Unity - A Database Integration Tool[1]
University of Calgary 2000-664-16
University of Manitoba TR-00-17

Ramon Lawrence
Advanced Database Systems Laboratory
Department of Computer Science
University of Manitoba
umlawren@cs.umanitoba.ca

Ken Barker
Advanced Database Systems and Applications Laboratory
Department of Computer Science
University of Calgary
barker@cpsc.ucalgary.ca

Abstract

The World-Wide Web (WWW) provides users with the ability to access a vast number of data sources distributed across the planet. Internet protocols such as TCP/IP and HTTP have provided the mechanisms for exchanging the data. 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 Internet 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.

This paper details the construction, function, and deployment of Unity, a database integration software package which allows database semantics to be captured so that they may be automatically integrated. Unity is the tool that we use to implement our integration architecture detailed in previous work. Our integration architecture focuses on capturing the semantics of data stored in databases with the goal of integrating data sources within a company, across a network, and even on the World-Wide Web. Our approach to capturing data semantics revolves around the definition of a standardized dictionary which provides terms for referencing and categorizing data. These standardized terms are then stored in semantic specifications called X-Specs which store metadata and semantic descriptions of the data. Using these semantic specifications, it becomes possible to integrate diverse data sources even though they were not originally designed to work together.

The centralized version of the architecture is presented which allows for the independent integration of data source information (represented using X-Specs) into a unified view of the data. The architecture preserves full autonomy of the underlying databases which are transparently accessed by the user from a central portal. Distributing the architecture would by-pass the central portal and allow integration of web data sources to be performed by a user's browser. Such a system which achieves automatic integration of data sources would have a major impact on how the Web is used and delivered.

Unity is the bridge between concept and implementation. Unity is a complete software package which allows for the construction and modification of standardized dictionaries, parsing of database schema and metadata to construct X-Specs, and contains an implementation of the integration algorithm to combine X-Specs into an integrated view. Further, Unity provides a mechanism for building queries on the integrated view and algorithms for mapping semantic queries on the integrated view to structural (SQL) queries on the underlying data sources.

1 Introduction

The World-Wide Web is a near endless source of information on almost every topic distributed across the globe. This information is exchanged and displayed using the standardized protocol TCP/IP and the standardized language HTML. However, the

[1]This research is partially sponsored by a Natural Science and Engineering Research Grant (RGP-0105566)

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.

Our integration architecture attempts to standardize the description of information. Behind most web sites is a database storing the actual data. Our goal is to capture the semantics of the data stored in each database, so that they may operate together. This work on capturing data semantics and integrating data sources has applications to companies and organizations with multiple databases which must work together. More importantly, 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.

Our model integrates database systems into a multidatabase using XML[21]. Unity is a software package which implements the models main components: a standardized dictionary of terms, X-Specs capturing database schema/metadata in XML, and an integration algorithm for combining X-Specs into an integrated view. Further, Unity implements query processing algorithms [15] for mapping from semantic to structural querying.

This paper briefly describes the integration architecture only as its relates to the implementation of Unity. Section 2 discusses the integration problem, and how capturing data semantics is fundamental to its solution. Previous work in the area is detailed in Section 3. Section 4 overviews the four components of Unity, and our integration architecture, which utilizes a standard dictionary for identifying similar concepts, a specification language (X-specs) used to exchange metadata on systems, and an integration algorithm for combining the metadata specifications together into a unified schema. A query processor provides for query execution, automatic conflict resolution, and results integration. The implementation of each of these components is discussed in following sections. We then briefly discuss the applications of our approach to the WWW. Two examples on how Unity is used to achieve integration are presented in Section 10. The paper closes with future work and conclusions.

2 Data Semantics and the Integration Problem

Integrating data sources involves combining the concepts and knowledge in the individual data sources into a "global view" of the data. The global view is a uniform, integrated view of all the knowledge in the data sources so that the user is isolated from the individual system details of the data sources. By isolating the user from the data sources and the complexity of combining their knowledge, systems become "interoperable", at least from the user's perspective, as they can access the data in all data sources without worrying about how to accomplish this task.

Constructing a global view of many data sources is difficult because they will store different types of data, in varying formats, with different meanings, and will be referenced using different names. Subsequently, the construction of the global view must, at some level, handle the different mechanisms for storing data (structural conflicts), for referencing data (naming conflicts), and for attributing meaning to the data (semantic conflicts).

Although considerable effort has been placed on integrating databases, the problem remains largely unsolved due to its complexity. 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). This paper will focus on schema-level integration. Schema integration is hard because at some level, both the operational and data semantics of a database need to be known for integration to be successful.

The schema integration problem is the problem associated with combining the diverse schemas of the different databases into a coherent global view by reconciling any structural or semantic conflicts between the component databases. Automating the extraction and integration of this data is difficult because the semantics of the data are not fully captured by its organization and syntactic schema.

Integrating data sources using schema integration is involved in constructing both a multidatabase system and a data warehouse. Both of these architectures are finding applications in industry because they allow users transparent access to data across multiple sites and provide a uniform, encompassing view of the data in an organization. On an even wider-scale, a standardized mechanism for performing schema integration would allow a user's browser to automatically combine data from multiple web sites and present it appropriately to the user. Thus, a mechanism for performing schema integration would be of great theoretical and practical importance.

The literature has proposed various methods for integrating data sources. However, the fundamental problem in these systems is the inability to capture data semantics. Automated integration procedures cannot be applied without a systematic way of capturing the meaning of the stored data. In this work, we propose a method for capturing data semantics which bridges work done in theory and the pragmatic approach used in industry. A standardized global dictionary is used to define words to reference identical concepts across systems. We then briefly show how a systematic method of storing data semantics (using these dictionary terms) can be used to integrate relational schemas.

3 Previous work

The database integration problem has been approached from two different ends by the database research community and the Internet standards/industrial communities. Although the approaches and applications of the two communities differ somewhat, they are both attempting to solve related problems. The overall goal is to get systems to interoperate by exchanging data in a meaningful, standardized form. In order for data to be exchanged, the semantics of the data must be known for the transfer to be successful.

The database community has attempted integration of data at the database level. They attempt to define algorithms for integrating database schema and transaction management protocols to achieve the necessary interoperation between systems. Work has been done on combining databases together using various architectures such as a multidatabase [4, 18] or a federated database [19]. These architectures divide the interoperability problem between databases into two 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 in any form (preserving full autonomy)

There has been some good work on transaction management in these architectures including Barker's algorithm [2] and serialization using tickets [11]. We have simulated several of these algorithms in previous work [16] and have determined via these simulations that Barker's algorithm is more efficient in practice. Schema integration has also been studied in depth, and appears to be a harder problem. The integration problem involves combining data from two or more different data sources, and is often required between applications or databases with widely differing views on the data and how it is organized. 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.

Previous research has focused on capturing metadata about the data sources to aid integration. This metadata can be in the form of rules such as the work done by Sheth [20], or using some form of global dictionary such as work done by Castano [5]. We believe that the best approach involves defining a global dictionary rather than using rules to relate systems. Rules are more subject to schema changes than a global dictionary implementation, and grow exponentially as the number of systems increase. Unfortunately, schema integration in the research community is a largely unsolved problem because of the difficulty in capturing data semantics.

Mediator and wrapper based systems such as Information Manifold [13], Infomaster [10], TSIMMIS [17], and others [1, 7, 3] do not tackle the schema integration problem directly. These systems focus on answering queries across a wide-range of data sources including unstructured data sources such as web pages and text documents. Some of these systems [1] do not attempt to provide an integrated view. Rather, information is integrated based on best-match algorithms or estimated relevance. Systems such as TSIMMIS [17] and Infomaster [10] that do provide integrated views typically construct these integrated views using designer-based approaches. Then, integrated views are mapped using a query language or logical rules into views or queries on the individual data sources. There is limited discussion on how the integrated global views and the associated mappings are actually created. This is probably because 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.

Thus, wrapper and mediator systems do not solve the schema integration problem. These systems solve the interoperability problem by providing mappings from a global view to individual source views and combining query results. To solve the schema integration problem, an automatic algorithm for producing the global view from data source information is necessary. This work proposes an automatic schema integration architecture based on removing all naming conflicts by utilizing a standard dictionary of terms to describe schema element semantics.

Internet and industrial standards organizations have taken the more pragmatic approach to the integration problem by standardizing the definition, organization, and exchange mechanisms for data in such a way that it can be communicated effectively. Instead of attempting to integrate entire database systems, their goal is to standardize the exchange of data from one system to another. This is a more straightforward problem because it is widely accepted that communications occur using standardized protocols and structures. Instead of determining data semantics for an entire system, it is only necessary to capture data semantics for just the data required in the communication, and format it using a recognized standard during transmission.

Work on capturing metadata information in industry has resulted in the formation of standardization bodies for exchanging data. An early data exchange standard was 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 extensable markable language (XML) and standardized XML schemas based on it. Microsoft has lead a consortium promoting BizTalk[8, 9] 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 has defined the Metadata Interchange Specification (MDIS) version 1.1 [6] as an emerging standard for specifying and exchanging metadata. Other standardization efforts include the Standard Interchange Language (SIL)[12].

It is important to notice the fundamental differences between the database research approach and the industrial approach. The research approach aims at defining mechanisms for combining entire database systems so that they can be queried and updated 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 has evolved on need rather than concept as businesses required the daily exchange of data between systems to do business. This has resulted in the definition of standards for communication. The important distinction is that the communication between systems is standardized not the actual data contained in the systems. These systems do not attempt transaction management 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 within a company because they do not capture the entire database semantics.

Our approach is to combine the two techniques. The goal is to integrate entire database systems by utilizing standardized dictionaries and languages developed in industry. On top of these standardized communication mechanisms, a multidatabase architecture is built allowing SQL-type queries and updates of the underlying, autonomous database systems utilizing the algorithms developed in the research community.

4 The Four Components of Unity

Our overall integration architecture has four components necessary to achieve it: a standardized dictionary of terms, 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. Our integration architecture [14] and query system [15] have been described in other work.

Unity is written in Visual C++ 6 and is intended for deployment on a Windows NT platform. The entire functionality of our 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, to begin the initial integration steps. The integration algorithm itself is highly portable C++ code and 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, a X-Spec document, an integrated view document, and a query document. Any combination of these four types of documents can be open in Unity at the same time and manipulated.

5 The Global Dictionary Editor

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 to allow the computer to determine semantically equivalent expressions.

The fundamental basis for our integration architecture is a standardized dictionary of terms which are used to construct XML tags to represent data semantics. The dictionary itself is defined and transmitted using XML. The standard dictionary is not a standardized schema which forces all data to be represented in one form and allows no flexibility. Rather, the dictionary simply consists of a set of phrases which represent concepts. These phrases are then combined into a semantic name for a data element. The semantic name is used as a XML tag for the data element to represent its semantics.

The standard dictionary is organized as a tree of concepts. All concepts are placed into the tree and are related using two types of relationships: 'IS-A' relationships and 'HAS-A' relationships. IS-A relationships are the standard subclass/superclass type of relationships and are used to model generalization/specialization data concepts. For example, consider two dictionary terms `customer` and `claimant`. In the dictionary, `claimant` would be a subclass of `customer` as presumably a claimant is a special type of customer. Component relationships relate terms using a 'Part of' or 'HAS A' relationship. For example, an address has (or may have) city, state, postal code, and country components. Similarly, a person's name may have first, last, and full name components. These components relationships are intended to standardize how these common concepts with subcomponents are represented.

Our initial dictionary contains a limited set of concepts commonly stored in databases.

Obviously, the definition of a dictionary storing all concepts is very difficult and time-consuming. Although we constantly expand the dictionary, it is not feasible to assume all concepts will be present as new types of data and very specialized data would not originally appear in the dictionary. Thus, we allow an organization to add nodes to the global dictionary to both the concept hierarchy and component relationships to capture and standardize names used in their organization which are not in the standardized global dictionary. These additional links are stored and transmitted along with the metadata information during integration. We expect that the evolution of the dictionary would be directed by some standardization organization to insure that new concepts are integrated properly over time.

The global dictionary editor component of Unity allows a user to create and maintain a standardized dictionary of terms. We have used this component of Unity to construct our basis dictionary and continue to add terms to it as required.

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 of the tree, 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 are used when the structure is graphically displayed, 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 uniquely identified. The unique key is constructed by combining the semantic name of the node with its definition number in the form: $key = sem\_name + " - " + str(def\_num)$. The semantic name of a node is the term name. It is probably an English word or phrase. The definition number is added by the system to insure 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. Node one would have a key of `Order-0` as its definition number would be `0`, and node two would have a key `Order-1` as its definition number would be `1`. 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 just one node would be used for the term "order", it may not be completely obvious what connotation of order should be used. Using this system, there is no 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 children 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 free-space. Once a node is created, it is first added to the list of all nodes in the `CGD` structure.

Then, the node is added to the list of children 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 children nodes. Finally, 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 the above classes can be found in Appendix A.

The previous paragraphs presented a good overview on the global dictionary implementation in terms of data storage, access, and updates. In order to simplify these procedures, updating the global dictionary is done through a GUI. 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.

The `CLeftView` class is derived from the `CTreeView` class which represents a tree-structure of items similar to how Windows Explorer displays the 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` will bring up a pop-up menu to add a new root only if the tree is empty, and right-clicking on empty space in `CGDView` brings up 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. At this time, the user is unable to move the display nodes, although this is seldom necessary as the tree is displayed in a very organized manner. The user can select nodes and links by left or right clicking on them, and they become highlighted in blue. A right click on a node or link brings up 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. Finally, 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. Also, 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 pop-up menu re-displays the entire tree from the normal root.

When one view is updated, the other view is also updated. Further, 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`).

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

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 basis standard dictionary, but allow organizations to add terms as required. Unlike a standardized 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. Rather, the context of the term as it is used in the metadata specification describing the database determines its meaning. Unity contains a global dictionary editor which allows the user to add, remove, and change terms. Each dictionary term consists of a standardized name, definition number, and text definition along with synonym and system name information.

6 The X-Spec Editor

Our definition of a standardized dictionary by itself is not enough to achieve integration because the dictionary is not defining a standard schema for communication; it is simply defining terms used to represent data concepts. These data concepts can be represented in vastly different ways in various data sources, and we are not assuming 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. The database schema is organized in relational form with tables and fields as basic elements.

A X-Spec consists of the relational database schema being described along with additional information about keys, relationships, and field semantics. More importantly, each table and field in the X-Spec has an associated name built from terms in the standardized dictionary. These semantic names are used to integrate X-Specs (based on matching names), and to provide a mechanism for formulating queries so that the user does not have to know the exact system names used.

Unity allows a user to quickly and easily construct a X-Spec for a given database. Unity contains a specification editor whose purpose is to parse existing database relational schema and format the schema information into a X-Spec. Then, 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 standardized dictionary to form a semantic name for the element. The semantic name is intended to capture the semantics of the data element using the standardized terms. During integration, the integration algorithm will parse this semantic name to determine how to combine the data 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 integration software is responsible for mapping the semantic names back to system names before the actual execution of a transaction.

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 [8] schemas, EDI, and other approaches, the required context is assumed because a standard dictionary/schema only applies to a very limited communication domain. For example, there is separate schema 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 provided both context and concept information. For our system, by utilizing a single dictionary, it is necessary to combine dictionary terms to provide both context and concept information.

The definition of a semantic name using dictionary terms follows this structure:

$semantic\_name = "[" \ CT \ [\,[;CT] \mid [,CT]\,] \ "]" \ \ [CN]$

$CT =< dictionary\_term >, CN =< dictionary\_term >$

That is, a semantic name consists of an ordered set of context terms separated by either a comma or a semi-colon, and an optional concept name term. Each context term and concept term is a single term of the standardized dictionary. The comma between terms A and B (A,B) represents that term B is a subtype of term A. A semi-colon between terms A and B (A;B) means that term A HAS-A term B, or term B represents a concept that is part of term A. The context terms are used to provide a context framework for the concept that describes them. Every semantic name should have at least one context term to provide this context framework for its semantics. The concept name is a single, atomic term describing the lowest level semantics. Fields have concept names to represent their base meaning void of any context information.

Consider, a database which stores information on orders and ordered items. A semantic name for the order table is `[Order]`. The semantic name for the ordered items table is `[Order;Product]`, as the ordered item table is storing ordered item records which are part of an order. The database schema storing order information has the structure *Order(id,customer,total_amount)* and *Order_item(order_num,item_id,quanity,price,amount)*. The X-Spec in Figure 1 conveys this information.

The resulting X-Spec is a XML schema storing the order database schema. The additional tags `sys_name` and `sys_type` are used to define the system name for the database element and indicate if it is a table or a field. The only other difference is the names assigned to the individual elements are semantic names consisting of multiple terms from the standardized dictionary. The semantic name uniquely identifies the element and provides both a concept term and context information. This X-Spec would normally be augmented with additional metadata about the field types and sizes.

Before we describe how X-Specs are created and modified in Unity, it is necessary to describe the data structure that they represent. Before a X-Spec is created, a data source schema must be parsed and translated. A relational schema is first translated

```
<?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] Name" sys_name = "customer" sys_type = "Field"/>
        <element type = "[Order;Total] Amount"  sys_name = "total_amount" sys_type = "Field"/>
        <element type = "[Order;Product]" maxOccurs="*"/>
</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 1: Order Database X-Spec

into a metadata document (`CMDDoc`) which contains the necessary metadata information (`CMDSource`) which 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 a X-Spec to describe the data source. Classes are used to display field properties (`CMDFldPropDlg`), table properties (`CMDTblPropDlg`), define keys (`CMDKeysDlg`), define joins (`CMDJoinsDlg`), and overall source properties (`CMDTblPropDlg`). The C++ definitions for these data classes are in Appendix B.

X-Specs are contained in specification documents (`CSpecDoc`) which contain the specification class (`CSpec`). Specification documents can be loaded and saved using MFC serialization routines. A X-Spec is represented by a `CSpec` class which contains all the information that a metadata class does plus additional information such as semantic names for the elements. Further, dialog screens allows specification of keys (`CSpecKeysDlg`) and joins (`CSpecJoinsDlg`). `CKey` and `CJoin` are classes for storing key and join information. A `CSpecDoc` has two views. 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.

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, a 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 some `CSpecTable` and represent specification fields (`CSpecField`) which map to data source table fields. The structure of 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. This is done by selecting the `Set GD` menu item, and finding a saved global dictionary file on disk. Once a 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 their construction of a semantic name.

The construction of a semantic name is performed by the class `CSnameDlg`. This dialog class is responsible for allowing the user to search for a dictionary term and combining 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 a 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 done 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. Generally, a semantic name for a table should not have a concept name, and the semantic name for a field should have one. Once semantic names are assigned to all fields and tables in a X-Spec, sufficient information is present for the X-Spec to be integrated by the integrated algorithm.

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 include Oracle, Sybase, and others using ODBC. However, we would like to implement a direct ODBC connection to the various databases to retrieve schema information. Also, work will be performed on remembering previous mappings from system names to a semantic name. 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.

In summary, a X-Spec is database schema and metadata information on a given data source encoded in XML. Although we use XML for transmission, a X-Spec is more than XML because it uses XML tags to capture metadata. A X-Spec is different than a BizTalk schema because it is not intended as a standardized schema for data communication. Rather, 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.

7 The Integration Algorithm

Integration algorithms for industrial systems such as EDI and BizTalk are relatively trivial because there is totally 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 only must 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. It may represent considerable amount of work to transform a company's data into the correct standard format. It also means that definitions of the standard schemas is increasingly complex.

Our integration algorithm is more complex because it must handle uncertainty. A X-Spec describing a database is not guaranteed to exactly match the X-Spec describing an almost identical database. There is 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 standardized dictionary to describe data elements.

Thus, integration is achieved not by the exact parsing of a document, but by matching standardized terms to relate concepts. The same term used in two different X-Specs is assumed to represent the identical concept regardless of its representation. This means that one X-Spec may represent a concept as a field while another represents it as a table. The integration algorithm must handle this, and organize the resulting integrated schema accordingly.

It is important to note the goal of the integration algorithm in this architecture. The algorithm is attempting to construct a unified schema of data sources rather than subsets of information contained in them or transmitted to other data sources. The integration algorithm receives as input one or more X-Specs describing database schema. It then uses the semantic names present in the X-Specs to match related concepts. If two related concepts are represented differently structurally, the algorithm must choose the correct representation or perform conflict resolution at query-time. Further, as the field types are not standardized, the integration algorithm will be responsible for field conversions. With sufficient metadata, field conversions by type, scale, or size are relatively straightforward and will not be discussed further here.

The product of the integration algorithm is a tree-structure representation of the integrated, unified schema. The unified schema is represented as a tree of entities with relationships linking the elements together. Users query the unified schema by choosing fields to include in the result, and the system handles the necessary joins based on the relationships between the data elements. This hiding of the relationships between elements is standard practice in high-level query tools such as those used for building data warehouse reports.

The integration algorithm is a straightforward term matching algorithm. The C++ code is provided in Appendix D, but the algorithm is easier described using an example. Let's look at the order database example. Consider starting with an empty integrated view V. The semantic name `[Order]` with one term is automatically added to V as it is empty. Then, when we attempt to add `[Order;Product]`, the first term `[Order]` matches with the order term already in V. The algorithm then attempts to find the term `[Product]` in V under `[Order]` which it cannot, 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 an integrated view V (see Figure 2).

Given this structural view of the data, a user issues queries on the integrated view simply by choosing which fields should be displayed in the final result. The required

```
V (view root)
    - [Order]
        - Id
        - [Customer]
            - Name
        - [Total]
            - Amount
        - [Product]
            - Id
            - Quantity
            - Price
            - Amount
```

Figure 2: Integrated View

joins between the tables are automatically inserted by the query processor. This example illustrates the integration of a X-Spec with an empty integrated view, but it is no more complex to now integrate with another 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 into V, the number of fields and concepts would grow, but assuming the semantic names are properly assigned, the integration procedure would be unchanged.

After integration, for each semantic name in V, there is one or more database elements that stores data relating to the semantic name. When the user issues a transaction against V, it is up to the query processor to generate SQL statements to access the appropriate data in the individual data sources. This is possible because the query processor has the schemas of all databases, knows all the data sources which store information on a given semantic name, and has a mapping from semantic names to system names. Thus, the query processor replaces the semantic names in the SQL query with the appropriate system names for a data source.

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. 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 a 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 as described above and update the schema view on the left-hand side.

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 an arbitrary depth, as there can be an arbitrary number of context terms in a semantic name. Correspondingly, a X-Spec typically 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.

8 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 [15] 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 the appropriate joins as required. The C++ definitions for these and other related query classes are in Appendix F. 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.

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. 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. Currently, the system does not support selection criteria for a query.

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 function out of the pop-up menu. The system insures that a semantic name cannot be added twice to the query.

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 ODBC, and normalizes and integrates the results as required. The results are then displayed in a result dialog (`CQryResDlg`). Results are stored in a record set structure (`CRecSet`) as they are returned via ODBC.

The algorithms for generating the SQL queries and performing normalization and integration are complex and will not be covered in detail here. They are available from other sources [15]. Most of these algorithms are implemented in the `CSpec` or `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 which 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 becomes multiple normalized rows in the query result. Currently, `CQryResDlg` is responsible for normalizing the result rows, although a more formal procedure is work in progress.

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. Joins across databases and multidatabase conflict resolution are currently active areas of study.

9 The Integration Architecture

The four components described in the previous section: a standardized dictionary of terms, X-Specs for storing database schema, an integration algorithm, and a query processor combine to form the basis of our integration architecture. The integration architecture actually consists of two separate and distinct phases: the *capture process*, and the *integration process*.

In the capture process, the X-Spec for a given data source is constructed, and the semantics of the data elements are mapped to semantic names consisting of terms from the standardized dictionary. This capture process is performed independently of the capture processes that may be occurring on other data sources because the only "binding" between individual capture processes at different data sources is the use of the dictionary to provide standardized 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 a X-Spec.

The integration process actually performs the integration of various data sources. For the purpose of this discussion, it is assumed that there is a central site where the integration is performed by combining the X-Specs of the data sources. Clients wishing to access the individual data sources submit their transactions to this central site which handles the necessary mappings and transaction management. Distributed integration and transaction management where there is no central integration site is an area of future work.

The key benefit to the two phase process is that the capture process is isolated from the integration process. This allows multiple capture processes to be performed concurrently and without knowledge of each other. Thus, the capture process at one data source is not affected by the capture process at any other data source. This allows the capture process to be performed only once, regardless on how many data sources may actually be integrated. This is a significant advantage as it allows application vendors and database designers to capture the semantics of their systems at design-time, and the clients of their products are able to integrate them with other systems with minimum effort.

The central site takes the X-Specs of the individual data sources and executes the integration algorithm to produce an integrated view. This integrated view is then provided to clients allowing them to issue SQL transactions against it. The integrated view displays to the user all elements described using their semantic names. A SQL query is then formed by the user using the semantic names. This query is sent to the central site which performs the necessary mapping from semantic names to system names and divides the query into subqueries against the data sources. The central site is assumed to implement the functionally of a MDBS manager which includes transaction management and query processing. Once results are returned from the individual data sources they are integrated based on the unified view and then returned to the user.

It is important to note that by the use of a central site no translational or wrapper software is required at individual data sources. Once the X-Spec has been provided for the data source and integrated by the central site, the software at the central site communicates directly with the data sources using ODBC or proprietary protocols. All translation, integration, and global transaction management is handled by software at the central site. This allows **full autonomy** of the underlying participating databases as the central site appears as just another client issuing transactions to the database.

10 Architecture Contributions

Our integration architecture contributes several new ideas:

- The definition of a hierarchical, standardized dictionary of terms which can be used across industries and organizations.
- A system for representing in XML entire database schema using the standardized dictionary and metadata about the data sources. (X-Specs)
- An integration algorithm which combines X-Specs into an integrated view allowing SQL queries.

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 integrated architecture which utilizes a standard protocol like XML to exchange data semantics on database systems and make them 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 be interoperable and provides exciting new opportunities on the integration of knowledge across systems.

11 Applications to the WWW

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 Net, besides the limited bandwidth, is in the inability to find and integrate the extensive databases of knowledge that exist. When a user accesses the Web for information, 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, it is up to the user to 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.

In our architecture, these types of queries are now possible. To achieve this, each web site would specify their database using a X-spec. The client’s browser would connect to the integrated central site to pose queries on data sources that it has integrated. A portal like Yahoo could then combine data sources together as a central site for the user to gather information. 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 was 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.

12 Integration Examples

In this section, we briefly describe two simple examples for this architecture, and how Unity is used to achieve the necessary integration. For the sake of brevity, some details have been omitted. However, these two examples provide a good overview on how Unity is used in practice.

12.1 Combining Order databases from two Merged Companies

Consider ABC Company which stores an order database as discussed previously. This database consists of an order table, *Order(id,customer,total_amount)*, and an ordered item table, *Order_item(order_num,item_id,quantity,price,amount)*.

The first step is to construct a X-Spec describing the database. To do this, we use 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 standardized dictionary to describe each table and field. The order table has a semantic name of `[Order]`, and its id field is `[Order] Id` as the field is an id for an order. Similarly, the ordered item table has a name `[Order;Product]` as it contains products for orders. After each field and table is assigned a semantic name, a X-Spec is produced like the one shown in Figure 1 (Section 6).

Next, the X-Spec is passed through the integration algorithm to produce the output shown in Figure 2 (Section 7). It is important to note that although only one data source was integrated, we already see some of the advantages of the approach. 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 central site running the integration algorithm becomes responsible for generating the SQL and join conditions, mapping from semantic to system names, and communicating with the database. Second, at the same time the user is isolated from the database, the full autonomy of the database is preserved as the central site appears as just another client issuing transactions.

Now, consider ABC Company buys out XYZ company in a similar business. For simplicity, assume that XYZ company has a very similar database as ABC company for storing their orders except that the table and fields names are different. Ideally, we do not want to change the database at either company right away. Preferably, we would want both company databases to function unchanged, but to have management able to see a global-view of all orders at both companies as required. Our architecture handles this case perfectly.

We already have a X-Spec for the ABC database which we do not have to change. We only 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 central site divides the 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 can be performed in an as-needed fashion, and does not have to be re-done when a new database must be added to the integrated view.

12.2 Comparison Shopping on the Net

The ultimate goal is a distributed version of the architecture where a user's browser is responsible for performing the integration algorithm as needed. The browser would contain the standardized dictionary and implement the integration algorithm, and receive from web sites their X-Specs which it integrates and presents 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 becomes comparison shopping. Just like 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 to them in the desired form. Our architecture allows a user to comparison shop only the sites that they are interested in, and query the information in a form that they are comfortable with.

The integration of web sites begins at each individual web site. For this discussion, we will consider two web sites that sell books. The first site, called **Books-for-Less**, stores its book catalog in a database using the structure: $Book(ISBN, Title, Author, Publisher, Price)$. The second site, called **Cheap Books**, stores its book database with this structure: $Book(ISBN, Author\_id, Publisher\_id, Title, Price, Description)$, $Author(id, name)$, and $Publisher(id, name)$. Each web site independently creates a X-Spec to describe their data without any knowledge of the X-Specs at other sites. Unity is used to parse the

```
<?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 3: Books-for-Less X-Spec

schema of their data source, and assign semantic names to the schema elements.

We will start with constructing the X-Spec for the **Books-for-Less** database because it 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 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 the *author_id* and *publisher_id* which have semantic names of `[Book;Author] Id` and `[Book;Publisher] Id` respectively. We 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 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 in Figure 4.

You may wonder about the decision to represent the *Author* and *Publisher* tables with the semantic names of `[Book;Author]` and `[Book;Publisher]`. The answer becomes obvious now when we attempt the integration. At this point, each web site has independently expressed their database using a 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. It then combines the X-Specs to produce the following integrated view in Figure 5.

As you can see, the information from both databases has been seemlessly combined

```
<?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 4: Cheap Books X-Spec

```
V (view root)
    - [Book]
        - ISBN
        - Title
        - Price
        - Description
        - [Author]
            - Id
            - Name
        - [Publisher]
            - Id
            - Name
```

Figure 5: Integrated View of Books Databases

```
V (view root)
    - [Book]
        - ISBN
        - Title
        - Price
        - Description
        - [Author]
            - Id
            - Name
        - [Publisher]
            - Id
            - Name
    - [Author]
        - Id
        - Name
    - [Publisher]
        - Id
        - Name
```

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

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. However, if we would have chosen the semantic names `[Author]` and `[Publisher]` for the *Author* and *Publisher* tables the integrated view would look slightly different (Figure 6).

This integrated view is as correct as the previous one, and the user can issue the identical queries as before. However, in this case the relationship between the author and publisher information is not as explicit to the user as in the previous view. The system still knows how to combine the information using the appropriate joins, but there is a slight disconnect for the user who begins to see more of the implementation structure than required. The choice of integrated view is application dependent, and there is nothing preventing the integration algorithm from "rolling up" 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 have in the query result. The query generator will then use 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 such query generation as it is guaranteed unique across databases similar to a social security number. Such keys allow the system to perform joins across databases. Query generation and implementation is a complicated subject in itself and is an area of continuing work.

13 Future Work and Conclusions

In this paper, we have detailed how a standardized global dictionary, a formalized method for capturing data semantics (X-Specs), and an integration algorithm can be combined into an integration architecture that integrates diverse data sources. Data sources are queried by semantic names, and a central site 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 such as EDI or BizTalk which are inflexible and only integrate a small subset of the data actually involved in communications. Applications of our approach include integration of web based databases and integration of entire company database systems allowing easier deployment of advanced technologies such as data warehouses and decision support systems.

Future work includes refining the standardized dictionary and improving the implementation of Unity. We plan to test the integration architecture on existing systems and develop new query mechanisms to complement the unique nature of the architecture.

A Global Dictionary Structure

Here is the C++ code implementing the global dictionary classes:

```
class CGD : public CObject
{
   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
{
   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 on rectangle used for calculations
   bool         visible;        // True if node should be currently visible
};

class CGD_link : public CObject
{
   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
}
```

B Metadata data classes

The following C++ classes are used to store metadata information in Unity:

```
class CMDField : public CObject
{
   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
{
   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
{
   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;
};
```

C X-Spec Classes

The following code implements the specification classes:

```
class CSpecField : public CObject
{
   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 CSpecTable : public CObject
{
   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
{
   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
};

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

// CKey class
class CKey : public CObject
{
   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;
};


// CJoin Class
class CJoin : public CObject
{
   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
};
```

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();

   if (spec == NULL)
      return;

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

   // 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);
   }
}
```

```
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)
      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;

   // Determine semantic name for the node at this depth
   s = terms.GetAt(aPos);
   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())+"]";
    }
    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
       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;

    // Insert node into global list of nodes
    nodes.AddTail(nd);

    // 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;
}
```

E Schema Class Definition

Here are the C++ definitions for the schema-related classes:

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

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

class CSch_node : public CObject
{
   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;                 // Position indicator for iterator function
   int          type;                 // 1 if context, 2 if concept
   CTypedPtrList<CObList,CSch_DBRec*> db_maps; // Semantic to system mappings
};

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

class CSchema : public CObject
{
   CTypedPtrList<CObList, CSch_node*> nodes; // List of schema nodes
   CSch_node*   root;                 // Root node of tree
   CDocument*   SchDoc;               // Ptr to document object for this schema
   CTypedPtrList<CObList, CSpecRef*> specs; // List of specs integrated into schema
   POSITION     aPos,aPos2;          // Position variable used for iterator
};
```

F Query Classes

Here are the C++ definitions for the query-related classes:

```
// CDT_node class
class CDT_node : public CObject
{
   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
};

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

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

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

// CTreeRef
class CTreeRef: public CObject
{
   CDT_node     *node;
   CDepTree     *tree;
};

class CDepNRef : public CObject
{
   CString      sname;
   CString      sys_name;
};

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

// Node class
class CQry_node : public CObject
{
   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
};

// The CQConcept class for retrieving concepts (semantic names) from data sources
class CQConcept : public CObject
{
   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
};

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

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

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

// The CSubQry class for accessing an individual data source
class CSubQry : public CObject
{
    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 iterator
};

// The Query class itself
class CQuery : public CObject
{
    CTypedPtrList<CObList, CQry_node*> nodes; // List of Query nodes
    CTypedPtrList<CObList, CSubQry*> subqry; // List of subqueries
    CTypedPtrList<CObList, CQConcept*> concept; // List of query concepts (semantic names)
    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
};

// CRecSet class
class CRecSet : public CObject
{
    CTypedPtrList<CPtrList,void*> recs; // List of pointers to records stored in 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
};
```

References

[1] 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.

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

[3] S. Bressan, C. H. 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.

[4] M.W. Bright, A.R. Hurson, and S.H. Pakzad. A taxonomy and current issues in multidatabase systems. *IEEE Computer*, 25(3):50–60, March 1992.

[5] 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.

[6] The Metadata coalition. Metadata interchange specification. Technical Report version 1.1, The Metadata coalition, August 1997.

[7] 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.

[8] Microsoft Corporation. BizTalk Framework 1.0 - Independent Document Specification. Technical report, Microsoft, November 1999.

[9] Microsoft Corporation. Microsoft Biztalk Server - Whitepaper. Technical report, Microsoft, May 1999.

[10] M. Genesereth, A. Keller, and O. Duschka. Infomaster: An information integration system. *SIGMOD Record*, 26(2):539–542, May 1997.

[11] 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.

[12] Uniform Code Council Inc. SIL - Standard Interchange Language. Technical report, January 1999.

[13] T. Kirk, A. Levy, Y. Sagiv, and D. Srivastava. The Information Manifold. In *AAAI Spring Symposium on Information Gathering*, 1995.

[14] R. Lawrence and K. Barker. Automatic integration of relational database schemas. Technical Report TR-00-15, Department of Computer Science, University of Manitoba, July 2000.

[15] R. Lawrence and K. Barker. Multidatabase querying by context. Technical Report TR-00-16, Department of Computer Science, University of Manitoba, July 2000.

[16] R. Lawrence, K. Barker, and A. Adil. Simulating MDBS transaction management protocols. In *Proceedings of the ISCA 11th International Conference*, November 1998.

[17] 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.

[18] W. Litwin, L. Mark, and M. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.

[19] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogenous and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[20] A. P. 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.

[21] W3C. Extensible Markup Langauge (XML) 1.0. Technical report, February 1998.