramatically.

This thesis proposes that a standard dictionary or ontology must be developed to enable automatic schema integration and improve database interoperability. The reasons for standardization are clear, and the benefits are enormous. Research addressing the integration problem has created solid algorithms for analyzing schema conflicts and resolving them manually or with the aid of software tools. The missing element to this point has been the lack of semantic standardization. Designers are required to manipulate the complex models and tools because the models do not have formalized methods for capturing data semantics. Thus, designers become responsible for "filling in" the missing gaps in information which the models lack.

While we are unwaivering in our belief of the necessity of standardization [58], the dictionary organization proposed in the thesis is neither complete nor optimal. A small research team can not possible enumerate a perfectly balanced ontology which caters to all domains, industries, and environments. The evolution of the dictionary and its initial creation can only truly be optimized by the co-operation of users, companies, and organizations working together. The best terms and their placement is a matter of debate among philosophers, computer scientists, and other interested parties.

However, organizing the dictionary hierarchically is an important insight, as it allows information to be categorized more effectively. Further, when used in conjunction with the new query model, hierarchical organization opens up new possibilities for semantic querying that will be explored further in the future work section. In addition, the very fact that the integration algorithm and query protocols can be effective despite a "sub-optimal" dictionary is a reflection of their power. An integrator does not require a dictionary which correctly organizes the entire world knowledge into a massive hierarchy. Rather, if domain integrators can agree on just their domain knowledge and determine appropriate dictionary terms, the algorithm will function correctly in any domain with the dictionary defined.

Defining and using a standard dictionary is an important contribution, but it is

not a total solution. The integration architecture defines specification documents, X-Specs, which use the basic semantic information present in the term dictionary, to build semantic information about database schemas. This semantic information includes the usual schema metadata including joins and keys. However, the notion of a *semantic name* for a schema element is where the standard dictionary is used in a practical application. The semantic name becomes the universal identifier coveted by integrators. It is like the attribute name in the Universal Relation, or an XML tag in XML documents. Given the semantic name for an element, you immediately know its semantics and know that these semantics are uniquely identified by name. Standardization has allowed the naming conflict between systems to be avoided. Further, it has decoupled the need for integrators to understand the semantics of every system that they are integrating. Integration is achievable by defining the semantics of each schema element independently using the standard dictionary. If the semantic names match across databases, they represent identical concepts and are integrated.

Semantic names relegate the integration algorithm to a trivial name matching algorithm. Complex transformations or schema equivalence models are not required. If two semantic names are identical, the concepts they represent are identical. The integration algorithm simply performs the term matching procedure and returns a set of semantic names present in the integrated databases.

At this point, there is an "integrated view" of the combined systems, but it is not the typical structural view consisting of tables and fields commonly used in other approaches. Although it may be possible to convert the integrated context view of semantic names into a "regular" structural view, there is no benefit. Structure is irrelevant to the user. The second major contribution of the thesis is that the integrated view is not a structural view of the data. The integrated context view produced by the architecture is actually a hierarchy of concepts and contexts. Essentially, it is a hierarchy of "ideas". The ideas are related by IS-A relationships and HAS-A relationships similar to how the individual terms in the semantic names are related. In fact, this hierarchy is automatically produced by the fact that in the process of describing the semantics of schema elements by combining dictionary terms to form semantic names, the designer is categorizing the "idea" or "essence" of the element and determining its appropriate place in the integrated context view.

It is easier to query by conceptual ideas than by structure because no semantic conversion is required. That is, when queries are formulated, the user has a conceptual model of the data which they require. However, in relational databases, this conceptual data model has been converted to the relational model and translated into fields and tables. Structural querying requires the user to understand how to reconcile their conceptual view of the data with the structural view of its current organization. Thus, the third major contribution of the thesis is providing a model which allows the user to transparently query any relational database by conceptual idea (semantic name) instead of by structure. This reduces the semantic burden on the user during query formulation, and significantly simplifies the task of querying multiple database systems.

By allowing the user to query by context [56, 57], the system becomes responsible for translating from semantic queries to structural queries. The fourth major contribution is a query system which automates this translation process. First, the query system must determine the correct ideas (semantic names) requested by the user and then using the supplied information in the X-Specs, generate local views for each data source which contain relevant information. Generating these local views requires determining the correct fields and tables to access, and then applying relevant local joins to connect the information. The thesis demonstrated intelligent query algorithms which are capable of constructing these local views for each data source by automatically generating SQL queries and executing them using ODBC.

Generation of local views for each database resolves many structural conflicts that exist between data sources. The second phase performed by the query processor automatically integrates the generated local views. The local views still may contain differences such as different normalization states. Further, the query processor may be able to perform joins across databases based on the presence of common keys. The architecture defines a simple but effective mechanism for specifying global keys and joins.

The fifth major contribution of the thesis is the implementation of the architecture. The architecture is implemented in a software package called Unity [59]. Unity is a production level tool which demonstrates the power and applicability of the architecture to many environments. The power of standardization and the architecture algorithms are

immediately apparent after using Unity.

In conclusion, this thesis has contributed an integration architecture [60, 55] which is unique. The combination of standardization and intelligent query and integration algorithms produces an architecture capable of automatic schema integration for relational databases. The architecture proves that automatic schema integration **is possible** if you accept standardization.

VII.2 Directions for Future Work

Future work includes refining the standard dictionary, improving the implementation of Unity, and developing new query mechanisms to complement the unique nature of the architecture. Performing transparent updates and edits to multidatabase data in this environment would be an especially challenging area of future work.

Refining the standard dictionary is an extremely interesting area of future work. Although the architecture can function with any standard dictionary, the development of a complete, general dictionary or ontology would be very exciting. There have been ontology development efforts such as the Cyc knowledge-base, but they tend to produce ontologies of massive size and complexity. The dictionary as defined in this architecture is simple and many terms are re-usable. Further, the dictionary is a strict hierarchy, unlike the graph-like nature of the Cyc knowledge-base, and is developed using the concepts of firstness, secondness, and thirdness as defined in Section III.2.1. These concepts are open to philosophical debate, which makes the placement of each dictionary term a matter of debate and compromise. It would be interesting to see how the dictionary evolves when used across many integration environments and application domains.

The query mechanisms defined in the thesis are sufficient for the majority of query environments. One major limiting factor, however, is that the join selection algorithm is only capable of determining joins when there is only one join connecting two tables (Section III.6.3). Extending the join selection algorithm to handle the possibility of multiple joins between two tables is required for a general solution. Further, there may be more elegant methods of determining the correct join tree to use when a join graph for a database is

cyclic. Although the current method of showing the user the interrelationships between semantic names (Section III.6.4) is functional and informative, there may be certain cases where it is insufficient or leads to conflicts between databases. For example, a join linking two ideas (semantic names) may be present in one database and not in another. Does it make sense to show this relationship to the user if it only applies to one database?

Another interesting query extension would exploit the hierarchical nature of the standard dictionary. For example, `Fax #` is a subconcept of `Phone #` in the dictionary. The user may query for "all phone numbers regardless of use" which would return all direct mappings to `Phone #`, plus mapping to its subconcepts such as `Fax #`, `Home Phone #`, *etc.* Such hierarchical querying could be extremely powerful.

A major and challenging query issue is handling global updates. The architecture is currently a query-only system. When the user is allowed to update the underlying databases from the global level, new issues concerning trust, concurrency, and data value conflicts arise. Simple trust issues include determining if a database should accept a user's modifications or attempt to validate them to ensure they make sense in respect to the database context. Even more interesting issues arise when databases can determine if they trust other databases. For example, a local database may update an address field value. Another local database which is connected to it in the federation receives notification of this update and contains a different address value. Should it accept the new address value? Maybe certain databases should be "authorities" on a certain subset of the information in the integrated view and ensure other databases have the correct information. This issue gets even more interesting when you factor in that the databases may be owned by different organizations and companies which may be competitors.

Concurrency control algorithms for the multidatabase environment have been studied extensively, but they would probably need to be optimized for a new co-operating and trusting environment. Further, since the architecture can be distributed in a web browser, implementing a global distributed transaction management protocol would be required. When the clients implement transaction management, whole new complexities arise.

Finally, Unity is evolving and improving with our architecture. We continue to add greater functionality to the system in all four components. Continuing work on the

global dictionary editor component includes an option to save the dictionary as an XML file instead of a binary file and better linking to the other document types which depend on its data.

Future work on the X-Spec editor involves improving the automation features. Currently, the editor is only capable of parsing a schema exported from Microsoft Access. This is not an overly limiting feature because Access itself is able to capture schemas from various data sources including Oracle, Sybase, and others using ODBC. However, a direct ODBC connection to the various databases to retrieve schema information would be desirable. Eventually, the system may be trained to remember previous mappings from system names to a semantic name, so that system names can be automatically mapped to a semantic name when encountered. For example, if the system name `tbl_id` is mapped extensively to the semantic name `id`, we would like the system to remember that mapping and automatically perform it (by assigning the correct semantic name) the next time the system name `tbl_id` is encountered. Further, although the query system implements result normalization, the system needs to implement data conflict resolution procedures and the algorithm for applying joins across databases to completely represent the total functionality of the architecture. Further, distributing Unity as a web browser component which interacts with component software at each database would be challenging future work.

In total, the architecture presented is a solid foundation for future work. As the query system functionality is expanded and the standard dictionary is refined, the architecture may become a total solution to the schema integration and database interoperability problems.

Bibliography

[1] S. Adali and R. Emery. A uniform framework for integrating knowledge in heterogeneous knowledge systems. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 513–520, 1995.

[2] A.V. Aho, C. Beeri, and J. Ullman. The Theory of Joins in Relational Databases. *ACM Transactions on Database Systems*, 4(3):297–314, September 1979.

[3] G. Arocena and A. Mendelzon. WebOQL: Restructuring documents, databases, and webs. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 24–33, 1998.

[4] G. Attaluri, D. Bradshaw, N. Coburn, P. Larson, P. Martin, A. Silberschatz, J. Slonim, and Q. Zhu. The CORDS multidatabase project. *IBM Systems Journal*, 34(1):39–62, 1995.

[5] M. Barja, T. Bratvold, J. Myllymaki, and G. Sonnenberger. Informia: A Mediator for Integrated Access to Heterogeneous Information Sources. In *Proceedings of the ACM International Conference on Information and Knowledge Management (CIKM-98)*, pages 234–241, New York, November 3–7 1998. ACM Press.

[6] K. Barker. *Transaction Management on Multidatabase Systems*. PhD thesis, University of Alberta, 1990.

[7] K. Barker. Quantification of Autonomy on Multidatabase Systems. *The Journal of System Integration*, pages 1–26, 1993.

[8] C. Batini, M. Lenzerini, and S. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.

[9] C. Beeri, A. Levy, and M. Rousset. Rewriting Queries Using Views in Description Logics. In *Principles of Database Systems (PODS'97)*, pages 99–108, 1997.

[10] J. Bell and L. Rowe. Human Factors Evaluation of a Textual, Graphical, and Natural Language Query Interfaces. Technical Report ERL-90-12, University of California, Berkeley, February, 1990.

[11] M. Bouzeghoub and E. Metais. Semantic Modeling of Object Oriented Databases. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 3–6, September 1991.

[12] S. Bressan, C. Goh, K. Fynn, M. Jakobisiak, K. Hussein, H. Kon, T. Lee, S. Madnick, T. Pena, J. Qu, A. Shum, and M. Siegel. The COntext INterchange Mediator Prototype. *SIGMOD Record*, 26(2):525–527, May 1997.

[13] M. Bright, A. Hurson, and S. Pakzad. A Taxonomy and Current Issues in Multidatabase Systems. *IEEE Computer*, 25(3):50–60, March 1992.

[14] M. Bright, A. Hurson, and S. Pakzad. Automated Resolution of Semantic Heterogeneity in Multidatabases. *ACM Transactions on Database Systems*, 19(2):212–253, June 1994.

[15] V. Brosda and G. Vossen. Update and Retrieval in a Relational Database Through a Universal Schema Interface. *ACM Transactions on Database Systems*, 13(4):449–485, December 1988.

[16] O. Bukhres, J. Chen, A. Elmagarmid, X. Liu, and J. Mullen. InterBase: A Multidatabase prototype system. *ACM SIGMOD Record*, 22(2):534–539, June 1993.

[17] S. Castano and V. Antonellis. Semantic Dictionary Design for Database Interoperability. In *Proceedings of the 13th International Conference on Data Engineering (ICDE'97)*, pages 43–54, April 1997.

[18] S. Castano, V. Antonellis, M. Fugini, and B. Pernici. Conceptual Schema Analysis: Techniques and Applications. *ACM Transactions on Database Systems*, 23(3):286–332, September 1998.

[19] T. Catarci and G. Santucci. Query by Diagram: A Graphical Environment for Querying Databases. *SIGMOD Record*, 23(2):515–515, June 1994.

[20] C. Chang and H. Garcia-Molina. Mind Your Vocabulary: Query Mapping Across Heterogeneous Information Sources. *SIGMOD Record*, 28(2):335–346, June 1999.

[21] J. Chen, A. Aiken, N. Nathan, C. Paxson, M. Stonebraker, and J. Wu. Extending a Graphical Query Language to Support updates, Foreign Systems, and Transactions. Technical Report S2K-93-38, University of California, Berkeley, 1994.

[22] W. Cheung and H. Cheng. The Model-Assisted Global query systems for multiple databases in distributed enterprises. *ACM Transactions on Information Systems*, 14(4):421–470, 1996.

[23] The Metadata coalition. Metadata Interchange Specification. Technical Report version 1.1. http://www.mdcinfo.com/MDIS/MDIS11.html, The Metadata coalition, August 1997.

[24] W. Cohen. Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity. *SIGMOD Record*, 27(2):201–212, 1998.

[25] C. Collet, M. Huhns, and W-M. Shen. Resource Integration Using a Large Knowledge Base in Carnot. *IEEE Computer*, 24(12):55–62, December 1991.

[26] B. Convent. Unsolvable problems related to the view integration approach. In *Proceedings of the International Conference on Database Theory*, pages 141–156, September 1986.

[27] D. Cruse. *Lexical Semantics*. Cambridge University Press, 1986.

[28] N. Fiddian: D. Karunaratna, W. Gray. Establishing a Knowledge Base to Assist

Integration of Heterogeneous Databases. In *Advances in Databases: 16th British National Conference on Databases*, pages 103–118, 1998.

[29] C. Date. *The SQL standard*. Addison Wesley, Reading, US, third edition, 1994.

[30] U. Dayal and H. Hwang. View definition and generalization for database integration in MULTIBASE: A system for heterogeneous distributed databases. *IEEE Transactions on Software Engineering*, SE-10(6):628–644, 1984.

[31] M. Fernandez, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: Experiences with a web-site management system. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 414–425, 1998.

[32] T. Finin, R. Fritzon, D. McKay, and R. McEntire. KQML – A Language and protocol for Knowledge and Information Exchange. In *Proceedings of the 13th International Workshop on Distributed Artificial Intelligence*, pages 126–136, Seattle, WA, July 1994.

[33] D. Florescu, A. Levy, I. Manolescu, and D. Suciu. Query Optimization in the Presence of Limited Access Patterns. *SIGMOD Record*, 28(2):311–322, June 1999.

[34] D. Florescu, A. Levy, and A. Mendelzon. Database techniques for the World-Wide Web: A Survey. *SIGMOD Record*, 27(3):59–74, 1998.

[35] S. Gadia. Parametric Databases: Seamless integration of spatial, temporal, belief and ordinary data. *SIGMOD Record*, 22(1):15–20, March 1993.

[36] M. Garcha-Solaco, F. Saltor, and M. Castellanos. *Semantic Heterogeneity in Multidatabase Systems*. Prentice Hall Inc., 1996.

[37] M. Genesereth, A. Keller, and O. Duschka. Infomaster: An Information Integration System. *SIGMOD Record*, 26(2):539–542, May 1997.

[38] D. Georgakopoulos, M. Rusinkiewicz, and A.P. Sheth. Using Tickets to Enforce the Serializability of Multidatabase Transactions. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):166–180, February 1994.

[39] C. Goh, S. Bresson, S. Madnich, and M. Siegel. Context Interchange: New Features and Formalisms for the Intelligent Integration of Information. *ACM Transactions on Information Systems*, 17(3):270–293, July 1999.

[40] T. Gruber. Toward principles for the design of ontologies used for knowledge sharing. Technical Report TR KSL 93-04, Knowledge Systems Laboratory, Stanford, 1993.

[41] T. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1995.

[42] M. Hammer and D. McLeod. Database Description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems*, 6(3):351–386, September 1981.

[43] S. Holzner. *XML Complete*. McGraw-Hill, New York, NY, USA, 1998.

[44] R. Hull and R. King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 19(3):201–260, September 1988.

[45] Uniform Code Council Inc. SIL - Standard Interchange Language. Technical Report http://www.uc-council.org/e_commerce/ec_sil_general_overview.html, January 1999.

[46] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An Adaptive Query Execution System for Data Integration. *SIGMOD Record*, 28(2):299–310, June 1999.

[47] R. Jakobovits. Integrating Autonomous Heterogenous Information Sources. Technical Report TR-97-12-05, University of Washington, Department of Computer Science, 1997.

[48] U. Johnen and M. Jeusfeld. An Executable Meta Model for Re-Engineering of Database Schemas. Technical Report 94-19, Technical University of Aachen (RWTH Aachen), 1994.

[49] W. Kent. The Breakdown of the Information Model in MDBSs. *SIGMOD Record*, 20(4):10–15, December 1991.

[50] W. Kim and J. Seo. Classifying Schematic and Data Heterogeneity in Multidatabase Systems. *IEEE Computer*, 24(12):12–18, December 1991.

[51] D. Konopnicki and O. Shmueli. Information gathering in the World-Wide Web: The W3QL query language and the W3QS system. *ACM Transactions on Database Systems*, 23(4):369–410, December 1998.

[52] H. Korth, G. Juper, J. Feigenbaum, A. Gelder, and J. Ullman. System/U: A Database System Based on the Universal Relation Assumption. *ACM Transactions on Database systems*, 9(3):331–347, September 1984.

[53] R. Krishnamurthy, W. Litwin, and W. Kent. Language features for interoperability of databases with schematic discrepancies. *SIGMOD Record*, 20(2):40–49, June 1991.

[54] E. Kuhn, T. Tschernko, and K. Schwarz. A language based multidatabase system. *SIGMOD Record*, 23(2):509, June 1994.

[55] R. Lawrence and K. Barker. Automatic Integration of Relational Database Schemas. Technical Report 2000-662-14, Department of Computer Science, University of Calgary, July 2000.

[56] R. Lawrence and K. Barker. Multidatabase Querying by Context. In *DATASEM2000 - 20th annual conference on the Current Trends in Databases and Information Systems*, pages 127–136, October 2000.

[57] R. Lawrence and K. Barker. Multidatabase Querying by Context. Technical Report 2000-663-15, Department of Computer Science, University of Calgary, July 2000.

[58] R. Lawrence and K. Barker. The Relational Integration Model for Integrating Relational Schemas. In *Knowledge Discovery For Business Information Systems*, pages 153–172. Kluwer Academic Publishers, 2000.

[59] R. Lawrence and K. Barker. Unity - A Database Integration Tool. Technical Report 2000-664-16, Department of Computer Science, University of Calgary, July 2000.

[60] R. Lawrence and K. Barker. Integrating Relational Database Schemas using a Standardized Dictionary. In *SAC'2001- ACM Symposium on Applied Computing*, March 2001.

[61] R. Lawrence, K. Barker, and A. Adil. Simulating MDBS Transaction Management Protocols. In *Proceedings of the ISCA 11th International Conference*, pages 93–97, November 1998.

[62] J. Lee and D. Baik. SemQL: A Semantic Query Language for Multidatabase Systems. In *Proceedings of the 8th International Conference on Information Knowledge Management (CIKM'99)*, pages 259–266, Kansas City, MO, November 1999.

[63] A. Levy, A. Mendelzon, D. Srivastava, and Y. Sagiv. Answering Queries Using Views. In *Principles of Database Systems (PODS'95)*, pages 95–104, 1995.

[64] A. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 251–262, 1996.

[65] C. Li, R. Yerneni, V. Vassalos, H. Garcia-Molina, Y. Papakonstantinou, J. Ullman, and M. Valiveti. Capability Based Mediation in TSIMMIS. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 564–566, June 1998.

[66] E.-P. Lim, J. Srivastava, S. Prabhakar, and J. Richardson. Entity Identification in Database Integration. In *International Conference on Data Engineering*, pages 294–301, Los Alamitos, Ca., USA, April 1993. IEEE Computer Society Press.

[67] E.-P. Lim, J. Srivastava, and S. Shekhar. Resolving Attribute Incompatibility in Database Integration: An Evidential Reasoning Approach. In *Proceedings of the 10th International Conference on Data Engineering*, pages 154–165, Houston, TX, February 1994. IEEE Computer Society Press.

[68] W. Litwin and A. Abdellatif. An overview of the multidatabase manipulation language MDSL. In *Proceedings of the IEEE*, pages 69–73, May 1987.

[69] W. Litwin, L. Mark, and M. Roussopoulos. Interoperability of Multiple Autonomous Databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.

[70] L. Liu, W. Han, D. Buttler, C. Pu, and W. Tang. An XML-based Wrapper Generator for Web Information Extraction. *SIGMOD Record*, 28(2):540–543, June 1–3 1999.

[71] D. Maier, M. Vardi, and J. Ullman. On the Foundations of the Universal Relation Model. *ACM Transactions on Database systems*, 9(2):283–308, June 1984.

[72] P. McBrien and A. Poulovassilis. A Formalisation of Semantic Schema Integration. *Information Systems*, 23(5):307–334, April 1998.

[73] P. McBrien and A. Poulovassilis. Automatic Migration and Wrapping of Database Applications - A Schema Transformation Approach. In *Conceptual Modeling - ER '99, 18th International Conference on Conceptual Modeling, Paris, France, November, 15-18, 1999, Proceedings*, volume 1728 of *Lecture Notes in Computer Science*, pages 96–113. Springer, 1999.

[74] A. Mendelzon, G. Mihaila, and T. Milo. Querying the world wide web. *International Journal on Digital Libraries*, 1(1):54–67, April 1997.

[75] J. Meseguer and X. Qian. A Logical Semantics for Object-Oriented Databases. *ACM SIGMOD Record*, 22(2):89–98, June 1993.

[76] Microsoft. Microsoft BizTalk Server - Whitepaper. Technical Report http://www.microsoft.com/biztalk/default.htm, Microsoft, May 1999.

[77] Microsoft. BizTalk Framework 2.0 - Independent Document Specification. Technical Report http://www.microsoft.com/biztalk/default.htm, Microsoft, December 2000.

[78] G. Miller, R. Beckwith, C. Fellbaum, D. Gross, and K. Miller. Five Papers on WordNet. Technical Report CSL Report 43, Cognitive Systems Laboratory, Princeton University, 1990.

[79] R. Miller, Y. Ioannidis, and R. Ramakrishnan. Schema Equivalence in Heterogeneous Systems: Bridging Theory and Practice. *Information Systems*, 19(1):3–31, January 1994.

[80] A. Motro and Q. Yuan. Querying database knowledge. *SIGMOD Record*, 19(2):173–183, June 1990.

[81] W. Ogden and S. Brooks. Query Languages for the Casual User: Exploring the ground between Formal and Natural Languages. In *Proceedings of the Annual Meeting of the Computer Human Interaction of the ACM*, pages 161–65, 1983.

[82] Y. Papakonstantinou and V. Vassalos. Query Rewriting for Semistructured Data. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 455–466, 1999.

[83] C. Parent and S. Spaccapietra. Issues and approaches of database integration. *Communications of the ACM*, 41(5es):166–178, May 1998.

[84] R. Pledereder, V. Krishnamurthy, M. Gagnon, and M. Vadodaria. DB Integrator: Open Middleware for Data Access. *Digital Technical Journal of Digital Equipment Corporation*, 7(1):7–22, Winter 1995.

[85] M. Roth and P. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pages 266–275, 1997.

[86] M. Rusinkiewicz, A. Sheth, and G. Karabatis. Specifying Interdatabase Dependencies in a Multidatabase Environment. *IEEE Computer*, 24(12):46–53, December 1991.

[87] Y. Sagiv. A Characterization of Globally Consistent Databases and Their Correct Access Paths. *ACM Transactions on Database Systems*, 8(2):266–286, June 1983.

[88] F. Saltor, M. Castellanos, and M. Garcia-Solaco. On Canonical Models for Federated DBs. *SIGMOD Record*, 20(4):44–48, December 1991.

[89] I. Schmitt and G. Saake. Merging Inheritance Hierarchies for Database Integration. In M. Halper, editor, *Proceedings of the International Conference on Cooperative Information Systems, CoopIS'98*, pages 322–331. IEEE Computer Society Press, 1998.

[90] E. Sciore, M. Siegel, and A. Rosenthal. Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems. *ACM Transactions on Database Systems*, 19(2):254–290, June 1994.

[91] A. Sheth and G. Karabatis. Multidatabase Interdependencies in Industry. In *Proceedings of the 1993 ACM SIGMOD Conference on Management of Data*, pages 483–486, June 1993.

[92] A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogenous and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[93] D. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, March 1981.

[94] M. Siegel and S. Madnick. A Metadata Approach to Resolving Semantic Conflicts. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 133–145, September 1991.

[95] John F. Sowa. Top-level ontological categories. *International Journal of Human-Computer Studies*, 43(5):669–685, 1995.

[96] M. Stonebraker, J. Chen, N. Nathan, C. Parson, A. Su, and J. Wu. Tioga: A Database-Oriented Visualization Tool. In *Proceedings of the Visualization '93 Conference*, pages 86–93, San Jose, CA, October 1993. IEEE Computer Society Press.

[97] C. Thieme and A. Siebes. An approach to schema integration based on transformations and behaviour. In *Proceedings of CAiSE'94*, pages 297–310, June 1994.

[98] J. Ullman. Information Integration Using Logical Views. In Foto N. Afrati and Phokion Kolaitis, editors, *Database Theory—ICDT'97, 6th International Conference*, volume 1186 of *Lecture Notes in Computer Science*, pages 19–40, Delphi, Greece, 8–10 January 1997. Springer.

[99] G. Vossen and J. Yacabucci. An extension of the database language SQL to capture more relational concepts. *SIGMOD Record*, 17(4):70–78, December 1988.

[100] W3C. Extensible Markup Langauge (XML) 1.0. Technical Report http://www.w3.org/XML/, February 1998.

[101] D. Weishar and L. Kerschberg. Data Knowledge packets as a Means for Supporting Semantic Heterogeneity. *SIGMOD Record*, 20(4):69–73, December 1991.

[102] D. Woelk, P. Attie, P. Cannata, G. Meredith, A. Sheth, M. Singh, and C. Tomlinson. Task scheduling using intertask dependencies in Carnot. *SIGMOD Record*, 22(2):491–494, June 1993.

[103] R. Yerneni, C. Li, H. Garcia-Molina, and J. Ullman. Computing Capabilities of Mediators. *SIGMOD Record*, 28(2):443–454, June 1999.

Appendix A

Unity Standard Dictionary Classes

```
class CGD : public CObject
{ // Global dictionary
   CTypedPtrList<CObList, CGD_node*> nodes; // List of GD nodes
   CGD_node*    root;           // Root node of tree
   CDocument*   GDDoc;          // Pointer to document object for this GD
   POSITION     aPos;           // Position variable used for iterator
   int          max_depth;      // Maximum depth to display tree
   int          node_types;     // Types of nodes/links to display
};

class CGD_node : public CObject
{ // Global dictionary node
   CString      key;            // Key for node: sname-def_num e.g. Node-1
   CString      sname;          // Semantic name of term
   int          def_num;        // Definition number of term
   CString      desc;           // Description of semantic term
   CList<CString,CString&> syn; // Synonyms for semantic term
   CGD_link     parent;         // Link to node parent - NULL if none
   CTypedPtrList<CObList,CGD_link*> children; // Pointer to children
   CRect        loc;            // Location of node on screen
   int          level;          // Level of node (only valid after BFS called)
   bool         visited;        // True if node has been visited in BFS
   CPoint       pt;             // Top of rectangle used for calculations
   bool         visible;        // True if node should be currently visible
};
```

```
class CGD_link : public CObject
{ // Global dictionary link
   long         loc;            // Not in use
   int          link_type;      // Type of link 1 - IS-A, 2 - Part Of
   double       value;          // Link value/weight
   bool         condense;       // True if often collapsed into parent
   CGD_node*    from_node;      // Origin of link
   CGD_node*    to_node;        // Destination of link
   CPoint       from_pt;        // From point of link when drawing on screen
   CPoint       to_pt;          // To point of link when drawing on screen
};
```

Appendix B

Unity Metadata Classes

```
class CMDField : public CObject
{ // Metadata field
   CString      field_type;     // Type of field (integer, string, etc.)
   long         size;           // Field size
   int          num_decimal;    // # of decimals in field
   int          precision;      // Field precision
   bool         required;       // True if field is required in table
   bool         empty_str;      // True if empty string is allowed for a value
   CString      comment;        // Comment
};

class CMDTable : public CObject
{ // Metadata table
   CTime        creation_date;  // Table creation date
   CTime        last_updated;   // Last update date
   long         record_count;   // # of records in table
   long         record_size;    // Size of a record
   CString      comment;        // Comment
   CTypedPtrList<CObList, CMDField*> fields;
   CTypedPtrList<CObList, CKey*> keys;
   CTypedPtrList<CObList, CJoin*> joins;
   POSITION     aPos,aPos2,aPos3; // For iterators
   CString      name;           // System name for table
};
```

```
class CMDSource : public CObject
{ // Metadata source
   POSITION     aPos;           // For iterator
   CString      name;           // Name for source
   CString      location;       // Source location/path
   CString      comment;        // Comment
   CString      group_name;     // Database group name
   CString      org_name;       // Database organization name
   CString      region_name;    // Database region name
   CString      national_name;  // Database national name
   CString      intl_name;      // Database international name
   CString      ODBC_name;      // Database ODBC connection name
   CString      connect_str;    // Database ODBC connection string
   CTypedPtrList<CObList, CMDTable*>    tables;
};
```

Appendix C

Unity X-Spec Classes

```
class CSpecField : public CObject
{ // X-Spec Field
   CString      sys_name;       // System name for field
   CString      field_type;     // Field type
   long         size;           // Field size
   int          num_decimal;    // # of decimal places
   int          precision;      // Precision
   bool         required;
   bool         empty_str;
   CString      comment;
   bool         is_key;         // True if key value
   bool         is_categorizer; // True if value is a categorizer
   bool         is_foreign_key; // True if field is a foreign key
   bool         is_reference;   // True if field used to reference other DBs
   bool         is_datetime;    // True if date time field
   bool         is_numeric;     // True if numeric field
   double       low_val;
   double       high_val;
   bool         requires_norm;  // True if field is a duplicate and can be normalized
   CString      parent_name;    // Parent group name
   bool         placed;         // Temporary field used to determine if field has been used
   CQry_FRef    *qfref;         // Pointer to a query field reference for this field
   CSpecTable*  parent_tbl;     // Parent table for this field
   CString      sem_name;       // Semantic name for field
};

class CPath : public CObject
{ // Path class (for shortest join path calculations)
   CTypedPtrList<CObList,CSpecTable*> nodes;
   CTypedPtrList<CObList,CJoin*> joins;
};
```

```
class CSpecTable : public CObject
{ // X-Spec Table
   CString      sys_name;       // System name for table
   int          scope;          // Table scope
   CTime        creation_date;
   CTime        last_updated;
   long         record_count;
   long         record_size;
   int          access_mech;    // Read-only, write-only, read-write
   int          rec_type;
   int          rec_grouping;
   bool         allow_duplicates;
   CString      comment;
   CTypedPtrList<CObList, CSpecField*> fields;
   CTypedPtrList<CObList, CKey*> keys;
   CTypedPtrList<CObList, CJoin*> joins;
   POSITION     aPos,aPos2,aPos3;
   CString      sem_name;       // Semantic name for table
};

class CSpec : public CObject
{ // X-Spec class
   POSITION     aPos;
   CString      name;           // Specification name
   CString      db_name;        // Database name
   CString      location;       // Database location
   CString      comment;
   CString      group_name;     // Database group name
   CString      org_name;       // Database organization name
   CString      region_name;    // Database region name
   CString      national_name;  // Database national name
   CString      intl_name;      // Database international name
   CString      ODBC_name;      // Database ODBC connection name
   CString      connect_str;    // Database ODBC connection string
   CTypedPtrList<CObList, CSpecTable*>  tables;
   CPath***     path_matrix;    // Stores shortest calculated join paths
   int          last_nodes;     // Last size of matrix
};

class CKey : public CObject
{ // CKey class
   int          scope;          // Scope of key: 1-Table, 2-Database, 3-Division/Group
                                //  4-Organization, 5-Regional, 6-National, 7-International
   int          type;           // Key type: 1-Primary key, 2-Secondary Key, 3-Foreign Key
   CString      name;           // Key name
   CTypedPtrList<CObList,CObject*> flds;        // Fields of the key (order is significant)
                                // Contains CMDField* (for MDSource documents) or
                                //   CSpecField* for CSpec documents
   POSITION     aPos;
};
```

```
class CJoin : public CObject
{ // CJoin Class
   CString      from_table;     // Join originating table
   CString      to_table;       // Join to table
   CKey         *from_key;      // Join from key
   CKey         *to_key;        // Join to key
   int          type;           // Type of join (cardinality): 1 - 1:1, 2 - 1:N, 3 - N:1,
                                //  4 - N:M, 5 - 1:C, 6 - C:1, 7 - C:D (from:to)
   int          from_card;      // From cardinality (for 5-7) if explicit cardinality
   int          to_card;        //  eg. 1:3, 3:1
   CString      name;           // Join name
};
```

Appendix D

The Integration Algorithm

```
void CSchDoc::OnSchAddall()
// Add all elements in the currently selected specification source into the view
{
   CSpecTable   *tbl;
   CSpecField   *fld;
   CSpec        *spec = (this->Get_SPV())->get_spec();
   CSpecRef     *spr;

   if (spec == NULL)
      return;

   // Root item - name of specification
   if (schema.is_null())
      schema.add_root();

   // Add schema context information
   schema.add_context(7,spec->get_intname());
   schema.add_context(6,spec->get_natname());
   schema.add_context(5,spec->get_regname());
   schema.add_context(4,spec->get_orgname());
   schema.add_context(3,spec->get_grpname());
   schema.add_context(2,spec->get_db_name());

   // Now add each one of the tables in the specification to the schema
   spec->init_iterator();
   while (spec->next_elt(tbl))
   {
      schema.add_table(tbl);

      // Now add each field of a table in the specification to the schema
      tbl->init_iterator();
      while (tbl->next_elt(fld))
         schema.add_field(fld);
   }

   // Add this specification to the list of specs in the schema
   spr = new CSpecRef;
   spr->spec_loc = (this->Get_SPV())->get_specdoc()->get_fname();
   spr->specdoc = new CSpecDoc;
   spr->specdoc->load_spec(spr->spec_loc);
   schema.AddSpec(spr);
}
```

```
bool CSchema::add_table(CSpecTable *tbl)
// Adds a table element from a specification to the schema
{
   CList<CString,CString&> terms;
   parse_sname(tbl->get_sem_name(),terms);
   match_sname(tbl,1,terms,this->get_root(),terms.GetHeadPosition());
   return true;
}

bool CSchema::add_field(CSpecField *fld)
// Adds a field element from a specification to the schema
{
   CList<CString,CString&> terms;
   parse_sname(fld->get_sem_name(),terms);
   match_sname(fld,2,terms,this->get_root(),terms.GetHeadPosition());
   return true;
}

int CSchema::match_sname(CObject *obj, int type, CList<CString,CString&> &terms,
                              CSch_node *cur_node, POSITION aPos)
/*
Matches a full semantic name separated into terms with a cur_node's
children recursively.
Adds references to DB list as proceeds and adds all terms regardless
of amount of matches. Type is 1 if table, 2 if field
Retuns:
- 1 - Perfect match of all terms with IV
- 2 - SN matches up to a certain point (remaining terms added)
- 3 - No match at any level
*/
{
   CString      term, sn, new_sn, s;
   CSch_link    *link;
   CSch_node    *nd, *res_node;
   POSITION     next_pos;
   CSpecTable   *tbl = (CSpecTable*) obj;
   CSpecField   *fld = (CSpecField*) obj;

   if (aPos == NULL)
   {
      add_DB_map(cur_node, obj, type, spec);   // Add database reference for element
      return 1; // Matched all terms
   }

   if (cur_node == NULL)
      return 3;

   // Otherwise, match all children of cur_node with the current term
   term = terms.GetAt(aPos);
   cur_node->init_iterator();
   while (cur_node->next_elt(link))
   {
      nd = link->get_to_node();
      // Note: May want to use depth_sname eventually as it is more efficient
      // Trick: Get sname of node we are matching with and remove last character
      //    if it has children (is a context). Otherwise, leave name unchanged.
      // The newly formatted sname should be found in the name we are matching
      //    if there will be a match
         if (nd->type == 1)
            sn = nd->sname.Left(nd->sname.GetLength()-1);   // Context, remove ]
         else
            sn = nd->sname;
```

```
         if (type == 1)
            new_sn = tbl->get_sem_name();
         else
            new_sn = fld->get_sem_name();

         if (new_sn.Find(sn) != -1)
         { // Found sn in new_sn - match at this level, proceed to next level
           //   if a context and record DB info
            next_pos = aPos;
            s = terms.GetNext(next_pos);
            return match_sname(obj,type,terms,nd,next_pos);
         }
   }
   // There was no further match - add all remaining terms (including this one)
   this->add_node(cur_node,obj,type,terms,aPos,res_node);
   term = terms.GetNext(aPos);
   while (aPos)
   {
      term = terms.GetAt(aPos);
      cur_node = res_node;
      this->add_node(cur_node,obj,type,terms,aPos,res_node);
      term = terms.GetNext(aPos);
    }
    return 2;
}

bool CSchema::add_node(CSch_node *parent, CObject *obj, int type,
                           CList<CString,CString&> &terms,
                           POSITION aPos, CSch_node* &res_node)
// Adds a node under the parent, constructing a semantic name out
//   of the POSITION and terms variables
{
   CSch_link    *link = new CSch_link;
   CSch_node    *nd = new CSch_node;
   int          link_type = 1,p,node_type;
   CString      sname,s;
   bool         full_term;

   // Determine semantic name for the node at this depth
   s = terms.GetAt(aPos);
   full_term = false;
   if (type == 1)
   {
      node_type = 1;         // Storing info on a table -> name is a context
      sname = ((CSpecTable*) obj)->get_sem_name();
      p = sname.Find(s);
      sname = sname.Left(p+s.GetLength())+"]";
      if (aPos == terms.GetTailPosition())
         full_term = true;   // Last term - use whole semantic name
    }
    else
    {
       sname = ((CSpecField*) obj)->get_sem_name();
       p = sname.Find(s);
       node_type = 1;
       if (p+s.GetLength()==sname.GetLength()) // Use whole name
       {
          node_type = 2;     // Storing a context
          full_term = true;
       }
       else // Deriving context
          sname = sname.Left(p+s.GetLength())+"]";
    }
```

```
    // Initialize new node
    nd->sname = sname;
    nd->key = sname;
    nd->type = node_type;
    nd->depth = parent->depth+1;
    nd->depth_sn = s;
    nd->parent.link_type = link_type;
    nd->parent.to_node = nd;
    nd->parent.from_node = parent;
    res_node = nd;
    // Initialize display properties for node
    nd->data_type = 0;
    nd->display_result = 1;
    nd->display_width = 10;
    nd->format_str = "";
    nd->num_decimals = 0;


    // Insert node into global list of nodes
    nodes.AddTail(nd);
    if (full_term)
       add_DB_map(nd,obj,type,spec);     // Add mapping to DB element
    // Construct link type to add to parent
    if (parent != NULL)
    {
       // Assign values to child link
       link->from_node = parent;
       link->to_node = nd;
       link->link_type = link_type;
       parent->children.AddTail(link);

       if (parent->type == 2 && node_type != 2)
       { // Parent node is a concept, but this node is a context
          parent->type = 1; // Promote parent node to a context
       }
    }
    return true;
}
```

Appendix E

Unity Schema Classes

```
class CSch_link : public CObject
{ // Schema link class
   CSch_node*   from_node;
   CSch_node*   to_node;
   int          link_type;            // 1 = IS-A, 2 = HAS-A
};

class CSch_DBRec : public CObject
{ // Schema database reference - Semantic name to X-Spec mapping
   CString      db_name;
   CString      db_loc;
   CString      sem_name;
   CString      sys_name;
   CString      type;
};

class CSch_node : public CObject
{ // Schema node
   CString      key;                // Key for node: same as sname
   CString      sname;              // Full sem. name of term (with contexts)
   int          depth;              // Depth of term from root (root level=1)
   CString      depth_sn;           // Single term at this depth eg.[A,B,C] D
                                    // if depth=2 this would be B
   CSch_link    parent;             // Link to node parent - NULL if none
   CTypedPtrList<CObList,CSch_link*> children;  // Pointer to children
   POSITION     aPos,aPos2;         // Position indicator for iterator function
   int          type;               // 1 if context, 2 if concept
   CTypedPtrList<CObList,CSch_DBRec*> db_maps;  // Semantic to system mappings
   int          data_type;          // 0-string,1-int,2-real,3-percent,4-date
   int          display_width;      // Width of field when displayed
   CString      format_str;         // C++ formatting string for date/time
   int          num_decimals;       // Number of decimals to display
   int          display_result;     // 0 - if field is not displayed in result
};
```

```
class CSpecRef : public CObject
{ // X-Spec reference
   CString      spec_loc;           // Location of specification file
   CSpecDoc     *specdoc;           // Pointer to spec in memory
};

class CContext : public CObject
{ // Context class
   int          type;               // Context level: 1 - Table, 2 - Database,
                                    // 3 - Division/Group, 4 - Organization,
                                    // 5 - Regional, 6 - National, 7 - Intl.
   CString      name;              // Name at this context level
   bool         selected;           // True if context is selected for query
   int          loc;              // Temp. var. storing position in list
};

class CSchema : public CObject
{ // Schema (integrated view) class
   CTypedPtrList<CObList, CSch_node*> nodes;   // List of schema nodes
   CTypedPtrList<CObList, CSpecRef*> specs;    // List of X-Specs in schema
   CTypedPtrList<CObList, CContext*> contexts; // List of schema contexts
   CSch_node*   root;               // Root node of tree
   CDocument*   SchDoc;             // Ptr to document object for this schema
   POSITION     aPos,aPos2;         // Position variable used for iterator
};
```

Appendix F

Unity Query Classes

```
class CDT_node : public CObject
{ // CDT_node class (dependency tree node)
   CString      sname;               // Semantic name of node
   CSpecField   *fld;                // Pointer to specification field
   CString      sys_name;            // System name of field
   CTypedPtrList<CObList,CDT_node*> children;  // Children nodes
   CDT_node     *parent;             // Pointer to parent node
   POSITION     aPos;
   bool         use_node;            // Set when filtering out duplicates during merge
};

class CDepTree : public CObject
{ // CDepTree class
   CTypedPtrList<CObList,CDT_node*>  nodes;  // List of pointers to nodes
   POSITION     aPos;
   CDT_node     *root;
   CArray<CString,CString&> lsname;   // Semantic name of levels
};

class CDepPath : public CObject
{ // Class to store tree dependency path
   CTypedPtrList<CObList,CDT_node*>  nodes;  // List of pointers to nodes
   POSITION     aPos;
};

class CResSet : public CObject
{ // CResSet stores all the valid paths (attribute combinations) for a query
   CTypedPtrList<CObList,CDepPath*>  paths;  // List of pointers to paths
   POSITION     aPos;
};

class CTreeRef: public CObject
{ // CTreeRef - linking class for a dependency tree
   CDT_node     *node;
   CDepTree     *tree;
};
```

```
class CDepNRef : public CObject
{ // Dependence node linking class
   CString      sname;
   CString      sys_name;
};

class CQry_link : public CObject
{ // Query Link class
   CQry_node*   from_node;
   CQry_node*   to_node;
   int          link_type;            // 1 = IS-A, 2 = HAS-A
};

class CQry_node : public CObject
{ // Query node class
   CString      key;                  // Key for node: same as sname for now
   CString      sname;                // Full semantic name of term (with all contexts)
   int          depth;                // Depth of term from root (root level=1)
   CString      depth_sn;             // Single semantic name term at this depth
                                      // eg. [A,B,C] D if depth =2 this would be B
   CQry_link    parent;               // Link to node parent - NULL if none
   CTypedPtrList<CObList,CQry_link*> children;  // Pointer to children (indexed by key)
   POSITION     aPos,aPos2;           // Position indicators for iterator functions
   int          type;                 // 1 if context, 2 if concept
   int          data_type;            // 0-string,1-int,2-real,3-percent,4-date
   int          display_width;        // Width of field when displayed
   CString      format_str;           // C++ formatting string for date/time
   int          num_decimals;         // Number of decimals to display
   int          display_result;       // 0 - if field is not displayed in result
};

class CQConcept : public CObject
{ // The CQConcept class for retrieving concepts (semantic names) from data sources
   CString      sname;                // Semantic name for concept and a key
   int          field_pos;            // Column position of field in result (from 0)
   long         size;                 // Size of field in bytes (chars)
   char         type;                 // Type of field: X-Fixed char, V-VarChar, I-Integer
                                      //  L-Long, F-Float, D-double, B-Boolean, C-Currency
   CQry_node*   qnode;                // Pointer to corresponding query node (if available)
   bool         mapped;               // True if concept has been mapped
   bool         status;               // True if in query, false if marked for deletion
};

class CQry_FRef : public CObject
{ // The CQry_FRef class for referencing fields in a subquery
    CSpecField* fld_ptr;             // Pointer to specification field
    CQConcept*  concept_ptr;         // Concept that this field is mapped to
    int         fld_pos;             // Position in result list
};
```

```
class CQry_TRef : public CObject
{ // The CQry_TRef class for referencing tables in a subquery
    CSpecTable* tbl_ptr;             // Pointer to specification table
    bool        req_norm;            // True if table requires normalization in subquery
    CDepTree    *dep_tree;
    CResSet     *rset;
};

class CSubQConcept : public CObject
{ // The CSubQConcept class for referencing which concepts are in a subquery
    CQConcept*  concept_ptr;         // Pointer to concept in parent query
    CTypedPtrList<CObList, CQry_FRef*> fields;  // List of field references
    POSITION    aPos;
};

class CSubQry : public CObject
{ // The CSubQry class for accessing an individual data source
    CString     key;
    CString     db_name;
    CString     db_loc;
    CString     SQL_st;
    CString     ODBC_name;
    CDatabase   db;                  // Database variable
    bool        valid_result;        // True if query returned valid result
    CRecSet     *rset;               // Recordset
    CSpec*      spec;
    bool        has_duplicates;      // True if contains duplicates to be normalized

    CTypedPtrList<CObList, CQry_FRef*> fields;  // List of field references
    CTypedPtrList<CObList, CQry_TRef*> tables;  // List of table references
    CTypedPtrList<CObList, CJoin*>  joins;  // List of joins used
    CTypedPtrList<CObList, CSubQConcept*> concepts;  // List of concepts queried
    POSITION    aPos,aPos2,aPos3, aPos4; // Position variables used for iterators
};

class CQuery : public CObject
{ // The Query class itself
    CTypedPtrList<CObList, CQry_node*> nodes; // List of Query nodes
    CTypedPtrList<CObList, CSubQry*> subqry; // List of subqueries
    CTypedPtrList<CObList, CQConcept*> concept; // List of query concepts
    CTypedPtrList<CObList, CContext*> contexts; // List of schema contexts
    CSchDoc     *schdoc;             // Schema on which query is posed
    CString     sch_loc;             // File location of schema
    CQry_node*  root;                // Root node of tree
    CDocument*  qrydoc;              // Pointer to document object for this query
    POSITION    aPos,aPos2,aPos3;    // Position variable used for iterator
    CString     qry_criteria;        // Query criteria (WHERE clause)
    CList<CString,CString&> orderby;  // Order by list - list of semantic names
    bool        all_ctx[7];          // True if want all context instances at level
    CList<CString,CString&> ctx[7];   // Context instances by type
    CResultSet  *result_set;         // Final result set
};
```

```
class CRecSet : public CObject
{ // CRecSet class
    CTypedPtrList<CPtrList,void*> recs; // List of pointers to records (buffers)
    int         num_cols;             // Number of columns in recordset
    int         num_recs;             // Number of records in recordset
    long        rec_size;             // Record buffer size
    POSITION    aPos;
    int         *CTypeArray;          // Array of column types
    long        *ColLenArray;         // Array of column lengths
    long        *OffsetArray;         // Array of column offsets into record buffer
};

// Classes for the entire record set (stores integrated results for all local views)
class CElement : public CObject
{ // Base field instance for one record/column combo from ONE database
    void        *data_ptr;            // Pointer to data
    CString     data_st;              // Data in string format
    CSubQry     *source_sq;           // Source subquery for data
    int         source_col;           // Source column
    POSITION    source_row;           // Source row
    int         data_type;            // Data type
};

class CFVal : public CObject
{ // CFVal class - field value for a given row (integrated view)
    CList<CElement, CElement&> values;// All values from all DBs
    void        *cur_data_ptr;        // Current data value pointer
    CString     cur_data_st;          // Current data (string form)
};

class CRow : public CObject
{ // CRow class - stores one data row
    CArray<CFVal, CFVal&> fvals;      // Array of field values
};

class CResultSet : public CObject
{ // CResultSet class - stores all returned results
    CTypedPtrList<CObList, CRow*> rows; // Data rows
    CQuery      *query_ptr;
    POSITION    aPos;
    int         num_cols;
};
```

Appendix G

The Standard Dictionary

This section displays the main standard dictionary constructed as the product of many test integrations. The terms are organized hierarchically as the dictionary is a tree. Terms related to the parent term by an IS-A relationship are connected using a "-", whereas terms related using a HAS-A relationship are connected using a "=". If a definition is present in the dictionary for a term, it is enclosed in parentheses. Note that a dictionary term consists of a term name plus a "-" then a definition number. Also, since some of the terms have been re-arranged in the current dictionary as compared to how they were inserted for Northwind, the dictionary for the Northwind example is listed separately.

```
T-0 (Ultimate, root)
   - Physical-0 (Physical matter; tangible concepts)
      - Entity-0 (Physical objects)
         - Physical Object-0
            - Aggregate-0
               - Configuration-0
                  - Pile-0
                  - Heap-0
                  - Stack-0
               - Thing-0
                  - Inanimate-0
                     - Mechanical-0
                     - Structure-0
                        - Building-0
                     - Mineral-0
                     - Book-0 (a collection of sheets of paper bound together)
                        = Page-0 (Book page)
                        = Pages-0 (# of pages in book)
                  - Animate-0
                     - Animal-0
                        - Rational-0
```

```
                        - Human-0
                           - Man-0
                              - Ramon-0
                           - Woman-0
                              - Carri-0
                     - Irrational-0
                        - Beast-0
               - Plant-0
      - Stuff-0
         - Material-0
            - Solid-0
               - Gold-0
            - Liquid-0
               - Water-0
            - Gas-0
               - Air-0
         - Immaterial-0
            - Spirit-0
            - Energy-0
               - Light-0
               - Heat-0
               - Fire-0
   - Social Object-0
      - Group-0
         - Team-0
         - Family-0
      - Institution-0
         - Organization-0
            - Education-0
               - School-0
               - University-0
            - Health-0
               - Hospital-0
         - Company-0
            - Proprietorship-0
            - Partnership-0
            - Incorporated-0
               - Subsiduary-0
               - Parent-0
- Role-0 (Facts of relatedness)
   - Physical Object-1
      - Material-1
         - Commodity-0
      - Location-0
         - Property-0
      - Structure-1
         - Building-1
            - Terminal-0
      - Product-0
         - Garment-0
         = Quantity per Unit-0 (# of discreet items in a product package)
         = Discontinued-0
         - Freight-0
   - Social Object-1
      - Person-0
```

```
            - Employee-1
               - Attorney-0
               - Pilot-0
               - Sales person-0
               - Author-0
               - Supervisor-0 (Employee supervisor)
            - Action-0
               - Pedestrian-0
               - Source-0
                  - Reference-0
            - Social-1
               - Mother-0
               - Wife-0
            - Discretionary-0
               - User-0
               - Contact-0
               - Consumer-0
                  - Customer-0
                     - Claimant-0
         - Organization-1
            - Company-1
               - Transportation-0
                  - Shipper-0
                  - Consignee-0
                  - Transmittal-0
               - Finance-0
                  - Bank-0
               - Real Estate-0
                  - Landlord-0
               - Commerce-0
                  - Soldto-0
                  - Shipto-0
                  - Supplier-0 (Company who supplies goods)
               - Media-0 (Media companies)
                  - Publisher-0 (Publishing, media company)
   - Circumstances-0 (Mediating circumstances that bring entity and role together)
      - Physical Object-2
         - Transaction-0
            - Order-0
      - Social Object-2
         - Person-1
            - Occupation-0
               - Aviation-0
               - Legal System-0
               - Employment-0
            - Action-1
               - Pedestrianship-0
            - Social-0
               - Motherhood-0
               - Marriage-0
            - Discretionary-1
               - Relationship-0
                  - Business-0
                  - Friendship-0
         - Organization-2
```

```
            - Reason-0
               - Cause-0
   - Information-0 (Pure structure that does not depend on the objects it describes or the
recording medium)
      - Form-0 (Eternal objects)
         - Measure-0
            - Size-0
               - Length-0
               - Area-0
               - Volume-0
            - Color-0
            - Weight-0
         - Magnitude-0
            - Scales-0
               - Numeric-0
            - Total-0 (Total (amount, number, etc.))
            - Quantity-0
         - Locators-0
            - Physical Address-0
               - Address-0
                  - Secondary address-0
                  - Home address-0
                  - Mailing address-0
                  - Primary address-0
                  = Address line 1-0
                  = Address line 2-0
                  = City-0
                  = Region-0 (state or province)
                  = Country-0
                  = Postal Code-0
            - Virtual Address-0
               - E-mail address-0
               - Web address-0
               - Phone #-0
                  - Home Phone#-0
                  - Business Phone#-0
                  - Fax #-0
                  - Cell #-0
                  = Extension-0 (Phone # extension)
         - Identifiers-0
            - Id-0
               - Probill Number-0
               - Reference Number-0
               - Login id-0
               - Login password-0
            - SSN-0
            - ISBN-0 (ISBN book number)
         - Descriptors-0
            - Comment-0
            - Name-0
               - Person-2
                  = Title-1 (Person's title (Dr., Mr., Mrs.))
                  = Last Name-0 (Person's last name)
                  = First Name-0 (Person's first name)
               - Organization-3
```

```
      - Picture-0 (Picture (visual description))
      - Subject-0
      - Title-0 (an inscription placed over a thing by which that thing is known)
   - Categorizers-0
      - Category-0
   - Temporal-0
      - Time-0
      - Date-0
- Proposition-0 (Propositions about facts of relatedness)
   - Temporal-1
      - Date-1
         - Start date-0
         - End date-0
         - Entered date-0
         - Valid date-0
         - Invalid date-0
         - Filled date-0
         - Shipped date-0
         - Inspection date-0
         - Status date-0
         - Update date-0
         - Ordered date-0
         - Allocated date-0
         - Picked date-0
         - Birthdate-0
         - Required Date-0
         - Published Date-0 (Date material was published)
      - Time-1
         - Best time to call-0
   - Legal-0
      - Claim-0
   - Numeric-1
      - Amount-0
         - Gross amount-0
         - Net amount-0
         - Paid amount-0
      - Percent-0
         - Gross %-0
         - Net %-0
         - Discount %-0 (Percentage discount)
      - Value-0
         - Salvage value-0
         - Price-0
         - Inventory-0
            - Available to ship-0
            - Available to sell-0
            - On Order-0 (Quantity of product on order)
            - Reorder Level-0 (Minimum inventory before automatic re-ordering)
         - Cost-0
   - Territory-0
      - Country-1
   - Monetary-0
      - Currency-0
         - CDN dollar-0
         - US dollar-0
```

```
      - Theory-0 (Multiplicities and contrasts; Mediating relationships)
         - Mathematics-0
         - Accounting-0
            - General Ledger-0
            - Account-0
         - Transactional-0
            - Transaction-1
               - Payment-0
         - Financial-0
            - Credit-0
               - Credit Terms-0
               - Credit Aging-0
            - Tax-0
               - GST-0
         - Government-0
```

The dictionary constructed using the Northwind example is as follows:

```
T-0
   - Physical-0
      - Entity-0
         - Social Object-0
            - Company-0
      - Role-0
         - Discretionary-0
            - Customer-0
            - Contact-0
         - Work-0
            - Employee-0
            - Supervisor-0
         - Product-0
            - Freight-0
            = Quantity-per-Unit-0
            = Discontinued-0
         - Company-1
            - Transportation Company-0
               - Shipper-0
            - ShipTo-0
            - Supplier-0
      - Circumstance-0
         - Employment-0
         - Commerce-0
            - Transaction-0
               - Order-0
   - Information-0
      - Form-0
         - Categorizers-0
            - Category-0
         - Identifiers-0
            - Id-0 (identifier)
         - Descriptors-0
            - Name-0
               - Person Name-0
                  = First Name-0
                  = Last Name-0
```

```
            = Title-1
         - Description-0
            - Note-0
         - Picture-0
         - Title-0
      - Locators-0
         - Physical Address-0
            - Address-0
               = Address Line 1-0
               = City-0
               = Region-0 (state or province)
               = Postal Code-0
               = Country-0
         - Virtual Address-0
            - Phone #-0
               - Fax #-0
               - Home Phone #-0
               = Extension-0
            - Web Address-0
      - Temporal-0
         - Date-0
      - Numeric-0
         - Quantity-0
      - Total-0
   - Proposition-0
      - Temporal-1
         - Date-1
            - Birthdate-0
            - Required Date-0
            - Shipped Date-0
      - Monetary-0
         - Price-0
            = Discount %-0
         - Cost-0
         - Amount-0
      - Inventory-0
         = On-Order-0
         = Re-Order Level-0
   - Theory-0
```