CONFLICT DETECTION AND RESOLUTION DURING RESTRUCTURING OF XML DATA

By

ANNA TETEROVSKAYA

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

2000

TABLE OF CONTENTS

Page

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

CONFLICT DETECTION AND RESOLUTION
DURING RESTRUCTURING OF XML DATA

By

Anna Teterovskaya

December 2000

Chairman: Joachim Hammer
Major Department: Computer and Information Science and Engineering

This thesis describes the underlying research, design and implementation for a Data Restructuring Engine (DRE). DRE is an integral part of the Integration Wizard system--a prototype system providing integrated access to multiple, heterogeneous sources containing semistructured data. DRE is part of the wrapper component that is associated with each source. Its main task is to translate an incoming query to the format understood by the source and to convert the query result into a representation utilized by IWiz. There are two major phases of the restructuring process: 1) During the initial built-time phase, mappings between concepts in the source and their counterparts in the integration system are discovered and documented in the form of a restructuring specification. 2) During the subsequent run-time, or querying phase, mapping functions are applied to the incoming queries and the source data to carry out the restructuring. IWiz uses XML as its internal data model; the common query language for IWiz users is XML-QL. We are currently assuming that the underlying sources are XML sources,

which allows us to focus on automatic restructuring within the same data model. Subsequent versions of DRE will be able to carry out data model conversions as well (e.g., from the relational model to XML). We have implemented a fully functional DRE, which is installed and used in the IWiz testbed in the Database Research and Development Center.

CHAPTER 1
INTRODUCTION

The World Wide Web (Web) is rapidly becoming a global data repository with virtually limitless possibilities for data exchange and sharing. New formats for storing and representing data evolve constantly [36], making the Web an increasingly heterogeneous environment. A large body of research is dedicated to overcoming this heterogeneity and creating systems that allow seamless integration of and access to the multitude of data sources. It has been noted [18] that in most cases, Web data retain some structure, but not to the degree where conventional data management techniques can be effectively used. Consequently, the term semistructured data emerged, and with it, new research directions and opportunities.

1.1 Characteristics of Semistructured Data

Before the advent of the Web, problems associated with storing large amounts of data were solved by using strictly structured databases. Conventional data storage formats in relational or object-oriented databases require that all data conform to a predefined schema, which naturally limits the variety of data items being stored, but allows for efficient processing of the stored data. On the other hand, large quantities of data are still being stored as unstructured text files in file systems. Minimal presence of constraints in unstructured data formats allows the expression of a wide range of information. However, automatic interpretation of unstructured data is not an easy task.

Semistructured data usually exhibit some amount of structure, but this structure might be irregular, incomplete, and much more flexible than traditional databases require. Semistructured data can come to existence in many different ways. The data can be designed with a semistructured format in mind, but more often the semistructured data format arises as a result of the introduction of some degree of structure into unstructured text or as the result of merging several collections of structured data with different schemas.

There are several characteristics of semistructured data that require special consideration when building an application for processing such data [1, 8, 38].

- The structure is irregular. The same information can be structured differently in parts of a document. Information can be incomplete or represented by different data types.
- The structure is partial. The degree of structure in a document can vary from almost zero to almost 100%. Thus, we can consider unstructured and highly structured data to be extreme cases of semistructured data.
- The structure is implicitly embedded into data, i.e. the semistructured data are self-describing. The structure can be extracted directly from data using some computational process, e.g., parsing.
- An a-priori schema can be used to constrain data. Data that do not conform to the schema are rejected. A more relaxed approach is to detect a schema from the existing data (recognizing that such a schema cannot possibly be complete) only in order to simplify data management, not to constrain the document data.

- A schema that attempts to capture all present data constructs can be very large due to the heterogeneous nature of the data.
- A schema can be ignored. Nothing prevents an application from simply browsing the hierarchical data in search of a particular pattern with an unknown location, since the data are self-describing and can exist independently of the schema.
- A schema can evolve rapidly. In general, a schema is embedded with the data and is updated as easily as data values themselves.

Amounts of electronic data produced, stored, and exchanged in a unit of time are growing at a constant rate, and it is safe to say, will grow at even faster rate in the future. One of the most rapidly evolving domains that requires access to large amounts of data is e-commerce. The nature of e-commerce also requires that data have to be exchanged constantly in large amounts. In addition, data coming from different origins have to be translated, processed, and incorporated into the existing body of information. In this context, the quest for the common data exchange format is closely related to problems of data integration. The following quote by Bosworth and Brown from Microsoft [10, p.36] concisely summarizes why semistructured data have the qualities to become a standard way of exchanging data: " . . . simplicity and flexibility beat optimization and power in a world where connectivity is key. These features of semistructured data - simplicity and flexibility - made the semistructured data in many cases an appropriate choice for data exchange between heterogeneous systems."

Using semistructured data as a data exchange format does not automatically answer all questions. The new mechanisms are required to solve the classical data exchange and management problems, such as providing persistent storage, query

optimization, or efficient data modeling in this new context [40]. Extensive research efforts over the last several years have concentrated on data models and query languages for semistructured data as well as on storage and processing of such data.

The term "semistructured data" does not imply any specific data model or format to represent the information. Several data models were created over the last decade [30, 34], but the Extensible Markup Language (XML) from the World Wide Web Consortium [44] has apparently become the dominant way to represent and model semistructured data. Created mainly as a technique for more effective document representation and Web publishing, XML also proved to be a convenient format for expressing and storing semistructured data. The intention of this work is to investigate theoretical as well as practical aspects of integration of XML data.

1.2 Challenges of Data Integration

The data integration process converts data from heterogeneous sources into a common format. The most common causes for heterogeneities are different data formats (e.g., HTML and Latex, HTML and text, etc.), differences in the underlying data model (e.g., relational, object-oriented, semistructured), and different representations. There are two general approaches to data integration: data-based and schema-based. For the reminder of this discussion, we assume that there exist a target format and a target schema that are supposed to represent the integrated information. Data from a number of information sources are integrated into this target schema.

In the data-based approach, data are sequentially parsed, analyzed, and converted to the target format with the use of a complex translation language. The translation

language should be complete enough to incorporate all possible translations for the particular combinations of data formats.

The schema-based approach works in two steps. The schemas of the heterogeneous sources are analyzed and compared to the target schema one by one. Conflicts between each source schema and the target schema are noted. Based on this knowledge, a set of rules for data translation is created for each source schema. These rules are valid for any data instance conforming to the source schema. Applying translations rules to the source information results in data instances fully conforming to the target schema.

As was mentioned in the beginning of this chapter, semistructured data can conform to an a-priori schema, or a schema can be deduced from the existing document. In any case, the problem of data integration can be reduced to finding and resolving discrepancies between two schemas and applying solutions to the document data converting these data to the desired format. The possible conflicts between a source and target schema cover all aspects of data definition mechanism for a particular data model. The richer the data definition facilities, the more conflicts are possible. One of the main challenges of data integration is to provide automatization to the process of conflict detection and resolution. This automatization is one of the main goals of our work which is discussed in the next section.

1.3 Goal of the Thesis

Integration Wizard (IWiz) project is under way in the Database Research and Development Center at the University of Florida [22]. IWiz is an integration system that

provides uniform access to the multiple heterogeneous sources. At this point, the sources are assumed to contain semistructured data in XML format. In the future, the system will provide access to information in other formats: text files, relational databases, image archives, etc. via wrapper components for various data models.

In this thesis, we describe the design and implementation issues for one of the architectural components of the IWiz, namely the Data Restructuring Engine (DRE). DRE is the module closest to a data source in the IWiz architecture. The primary DRE activity from the system point of view is querying the source while returning results in the common IWiz data model and representation. DRE uses a schema-based integration algorithm, therefore, the two main procedures carried out by DRE are 1) detection and resolution of possible conflicts between the source and global IWiz schema and 2) translation of a data request into a form understandable by the source, and translation of a result into a form understandable by the rest of the system.

To accomplish these tasks, we need to understand the classification of possible conflicts between two schemas and devise the algorithms to detect and resolve those conflicts. Another important aspect of data integration used in IWiz is to formulate the conversion rules and devise ways to apply the set of rules to actual data when querying the source. Our approach concentrates specifically on transforming XML data. DRE converts a set of source XML data to XML data based on the common system schema.

The rest of the thesis is composed as follows. Chapter 2 gives an overview of XML and related technologies. Chapter 3 is dedicated to an overview of related research on integration systems. Chapter 4 describes the IWiz architecture and the place of DRE in relation to other components. Chapter 5 analyzes fundamental concepts of conflict

detection and resolution. Chapter 6 focuses on our implementation of DRE. Chapter 7 gives performance analysis of the implementation, and Chapter 8 concludes the thesis with the summary of our accomplishments and issues to be considered in future releases.

CHAPTER 2
XML AS AN EXCHANGE FORMAT AND DATA MODEL FOR SEMISTRUCTURED DATA.

Semistructured data can be represented in different ways. Numerous research projects have been using various representations and data models to manage collections of irregular structured data [27, 30, 34]. XML has emerged as one of the contenders and has quickly turned into the data exchange model of choice. Initially, it started as a convenient format to delimit and represent hierarchical semantics of text data, but was quickly enriched with extensive APIs, data definition facilities, and presentation mechanisms, which turned it into a powerful data model for semistructured data.

The popularity of XML as a way of modeling and representing semistructured data can be attributed to the following practical aspects [28]:

- XML is platform-independent. It can be used to exchange data between users and programs.
- XML is extensible. Countless sets of tags may be constructed, describing any imaginable domain.
- XML is self-describing. Each data element has a descriptive tag. Using these tags, the document structure can be extracted without knowledge of the domain or a document description.
- XML is able to capture hierarchical information and preserve parent-child relationships between real-world concepts.

In this chapter, we describe the principles on which XML is based and give an overview of XML-related technologies that are relevant to our work.

2.1 Basic XML

The Extensible Markup Language (XML) is a subset of SGML [46]. The World Wide Web Consortium took the initiative in developing and standardizing the XML, and their recommendations from 10 February 1998 outline the essential features of XML 1.0 [44].

```
<?xml version="1.0"?>
<bibliography>
        <book>
                <title>"Professional XML"</title>
                <author>
                        <firstname>Mark</firstname>
                        <lastname>Birbeck</lastname>
                </author>
                 <author>
                        <lastname>Anderson</lastname>
                </author>
                <publisher>
                        <name>Wrox Press Ltd</name>
                </publisher>
                <year>2000</year>
        </book>
        <article type = "XML">
                <author>
                        <firstname>Sudarshan</firstname>
                        <lastname>Chawathe</lastname>
                </author>
                <title>Describing and Manipulating XML Data</title>
                <year>1999</year>
                <shortversion> This paper presents a brief overview of
                        data management using the Extensible Markup
                        Language(XML).  It presents the basics of XML
                        and the DTDs used to constrain XML data, and
                        describes metadata management using RDF.
                </shortversion>
        </article>
</bibliography>
```

Figure 1: An example of an XML document

XML is a markup language; nested markup, in the form of tags, describes the structure of an XML document. Markup tags can convey semantics of the data included

between the tags, special processing instructions for applications, and references to other data elements either internal or external.

The XML document in Figure 1 illustrates a set of bibliographic information consisting of books and articles, each with its own specific structure. Data items are delimited by tags. XML requires a mandatory closing tag for each opening tag. Tags can be nested, with child entities placed between the parent's opening and closing tags, no limits are placed on the depth of the nesting.

The fundamental structure composing an XML document is the *element*. An element can contain other elements, character data, and auxiliary structures, or it can be empty. All XML data must be contained within elements. Examples of elements in Figure 1 are `<bibliography>`, `<title>`, and `<lastname>`. Simple information about elements can be stored in attributes, which are name-value pairs attached to an element. Attributes are often used to store the element's metadata. Only simple character strings are allowed as attribute values, and no markup is allowed. The element `<article>` in our example has an attribute "`type`" with an associated data value "XML." The XML document in Figure 1 is an example of a *well-formed* XML document, i.e. an XML document conforming to all XML syntax rules.

2.2 Advanced XML Features

An XML grammar defines how to build a well-formed XML document, but it does not explain how to convey the rules by which a particular document is built. Other questions requiring answers are how to constrain the data values for a particular document, and how to reuse an XML vocabulary created by somebody else. This

sections touches on XML-related standards and proposals that solve these and other problems.

2.2.1 Document Type Definition

A Document Type Definition (DTD) is a mechanism to specify structure and permissible values of XML documents. The schema of the document is described in a DTD using a formal grammar. The rules to construct a DTD are given in the XML 1.0 Recommendation. The main components of all XML documents are elements and attributes. Elements are defined in a DTD using the `<!ELEMENT>` tag, attributes are defined using the `<!ATTLIST>` tag. The declarations must start with a `<!DOCTYPE>` tag following by the name of the root element of the document. The rest of the declarations can follow in arbitrary order. Other markup declarations allowed in a DTD are `<!ENTITY>` and `<!NOTATION>`. `<!ENTITY>` declares a reusable content, for example, a special character or a line of text repeated often throughout the document. An entity can refer to a content defined inside or outside of the document. A `<!NOTATION>` tag associates data in formats other than XML with programs that can process the data.

Figure 2 presents a DTD for the XML document in Figure 1. Every DTD defines a set of well-formed XML documents; by the same token, a well-formed XML document can conform to a number of DTDs. When a well-formed XML document conforms to a DTD, the document is called *valid* with respect to that DTD. Next, we provide detailed standards for forming element and attribute declarations in DTDs, as the format of these declarations play a key role in the conflict detection process described later in this thesis.

```
<?xml version="1.0"?>
<!DOCTYPE bibliography [
<!ELEMENT bibliography (book|article)*>
<!ELEMENT book (title, author+, editor?, publisher?, year)>
<!ELEMENT article (author+, title, year ,(shortversion|longversion)?)>
<!ATTLIST article type CDATA #REQUIRED
                  month CDATA #IMPLIED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (firstname?, lastname)>
<!ELEMENT editor (#PCDATA)>
<!ELEMENT publisher (name, address?)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT shortversion (#PCDATA)>
<!ELEMENT longversion (#PCDATA)>
]>
```

Figure 2: Sample DTD

Each element declaration consists of the element name and its contents. The contents of the element can be one of four types: *empty, element, mixed,* or *any*. An *empty* element cannot have any child elements (but can contain attributes). An element whose content has been defined as *any* can have any number of different contents conforming to XML well-formed syntax. *Element* content refers to the situation in which an element can have only other elements as children. *Mixed* content allows combinations of element child nodes and parsed character data (#PCDATA), i.e. text. For example, in Figure 2, the `bibliography` element has element content, and the `year` element has mixed content.

The DTD also allows to specify the cardinality of the elements. The following explicit cardinality operators are available: ? which stands for "zero-or-one," * for "zero-or-more," and + for "one-or-more." In the case when no cardinality operator is used, the element can be present exactly once (i.e., the default cardinality is "one"). In our example in Figure 2, a `book` can contain one or more `author` child elements, must

have a child element named `title`, and the `publisher` information can be missing. Order is an important consideration in XML documents; the child elements in the document must be present in the order specified in the DTD for this document. For example, a `book` element with a `year` child element as the first child will not be considered a part of a valid XML document conforming to the DTD in Figure 2.

Attributes provide a mechanism to associate simple properties with XML elements. Each attribute declaration includes name, type, and default information. The attribute type can be one of the following CDATA, ID, IDREF, IDREFS, ENTITY, ENTITIES, NOTATION, ENUMERATION, NMTOKEN, or NMTOKENS.

CDATA attributes can contain character strings of any length, like the `month` attribute of the element `article` in our example. An element can have at most one attribute of type ID. This attribute must be assigned a value that is unique in the context of the given document. The ID value can be referenced by an attribute of type IDREF in the same document. In a sense, the ID-IDREF pairs in XML play the same role as primary key-foreign key associations in the relational model. A value for an attribute of type IDREFS is a series of IDREF references of unspecified length. ENTITY(ENTITIES) and NOTATION attributes have the same meaning as the ENTITY and NOTATION elements described previously. ENUMERATION specifies a set of permissible values for a particular attribute. The NMTOKEN(S) declaration, which is short for "name token", forces the value of the attribute to conform to XML element naming rules, i.e. be composed only of letters, digits, colons, underscores, hyphens, and periods.

Default information for an attribute combines cardinality constraints with information about its default value if one exists. Each attribute declaration must include one of the following descriptors #REQUIRED, #IMPLIED, #FIXED or provide a default value. A value for a #REQUIRED attribute must be provided for every instance of the element. An #IMPLIED attribute can be missing from some or all instances. The attribute whose name followed by a value in the declarations is assigned that value in case the data are missing. The #FIXED tag must be followed by a default value. In this case, the default value cannot be overwritten by any other value.

The entire DTD structure can be placed in the beginning of the associated XML document or in a separate location, in which case the document contains only a `<!DOCTYPE>` tag followed by the root element name and the location of the DTD file in form of a URI. Separation of a schema and data permits multiple XML documents to refer to the same DTD and allows a DTD file to be assigned an access mode that differs from the access mode of the data.

2.2.2 Namespaces

Namespaces enable reuse of existing XML vocabularies and mixing of concepts which are defined in different vocabularies without running into conflicts when concepts from different origins share the same name. The W3C issued its Recommendation "Namespaces in XML" on January 14, 1999 [45], where a namespace is defined as "a collection of names, identified by a URI reference, which are used in XML documents as element types and attribute names." The term namespace can refer to a DTD or a file in some other format that contains a series of names and can be located by the application that processes the XML document.

```
<Bibilography xmlns = http://www.bookcatalog.com/book.dtd
       xmlns:author1  = "book-author-list"
       xmlns:author2 = "article-author-list">
       <book>
              <author1:Author>Andrew Williams</author1:Author>
              …
       </book>
       <article>
              <author2:Author>David Stark</author2:Author>
              …
       </article>
</Bibliogrpahy>
```

Figure 3: Example of declaration and usage of namespaces.

A namespace for a particular element is declared as an attribute with the reserved name "xmlns"; its value is a URI pointing to the desired namespace. Several namespaces can be declared simultaneously. In order to distinguish between elements associated with different namespaces, each namespace declaration is identified with an alias. For example, in Figure 3, three different namespaces are used for the `<Bibliography>` element. The first namespace (`xmlns = http://www.bookcatalog.com/book.dtd`), which is declared without an alias, becomes the default namespace for the element. Definitions for elements whose name does not have a prefix (unqualified names in XML terminology) come from the default namespace. The definitions for the two `<Author>` elements in the example come from two different sources and can be distinguished by their prefixes, `author1` and `author2`.

The XML 1.0 Recommendation makes no provision for namespace support, so not all of the available XML parsers can handle documents containing elements from different namespaces. Current working drafts for newer versions include support for the namespace technology. Including namespace support in an application greatly increases the range of XML documents that can be processed by this application and allows to

efficiently utilize existing XML vocabularies. We intend to support the namespace technology in the IWiz implementation.

2.2.3 XML Schema

At the moment of writing, a DTD is the only officially approved mechanism to express and restrict the structure of XML documents. There are obvious drawbacks to DTDs. Their syntax is different from the XML syntax (this is one reason why most parsers do not provide programmatical access to DTD structure). In addition, DTDs do not provide any inherent support for datatypes or inheritance. Finally, the format of cardinality declarations permits only coarse-grained specifications.

```
<schema ...>
      <element name = "bibliography"
               type = "string"
               minOccurs = "0"
               maxOccurs = "unbounded">
            <type>
                  <group order = choice>
                        <element type = "book">
                        ...
                        </element>
                        <element type = "article">
                              <attribute name = "type" type ="string">
                              <attribute name = "month"
                                          type =  "integer"
                                          default = "1">
                        ...
                        </element>
                  </group>
            </type>
      </element>
</schema>
```

Figure 4: A fragment of an XML Schema representing the information from Figure 2.

W3C has recognized these existing problems with DTDs and has been working on new specifications called XML Schema since 1999 [48, 49]. Eventually, this new data definition mechanism will express schema information using XML grammar and feature

strong typing and support for multiple namespaces. Proposed data types include types currently present in XML 1.0 and additional data types such as boolean, float, double, integer, URI, and date types. An XML Schema is a series of one or more special XML documents. Figure 4 shows an XML Schema containing the same information as the DTD in Figure 2 (internal details of `<book>` and `<article>` elements are omitted). In an XML Schema, cardinalities can be specified precisely, and type information is included in the declarations. In addition, type definitions can be derived from other type definitions by restricting or extending the source type. The syntax given in this section is taken from W3C Working Draft for XML Schema and can still be changed by the time when W3C issues final recommendations on XML Schema.

2.3 APIs for Processing XML Documents

A file containing an XML document is nothing more than a plain text file. In order to search or modify XML data, delete parts of the document, or add new data items, some effective mechanisms for accessing these data items are required. The two alternative ways to access contents of an XML document from a program are the tree-based approach and the event-based approach. In the tree-based approach, an internal tree structure is created that contains the entire XML document in memory. An application program can now freely manipulate any part of the document as small as a single data value or as big as the whole document. In case of the event-based approach, an XML document is scanned, and the programmer is notified about any significant events such as start or end of a particular tag that are encountered during scanning. The realizations of these approaches that have gained widespread popularity are the

Document Object Model (implementing the tree-based model) and the Simple API for XML (in case of the event-based model). Overviews of both mechanisms are given in the next two sections.

2.3.1 Document Object Model

The Document Object Model (DOM) specifications are produced by W3C like most of the XML-related technologies. The DOM Level 1 Recommendation dates back to October 1, 1998 [42, 43]. DOM is a language- and platform-neutral definition and specifies the APIs for the objects participating in the tree model. However, it provides no implementation details or language restrictions.

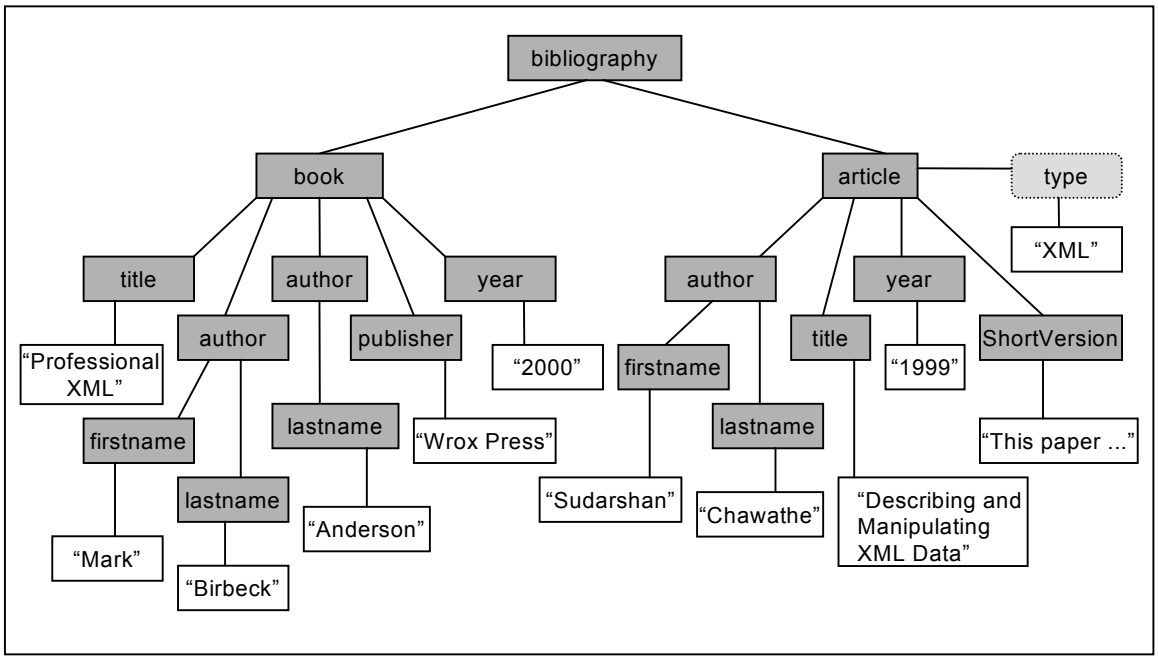

Figure 5: DOM Representation for XML Document from Figure 1.

A DOM parser parses the XML file identifying individual elements, attributes, and other XML structures and creates a node tree. DOM treats every item in an XML document as a *node*. Special types of nodes exist for different XML components, such as

Element, Attribute, or *Document*, but all of them extend the common class *Node*, which facilitates operations on the node tree. The XML document in Figure 1 can be parsed into a node tree shown in Figure 5. We can see that the actual data values are contained in special *Text* nodes. The other nodes in a DOM tree cannot directly contain a string value, a Text node with a value has to be appended as a child to an *Element* or an *Attribute* node. After the DOM tree is constructed, the existing nodes can be deleted, rearranged, or new nodes can be inserted. Throughout this process, the DOM API ensures that the document stays well-formed and the grammar is maintained. Actually, the programmer does not have to be aware of the XML grammar at all, since DOM abstracts the contents of the document from the grammar and presents the contents to the programmer as a hierarchical tree.

2.3.2 Simple API for XML

The Simple API for XML (SAX) represents a different approach to parsing XML documents [31]. A SAX parser does not create a data structure for the parsed XML file. Instead, a SAX parser gives the programmer the freedom to interpret the information from the parser, as it becomes available. The parser notifies the program when a document starts and ends, an element starts and ends, and when a text portion of a document starts. The programmer is free to build his/her own data structure for the information encountered or to process the information in some other ways. At first, the process seems very involved and tedious, but there are applications that can benefit from the SAX approach.

Examples where SAX would be the API of choice are applications which are interested only in the total count of occurrences for a particular element, applications

searching for a particular text string, or applications requiring a data structure other than DOM to process and manipulate an XML document. Another important consideration is memory management. SAX does not load the entire document into memory and, subsequently, can operate on XML documents of any size without running into any memory limitations.

As we have seen, both approaches have their own benefits and drawbacks. The decision to use one or the other should be based on a thorough assessment of application and system requirements.

2.4 Querying XML Documents

Data represented in XML can be utilized by many applications. However, XML data are useful only if the information can be effectively extracted from an XML document according to specified conditions. W3C is currently coordinating the process of creating a query language for XML. Research query languages for semistructured data have been designed and implemented such as Lorel [2] and UnQL [9]. With the emergence of XML, those languages migrated to accommodate XML syntax [21] and an array of new query languages emerged. As in many other instances, the W3C has coordinated the quest for a new flexible and efficient query language for XML. Several proposals were submitted to W3C committee including XML-QL, XQL, XPath, XML-GL [13, 17, 47], just to name a few.

At the time of writing, there is no standard query language for XML. The committee issued requirements for such a language named XML Query Language and the

work in this direction continues. The main points of the latest version of the requirements from August 15, 2000 [50] for XML query language are:

- The XML Query Language must support operations on all data types represented by the XML Query Data Model.
- Queries must be able to express simple conditions on text, including conditions on text that spans element boundaries.
- The XML Query Language must be able to combine related information from different parts of a given document or from multiple documents.
- The XML Query Language must be able to compute summary information from a group of related document elements.
- The XML Query Language must be able to sort query results.
- The relative hierarchy and sequence of input document structures must be preserved in query results.
- Queries must be able to transform XML structures and create new XML structures.
- Queries should provide access to the XML schema or DTD, if there is one.
- Queries must be able to perform simple operations on names, such as tests for equality in element names, attribute names, and processing instruction targets and to perform simple operations on combinations of names and data.

IWiz is an integration system that provides access to multiple XML sources. One of its basic functionalities is the ability to query XML documents. Therefore, there was a need for a query language for semistructured data, specifically in XML format. We have examined possible choices and selected XML-QL [14]. XML-QL does not provide all functionalities required by W3C committee, but it is a data-oriented language (whereas

XQL is document oriented) and provides most of the necessary functionalities (e.g., joins and aggregations are missing from XSL). In addition, our decision was influenced by the existence of a robust implementation by AT&T [51].

XML-QL can extract data from XML documents, transform XML data or integrate data from different XML documents. XML-QL queries consist of `WHERE` and `CONSTRUCT` clauses that are similar to the `SELECT-WHERE` clauses in SQL. XML-QL uses an XML-like format to specify element patterns, but XML-QL queries are not well-formed XML documents. Figure 6 shows an example of a simple XML-QL query requesting all book titles authored by "A.B.Cooper".

```
WHERE
      <book>
            <author>"A.B.Cooper"</author>
            <title>$t</title>
      </book> IN "catalog.xml"
CONSTRUCT
      $t
```

Figure 6: Example of XML-QL Query.

Variables in XML-QL are preceded by '$'. The query above scans the "`catalog.xml`" file and finds all `<book>` elements that have at least one `<author>` and one `<title>` child element and whose `<author>` element has the value of "`A.B.Cooper`". The `CONSTRUCT` clause constructs a list of values bounded to `$t`, i.e. list of titles.

`WHERE-CONSTRUCT` blocks can be nested as shown in Figure 7. The variable $p is bound to contents of a <book> element. Expression `IN $p` now refers to the contents of the current `<book>` element. Due to this modification, the result of the

query in Figure 7 contains author names grouped by the title, whereas in the result of the query in Figure 6, the title is repeated for each author of the book.

```
    WHERE
          <book year = $y> $p </book> IN "catalog.xml",
          <title> $t </title> IN $p,
          $y > 1998
     CONSTRUCT <resultSet>
          <title> $t </title>
          WHERE
                <author> $a </author> IN $p
          CONSTRUCT
                <author> $a </author>
          </resultSet>
```

Figure 7: Nested XML-QL Query.

XML-QL uses a labeled graph as a data model, in which attributes are associated with nodes, and elements are represented by the edge labels. Only leaf nodes can contain data values. Each node in a graph has a unique identifier, which may be an ID attribute or system assigned identifier if an ID for the element is not defined. In most cases, XML-QL relaxes the data model for XML data and assumes no order between child subelements of an element.

Variables in XML-QL can be bound to any intermediate node in data graph, not only to leaf nodes. Since intermediate nodes do not contain values, variables are bound to the underlying structure of the element. These variables are called tag variables. An example illustrating the use of tag variables is presented in Figure 8. The $p variable in Figure 7 is bound to any child element of the root element in "catalog.xml." Assuming that the top-level elements in the "catalog.xml" document are `<book>` and `<article>`, the query in Figure 8 will produce the title of all publications that appeared in 1999.

```
WHERE <$p>
       <title> $t </title>
       <year> 1999 </year>
      </> IN "catalog.xml"
CONSTRUCT
       <$p>
              <title> $p </title>
       </>
```

Figure 8: Example of Using Tag Variables.

The `WHERE` clause can contain several element patterns and variable bindings that belong to different XML source documents. By creating a new structure in the `CONSTRUCT` clause that includes variables from different sources, we can integrate data from several sources into one XML document with new structure. In case of a single original document, this document can be restructured by specifying new element structure in the `CONSTRUCT` clause.

This chapter has provided background on various XML technologies that directly affect the design and implementation of DRE and the IWiz project. Next, we proceed to an overview of the state-of-the-art in integration systems.

CHAPTER 3
OVERVIEW OF INTEGRATION SYSTEMS

Systems for integration of heterogeneous information continue to receive much attention from the research community. The subject of this thesis, Data Restructuring Engine, is an architectural component in IWiz [22], an integration system for semistructured data. Before introducing the proposed architecture for IWiz, we present research aspects of integration systems relevant to our work and give an overview of similar research projects.

3.1 Architectures for Integration Systems

Most information integrated system architectures conform to one of the three design approaches: federated databases, data warehousing, or mediation approach [20]. Federated databases do not employ any centralized module to regulate information flow between separate autonomous databases. Each unit operates independently and has a mechanism to communicate (query) with other databases in the system when it is necessary.

The data warehousing scheme assumes presence of a single centralized data storage facility, which physically holds a copy of data from multiple sources. The data in a warehouse conform to a certain schema, usually called a global schema.

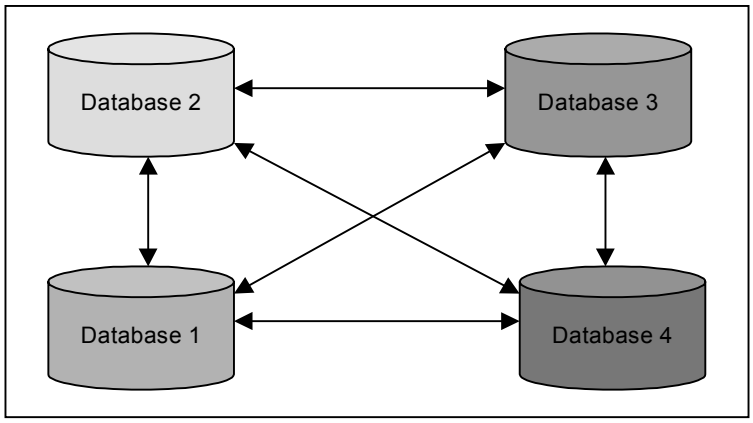

Figure 9: Federated Databases

When a new source becomes available to the warehouse, the source data must be processed to conform to the global warehouse schema and can be combined with the existing data in the warehouse. All data requests are processed directly by the warehouse resulting in faster response times, but creating the possibility of stale data.

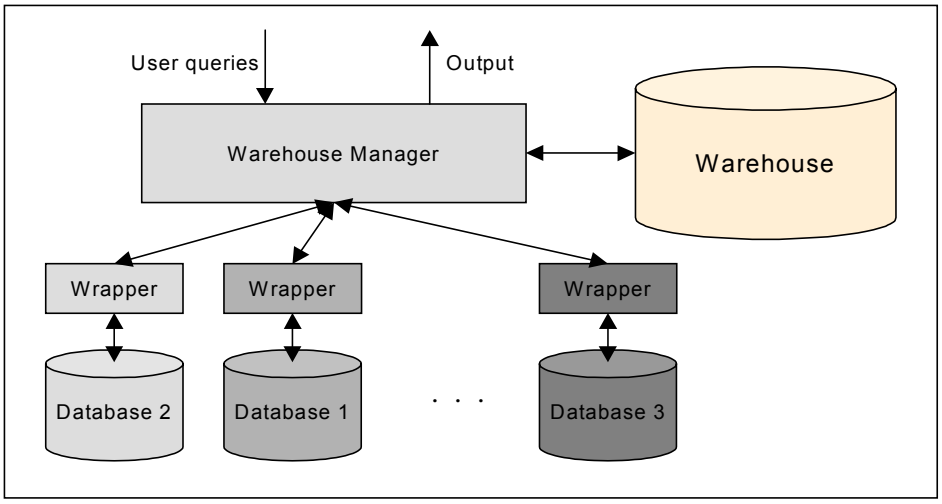

Figure 10: Generic warehouse-based architecture.

Systems based on the mediation approach do not retrieve data from the sources until the data are requested. The user query is decomposed by the mediator component--

a software module responsible for creating a virtual integrated view of the data sources in the system. The mediator determines which data sources contain relevant information and queries those sources. The mediation approach guarantees that the retrieved data are always up to date. However, accessing distributed sources and integrating results before presenting them to the user can take considerably longer than accessing data in a warehouse.

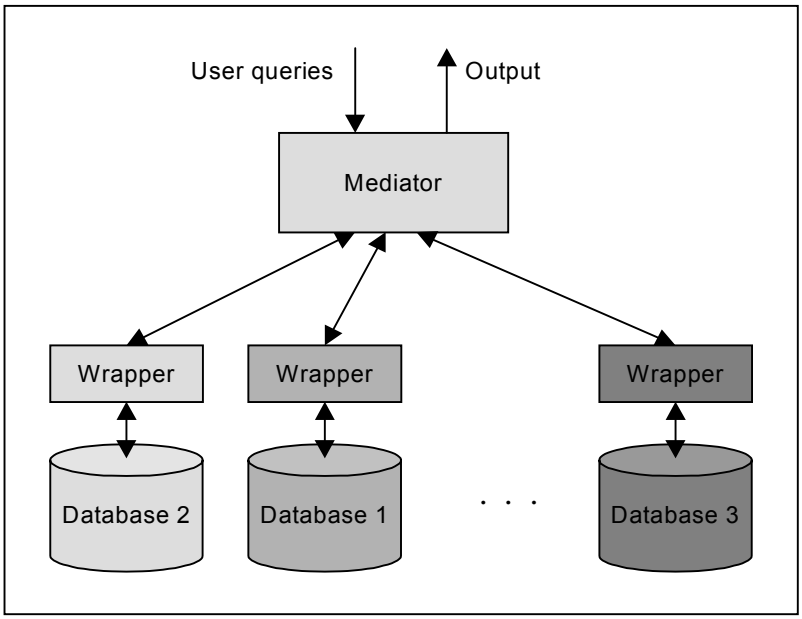

Figure 11: Generic mediator-based architecture.

Both the mediator and warehousing architectures feature an integrated view of all system data, i.e. a "virtual relation" that incorporates concepts represented in the underlying sources. The integrated view can be constructed in "bottom-up" or "top-down" fashion [25]. The objective of the former approach is to build an exact union of information from the underlying source schemas. In the latter case, the integrated view attempts to encompass all information relevant to the given knowledge domain.

Subsequently, the integrated schema may represent only a subset of source information when the "top-down" approach is used.

The warehousing and mediator approaches have been successfully used in research integration systems [19, 23, 24] as well as commercial applications for data integration. The comparative analysis in [25] shows strengths and limitations of both methodologies and identifies future research challenges.

IWiz is an integration system that provides uniform access to the multiple heterogeneous sources. The architecture of the IWiz combines both the mediator and warehouse approaches in an effort to achieve both flexibility and shorter query response time. Query results are cached in the warehouse and can be retrieved efficiently in the case of repeated or similar requests. The replacement policy guarantees that current data always replace the older information. Each data item is assigned a time interval when the data can be regarded as valid, after this time the information has to be retrieved directly from the source.

3.2 Integration and Translation System Prototypes

Many research efforts in recent years are directed toward designing systems for integration of heterogeneous data [24, 27, 29]. A variety of design approaches to data conversion have been proposed. In this section, we will focus of systems that employ schema-based translation and integration.

The TranScm system is implemented at the Computer Science Department at Tel-Aviv University [32]. The TranScm is a translation system that handles data translation between multiple data formats. Schemas for different formats are converted to an

internal common schema model and compared. The system uses the data translation approach similar to the IWiz approach. The data translation process in TranScm is schema-based. The assumption is made that the source and the target schemas are closely related. The discrepancies between the two related schemas are found automatically based on set of predefined matching rules. Each rule contains translation functions applied at the data conversion phase. Input from an expert user is required to assist in cases where automatic conflict detection fails.

The MOMIS project (Mediator envirOnment for Multiple Information Sources) [7] is a mediator-based integration system for structured and semistructured data. It supports read-only view of data from multiple sources. The common thesaurus plays the role of a shared ontology; it is built by extracting terminological relationships from source schemas. The wrapper components translate heterogeneous source schemas into a common object-oriented model. The translation is based on relationships found in the common thesaurus. Source schemas are analyzed, and the mediator-integrated view is (semi)automatically constructed based on relationships between the source schemas' concepts. The system uses extensive query optimization techniques in order to ensure the effective data retrieval in distributed environment.

The ARANEUS project [30] was started as an integration system for HTML documents on the Web. With introduction of XML, the system has evolved to handle XML sources as well. ARANEUS uses its own data model (ADM) to describe source structure (instead of conventional DTDs). The ADM has stronger typing mechanism than DTDs and completely abstracts logical aspects of a schema from physical characteristics. Queries to the source are written against the ADM schema of the source

using custom-developed query algebra. In addition to querying HTML and XML sources, the system includes the tools to design and generate HTML or XML documents from an ADM object.

The last project on which we will comment and which has provided a direct foundation for the current implementation of DRE is the work of R.S. de Oliveira at the Database Research and Development Center at the University of Florida. Mr.Oliveira's designed and implemented the first version of DRE as a part of his Master's thesis research. The conflict taxonomy used in our DRE implementation (Chapter 5) and our current approach to data restructuring (Chapter 6) are partially based on Mr.Oliveira's work. The work presented in this thesis represents a series of significant improvements and innovations in the design approach and the implementation of DRE. Our DRE implementation detects and resolved several times more conflicts between a source and target schema than the previous version. A fully-functional graphical user interface is available to the user. One of the most significant additions was providing DRE with querying capabilities. This change indicates evolution of the entire IWiz architecture from an exclusively warehouse-based system to a hybrid architecture that combines both the mediation and the warehouse approaches. Altogether, the algorithms involved were revised to increase efficiency and robustness of the module. Chapter 8 provides a more detailed list of implemented innovations and improvements.

This concludes the discussion on related research dedicated to data integration systems. Next, we briefly describe the IWiz architecture and roles of its major structural components.

CHAPTER 4
IWIZ ARCHITECTURE

The architecture of the IWiz is presented in Figure 12 [22]. The main components of the system are the Querying and Browsing Interface, the warehouse component (IWiz Repository), Mediator, and Data Restructuring Engine (DRE).

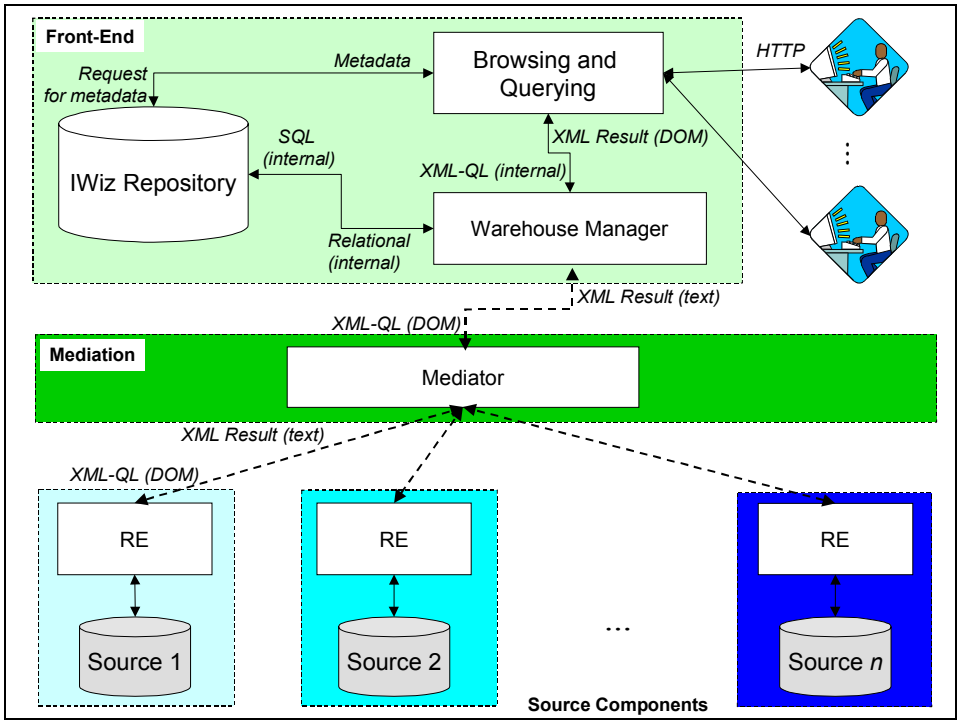

Figure 12: Architecture of the IWiz system

The Querying and Browsing Interface is an interactive user interface. It allows the user to examine an integrated view of all the data contained in the system without being aware of the fact that data are physically stored in different locations and different formats.

The user can explore the global schema and pose queries against the global schema by selecting data elements and specifying conditions on those elements.

The warehouse is a relational database that contains recent information retrieved from the sources. Oracle 8i database system is used to manage relational data. The warehouse manager is the module that decides if the information in the warehouse is sufficient to answer the query right away or the information from the source should be retrieved and maybe joined with information from the warehouse.

The mediator is the component that is in charge of all sources. It maintains information about location and contents of system sources. The work of the mediator is to analyze a query coming from the warehouse manager and to pass the query, split and adjusted to each particular source, to the respective DRE modules.

The DRE component, which is a main topic of this thesis, serves as an interpreter between the sources and the rest of the system. It is the only component that has direct access to the source data. The DRE component is source-specific, meaning that there is exactly one DRE associated with each source. The two major phases in the DRE operation are the run-time and built-time phases. The DRE determines the correspondence between concepts represented in global schema and those in the source schema at built-time. Further, mappings are augmented with translation procedures for each data element and verified by an expert user. The mappings are defined once and not changed during the system operation unless the source or ontology schemas are altered. During run-time, the DRE extracts information from the source document and transforms it to the ontology terms.

The following example illustrates the data and control flow in IWiz. We will follow the information retrieval process from the moment an end user issues a request to the point when the query results are displayed at the user's monitor. From now on, unless otherwise noted, the term "user" refers to the casual end user with no administrator or source-specific knowledge. The term "expert user" refers to a person responsible for setting up and maintaining the IWiz system.

The user interacts with the system via a graphical user interface. The interface displays the global schema represented in a tree-like form. The user builds a query by selecting data elements and providing constraints for data values of these elements. The user is not assumed to be familiar with any specific querying language or know any details about number, format, or contents of the underlying sources participating in IWiz.

Internally, the query is transformed into an XML-QL query and passed to the warehouse component. The warehouse manager decides if the current data in the warehouse are sufficient to satisfy the user's request. In case of sufficient data, the answer is forwarded directly to the user interface where it is displayed. The browsing interface and the warehouse component are assumed to reside in the same address space and therefore, interaction between those modules should be quite efficient. In case when requested information or its part is not present in the warehouse, the query is passed to the mediator by the warehouse manager.

The mediator determines which sources contain information necessary to satisfy the query. In case some sources contain only partial information, arrangements are made to merge the information coming from the sources. The incoming query is re-written into

several queries appropriate for each source. These so-called mediated queries are passed to the DRE component associated with each source.

DRE is responsible for transforming the mediator query into a query that is understood by the source and submitting the new, source-specific query to the source. The built-time process involves comparison of the source and global schema and detection and resolution of conflicts between the concept definitions in the two schemas. The information about detected conflicts together with proposed methods of their resolution, called conversion specifications, is saved as a valid XML document accessible by the run-time process. The built-time phase of the DRE activity is executed during system set-up once for each data source.

The run-time DRE component is invoked to process a user query. Based on the mappings found at the built-time, a query is translated into source terms. Translation procedures from the restructuring specifications are incorporated into the query or applied directly to the XML documents before or after query execution. The query is applied to the source document, and the result is passed back to the mediator.

The mediator receives responses from multiple sources in the form of XML documents. The next step is integration of the data. It is important to note that, at this point, the individual XML documents returned by the sources may not conform to a global schema. It is the task of the mediator to produce an integrated result and to validate the integrated result against the global schema. Data from different sources are joined, duplicates coming from overlapping sources are removed, and inconsistencies in the data are reconciled. In general, data are transformed into a form requested by the user.

The transformed result is returned to the warehouse manager that decides whether the data or part of it should be added to the warehouse. The warehouse manager also may integrate the result from the mediator with existing data already in the warehouse. At the same time, the final result is returned to the browsing interface and displayed to the user.

All major components of the IWiz are currently under development at the Database Research and Development Center at the University of Florida. The integrated system prototype is expected to become functional in 2001. Based on this understanding of relationships between IWiz components, and especially the role of DRE in the process of data integration, we now proceed to a detailed description of the DRE functionalities and our implementation by starting with the details of the main built-time activity, namely conflict detection and resolution.

CHAPTER 5
DATA RESTRUCTURING PROCESS

As was pointed out in the previous chapter, the main task of DRE is to convert data coming from a source to the format of the global schema used by IWiz. This global schema is defined using the "top-down" approach (see Section 3.1); i.e. the global schema is a set of concepts pertinent to a particular application domain and may not be the exact union of the source information. From here on, we use the term "ontology" when referring to the global schema used by IWiz.

DRE is the only component that has access to the source data and full knowledge of the underlying source structure. The other components operate on data which are expressed in terms of the ontology. Therefore, it is the purpose of DRE to restructure the data in order for the other components to use the source information.

The process of data restructuring relies on information that specifies actions that are necessary for accurate data conversion from source context to the IWiz context. Creating this so-called restructuring specification is the focus of the built-time component of DRE. This involves identifying similarities and discrepancies between various structural aspects of the source and ontology schemas. Effectively, the built-time process attempts to find ways to convert elements of the source schema to elements of the ontology schema. The methodologies for schema conversion and integration have existed for years in context of relational and object-oriented databases [3, 12, 16]. Recently, they have developed into new techniques applicable to semistructured data [6,

15]. A typical integration process goes through four basic steps [5]: preintegration, schema comparison, conforming of the schemas, and merging and restructuring of intermediate results. Preintegration refers to general schema analysis that helps determine the integration strategy. In the IWiz context, preintegration would correspond to human assessment of a data source that can be potentially added to the system. The second step, schema comparison, directly corresponds to the conflict detection by DRE at built-time. Conforming the schemas is the process of resolving conflicts discovered in the step two. The last phase, merging and restructuring, refers to combining of schemas. This step is used in methodologies where an integrated system view is formed directly from source concepts, in "bottom-up" fashion, and is omitted in IWiz.

In terms related to XML documents, the built-time process compares the source and ontology DTDs and attempts to find a mapping for each source concept in the ontology schema. A valid XML document must have a single root element, which is an ancestor of every other element in the document, therefore, any DTD compliant with XML grammar can be viewed as a tree structure. The mapping process starts at the roots of both source and ontology trees and proceeds recursively down the trees advancing in preorder traversal fashion. At each point in time, the mapping process works with a single source tree node trying to determine if the ontology tree contains a node that can be regarded semantically equivalent to the source node. If such a node is found, the mapping between the source and ontology nodes is recorded in restructuring specifications. The information for each mapping includes paths to the nodes from the root of the respective tree, an optional data conversion function, and auxiliary processing

data. The detailed description of the algorithm and the format of restructuring specifications are given in Chapter 6.

Before we proceed with the our detailed description of conflict detection and resolution algorithms, we first need to state the conditions that are sufficient in order for two concepts to be considered semantically equivalent during the mapping process. The goal of the built-time process can now be restated as finding a semantically equivalent concept in the IWiz ontology for each concept occurring in the source. The following five assertions must hold in order for two concepts to be considered semantically equivalent:

1. Nodes in the ontology schema represent unique real-world concepts. The same is not required for the source DTD. The unique concepts in the source schema will produce one-to-one mappings. Presence of analogous concepts in the source will produce many-to-one mappings. (To handle one-to-many and many-to-many mappings the methodology to choose the best mapping out of several possibilities is necessary. See "Future Work" in Chapter 8).
2. Both the source and ontology concepts are specified by their full paths starting at the root of the tree. The full path information allows to exactly locate each node in the respective tree. This knowledge is crucial in resolving conflicts that involve parents/children of current nodes.
3. The source concept is determined to be a semantic match for the ontology concept with the help of the customized thesaurus or by an expert user.

4. The source path is fully or partially mapped to the ontology path, i.e. one or more nodes in the path to the source concept that is currently being mapped correspond to nodes in the path to a potential ontology counterpart.
5. Detected conflicts are unambiguously resolved. In case of several conflicts, ALL of them must be satisfactory resolved in order for a mapping to be valid. Conflict resolution may be automatic or user-assisted.

If all of the conditions above are satisfied, a source concept can be successfully mapped into its ontology counterpart. After seeing how conflicts originate and the role they play in the mapping process, we proceed to a detailed classification. For each category, we will present a definition, an example illustrating the conflict, an algorithm to determine if the conflict prevents a successful mapping from taking place, and implications that the conflict might have for the data conversion at run-time.

As mentioned before, data can be modeled in XML in form of elements or attributes. Hence, the only possible mappings are between elements and attributes. Each conflict can be applied to any of the following combinations Element-Element, Attribute-Element, Element-Attribute, or Attribute-Attribute. For each conflict, we show an example of each category. In most cases, the solutions for a given conflict are identical for all four combinations, in which case a single solution is provided. In case when a specific solution is required for each combination, four corresponding approaches are shown.

In the figures illustrating the conflicts, a fragment of the source tree is always presented on the left, and a fragment of the ontology tree is presented on the right. No assumptions about the contents of the elements (structure of the trees rooted in these

elements) are made, unless their structure is explicitly shown. The elements are represented by darker rectangles with solid contours. The attributes are lighter rectangles with broken-line contours.

In general, conflicts between two XML documents can stem from differences in the document structure or from differences in data representation. The conflict classification for valid XML documents used in our project was developed during the course of the IWiz project at the Database Research and Development Center [35]. According to this classification, all conflicts can be divided into three major groups: structural, data-structural, and data conflicts.

Table 1: Conflict classification

Classes	Categories	Section
Structural Conflicts	Case Sensitivity	5.1.1
	Synonyms	5.1.2
	Acronyms	5.1.3
	Homonyms	5.1.4
	Generalization/Specialization	5.1.5
	Aggregation	5.1.6
	Internal Path Discrepancy	5.1.7
	Missing Item	5.1.8
	Element Ordering	5.1.9
	Constraint Mismatch	5.1.10
	Type Mismatch	5.1.11
Structural-Data Conflicts	Schematic Discrepancy	5.2.1
	Scale or Unit	5.2.2
	Precision	5.2.2
	Data Representation	5.2.3
Data Conflicts	Case Sensitivity	5.3.1
	Synonyms	5.3.1
	Acronyms	5.3.1
	Homonyms	5.3.2
	ID-value	5.3.3
	Missing Data	5.3.4
	Miss-spelling	5.3.5

As was noted in Chapter 2, each valid XML document consists of data definition part (DTD) and a part containing data conforming to this DTD. Structural conflicts are concept representation discrepancies between source and ontology DTDs. They can be detected without analyzing the data instances that make out the document. Structural-data conflicts also involve different schematic representations of related concepts in source and ontology, but require knowledge of some characteristics of the data in the documents for their resolution. Data conflicts are semantic conflicts; they stem from variations in data representations of related concepts. To recognize data-structural and data conflicts and provide ways to solve them, the actual source data have to be inspected. The rest of the chapter delineates different categories of the possible heterogeneities between two valid XML documents and methods to resolve those heterogeneities.

5.1 Structural Conflicts

The first four conflicts in this class, namely Case Sensitivity, Synonyms, Acronyms, and Homonyms, represent naming conflicts, i.e. differences between names used for source and ontology concepts.

5.1.1 Case Sensitivity Conflict

Case sensitivity means differences in text cases of element names that are identical otherwise. For example, elements <Book> and <book> in Figure 13 have case sensitivity conflict.

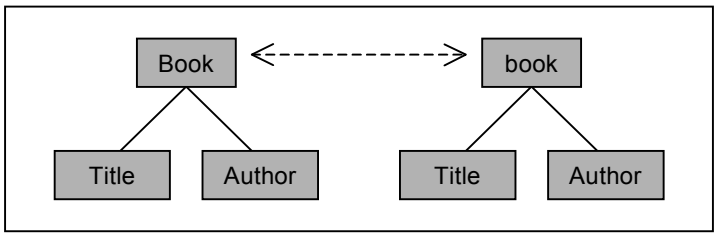

Figure 13: Structural Conflicts. Case Sensitivity.

Our system can recognize conflicting items as semantically equivalent or not depending on situation-dependent preferences. For example, the expert user can choose to treat terms with differences in text case as related or unrelated.

5.1.2 Synonym Conflict

Synonyms are morphemes with similar meaning and different appearance. Since synonyms describe the same real-world concept, the elements represented by synonyms are considered related.

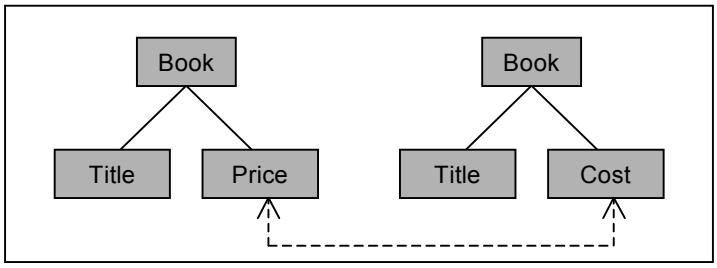

Figure 14: Structural Conflict. Synonyms.

Our approach to resolving synonym conflicts (as well as other naming conflicts) is to employ the customized thesaurus. The thesaurus includes all terms defined in the ontology DTD and provides a list of synonyms for each entry. The detailed description of building and employing the thesaurus by DRE is given in Section 6.1.

If the source concept name is found among the synonyms for the ontology concept name in the thesaurus, the ontology concept is listed as a potentially valid

mapping for the source concept, otherwise, the automatic mapping process for the source concept fails. In the example in Figure 14, if the synonyms for the name Price in the thesaurus include the word Cost, this ontology element is a candidate for a valid mapping for the source element < Price >.

5.1.3 Acronym Conflict

An acronym conflict can be seen as a special case of the synonym conflict. The term MS in Figure 15 is an acronym for Master of Science. The acronyms (if there are any) for each ontology concept are included in the thesaurus, the acronym conflicts are resolved in the same fashion as the synonym conflicts by the system.

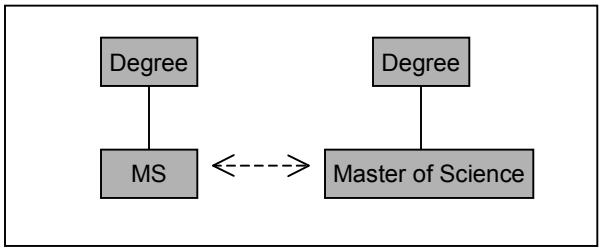

Figure 15: Structural Conflicts. Acronyms.

5.1.4 Homonym Conflict

Another example of a naming conflict is a homonym conflict. The same set of characters can have several unrelated meanings or represent unrelated concepts, like the <Name> elements in Figure 16. In the case when the source and ontology concepts are represented by the same morpheme, but differ in the meanings which are attached to them, automatic conflict detection is not able to capture this semantic inconsistency. The concepts are classified as semantically equivalent, and expert user input on later stages of the mapping process can correct the situation.

Although it seems that many incorrect mappings will find their way into the restructuring specifications, in reality, the decision about semantic equivalency of two concepts is based on several factors as described in the beginning of this chapter. Since the concepts represented by homonyms are not related to each other, there is a good chance that other factors will prevent the incorrect mapping from appearing in restructuring specifications.

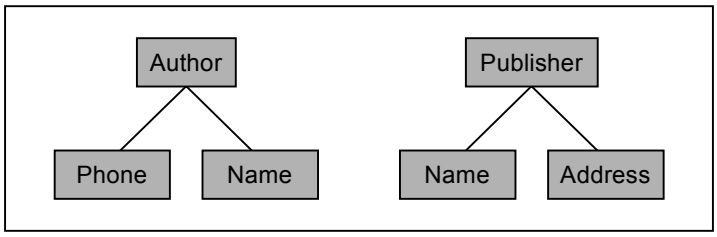

Figure 16: Structural Conflicts. Homonyms.

It is important to note that during the automatic conflict detection, synonym, acronym, and homonym conflicts are not encountered as separate heterogeneities. They present different aspects of the same conflict. When the mapping process compares the names of the source and ontology concepts they are either synonyms, or acronyms, or homonyms, or unrelated. The first two cases produce a valid mapping between the two concepts. In the last two cases, the concepts are incompatible. A concept name by itself does not provide any information or constraints for contents of an element/attribute represented by this name, therefore, naming conflicts do not entail any data conversion actions.

5.1.5 Generalization/Specialization Conflict

This conflict arises in the situation where a source concept has a more general (special) meaning than the related ontology concept. To define a mapping between a

more general and a more special term, these terms can be treated as synonyms, or, alternatively, the mapping algorithm can take into account various degrees of the equivalent relationship (e.g., equivalent, related, broader/narrower term). In context of the current DRE prototype, we equate generalization/specialization conflicts to naming conflicts and apply the same strategy to resolve them, i.e. using the customized thesaurus. For discussion on an alternative approach see "Future Work" in Chapter 8.

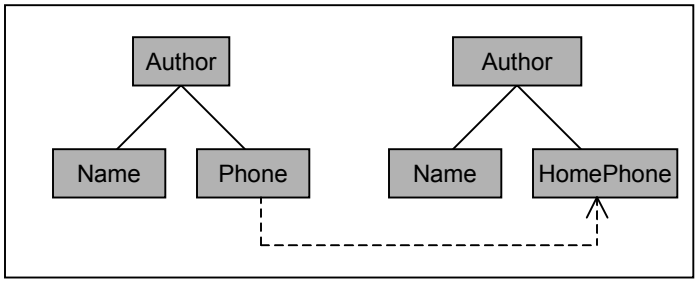

Figure 17: Structural Conflicts. Generalization/Specialization.

5.1.6 Aggregation Conflict

An aggregation conflict arises when an element whose contents include character strings (#PCDATA nodes) is mapped to an element with solely element content and vice versa. The source data values must be combined (divided) to create a new value for the restructured result. Figure 18 shows an example of an aggregation conflict. The source element <Author> is in the process of being mapped to the ontology element <Author>. In the source tree, the name information spans across two elements, <FirstName> and <LastName>, in the ontology tree, the <Author> element has #PCDATA contents. In order to create a new structure conforming to the ontology DTD the data values for <First Name> and <Last Name> are combined into one string.

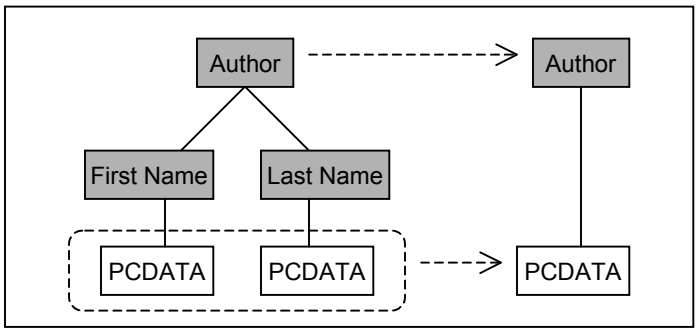

Figure 18: Structural Conflicts. Intra-aggregation

Another example of aggregation is inter-aggregation, which refers to combining the data values of the several instances of the same element. In Figure 19, the element <Authors> is declared as `<!ELEMENT Authors (Author+)>` in the source DTD, whereas in the ontology, the <Authors> element contains only #PCDATA. To resolve this inter-aggregation conflict, the data values of all <Author> elements in a source document will be aggregated to form a value for the <Authors> element in the resulting document.

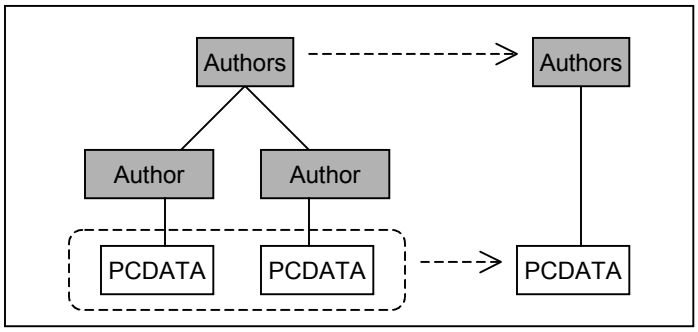

Figure 19: Structural Conflict. Inter-aggregation.

The aggregation conflict is the first type of conflict we have seen so far that requires a data transformation step for its resolution. The detection of these conflicts is based on the content model comparison for the elements involved. In our prototype, this step is handled in the context of Constraint Mismatch detection (See Section 5.1.10).

5.1.7 Internal Path Discrepancy Conflict

As stated in the beginning of this chapter, one of the conditions for determining semantic equivalency between two concepts is successful mapping of a path leading to the current source node to a path leading to the corresponding ontology node. An internal path discrepancy conflict results in only a partial mapping between source and ontology paths. For example, in Figure 20, the path to the <WorkPhone> element in the ontology tree does not include the <Phone> element that is present in the source. Note that the internal path discrepancy conflict does not apply to Attribute-Attribute mappings, because attributes in XML may not have hierarchical structure of their own.

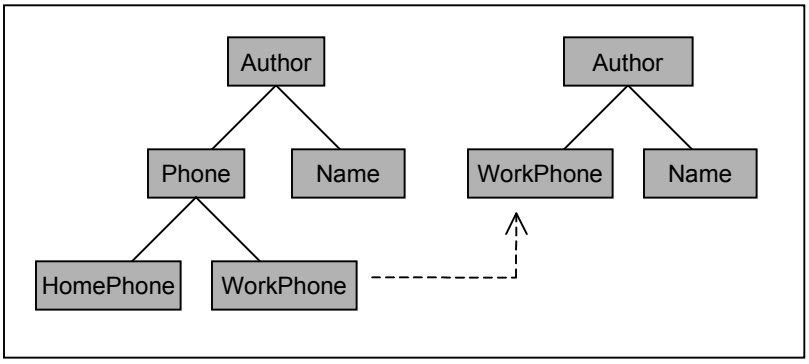

Figure 20: Structural Conflicts. Internal Path Discrepancy.

The internal path discrepancy conflict can be resolved if we can show that at least a part of a source path can be mapped to an ontology path. To determine the partial mapping, the source path is gradually reduced and compared with the ontology path after each reduction. If the reduced path can be mapped to the original ontology path, the source and ontology concepts are considered semantically equivalent. In case where the source path is exhausted and no valid mapping can be established, the ontology path is gradually reduced and attempted to be mapped to the original source path. Path

information does not provide knowledge about node contents, and presence of these conflicts does not result in any data conversion.

5.1.8 Missing Item Conflict

A missing item conflict reflects the situation where the definition of a concept in the source schema contains an element/attribute that is missing in the ontology schema and vice versa. In Figure 21, the publisher information is missing from the ontology schema.

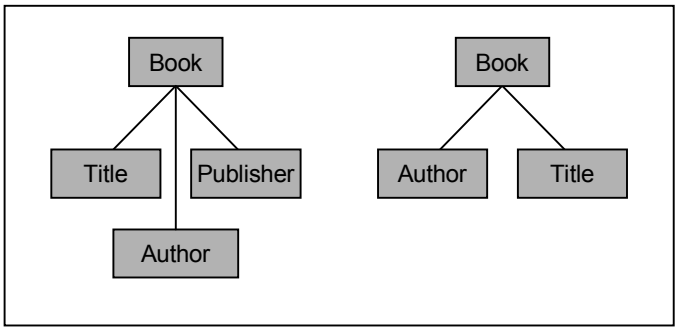

Figure 21: Structural Conflicts. Missing Item.

Our current approach to determining semantic equivalency between two nodes does not include analysis of child elements. Therefore, the missing item conflict is not detected in the form given in the definition above, i.e. as a difference between two sets of child nodes. However, it can be partially detected and resolved when the algorithm advances to the level of the missing element, for example from the level of the <Book> element to the level of the <Title>, <Publisher>, and <Author> elements in Figure 21.

If the missing item belongs to the ontology schema, the corresponding source concept is left unmapped at this point (it may be mapped to another ontology concept when the algorithm advances to deeper tree levels). The missing source element or attribute can cause a problem if the ontology requires the data for this element to be

present in order to form a valid XML document. A general solution minimizing data loss during restructuring is to provide a default/empty value for the missing concept. Other custom, user-defined data conversion actions are possible.

5.1.9 Element Ordering Conflict

The XML data model is an ordered graph. Two elements that consist of the same subelements arranged in a different order cannot be mapped to each other without additional restructuring. Since our current approach does not analyze child composition of an element in order to produce a mapping from the source to the ontology concept, the fact that the children elements of the two <Book> elements in Figure 21 have different order does not affect the outcome of the mapping process for the <Book> elements. However, as in the case of the missing item conflict, the element ordering conflict can be detected and resolved on the subelement tree level.

The reordering happens automatically when mappings for each subelement are created. The data values are placed in the right order. For example, in Figure 21, the <Author> and <Title> child elements of the <Book> have a different order in the source and ontology trees. When mappings between the two <Author> nodes and the two <Title> nodes are defined, the appropriate data value will be appended to the appropriate node at the data restructuring phase. The element ordering conflict is not applicable to attribute lists, since XML does not impose any order on attribute collections.

5.1.10 Constraint Mismatch Conflict

The W3C Recommendation for XML provides a way to specify cardinality constraints for elements and attributes in a DTD. The Kleene star "*" denotes that the

element can be present zero-or-more times, one-or-more times if marked by a "+", or zero-or-one time if marked by "?". If no constraint is present in a DTD, the element is assumed to be present exactly once. Each attribute definition must include one of the following constraints: #REQUIRED, #IMPLIED, #DEFAULT, or #FIXED. In the mapping process, source and ontology concepts can have conflicting constraints. In general, conflicts arise when an element/attribute with the least restrictive constraint is mapped to an element/attribute with a more restrictive constraint. For example, in Figure 22, the optional element <Year> in the source tree is mapped to the required attribute <Year> in the ontology tree.

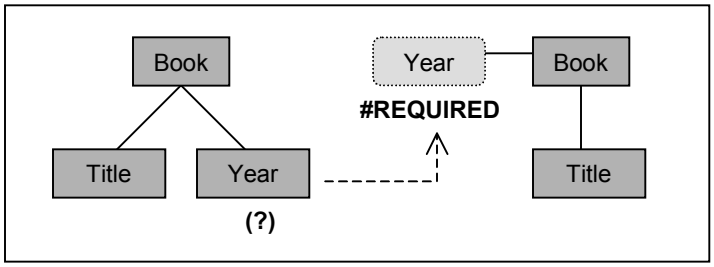

Figure 22: Structural conflicts. Constraint mismatch.

Since an element can be mapped to either an element or an attribute or vice versa, we have to investigate all possible constraints pairs. The tables below enumerate the constraint combinations and provide solutions for various resulting conflicts.

The first column contains a cardinality constraint for the source element; the second column represents a constraint for the ontology attribute; and the last column states an action that should be taken to resolve a conflict if one is present. In the case where the constraints of the source and ontology concepts do not conflict, the last column contains a "No conflict" entry.

Table 2: Constraint combinations for Element-Attribute mapping

Source Element Constraint	Ontology Attribute Constraint	Proposed Action
Exactly One	#REQUIRED	No conflict
	#IMPLIED	No conflict
	#DEFAULT	No conflict
	#FIXED	User validation
Zero-or-One	#REQUIRED	No conflict
	#IMPLIED	No conflict
	#DEFAULT	No conflict
	#FIXED	User validation
One-or-More	#REQUIRED	Inter-aggregation
	#IMPLIED	Inter-aggregation
	#DEFAULT	Inter-aggregation
	#FIXED	User validation
Zero-or-More	#REQUIRED	Inter-aggregation
	#IMPLI ED	Inter-aggregation
	#DEFAULT	Inter-aggregation
	#FIXED	User validation

Table 3: Constraint combinations for Element-Element mappings

Source Element Constraint	Ontology Element Constraint	Proposed Action
Exactly One	Exactly One	No conflict
	Zero-or-One	No conflict
	One-or-More	No conflict
	Zero-or-More	No conflict
Zero-or-One	Exactly One	No conflict
	Zero-or-One	No conflict
	One-or-More	No conflict
	Zero-or-More	No conflict
One-or-More	Exactly One	Inter-aggregation
	Zero-or-One	Inter-aggregation
	One-or-More	No conflict
	Zero-or-More	No conflict
Zero-or-More	Exactly One	Inter-aggregation
	Zero-or-One	Inter-aggregation
	One-or-More	No conflict
	Zero-or-More	No conflict

Table 4: Constraint combinations for Attribute-Attribute mappings

Source Attribute Constraint	Ontology Attribute Constraint	Proposed Action
#REQUIRED	#REQUIRED	No conflict
	#IMPLIED	No conflict
	#DEFAULT	No conflict
	#FIXED	User validation
#IMPLIED	#REQUIRED	No conflict
	#IMPLIED	No conflict
	#DEFAULT	No conflict
	#FIXED	User validation
#DEFAULT	#REQUIRED	No conflict
	#IMPLIED	No conflict
	#DEFAULT	No conflict
	#FIXED	User validation
#FIXED	#REQUIRED	No conflict
	#IMPLI ED	No conflict
	#DEFAULT	No conflict
	#FIXED	Can't be mapped, unless the fixed values are equal

Table 5: Constraint combinations for Attribute-Element mappings

Source Attribute Constraint	Ontology Element Constraint	Proposed Action
#REQUIRED	Exactly One	No conflict
	Zero-or-One	No conflict
	One-or-More	No conflict
	Zero-or-More	No conflict
#IMPLIED	Exactly One	No conflict
	Zero-or-One	No conflict
	One-or-More	No conflict
	Zero-or-More	No conflict
#DEFAULT	Exactly One	No conflict
	Zero-or-One	No conflict
	One-or-More	No conflict
	Zero-or-More	No conflict
#FIXED	Exactly One	No conflict
	Zero-or-One	No conflict
	One-or-More	No conflict
	Zero-or-More	No conflict

Conflicting cardinality constraints of the compared source and ontology concepts result in possible data conversion. The "Inter-integration" in the action column refers to

integrating data values of multiple instances of the source concept in order to form a single data value for the ontology node (details are presented in Section 5.1.6). The "User validation" action refers to a situation where the constraint mismatch conflict cannot be resolved automatically, and expert user intervention is required. From the information in the tables we can conclude that mismatching constraint conflicts are rarely fatal, they entail some data conversion, but almost never change a so far valid mapping to invalid.

5.1.11 Type Mismatch Conflict

An XML element can have one of the following content types PCDATA, ANY, EMPTY, MIXED, or ELEMENTS. XML provides an explicit notation to denote the first three types. MIXED and ELEMENTS types can be inferred by a direct evaluation of element contents. An attribute type can be one of the following CDATA, ID, IDREF, IDREFS, ENTITY, ENTITIES, NOTATION, ENUMERATION, NMTOKEN, or NMTOKENS. A type is associated with an attribute at declaration time. When we try to map a source concept to an ontology concept, we have to make sure that a source concept type can be converted to an ontology concept type without loss of structure or semantics.

Figure 23 shows examples of compatible and incompatible types for elements and attributes. The element <Title> of type PCDATA is mapped to the attribute of type CDATA. Both types represent character strings and can be safely mapped to each other. The element <Cover> of type ANY is mapped to the element of type PCDATA. In general, these types are incompatible. A custom conversion function can be used to convert the data values.

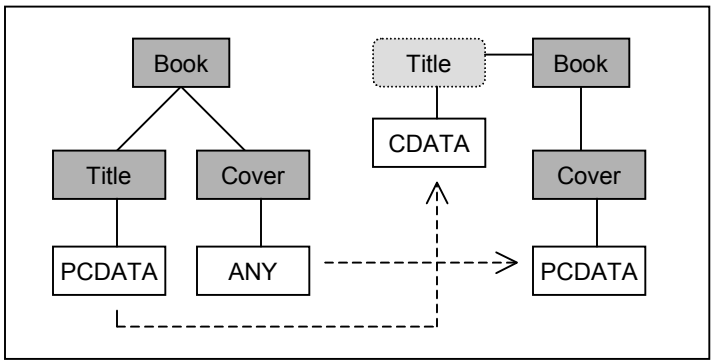

Figure 23: Structural Conflicts. Type Mismatch

Typing of elements and attributes in XML is based on different concepts. An element type represents structure of the element contents, i.e. how many children the element can have (if any) and types of those children. On the other hand, attribute values can only be character strings, and an attribute type, in fact, represents the semantics of that string. These features present problems for automatic detection of many types of conflicts. Considering only structural aspect, any attribute can be mapped to another attribute; string value from one type can be easily assigned to any other type. Expert user input is necessary to ensure that semantics is not compromised. The tables below show all possible type combinations and their effects on the mapping process. As in the previous section, the first column contains type information for the source element; the second column represents a type for the ontology attribute; and the last column states an action taken to resolve a conflict if one is present. In the case where the types of the source and ontology concepts do not conflict, the last column contains a "No conflict" entry.

Conflicting content types of the compared source and ontology concepts result in possible data conversion actions that are presented in the last column in the tables below. The "Intra-integration" in the action column refers to integrating data values of

subelements of the source concept in order to form a single data value for the ontology node. The "Divide" action refers to the opposite process, namely separating the source data value into two or more parts and assigning these parts as data values for the ontology elements (details are presented in Section 5.1.6). The "User validation" action refers to a situation where the type mismatch conflict cannot be resolved automatically, and an expert user intervention is required.

Table 6: Type combinations for Element-Attribute mappings

Source Element Type	Ontology Attribute Type	Proposed Action
ANY	CDATA	Intra-aggregation
	Other types	User Validation
EMPTY	CDATA	Conflict
	Other types	Conflict
PCDATA	CDATA	No conflict
	Other types	User validation
ELEMENTS	CDATA	Divide
	Other types	User validation
MIXED	CDATA	Divide
	Other types	User validation

Table 7: Type combinations for Attribute-Element mappings

Source Attribute Type	Ontology Element Type	Proposed Action
CDATA	ANY	No conflict
	EMPTY	Data loss
	PCDATA	No conflict
	ELEMENTS	Divide
	MIXED	Divide
Other Types	ANY	User validation
	EMPTY	
	PCDATA	
	ELEMENTS	
	MIXED	

Table 8: Type combinations for Element-Element mappings

Source Element Type	Ontology Element Type	Proposed Action
ANY	ANY	No conflict
	EMPTY	Data loss
	PCDATA	Intra-aggregation
	ELEMENTS	User Validation
	MIXED	User Validation
EMPTY	ANY	No conflict
	EMPTY	No conflict
	PCDATA	Conflict
	ELEMENTS	Conflict
	MIXED	Conflict
PCDATA	ANY	No conflict
	EMPTY	Data loss
	PCDATA	No conflict
	ELEMENTS	Divide
	MIXED	Divide
MIXED	ANY	No conflict
	EMPTY	Data loss
	PCDATA	Intra-aggregation
	ELEMENTS	No conflict
	MIXED	No conflict
ELEMENTS	ANY	No conflict
	EMPTY	Data loss
	PCDATA	Intra-aggregation
	ELEMENTS	No conflict
	MIXED	No conflict

Several type combinations involving an EMPTY element as a source type cannot be resolved either automatically or manually, they are marked with a "Conflict" entry in the last column. "Data loss" action refers to situations where an element/attribute is mapped to an EMPTY ontology element. Technically, such a mapping is legal, but source data will be lost in this transformation.

Instances of this type of structural conflicts require user intervention much more often than any other structural conflicts. To a certain degree, this is a result of a weak typing mechanism of XML. Type checking is also a problem that is not addressed in the

current version of XML Recommendation and has to be solved on the application level. Some of type mismatch conflicts can be solved by applying data conversion functions, but in many instances, these conflicts indicate irreconcilable differences that prevent the two concepts from being recognized as semantically equivalent.

5.2 Structural-Data Conflicts

Structural-data conflicts are structural conflicts that can be detected and resolved based on knowledge about domains of data values which make up the underlying XML documents. Without this information structural-data conflicts will appear as certain types of structural conflict or do not manifest themselves at all, this could be seen in examples in the following sections.

5.2.1 Schematic Discrepancy Conflict

This conflict reflects the situation where a data value for an element or an attribute in the source (ontology) schema corresponds to a node name in the ontology (source) schema, where the node also represents an element or an attribute. For example, in Figure 24, the domain for data values of the <Category> element in the source tree is {"Fiction", "Non-Fiction"}.

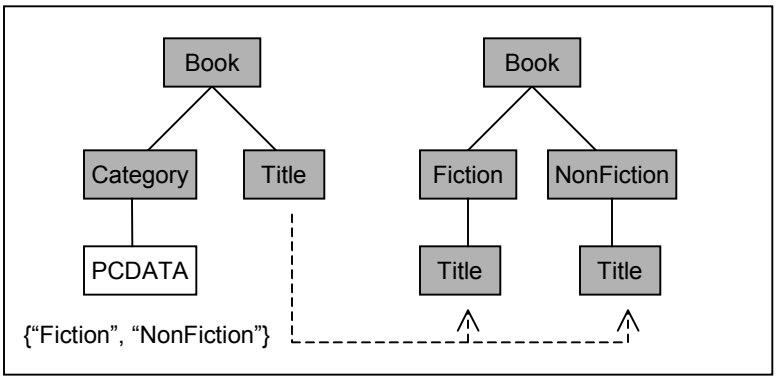

Figure 24: Structural-Data Conflicts. Schematic Discrepancy.

Clearly, without information about the domain of <Category> values, the conflict between <Title> nodes in the source and ontology trees will be classified as the internal path discrepancy conflict; and the <Title> node in the source can be mapped to either of the <Title> nodes in the ontology. The knowledge about the domain of the <Category> helps convert data in a more precise way. A title of a fiction book is appended to the node labeled <Fiction>, whereas a title of a non-fiction book is appended to the <NonFiction> node.

IWiz works with external, possibly autonomous sources that are available online, for example on the Web. IWiz is assumed to operate with only partial information about the source structure and have no control over the source contents. Under these assumptions, automatic detection of conflicts involving source data values is virtually impossible. At the time of validation of the restructuring specifications, an expert user can provide a custom function to be applied at the restructuring phase (run-time) to address the schematic discrepancy. Even then, we cannot be completely sure that examining the source gave us the most comprehensive information about the domain of the particular data item, and that a data instance with a different value will not be added to the source document later. The argument above applies to all categories of structural-data and data conflicts.

5.2.2 Scale, Unit, or Precision Conflicts

The conflict arises when two elements/attributes associated with the same concept use different scale, unit, or precision to express the same value. For example, in Figure 25, the value of the element <Weight> in the source document is represented in kilograms, while the ontology requires representing weight in pounds.

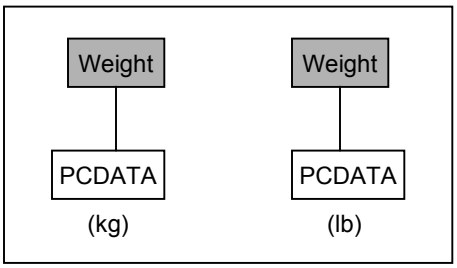

Figure 25: Structural-Data Conflicts. Unit

In general, these conflicts can be detected and the automatic unit conversion can be achieved if the data sources provide sufficient metadata (in form of attribute values in XML) to convey the used unit system. In this case, a conversion function can be chosen from the set of predefined conversion functions, or provided by an expert user. In other cases, only an expert user can detect discrepancies and take steps for resolving the conflict.

5.2.3 Data Representation Conflict

This type of conflicts stems from the fact that XML treats any data value as a string. Also, a format of a string (length or allowed characters) cannot be specified or enforced. Again, attached metadata with data type information can provide the necessary insight and allow for automatic data conversion with a default or custom conversion function. Otherwise, the task of detection the conflict is left to an expert user.

5.3 Data Conflicts

Data conflicts can only be detected and resolved at run-time during the data restructuring phase. After the restructuring by DRE, all structural conflicts are resolved and the data are considered well-formed, conforming to XML syntax. However, the

semantic discrepancies can still present a problem when data values are compared during data merging and may result in a semantically incorrect query result.

5.3.1 Case Sensitivity, Synonym, and Acronym Conflicts

The definitions of these conflicts are identical to similar structural conflicts except that the actual data values are compared instead of node names. Structural differences in naming present problems during structural transformations; semantic conflicts involving naming can be a problem at the data conversion phase. Specifically, unrecognized semantically equivalent values (synonyms, acronyms, or terms using differing text cases) can avert value-based joins, contribute to propagation of duplicate information, or be responsible for incorrect results in queries involving count or aggregation functions.

To resolve naming structural conflicts we use the custom thesaurus that includes all ontology concepts and a list of synonyms/acronyms for each concept. A similar approach could be envisioned for automatic resolution of semantic discrepancies among data values. This strategy might work particularly efficiently for sources with reasonably limited and stable data sets, or data sets consisting of common terms. Building the appropriate thesaurus for such a data set is a tedious yet feasible task. In the case when the source is large or contains an abundance of proper names or special terms as data values, the thesaurus may be large and inefficient.

5.3.2 Homonym Conflict

When two identical data items represent unrelated real-world concepts, for example, a combination of the first and last name identical for two different people, the homonym relationship can cause inaccurate results, which occur when unrelated data

items are joined. In case of homonyms among data values, these problems can be prevented by imposing additional constraints on the data sets, for example comparing not only names, but also dates of birth or SSNs, or consulting with an end user to decide if the presented information is consistent.

5.3.3 ID-value Conflict

An XML element can be associated with an attribute of type ID. ID values are unique in context of the original XML document, but they may not be unique across several integrated documents. Identical IDs can point to unrelated entities in the new integrated document, or the same data item can be represented by several different ID values in different parts of the integrated document.

At each point in time, an instance of DRE is associated with a single data source, which means the ID conflict cannot occur at the DRE level. However, it can be detected at the levels, where data from multiple sources are merged, namely the mediator and the warehouse manager.

5.3.4 Missing Data Conflict

This type of conflict can arise when two data instances represent the same real-world object, and data values for one or more components is missing. This conflict is likely to occur during the merging of data from different sources and, therefore, is not detected by DRE, which is associated with a single source only.

5.3.5 Miss-spelling Conflict

Since all XML values are of type string, correct spelling plays an important role in applications dealing with XML data. In a sense, miss-spelled data values can be

treated as synonyms for the correctly spelled entries; the conflict resolution strategy used for synonym conflicts can be used for this type of conflict (aside from basic spell checking).

We can see that the number of semantic conflicts that arise during the data transformation for a particular source is directly proportional to the quality of the data in the source. Extensive error-checking at the time of data entry can eliminate most of the naming, missing data, and incorrect spelling conflicts.

From the description of the conflicts in this chapter, we can conclude that structural conflicts have a much better chance to be automatically detected at built-time and subsequently resolved at run-time. The structural-data conflicts sometimes can present themselves as structural conflicts and be partially solved as ones. The data conflicts cannot be seen at all during the structural comparison. They might be resolved in DRE at run-time with the help of user-defined custom functions or at the higher level in the system, for example in the mediator or the warehouse manager.

In addition, we have to note that in the process of finding a mapping from a source concept to a related ontology concept, several conflicts from several different classes of conflicts can be present at the same time. For example, two concepts can be structural synonyms, have a constraint mismatch, use conflicting unit representations, and the data value in one of them can be miss-spelled. This work considers possible interactions between different instances of structural conflicts in sections devoted to implementation of automatic conflict detection. However, interactions between structural and other classes of conflicts as well as interactions within other conflict classes are not in the scope of this thesis and represent an interesting topic in themselves.

CHAPTER 6
THE DRE ARCHITECTURE AND IMPLEMENTATION

The Data Restructuring Engine (DRE) is a software layer that can communicate with the source, has an access to a source DTD and, hence, full knowledge of the source structure. DRE is invoked by the mediator and its main goals are to extract data from the source based on queries received from the mediator, restructure the data to conform to the global IWiz schema, and return the restructured data back to the mediator.

The two major functions carried out by DRE are generation of the restructuring specifications at built-time and querying of the source at run-time. The built-time activities are carried out once for a given combination of source and ontology schemas. Figure 26 presents the architecture of DRE and shows control flow between the DRE components. Darker entities represent DRE components; lighter elements represent input/output XML documents.

The major DRE components are the Thesaurus Builder, the Conversion Specification Generator (Spec Generator), a graphical user interfaces for validation of the thesaurus and the conversion specifications, the Query Rewriter, and the Data Processor. The DRE prototype is implemented using Java (SDK 1.3) from Sun Microsystems. Other major software tools used in our implementation are the XML Parser from Oracle version 2.0.2.9 [33] and the XML-QL processor from AT&T version 0.9 [51].

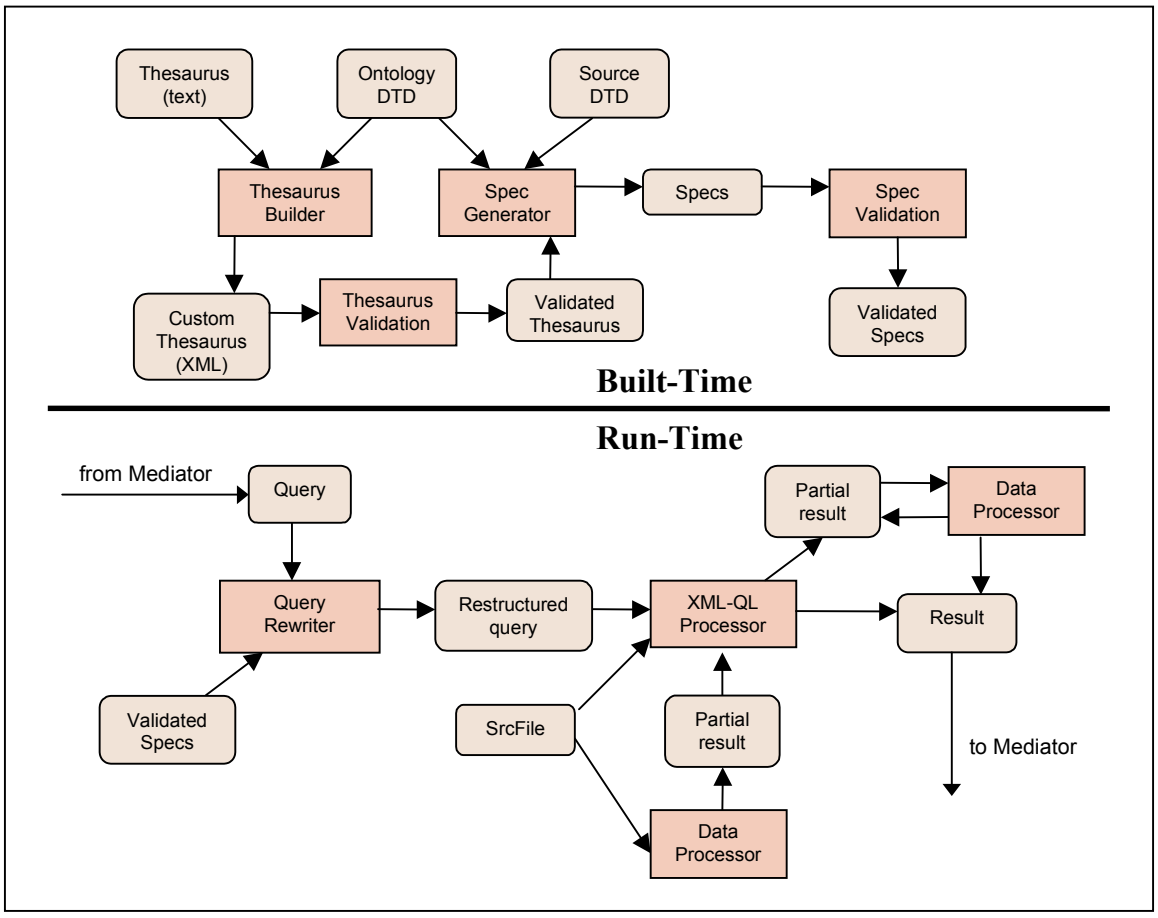

Figure 26: DRE architecture

Each major DRE component corresponds to a Java class. Each of these classes implements the `DREConstants` interface. The interface contains constant values used by all classes throughout the restructuring process. The description of each class includes functionality of constructors and public methods as well as discussion of algorithms used in the implementation.

6.1 Thesaurus and Thesaurus Builder

The build-time process starts with creation of the thesaurus. The thesaurus provides synonyms for the ontology concepts, and thus, makes it possible to map source concepts to their ontology counterparts. The quality and completeness of the thesaurus

strongly determine the accuracy of the mapping process, i.e. number of correct mappings automatically found during built-time. The process of building the thesaurus is intended to be automatic. A customized user interface is available for an expert user to validate the resulting thesaurus document. An expert user ensures semantic soundness of the found synonyms and enhances the thesaurus with acronyms, abbreviations, and other ad hoc terms. The thesaurus is a valid XML document conforming to the DTD in Figure 27.

The `<Concept>` element represents a concept in the ontology schema identified by its path. For each concept, a list of synonyms is stored in a series of the `<Synonym>` elements.

```
<?xml version="1.0">
<!DOCTYPE Thesaurus [
<!ELEMENT Thesaurus (Concept*)>
<!ELEMENT Concept (TargetPath, Synonym+)>
<!ELEMENT TargetPath (#PCDATA)>
<!ELEMENT Synonym (#PCDATA)>
]>
```

Figure 27: DTD for the customized Thesaurus.

The Thesaurus Builder component traverses the ontology DTD and extracts all concepts (elements and attributes) found in the DTD. Then, a list of synonyms is created for each ontology concept. A general purpose or special thesaurus can be used to provide the synonyms.

The constructor for the `Thesaurus` class takes the ontology XML file name as an argument. The constructor parses the XML file and coverts it into a DOM object. (Here and in other DRE components the Oracle XML parser is used for processing XML files.) The `build` method traverses the DOM tree and records each element or attribute found in the ontology by creating a new `<Concept>` element and adding its path to the

current node to the `<TargetPath>` element. Next, the `findSynonyms` method adds `<Synonym>` nodes for each concept. We use the ThesDB product from Wintertree Software [41], a Java package that allows programmatical extraction of synonyms from a general-purpose thesaurus containing over 77,000 English synonyms for 35,000 key words.

The DOM object created as a result of this process is then converted to the Java `Hashtable` object, with the `<Synonym>` values as the keys of the hashtable. Now we can efficiently extract `<TargetPath>` elements corresponding to each particular concept.

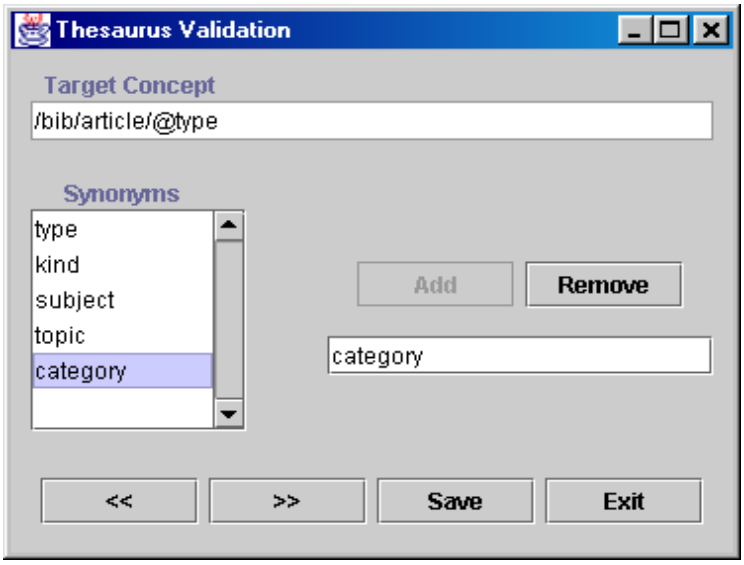

Figure 28: User interface for Thesaurus validation.

The thesaurus built by the `Thesaurus` class is validated by an expert user. The validation interface is written using the Java Swing package and is invoked automatically when the thesaurus document is constructed. The user interface displays synonyms found automatically for each ontology concept and provides options to remove synonyms in the list or add new entries as shown in Figure 28. The navigation buttons (**>>** and **<<**)

allow the user to move back and forth along the concept list. Changes made by the user are reflected in the "thesaurus.xml" file when the **Save** option is chosen or after choosing the corresponding option at the exit prompt (not shown). An error-checking routine ensures that only legal options are available to the user at any moment in time. Duplicate entries are not inserted, and the user is prompted to save or discard changes in case of unsaved changes on exit.

6.2 Restructuring Specifications and SpecGenerator

The *Spec Generator* module carries out the main function of the built-time process: it constructs the restructuring specifications for a particular source document (Figure 29). The specifications provide guidelines for converting source data to the data with the structure that conforms to the ontology DTD.

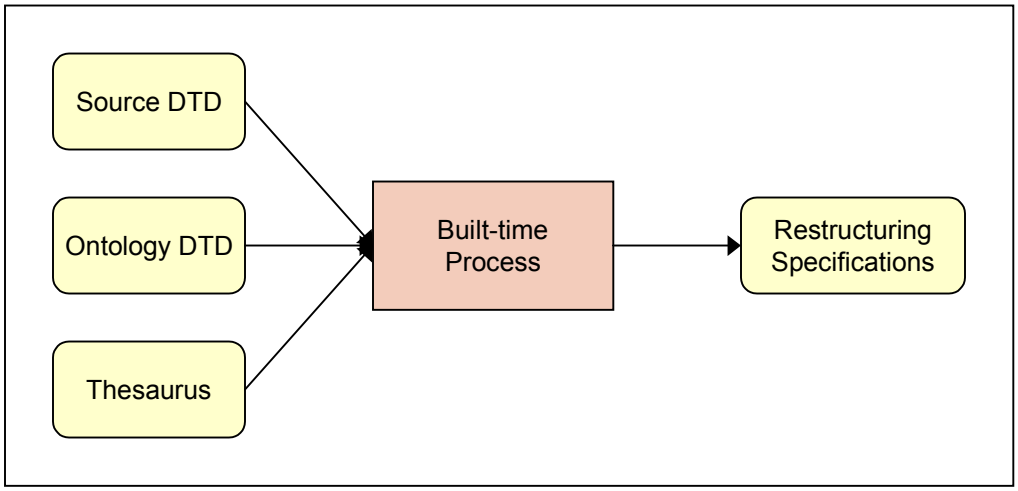

Figure 29: Built-time process.

The *Spec Generator* component detects the structural conflicts between source and ontology document structures as described in Chapter 5. The process starts by parsing the source and ontology DTDs. If the source DTD is not available, the *Spec*

Generator invokes an inferencing engine (not shown in Figure 26). The inferencing engine deduces the source DTD based on the structure of the XML source file and saves it in a "source_name.dtd" file.

The main function of the *Spec Generator* traverses the tree that is constructed during the parsing of the source DTD and attempts to find a mapping for each node in the source tree. The traversal starts at the root node. First, the path to the current element in the source tree is mapped to the corresponding path in the ontology tree. If this mapping is successful, the program determines if a content model and other constraints for source and ontology elements are compatible. The program provides the solutions based on algorithms from Section 5.1. The mappings are validated by an expert user and can be further refined via custom conversion functions. (See Section 6.3 on more details on custom functions).

The class that creates the specifications is called `SpecGenerator`. The `SpecGenerator` constructor takes the ontology and source file names as arguments. The constructor parses the source and ontology documents and creates a thesaurus by creating a `Thesaurus` object and calling `build` method of the `Thesaurus` class on that object.

The conversion specifications, created as a result of the built-time process, is a valid XML document conforming to the DTD in Figure 30.

```
<?xml version="1.0">
<!DOCTYPE ConversionSpec [
<!ELEMENT ConversionSpec (Mapping)*>
<!ELEMENT Mapping (SourcePath, TargetPath, ConversionFunction*, Warning?)>
<!ELEMENT SourcePath (#PCDATA)>
<!ELEMENT TargetPath (#PCDATA)>
<!ELEMENT ConversionFunction (FunctionName, (Element+|FunctionBody))>
<!ELEMENT Warning ((#PCDATA)?, HoldConditions?, IncludeAttributes?)>
<!ELEMENT FunctionName (#PCDATA)>
<!ELEMENT Element (#PCDATA)>
<!ELEMENT FunctionBody (#PCDATA)>
<!ELEMENT HoldConditions EMPTY>
<!ELEMENT IncludeAttributes EMPTY>
]>
```

Figure 30: DTD for the Conversion Specifications.

The principal method in the `SpecGenerator` class is called `genSpec`. This method takes the parsed DTD schemas represented by DOM objects and traverses the source tree in preorder trying to match each element or attribute in the source tree to the one(s) semantically equivalent to it in the ontology tree. In each instance, the process starts by finding entries in the thesaurus that match the current node of the source tree. In this step, the different categories of naming conflicts are detected and resolved. Next, matching entries (i.e. references to ontology nodes) are examined. A mapping can be considered final if the parent of the current node in the source tree is found to be mapped to the parent node of the examined ontology node. If no ontology concept is established as a match for the source concept at this point, the algorithm attempts to detect any internal path discrepancies that might be present. The algorithm from Section 5.1.7 is applied to handle these types of conflicts.

If the mapping for the current source node is not established at this point, the node is labeled unmapped. Unmapped concepts are still added to the conversion specifications, since a mapping for those concepts can be added during specification validation.

Next, the genSpec method determines whether there are mismatches between content models and presence constraints of the mapped source and ontology nodes. Content model types and presence constraints are computed and compared for the two nodes. All possible combinations and solutions for conflicting pairs are listed in Sections 5.1.10 and 5.1.11. If a conflict is detected, a child node named <ConversionFunction> is added to the current <Mapping> element and the default conversion function is formed. A <Mapping> element can have up to two <ConversionFunction> nodes, one with the solution for a type mismatch conflict and the other with the solution for a constraint mismatch. Each conversion function must have a name which is saved in a <FunctionName> element. As follows from Sections 5.1.10 and 5.1.11, the three possible types (and, subsequently, names) of conversion functions are DIVIDE, COMBINE SAME (refers to inter-aggregation), and COMBINE DIFFERENT (refers to intra-aggregation).

For a DIVIDE function the genSpecElement method finds all the children of the ontology node and creates an <Element> node for each of them. Thus, the default divide function, when applied to the source data, will parse the source data value and assign each token to the next element in <Element> list. For instance, in Figure 31 during the data conversion step, the default divide function is applied to the <Publisher> element in the source document. Its value is parsed into two tokens; these tokens are appended to the <Name> and <Address> elements in the restructured tree.

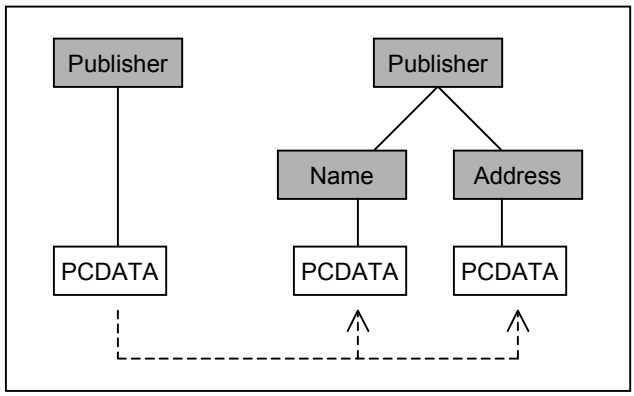

Figure 31: Example of a DIVIDE data conversion function

To create specifications for a COMBINE SAME function, the name of the current source element is saved in the <Element> node. The default function combines data of all instances of the specified element. In Figure 32, the <Author> element in source is declared to have element content model in the source DTD. At built-time, the <Authors> element is mapped to the <Authors> element in the ontology, which is declared to contain only character data in the ontology DTD. During data transformation at run-time, the data of all, in this example two, instances of the <Author> elements are combined in one string before it is appended to the <Authors> node.

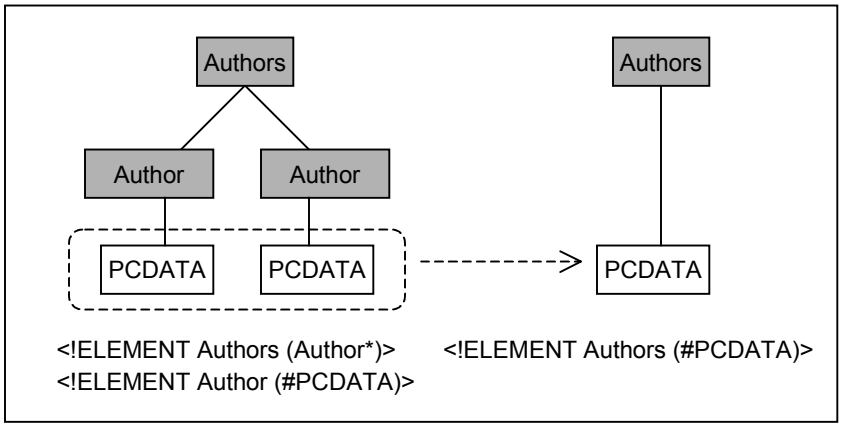

Figure 32: Example of a COMBINE SAME data conversion function

The effects of the COMBINE DIFFERENT function are the inverse of the DIVIDE action. For the default function, all child nodes of the source element are found and saved as a series of `<Element>` nodes. At the time of data conversion, data values of these elements will be combined and become a single value assigned to the node created according to the ontology DTD as illustrated in Figure 33.

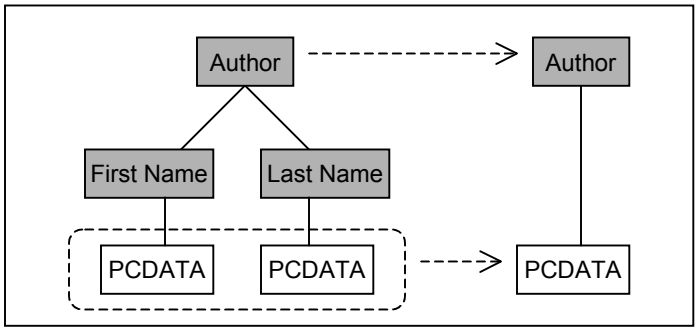

Figure 33: Example of a COMBINE DIFFERENT data conversion function

An optional `<Warning>` node can be appended to the current `<Mapping>` node. The `<Warning>` node can contain warning messages displayed to the user during the validation, and/or processing instructions (`<HoldConditions>` and `<IncludeAttributes>` flags) for the run-time procedures.

The `genSpecElement` method recursively progresses down the source tree and examines all elements and attributes. Since attribute nodes do not have child nodes, a non-recursive method `genSpecAttribute` is called if the current node is an attribute. The algorithm employed by the `genSpecAttribute` method resembles the algorithm described above. The differences in provided conflict solutions in the `genSpecAttribute` method stem from differences between element and attribute status in XML. The element nodes are mapped by the recursive method

`genSpecElement`. If the mapping is successfully established for the element, all the child nodes are identified and `genSpecElement` is applied to each of them.

As stated before, the restructuring specifications are in the form of a valid XML document. The specifications are saved in the "spec.xml" file and are later validated by an expert user via a customized interactive interface.

6.3 Specification Validation Interface

The validation of the automatically generated specification completes the run-time process. The conflict detection process described in the previous section discovers majority of the structural conflicts and provides solutions for them. However, semantic nuances, homonym relationships, irregularities in the source document, and other reasons make it virtually impossible to automatically produce absolutely reliable conversion specifications. We will show the estimated percentage of the entries that require human intervention in the chapter dedicated to the performance issues.

The specification validation interface is automatically invoked by the `SpecGenerator` class after the `genSpec` method terminates. A general view of the interface is presented in Figure 34. The left panel titled Source DTD displays the DTD of the source document. The right panel titled Ontology DTD displays the DTD of the ontology. The DTDs are presented for references.

The central panel is the most important part of the interface, where modifications to the mappings can be made. The tool bar in the top portion of the screen provides navigating capabilities (**<<**, **>>**, **2ndFunction** buttons), **Save**, **Undo**, and **Exit** functions as well as a status line that displays current warnings and/or status information.

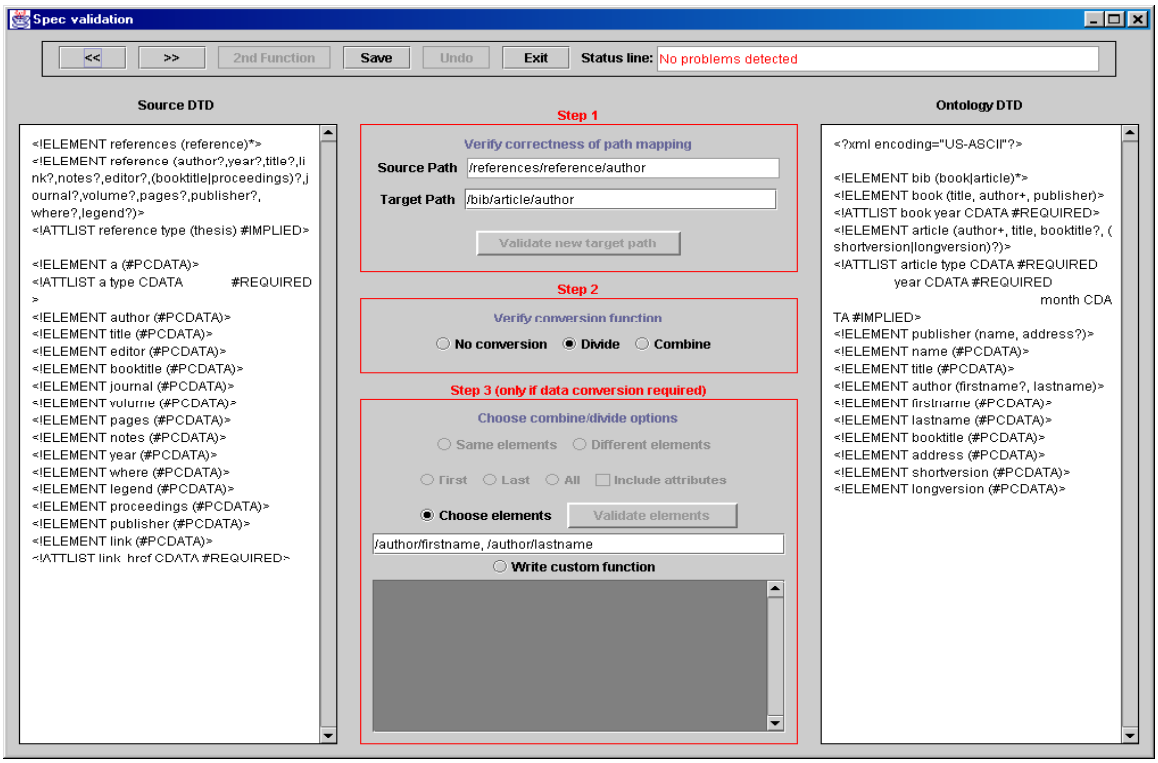

Figure 34: User interface for validation of the Restructuring Specifications.

At each point in time, the interface displays mapping details of a single element of the source DTD, i.e. the contents of a single `<Mapping>` element of the Conversion Specifications XML document. The verification process advances from top to bottom in the middle panel.

The first step is to verify the correctness of the path mapping. The fields labeled **Source Path** and **Target Path** display the contents of the `<SourcePath>` and `<TargetPath>` elements of the current `<Mapping>` element respectively. Correct path mapping resolves all the structural conflicts (see Table 1 in Chapter 5) except for Constraint Mismatch and Type Mismatch. If the suitable mapping is found by the `SpecGenerator`, it is displayed in **Target Path** field. Otherwise, the field is empty, and the warning is displayed that no suitable mapping could be found. The user has an

option to provide own mapping. **Validate new target path** function makes sure that the provided element is, in fact, present in the ontology DTD. The program blocks all other functions until an expert user enters a valid ontology path.

The second step of the validation procedure is to determine if any special data conversion has to be performed on the value that comes from the element determined by source path before this value is assigned to the element described in target path. The need for data conversion arises from discrepancies in cardinality constraints or content model types of the mapped elements as described in Chapter 5. The initial options for data conversion are **No conversion**, **Divide**, and **Combine**. If during conflict detection phase the `SpecGenerator` discovers that a data conversion has to be performed, a conversion function is automatically chosen and information describing the default function is saved in a `<ConversionFunction>` node of the specification document. When the current `<Mapping>` element has one or more `<ConversionFunction>` child nodes, the information is displayed in the validation interface via selected radio buttons/check boxes and text. The available options will be described later in this section. When a mapping has two conversion functions, the **2ndFunction** button is enabled, and the status line displays a warning that the second conversion function is present and needs validation. After pressing the button, the details of the second conversion function are displayed, and the button label becomes **1stFunction**. Thus, the button assists navigation between two conversion functions of the same mapping.

The **No conversion** option, chosen in the second step, concludes validation of the element. This option implies that no conflicts between constraints and types of the

source and target elements exist. New element can be created by simply copying the contents of the source element into target element.

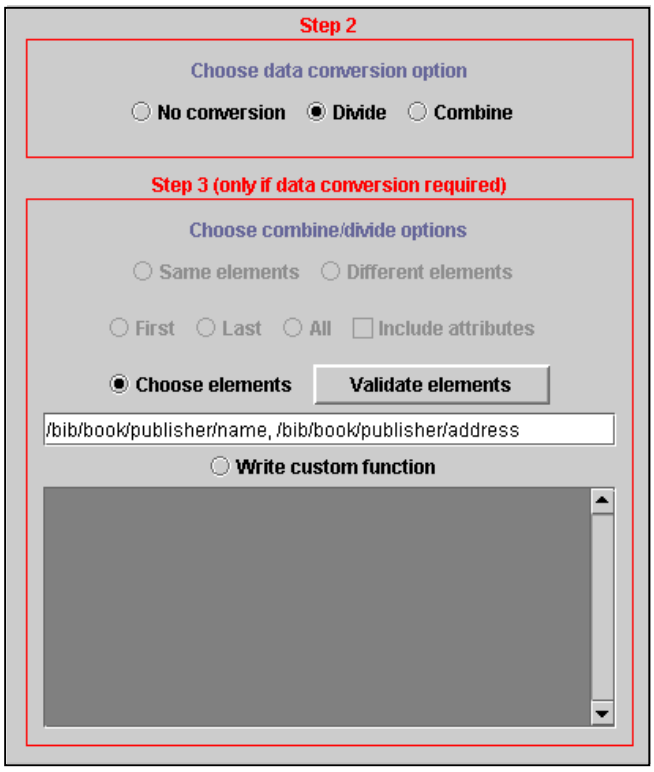

Figure 35: Options Available for DIVIDE Conversion Function

Initially, the **Divide** option is selected if the name of the current conversion function is DIVIDE. A text field will display the elements chosen by the default function. The available options to modify a divide function are shown in Figure 35. There are two ways to modify a divide function. The user can choose different set of elements/attributes or write a custom Java function in the text box provided. The default element set contains all child elements of the divided source concept. For example, Figure 35 depicts a default element set for the data conversion action in Figure 31. A default conversion function tokenizes the source data value and assigns one token per an ontology element (<Name> and <Address> in the example). A custom conversion

function might be required if we want to assign tokens in a different manner, for example the first two tokens should be assign to the <Name> element, the third token is skipped, and the rest of the string is assigned to the <Address> node. A custom conversion function is a static Java method written with the assumption that a source XML document is represented as a DOM object. DRE assigns a unique method name to each user-defined conversion function. A method receives a single argument, namely a source element that has to be divided (the <Publisher> element in our example). A person validating specifications can define various custom actions on the element using the DOM API.

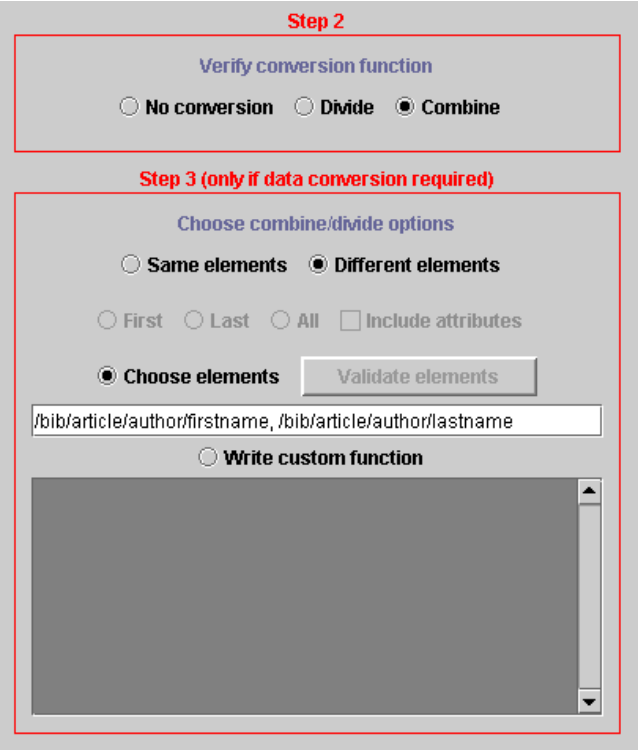

Figure 36: Options Available for COMBINE DIFFERENT Conversion Function.

In case the set of default elements is edited, the **Validate elements** button is enabled. This function makes sure that the newly chosen elements are valid components

of the ontology DTD. The selected **Combine** option indicates one of the two possible variations of the combine function. The SpecValidation interface provides options to customize both types of conversion functions. Options available to the user in both cases are shown in Figure 36 and Figure 37. As seen in the figure, when combining instances of the same element, the user can choose to retain the data value from the first instance, the last instance, or use the default option **All**.

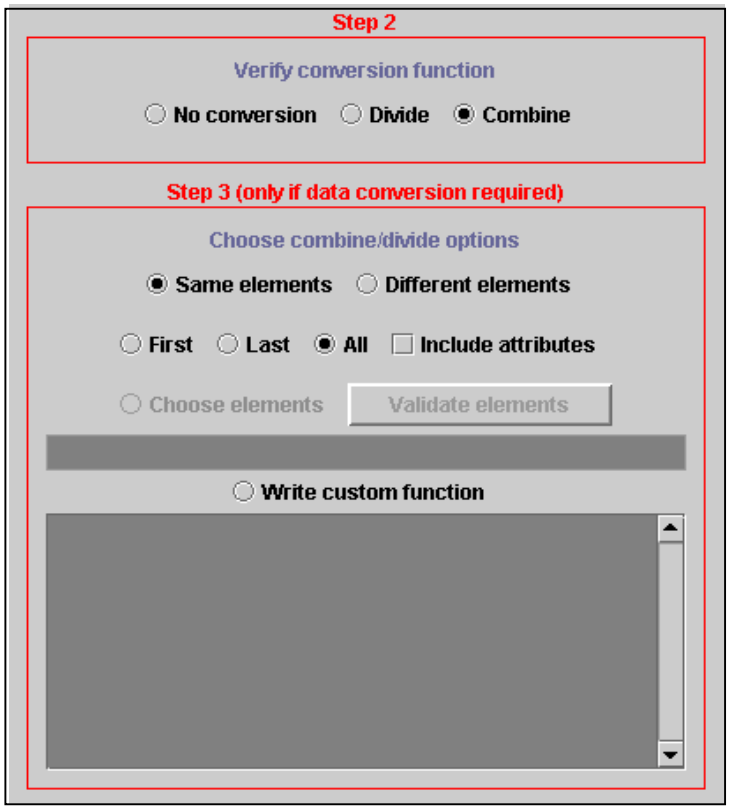

Figure 37: Options Available for COMBINE SAME Conversion Function.

The **Include attributes** check-box allows the user to combine data values for attributes with element values to produce the result. Options for combining different elements are similar to those for divide function. Both functions can be user-defined. Custom functions for combining elements are formed using the same strategy as custom divide functions.

The extensive error-checking capabilities of the user interface program make it impossible for the user to enter incorrect information affecting the document structure (e.g., enter a path that does not exist in the actual document or choose an option that does not correspond to a chosen type of the conversion function). The program ensures that created specifications are a valid XML document that conforms to the Restructuring Specifications DTD (see Figure 30).

6.4 Query Restructuring and QueryRewriter

The main objective of the run-time process is to query the source file and return the information back to the calling process, the mediator, as shown in Figure 38. We have chosen the XML-QL query language developed by Suciu, et al. [14] as the query language used by DRE and throughout the IWiz system.

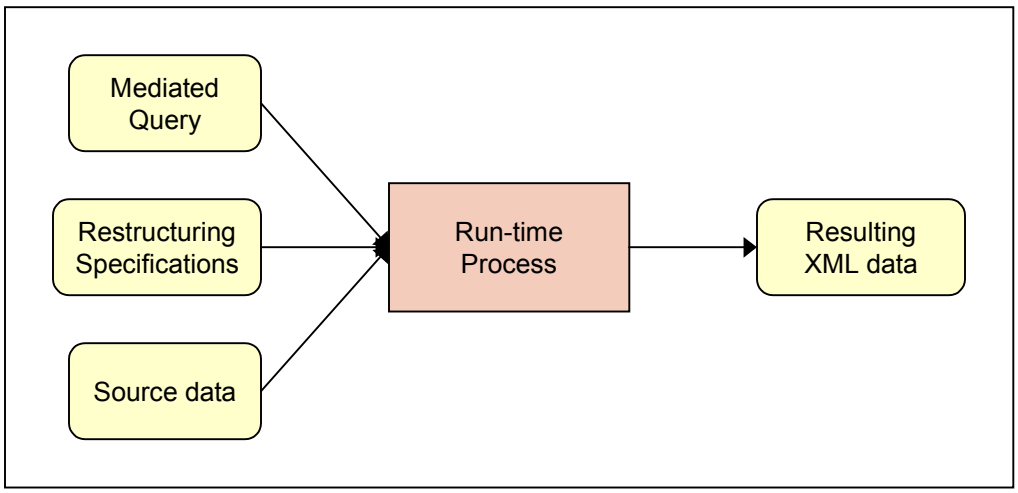

Figure 38: Run-time process.

The principal module of the run-time system, the Query Rewriter, is invoked by the mediator. The mediator provides an XML-QL query and expects the data extracted from the source according to the query as the return value.

The query can be passed to DRE as plain text or a DOM object (constructed according to certain specifications). In case of a text input, the incoming query is converted to a valid XML document and parsed into a DOM structure. Regardless of the format, the information request is expressed in terms of the ontology. The goal of the run-time process is to translate this request into terms used by the source document.

Another input to the Query Rewriter is the validated restructuring specifications created at the built-time. These are also parsed into a DOM structure. Using information in the specifications, the Query Rewriter restructures the query DOM object replacing each node with its corresponding mapping in the source document. After the restructuring, the query DOM object is converted into text form to be sent to the XML-QL processor.

The class that implements the main functionality of the run-time component is called `QueryRewriter`. The `QueryRewriter` class has two constructors. The arguments to a constructor are an XML-QL query (a string or a DOM object) and a query ID. The query ID is concatenated with the source ID, and the resulting string is used as a name for the result file. The constructors parse the restructuring specifications and the incoming query (if needed). The `rewrite` method of the `QueryRewriter` class is called on the `QueryRewriter` object to restructure the query DOM tree and to query the source.

A typical XML-QL query consists of one or more `WHERE` clauses and one or more `CONSTRUCT` clauses. Both `WHERE` and `CONSTRUCT` clauses of the incoming query are composed of the ontology terms. The `CONSTRUCT` clause specifies the format of the query result and, as a rule, does not need any restructuring. A typical `WHERE`

clause consists of the concepts and conditions that specify the information to be extracted from the particular document. Thus, the `WHERE` clause must be written in terms of that particular document, and, in our case, has to be restructured to represent concepts of the source file.

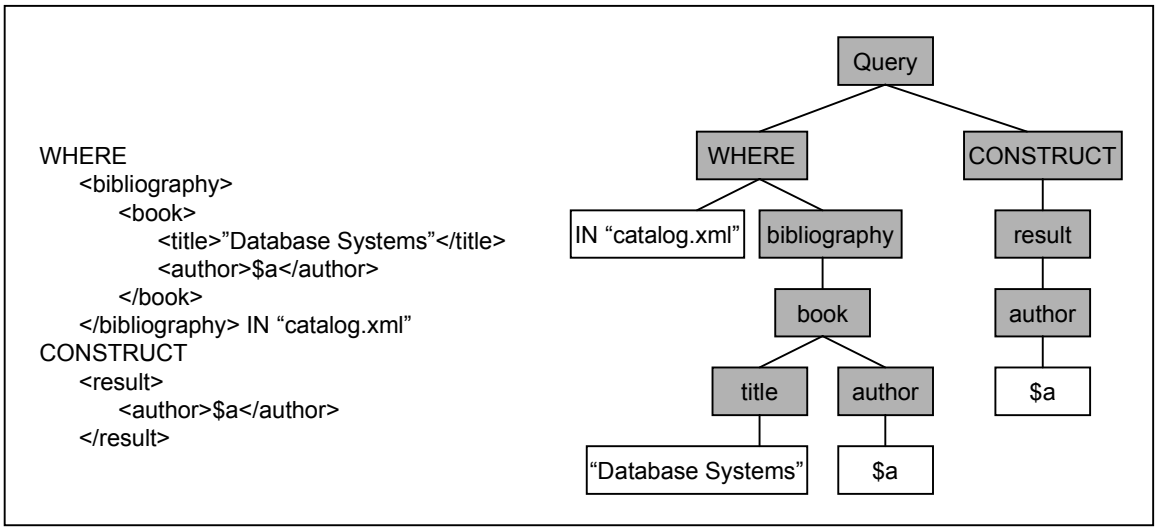

Figure 39: An example of an XML-QL query and the corresponding DOM tree.

The `rewrite` method starts by finding all `<WHERE>` elements in the query DOM tree. For each of those elements the concepts (and their respective paths) are extracted, replaced by matching source concepts from the conversion specs, and then assembled back into a DOM structure. If the mappings of the concepts, involved in this operation, contain data conversion functions, these functions are incorporated into the new query or saved to be applied separately. The newly created DOM structure for each `<WHERE>` element replaces the old `<WHERE>` element in the query DOM structure. The process is illustrated in Figure 40. The new query DOM structure is converted into a string and passed to the XML-QL processor.

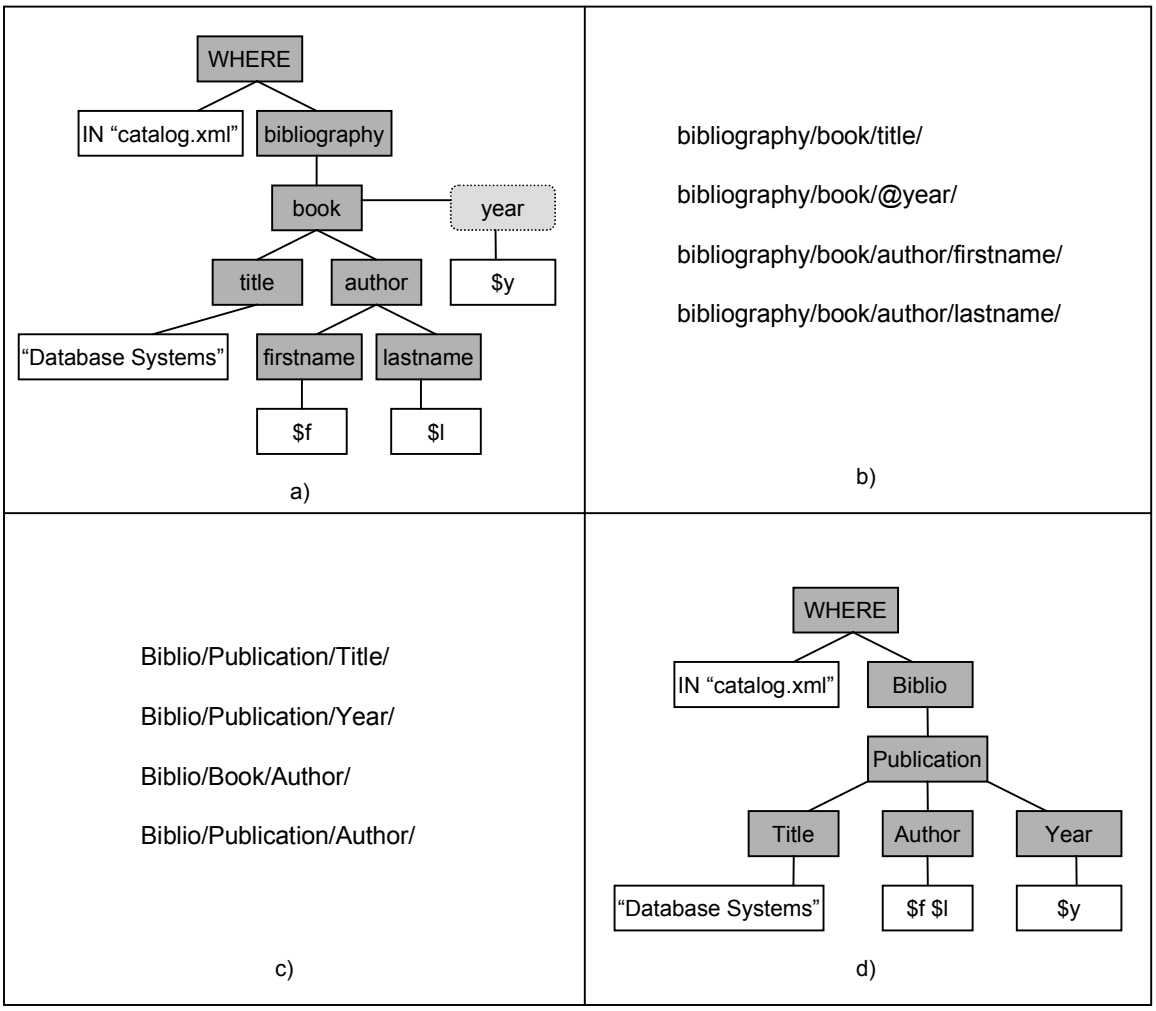

Figure 40: Query restructuring process. a) Original `WHERE` clause; b) set of source paths extracted from a); c) set of ontology paths replacing source paths in b); d) restructured `WHERE` clause.

Because of several limitations of the XML-QL processor, in some cases information has to be extracted from the source in several steps. Queries that contain elements with associated conversion functions are the most notable example. The COMBINE DIFFERENT function can be incorporated into the query as shown in Figure 41.

```
WHERE
    <biblio>
        <book>
            <author>
                <firstname>$F</firstname>
                <lastname>$L</lastname>
            </author>
        </book>
    </biblio> IN “catalog.xml”
CONSTRUCT
    <result>
        <name>$F $L</name>
    </result>
```

Figure 41: Query combining different elements

However, if the original query in Figure 41 contains a condition involving the combined data value, the condition has to be applied to the data after the query execution. Other cases that require additional data processing are the presence of custom, COMBINE SAME, or DIVIDE functions. The module responsible for data processing outside of the XML-QL processor is the Data Processor. It can be invoked before and/or after the XML-QL processor by the Query Rewriter. After all necessary data transformations, the result is serialized and sent to the mediator.

6.5 Source Querying and XML-QL Processor

The actual querying process is performed by the XML-QL processor. For the DRE prototype, we used the XML-QL processor implemented by a research group at AT&T [51]. The processor is a set of Java packages and has a command line interface as well as an API. The arguments for the querying process are a query string, the name of the source file, and the name of the output file. The output of the XML-QL processor is a DOM structure that can be saved as an XML file or processed further.

6.6 Data Processor

The Data Processor module carries out data transformations is addition to the XML-QL processor. Some transformations need to be applied to the source data and the others to the data returned by the XML-QL processor. In general, XML data can be easily manipulated after the data are converted to a DOM tree, i.e. all of the data are brought into memory at once. Since the IWiz system is designed to work with practically any external data sources, this requirement does not allow us to make any assumptions about the size of each source. We chose to use the SAX mechanism to implement this part of the system to have as much flexibility as possible with respect to memory requirements. We also assume that the query result is of manageable size and can be safely converted to a DOM tree.

Additional data processing involves applying conditions, applying custom functions, applying divide functions, and applying combine same functions. Only COMBINE SAME functions are have to be applied to the source data; the other cases are applied to query results.

Custom functions provided by the user during the conversion specifications validation are saved by the program in the designated text file. This file is compiled by the expert user as a part of the system setup. The method name for each custom function is a combination of the source and ontology element paths to ensure its uniqueness across the program. The methods are invoked as need with the appropriate arguments from inside of the *Data Processor* component using the `invoke` method of the `Method` class.

The conversion specifications contain the information sufficient to apply the divide function. The <TargetPath> element contains the path to the node whose value has to be transformed. The <Element> entries specify the resulting elements/attributes. For each occurrence of the element/attribute specified in the <TargetPath> the data value is extracted, divided into tokens, and the element itself is replaced with a set of elements/attributes created according to the information from the <Element> nodes.

For more efficient processing and because of limitations of XML-QL processor, the COMBINE SAME functions need to be applied independently from and ahead of the query execution step. As was stated in Section 2.3, there are two APIs to process XML data, namely DOM and SAX. The DOM approach requires presence of the full data tree in memory. Since the size of a source cannot be easily determined in advance, the program employing DOM interface can encounter the situation when system memory limitations may prevent successful data processing. Conversely, SAX memory requirements do not grow with the increase of the source file. The DRE implementation uses SAX interface to process possibly large XML source files.

In order to apply a COMBINE SAME conversion function, the *Data Processor* reads one top-level element at a time and applies the conversion function if needed. The modified data are written to a temporary file. Representation for only one element resides in memory at any given time.

This concludes the description of DRE implementation. The next chapter is dedicated to our evaluation of the performance of DRE.

CHAPTER 7
PERFORMANCE EVALUATION

To evaluate the performance of a system, several prerequisites are needed. In particular, sets of suitable data, sets of benchmark programs, and performance characteristics observed by comparable systems during testing. Practically all of these components are non-existent at this point in the XML world. The only XML-processing programs that are tentatively benchmarked are several XML parsers [11]. In addition, a variety of products exists to test validity and well-formedness of the XML documents [26, 33, 39]. But notwithstanding the development of all XML-related technologies and products, large sets of diverse and "real" XML data are still difficult to come by. Aside from size consideration, other characteristics of semistructured data such as markup density and degree of structure also affect the test result. For the purpose of testing of IWiz components, we use several XML sources containing bibliographic data. One of these sources is an XML representation of SIGMOD records [4] (500Kb); other sources contain smaller bibliographic collections in XML format found on the Web or were created with the help of XML authoring tools. All of these sources are reasonably well structured and conform to moderately complex DTDs.

Since no tools or methodologies for formal testing of XML-processing systems are available at the time of writing, we intend to demonstrate in an informal manner that our design approach to XML data restructuring is valid and that the implemented functionality and performance satisfy the requirements of the IWiz architecture.

The functionality of the DRE architecture can by analyzed by its applicability to IWiz. Since DRE was designed in the context of the IWiz, its functionality was defined based on the following requirements. First and foremost, DRE must be able to restructure data according to a mediator query and return an XML result conforming to a representation utilized by IWiz. DRE should take as input an XML-QL query and produce an XML document containing XML data restricted by the input query and at the same time conforming to the ontology schema. The restructured data from multiple DREs are passed to the mediator for further processing. Second, a human expert should be able to validate and/or refine the mappings generated by DRE. This step is necessary because of possible semantic inconsistencies in concept definitions that cannot be captured by DRE. Third, DRE should operate as an integral part of the IWiz system, i.e. its performance should not be a bottleneck for the system operation.

The remainder of the chapter is dedicated to DRE performance evaluation. We provide evaluation metrics for all major DRE functions, namely specification generation, query restructuring, source querying, and local data processing.

The evaluation criteria for the two main functions carried out by DRE, the built-time conflict detection and the run-time data restructuring, are significantly different. The built-time process is executed once for the combination of the source and ontology schemas. The accuracy and completeness of the mapping process are of the utmost importance, and speed is only of secondary concern. The ultimate goal of the built-time component is to automatically and correctly determine mappings for all source concepts or to correctly assert that for the given concept a mapping does not exist in the ontology schema.

The comprehensive conflict classification used in our approach is devised exclusively for the XML data model and includes structural conflicts caused by discrepancies in element/attribute declarations in DTDs. This classification covers all aspects of the standard DTD syntax rules. The built-time component implementation is closely based on this classification and, thus, is able to resolve structural conflicts between two submitted DTDs. In order to illustrate this fact, we have created an ontology and several source files for testing the IWiz system. The ontology includes 54 entities and describes various instances of bibliographic data such as books, journals, articles, proceedings, etc. The sources were described earlier in this chapter, Appendix contains DTDs for the ontology and sources used in experiments. The DRE run-time component was used to create conversion specifications for the three sources. The results are shown in Table 9.

Table 9: Accuracy of automatic conflict detection in various sources.

	Source 1	Source 2	Source 3
Total number of concepts	16	18	26
Number of conflicts present	22	22	29
Number of conflicts detected	16	20	21

The average resolution accuracy in Table 9 is 78%. To improve this number we need to analyze the reasons why the remaining mappings were not found. For example, with an automatically built thesaurus containing 841 synonyms, correct mappings for 7 concepts defined in a DTD for Source 1 were not found. Out of those seven concepts, four elements did not have a corresponding concept defined in the ontology DTD, one concept was not a part of the synonyms in the thesaurus, and mappings for two concepts required adjustments because of invalid semantics (homonym relationships). The only

factor that can be programmatically improved, out of the three reasons above, is the contents of the thesaurus. In other words, the correct resolution of detected conflicts depends on completeness (or size) of the custom thesaurus. The results in Table 9 were achieved using the customized thesaurus as described in Section 6.1. The thesaurus contained 841 concepts. As it might be expected, the size of the thesaurus affects the accuracy of the mapping process.

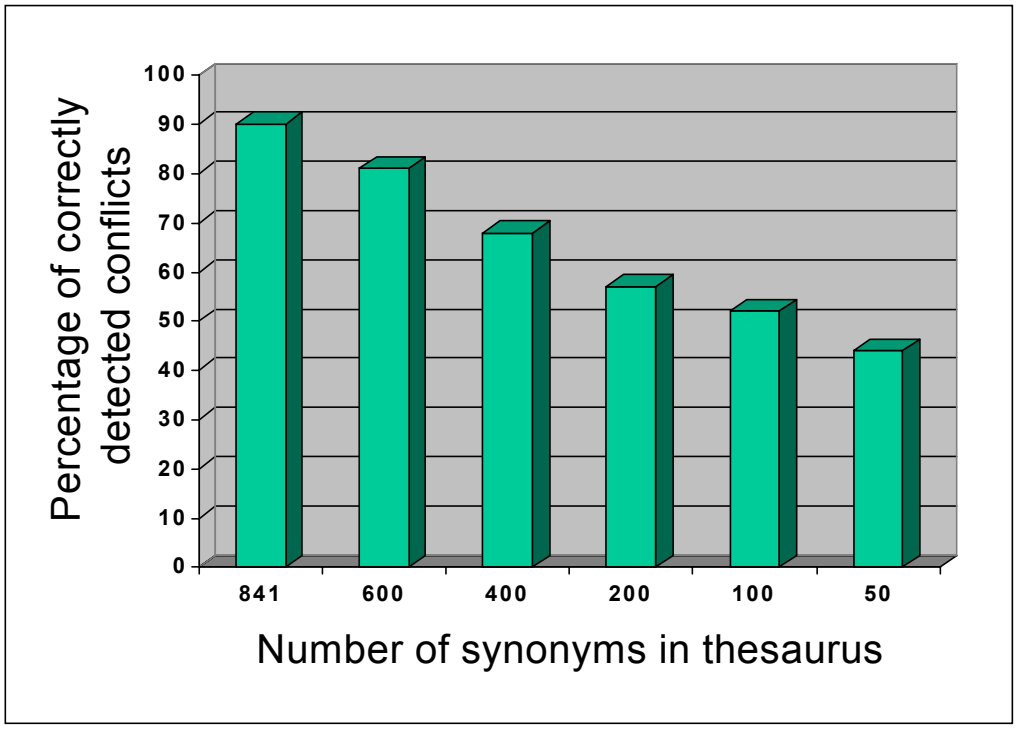

Figure 42: Conflict resolution accuracy as a function of the thesaurus size

Figure 42 shows the relationship between the percentage of the correct mapping found automatically for concepts in Source 3 and number of entries in the thesaurus. The thesaurus size was reduced by removing random synonyms. We can see that the rate of correctly detected conflicts drops, but not as dramatically as might be expected. Because the sources and ontology describe the same knowledge domain, a number of concepts are represented by the same name in both schemas. These concepts can be correctly mapped

by DRE even without the use of the thesaurus; i.e. DRE can be reasonably efficient at built-time when the thesaurus is not complete.

Unlike conflict detection and resolution, the data restructuring occurs in real-time as a response to an end user request. The run-time process can be divided into three successive steps: query restructuring, source querying, and data processing. We will separately analyze performance of each of these steps.

Each DRE component in the IWiz architecture handles data restructuring for a single data source. DRE accepts an XML-QL query in form of a DOM tree and applies restructuring algorithms to that tree. Size of a typical XML-QL query that involves concept from a single source, i.e. the size of the corresponding DOM tree, is proportional to the number of concepts in the source DTD plus the number of concepts in the ontology DTD. In most cases, the query (and DOM tree) size is considerably smaller. We have analyzed 20 random DTDs found on the Web and found that the average DTD size was 142 entities, and a DTD size ranged from 17 to approximately 500 entities. The restructuring algorithms are only applied to `WHERE` clauses of the query, which consist solely of concepts found in a particular source. Therefore, we estimate the maximum size of the DOM tree that can be involved in query restructuring is approximately 500 nodes, in most cases the size is considerably smaller. The query restructuring algorithms do not traverse the entire tree and do not involve recursion. The total time complexity for this part of the run-time process is $O(n^2)$, where n is a number of nodes in the DOM query tree. Figure 43 shows run times for the query restructuring as a function of a number of nodes in the DOM tree. A computer with an Intel Pentium II processor 400 MHz and 192 Mbytes of RAM running Windows 98 was used for this and the following experiments.

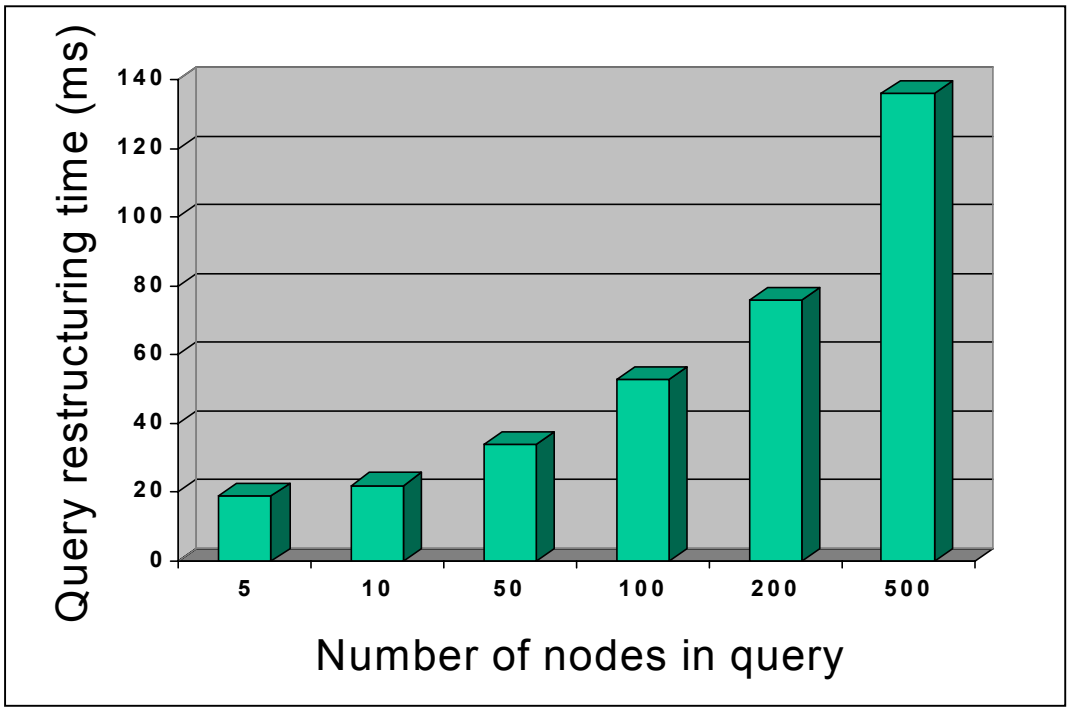

Figure 43: Query restructuring time as a function of tree size

The second step, source querying, is implemented using the XML-QL processor implemented by AT&T. The performance of this step is determined by the quality of this implementation and the size of the source file.

Figure 44 shows the relationship between query execution time and source size. A query with 4 nodes was applied to source files of different size. Figure 45 shows query execution time as a function of query size. All queries were applied to the same source with the size of 500Kb. Query run time grows approximately linearly with growth of either source or query size.

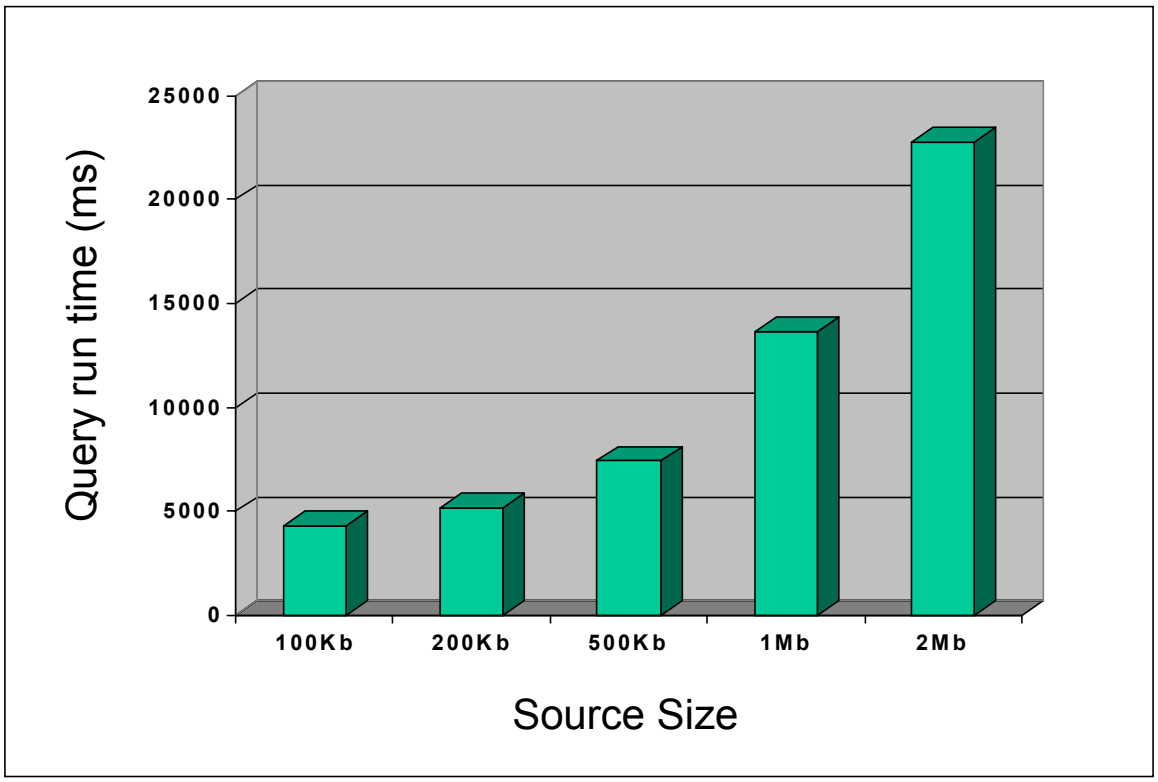

Figure 44: Query execution time as a function of source size

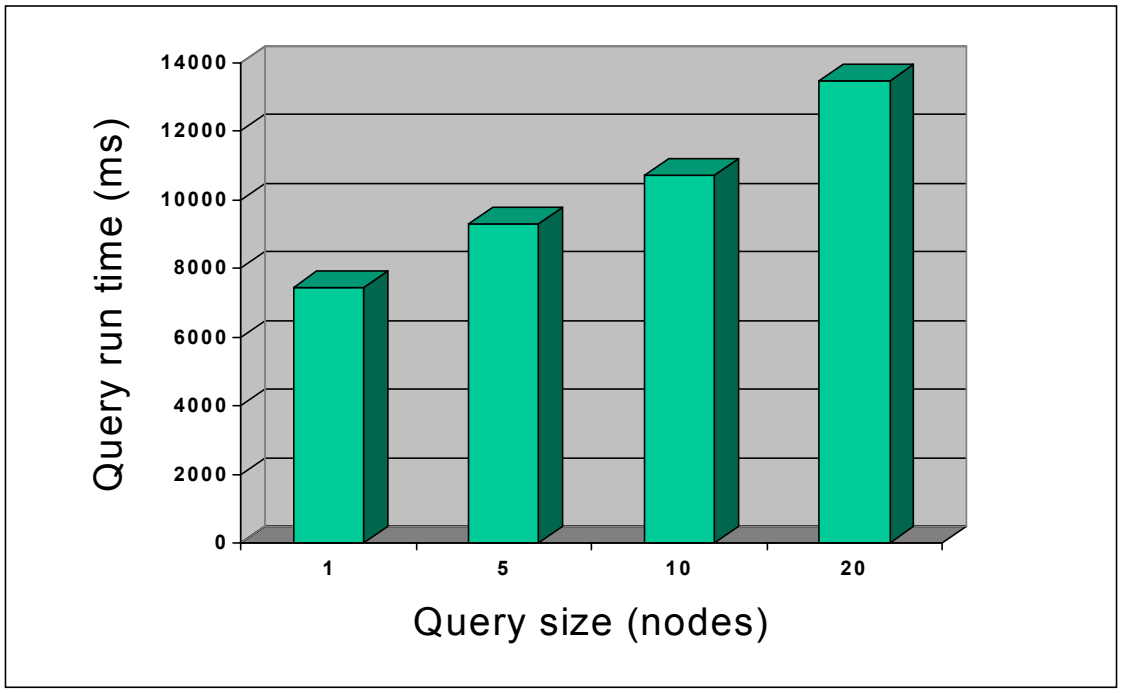

Figure 45: Query execution time as a function of query size

The last step in data restructuring is the additional data processing, which is performed outside the XML-QL processor under control of DRE. We were especially concerned with situations when conditions have to be applied to the entire set of source data. The size of a source cannot be easily determined in advance, and it can change dramatically over the time. These considerations have influenced the choice of SAX as parsing mechanism for data processing at this step. As pointed out in Section 2.3, memory requirements do not increase with growth of file size for SAX parsing. The experiment results are summarized in Table 10. Time is given in milliseconds.

Table 10: Comparative run-times for different size sources

	Source 1 100KB	Source 2 500KB	Source 2 1MB	Source 3 5MB	Source 4 10MB
DOM processing	1590	1813	2853	9776	Out of memory
SAX processing	820	1173	1443	3476	6020

It follows from the table that the functionality of the DOM API is limited by system memory. In the experimental runs the program using the DOM API ran out of memory while processing a 10Mb XML document, which corresponds to approximately 13,000 data entries, or 18,200 tree nodes. As was expected, a program using SAX did not show any increase in memory requirements and completed its task significantly faster. The drawback of the SAX approach is the fact that any non-trivial data manipulations require considerably complex programming especially in general case. To alleviate this problem, the SAX approach is used only to read one document fragment at a time, all necessary data manipulations are performed on a DOM tree built from the read fragment.

Comparing all three run-time steps, query restructuring, source querying, and data processing, we can conclude that data extraction, i.e. querying the source is the most time consuming operation of the three. The time for query restructuring is at least 2 orders smaller and, therefore, is negligible. Since query execution time is determined by the AT&T's XML-QL processor, future improvements to the run-time component should be concentrated on the third step, data processing algorithms and implementation that constitute perceptible portion of the total execution time.

We showed that the system has the required functionality and shows adequate performance. Creating evaluation methodologies and tools for semistructured data is important and complex task, which can be a topic for a substantial project in itself. We recognize the fact that evaluation of the isolated system does not provide the most comprehensive information about system performance. We hope to continue the evaluation with new tools and methodologies created specifically for XML-processing applications.

At this point, we close our discussion on DRE performance issues. The final chapter is dedicated to concluding remarks and issues to be considered in future development.

CHAPTER 8
CONCLUSION

The field of semistructured data is relatively new and not well researched yet. The concept of semistructured data is expected to change the way data are exchanged and represented on the Web. At the same time, a new class of data management problems is generated. The intent of this thesis work is to contribute to the solution of one of such problems, namely integration of heterogeneous XML data.

In this thesis, we present a solution for (semi-)automatic restructuring of heterogeneous XML data. Our solution is based on a taxonomy that classifies the conflicts that arise when the same concept is represented differently in two related DTDs. A prototype version of our algorithms, called Data Restructuring Engine (DRE), is a part of the Integration Wizard project, currently under way in the database group. DRE translates an incoming query to the format understood by the source and converts the query result into a representation utilized by IWiz. DRE algorithms rely on the existence of a customized thesaurus to establish the initial mappings between the source and the ontology concepts.

The implementation of all DRE components uses XML as an internal data model. All intermediate results generated by DRE are represented as XML documents; the internal representation of the data is based on the Document Object Model (DOM).

8.1 Contributions

The most notable contributions of this work are the following. First, we designed and implemented algorithms for automatic detection and solving multiple types of structural conflicts in XML documents based on the existing conflict taxonomy. Second, the DRE implementation includes practical interactive tools that enable expert users to verify and refine automatically found solutions. The user interface abstracts the semantics of the restructuring specifications from XML syntax and simplifies the validation process. The built-time component detects conflicts and provides specifications for their resolution in the form in many ways independent of the realization of the run-time procedures. We believe that this useful interactive tool can be used with minor modifications in other schemes that require schema translation for two related XML documents. Third, we designed and implemented algorithms for automatic utilization of conversion functions during data restructuring.

As we mentioned in Section 3.2, the current design and implementation of DRE is based on the work previously done in this field by Renilton Soares de Oliveira. Our current design approach and implementation feature a series improvements and new functionalities that significantly enhance DRE and make it a very powerful and effective tool for XML data conversion. The new augmented conflict classification used in the current DRE version has provided basis for detection and resolution of additional types of structural conflicts that allow for more precise data conversion. The most significant improvement was the addition of content model conflicts, namely constraint and type mismatch conflicts. In addition to Element-Element conflicts detected and resolved by the first DRE version, the current implementation handles Attribute-Element and

Attribute-Attribute conflicts as well. Another feature originated in the current DRE version is automatic thesaurus generation. User interfaces for thesaurus and conversion specification validation were added to simplify verification of structural and semantic transformations. The last notable development reflects the conceptual evolution of the IWiz system from the data warehouse architecture to a hybrid scheme that combines both data warehousing and mediation capabilities. This modification required an addition of query functionalities to DRE.

8.2 Future Work

The world of XML is evolving extremely rapidly. Very few aspects of managing XML documents are in the form of final recommendations from W3C, which is the leading organization coordinating the research and development efforts of the XML community. That is why we cannot expect to thoroughly cover and employ all aspects of XML technology at this point in time.

In the near future, the mechanism of data definition for XML will shift from using DTD to using XML Schemas. At the time of writing, the XML Schema working group at W3C is close to releasing candidate recommendation for XML Schemas. This means that new XML sources will use XML Schema as the data definition mechanism, and this will require modification of DRE. Future releases of DRE will have to be able to deal with sources defined using either a DTD or an XML Schema. XML Schemas provides richer data definition facilities comparing to DTDs, hence, new additional types of conflicts will arise between a source and target schema. New versions of the built-time

DRE component will have to reflect updated conflict classification by implementing algorithms for resolving conflicts in context of XML Schemas.

Another expected addition to XML technology is the development of new XML Query language. We covered the main features of this language as they stated in W3C Requirements in Section 2.4. This will entail changes to the run-time component.

Possibilities exist for the further refinement of the built-time procedures. In the current version of DRE, we work under the assumption that the ontology concepts are unique, and that each source concept can be mapped to exactly one ontology concept or not mapped at all. In practice, the ontology can contain several concepts related to one source concept to a certain degree. We plan to devise a scheme to evaluate and assign the degree of compatibility for each mapping. The relationships between concepts in thesaurus can include not only the strict equivalency, but also generalization/ specialization and other degrees of compatibility. In case when a source concept corresponds to several related concepts, the concept with the highest degree of compatibility must be chosen. Another extension that can improve the accuracy of mapping process is to include the assessment of descendants of each node in order to determine the most accurate mapping for the source node.

Another proposed extension affects the data processing phase during run-time. In the current prototype, we operate under assumption that the query processor always returns result of a manageable size, i.e. the result can always be converted to a DOM tree and placed in memory. In the future, we plan to evaluate the result size and switch between DOM and SAX interfaces as necessary.

Finally, we plan to change the DRE mode of operation in relation to other IWiz components. Currently, DRE is invoked by the mediator, i.e. it is assumed to function in the same address space as the mediator. In the IWiz architecture, DRE modules and the mediator can be distributed across several computers connected to a network. To accommodate this arrangement, a DRE component has to be turned into a standalone application that communicates with the mediator through RMI.

APPENDIX
DOCUMENT TYPE DEFINITIONS (DTDS) FOR XML DATA SOURCES

Ontology.dtd

```
<!--  IWIZ's ONTOLOGY in One single Integrated DTD -->

<!ELEMENT Ontology (Bibliography*)>
<!ELEMENT Bibliography  (Article | Book | Booklet | InBook
| InCollection | InProceedings | Manual | MastersThesis |
PhdThesis | Misc | Proceedings | TechReport | UnPublished
)*>
<!ELEMENT Article   (Author+,
                Title,
                Journal,
                Year,
                Month?,
                Volume?,
                Number?,
                Pages?,
                Month?,
                Note?    )>
<!ATTLIST Article id ID #REQUIRED>
<!ELEMENT Author (firstname?,lastname,Address?)>
<!ELEMENT firstname (#PCDATA) >
<!ELEMENT lastname (#PCDATA) >
<!ELEMENT Title (#PCDATA) >
<!ELEMENT Journal (Author+, Year, Month?)>
<!ELEMENT Year (#PCDATA) >
<!ELEMENT Month (#PCDATA) >
<!ELEMENT Volume (#PCDATA) >
<!ELEMENT Number (#PCDATA) >
<!ELEMENT Pages (#PCDATA) >
<!ELEMENT Note (#PCDATA) >
<!ELEMENT Book      ((Author+| Editor+),
                Title,
                Publisher,
                Year,
                Month?,
                (Volume | Number)?,
                Series?,
```

```
                Address?,
                Edition?,
                ISBN,
                Cost?,
                Note?    )>
<!ATTLIST Book id ID #REQUIRED>
<!ELEMENT Editor (firstname?,lastname,Address?) >
<!ELEMENT Publisher (firstname?,lastname,Address?) >
<!ELEMENT Edition (#PCDATA)>
<!ELEMENT Address (#PCDATA)>
<!ELEMENT Series (#PCDATA)>
<!ELEMENT ISBN (#PCDATA)>
<!ELEMENT Cost (#PCDATA)>
<!ELEMENT Booklet   (Author?,
                Title,
                HowPublished?,
                Year?,
                Month?,
                Address?,
                Note?    )>
<!ATTLIST Booklet id ID #REQUIRED>
<!ELEMENT HowPublished (#PCDATA) >
<!ELEMENT InBook  ( (Chapter | Pages ) )>
<!ATTLIST InBook  id     ID     #REQUIRED
                  bookID IDREF  #REQUIRED >
<!ELEMENT Chapter (#PCDATA) >
<!ELEMENT InCollection   (Author+,
                Title,
                (Chapter | Pages)?,
                Type?,
                Note?    )>
<!ELEMENT Type (#PCDATA) >
<!ATTLIST InCollection   id   ID   #REQUIRED>
<!ATTLIST InCollection   bookID    IDREF     #REQUIRED>
<!ELEMENT InProceedings (Author+,
                Title,
                Pages?,
                Note?    )>
<!ATTLIST InProceedings id         ID   #REQUIRED>
<!ATTLIST InProceedings proceedingsID   IDREF
     #REQUIRED>
<!ELEMENT Manual    (Title,
                Author*,
                Year?,
                Month?,
                Edition?,
                Address?,
```

```
                Organization?,
                Note?    )>
<!ATTLIST Manual    id ID #REQUIRED>
<!ELEMENT Organization (#PCDATA) >
<!ELEMENT MastersThesis  (Author,
                Title,
                School,
                Year,
                Month?,
                Type?,
                Address?,
                Note?    )>
<!ATTLIST MastersThesis  id ID #REQUIRED>
<!ELEMENT School (#PCDATA) >
<!ELEMENT Misc      (Author?,
                Title?,
                HowPublished?,
                Year?,
                Month?,
                Note?    )>
<!ATTLIST Misc      id ID #REQUIRED>
<!ELEMENT PhdThesis (Author,
                Title,
                School,
                Year,
                Month?,
                Type?,
                Address?,
                Note?    )>
<!ATTLIST PhdThesis id ID #REQUIRED>
<!ELEMENT Proceedings    (Title,
                Year,
                Month?,
                Editor*,
                (Volume | Number)?,
                Series?,
                Address?,
                Organization?,
                Publisher?,
                Note?    )>
<!ATTLIST Proceedings id ID #REQUIRED>
<!ELEMENT TechReport     (Author+,
                Title,
                Institution,
                Year,
                Month?,
                Type?,
```

```
                Number?,
                Address?,
                Note?    )>
<!ATTLIST TechReport     id ID #REQUIRED>
<!ELEMENT Institution (#PCDATA) >
<!ELEMENT UnPublished    (Author+,
                Title,
                Year?,
                Month?,
                Note     )>
<!ATTLIST UnPublished    id ID #REQUIRED>
```

Source1.dtd

```
<!ELEMENT references (reference)*>
<!ELEMENT reference
(author?,year?,title?,link?,notes?,editor?,(booktitle|proce
edings)?,journal?,volume?,pages?,publisher?,
where?,legend?)>
<!ATTLIST reference type (thesis) #IMPLIED>
<!ELEMENT a (#PCDATA)>
<!ATTLIST a type CDATA   #REQUIRED>
<!ELEMENT author (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT editor (#PCDATA)>
<!ELEMENT booktitle (#PCDATA)>
<!ELEMENT journal (#PCDATA)>
<!ELEMENT volume (#PCDATA)>
<!ELEMENT pages (#PCDATA)>
<!ELEMENT notes (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT where (#PCDATA)>
<!ELEMENT legend (#PCDATA)>
<!ELEMENT proceedings (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT link (#PCDATA)>
<!ATTLIST link  href CDATA #REQUIRED>
```

Source2.dtd

```
<?xml encoding="US-ASCII"?>
<!ELEMENT bib (book|article|person)*>
<!ELEMENT book (title, publisher)>
<!ATTLIST book year CDATA #REQUIRED>
<!ATTLIST book authors IDREFS #REQUIRED>
<!ELEMENT article (title, booktitle?,
(shortversion|longversion)?)>
```

```
<!ATTLIST article authors IDREFS #REQUIRED>
<!ATTLIST article year CDATA #REQUIRED>
<!ATTLIST article type CDATA #REQUIRED>
<!ELEMENT publisher (name, address?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT person (firstname?, lastname)>
<!ATTLIST person ID ID #REQUIRED>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT booktitle (#PCDATA)>
```

Source3.dtd

```
<!ELEMENT ARTICLE
((TITLE,AUTHOR+,URL+,DATE,RELATED,((SUMMARY,PUBLICATION,AUT
HOR*)|(((SUMMARY,URL+)|(URL+,SUMMARY))?,PUBLICATION)))|(TIT
LE,AUTHOR+,PUBLICATION,URL,DATE,RELATED?)|(TITLE,AUTHOR,DAT
E,((RELATED,SUMMARY,URL+,PUBLICATION)|(PUBLICATION,URL+))))
>
<!ELEMENT AUTHOR
(#PCDATA|LAST|FIRST|ORGANIZATION|URL|POSITION)*>
<!ELEMENT COLLECTION
((TITLE,AUTHOR,PUBLICATION,URL,(DATE,SUMMARY)?)|(TITLE,AUTH
OR,URL,DATE,RELATED,SUMMARY,PUBLICATION))>
<!ELEMENT DATE
((MONTH,DAY,YEAR,(QUARTER,URL?)?)|((DAY|MONTH)?,YEAR,QUARTE
R)|YEAR|QUARTER)>
<!ELEMENT DAY (#PCDATA)>
<!ELEMENT EDITOR (#PCDATA)>
<!ELEMENT EM (#PCDATA)>
<!ELEMENT FIRST (#PCDATA)>
<!ELEMENT I (#PCDATA)>
<!ELEMENT LAST (#PCDATA)>
<!ELEMENT MONTH (#PCDATA)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT NUMBER (#PCDATA)>
<!ELEMENT ORGANIZATION (#PCDATA)>
<!ELEMENT PAGE (#PCDATA)>
<!ELEMENT POSITION (#PCDATA)>
<!ELEMENT PUBLICATION
((NAME,VOLUME,NUMBER,((PAGE,URL?)|URL))|(NAME,URL,(VOLUME,(
(NUMBER,PAGE)|(EDITOR,NUMBER,PAGE,PUBLISHER)))?)|NAME)?>
<!ELEMENT PUBLISHER (#PCDATA)>
<!ELEMENT QUARTER (#PCDATA)>
<!ELEMENT RADIO (SUMMARY,TITLE,AUTHOR,PUBLICATION,URL)>
<!ELEMENT RELATED (#PCDATA)>
```

```
<!ELEMENT RESOURCES
(ARTICLE+,COLLECTION+,RADIO,ARTICLE+,COLLECTION,ARTICLE,COL
LECTION,ARTICLE+,COLLECTION+,ARTICLE+)>
<!ATTLIST RESOURCES
     xmlns:html CDATA #IMPLIED
     xmlns CDATA #IMPLIED
>
<!ELEMENT STRONG (#PCDATA)>
<!ELEMENT SUMMARY (#PCDATA|I|A|STRONG)*>
<!ELEMENT TITLE (#PCDATA|I|EM)*>
<!ATTLIST TITLE
     SUBTITLE CDATA #IMPLIED
>
<!ELEMENT URL (#PCDATA)>
<!ATTLIST URL
     TITLE CDATA #IMPLIED
>
<!ELEMENT VOLUME (#PCDATA)>
<!ELEMENT YEAR (#PCDATA)>
```

REFERENCES

1. S. Abiteboul, "Querying Semistructured Data," *Proceedings of the International Conference on Database Theory*, 1997.

2. S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener, "The Lorel Query Language for Semistructured Data," *International Journal of Digital Libraries*, vol.1, 1997.

3. R. Abu-Hamdeh, J. Cordy, P. Martin, "Schema Tanslation Using Structural Transformation," *Proceedings of CASCON'94*, Toronto, Canada, pp. 202-215, 1994.

4. ACM SIGMOD Record: XML Version, October 2000, http://www.acm.org/sigs/sigmod/record/xml/.

5. C. Batini, M. Lenzerini, S. Navathe, "A Comparative Analysis of Methodologies for Database Schema Integration," *ACM Computing Surveys*, vol.18, pp. 323-364, 1986.

6. C. Beeri, T. Milo, "Schemas for Integration and Translation of Structured and Semi-Structured Data," *Proceedings of the International Conference on Database Theory*, Jerusalem, Israel, 1999.

7. S. Bergamaschi, S. Castano, M. Vincini, "Semantic Integraion of Semistructured and Structured Data Sources," *SIGMOD Record*, vol.28, 1999.

8. P. Buneman, "Semistructured Data," *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Tuscon, Arizona, 1997.

9. P. Buneman, S. Davidson, G. Hillebrand, D. Suciu, " A query language and optimization techniques for unstructured data," *Proceedings of ACMSIGMOD International Conference on Management of Data*, pp. 505-516, Montreal, Canada, 1996.

10. A. Bosworth, A. Brown, "Microsoft's Vision for XML," *Data Engineering Bulletin, Special Issue on XML*, vol. 22, 1999.

11. C. Cooper, "Benchmarking XML Parsers: A performance comparison of six stream-oriented XML parsers," 1999, available at http://www.xml.com/pub/Benchmark/article.html.

12. S. Davidson, P. Buneman, A. Kosky, "Semantics of Database Transformations," Technical Report MS-CIS-95-25, University of Pennsylvania, 1995, available at ftp://ftp.cis.upenn.edu/pub/papers/db-research.transsurv.ps.Z.

13. E. Derksen, P. Fankhauser, E. Howland, G. Huck, I. Macherius, M. Murata, M. Resnick, H. Schöning, "XQL (XML Query Language)," Submission to the World Wide Web Consortium, 1999, available at http://www.ibiblio.org/xql/xql-proposal.html.

14. A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu, "XML-QL: A Query Language for XML," *Proceedings of 8th International World WideWeb Conference (WWW8)*, 1999.

15. M. Erdmann, R. Studer, "How to Structure and Access XML Documents With Ontologies", to appear in *Data and Knowledge Engineering, Special Issue on Intelligent Information Integration*, 2000.

16. C. Eick, "A methodology for the design and transformation of conceptual schemas", *Proceedings of the 17th International Conference on Very Large Databases*, Barcelona, Spain, pp. 25-34, 1991.

17. M. Fernandez, J. Simeon, P. Walder, S. Cluet, A. Deutsch, D. Florescu, A. Levy, D. Maier, J. McHugh, J. Robie, D. Suciu, J. Widom, "XML Query Languages: Experiences and Exemplars", available at http://www-db.research.bell-labs.com/user/simeon/xquery.html.

18. D. Florescu, A. Levy, A. Mendelzon, "Database techniques for the World-Wide Web: A survey," *SIGMOD Record*, vol. 27, pp. 59-74, 1998.

19. H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, J. Widom, "Integrating and Accessing Heterogeneous Information Sources in TSIMMIS," presented at *AAAI Symposium on Information Gathering*, Stanford, California, 1995.

20. H. Garcia-Molina, J. Ullman, J. Widom, "Database System Implementation", Prentice Hall, 2000.

21. R. Goldman, J. McHugh, J. Widom, "From Semistructured Data to XML: Migrating the Lore data Model and Query Language," *Proceedings of the 2nd International Workshop on the Web and Databases*, 1999.

22. J. Hammer, "The Information Integration Wizard (IWiz) Project," Project Description TR99-019, University of Florida, Gainesville, Florida, 1999.

23. J. Hammer, H. Garcia-Molina, W. Labio, J. Widom, Y. Zhuge. "The Stanford Data Warehousing Project", *Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, vol. 18, pp. 41-48, 1995.

24. J. Hammer, J. McHugh, H. Garcia-Molina, "Semistructured Data: The TSIMMIS Experience," *Proceedings of the First East-European Workshop on Advances in Databases and Information Systems-ADBIS '97*, St. Petersburg, Russia, 1997.

25. R. Hull, "Managing semantic heterogeneity in databases: A theoretical perspective," *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Tuscon, Arizona, 1997.

26. IBM's alphaWorks, "XML Parser for Java," October 2000, http://alphaworks.ibm.com/tech/xml4j.

27. T. Lahiri, S. Abiteboul, J. Widom, "Ozone: Integrating structured and semistructured data," *Proceedings of the Seventh International Workshop on Database Programming Languages*, Kinloch Rannoch, Scotland, 1999.

28. D. Martin, M. Birbeck, M. Kay, B. Loesgen, J. Pinnock, S. Livingstone, P. Stark, K. Williams, R. Anderson, S. Mohr, D. Baliles, B. Pear, N. Ozu, "Professional XML," Wrox Press, 2000.

29. J. McHugh, S. Abiteboul, R. Goldman, D. Quass, J. Widom, "Lore: A Database Management System for Semistructured Data", *SIGMOD Record*, vol. 26, pp. 54-66, 1997.

30. G. Mecca, P. Merialdo, P. Atzeni, "Araneus in the Era of XML," *Data Engineering Bulletin, Special Issue on XML*, vol. 22, 1999.

31. D. Megginson, "SAX 2.0: The Simple API for XML," October 2000, available at http://www.megginson.com/SAX/index.html.

32. T. Milo, S. Zohar, "Using Schema Matching to Simplify Heterogeneous Data Translation", *Proceedings of the 24th Conference on Very Large Databases*, New York, USA, 1998.

33. Oracle XML Developer's Kit for Java, October 2000, available at http://technet.oracle.com/tech/xml/parser_java2/index.htm.

34. Y. Papakonstantinou, H. Garcia-Molina, J. Widom, "Object exchange across heterogeneous information sources," *Proceedings of the 11th International Conference on Data Engineering*, Taipei, Taiwan, 1995.

35. C. Pluempitiwiriyawej, J. Hammer, “A Classification Scheme for Semantic and Schematic Heterogenities in XML Data Sources”, TR00-004, University of Florida, Gainesville, FL, 2000.

36. Programmers Vault, "File Format DataBank," October 2000, http://www.chesworth.com/pv/file_format/filex/index.htm.

37. J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, J. Naughton, “Relational Databases for Quering XML Documents: Limitations and Opportunities”, *Proceedings of the 25th Conference on Very Large Databases*, Edinburg, Scotland, 1999.

38. D. Suciu, "An Overview of Semistructured Data", SIGACT News, 29(4), 1998.

39. S. Voshmgir, "XML Tutorial: The Parser," 1999, http://indi.wu-wien.ac.at/~student/xml/parser.htm.

40. J. Widom, "Data Mangement for XML: Research Directions," *Data Engineering Bulletin, Special Issue on XML*, vol. 22, 1999.

41. Wintertree Software, "ThesDB Thesaurus Engine for Java", October 2000, http://www.wintertree-software.com/dev/thesdb/javasdk.html.

42. World Wide Web Consortium, "Document Object Model (DOM) Level 1 Specification," 1998, available at http://www.w3.org/TR/REC-DOM-Level-1/.

43. World Wide Web Consortium, "Document Object Model (DOM) Level 2 Specification," 2000, available at http://www.w3.org/TR/DOM-Level-2/.

44. World Wide Web Consortium, "Extensible Markup Language (XML) 1.0," W3C Recommendation, 1998, available at http://www.w3.org/TR/1998/REC-xml-19980210.

45. World Wide Web Consortium, "Namespaces in XML," 1998, available at http://www.w3.org/TR/REC-xml-names/.

46. World Wide Web Consortium, "Overview of SGML Resources," October 2000, available at http://www.w3.org/MarkUp/SGML.

47. World Wide Web Consortium, "The Query Languages Workshop," October 2000, available at http://www.w3.org/TandS/QL/QL98/.

48. World Wide Web Consortium, "XML Schema Part 1: Structures," Working Draft, 22 September 2000, available at http://www.w3.org/TR/xmlschema-1/.

49. World Wide Web Consortium, "XML Schema Part 2: Datatypes," Working Draft, 22 September 2000, available at http://www.w3.org/TR/xmlschema-2/.

50. World Wide Web Consortium, "XML Query Requirements," Working Draft 15 August 2000, available at http://www.w3.org/TR/xmlquery-req.

51. XML-QL : A Query Language for XML, Version 0.9, 2000, available at http://www.research.att.com/~mff/xmlql/.

BIOGRAPHICAL SKETCH

Anna Teterovskaya was born in Gorky, Russia. She received her first master's degree in biophysics from Nizhny Novgorod State University (Nizhny Novgorod, Russia) in 1993. In 1999, Anna was admitted into the CISE graduate program at the University of Florida. She graduated in December 2000 with the Master of Science degree. Her research interests include semistructured data, data integration, and database technologies.