Merging of XML Documents

By

Wanxia Wei

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

August 5, 2004

© Copyright
2004, Wanxia Wei
The author has granted a non-exclusive license allowing the Library and Archives Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.
Abstract

XML data extracted from the Web is usually irregular. In addition, information about real world objects may spread over heterogeneous XML documents and may be incomplete. Moreover, it is critical to identify XML data instances representing the same real world object when merging XML documents, but each XML document may have different elements and/or attributes to identify objects. Furthermore, conflicts may emerge when merging these XML documents. Therefore, how to deal with the heterogeneous structures of XML documents, identify XML data instances, solve conflicts, and effectively merge XML documents to obtain complete information is a challenge. In this thesis, we define a merging operation over XML documents that can merge two XML documents with different structures. It is similar to a full outer join in relational algebra. We describe an algorithm for this operation. In addition, we present a mechanism that combines Skolem function and Boolean functions defined for designers to identify XML instances. Moreover, we propose a method to merge XML elements and handle typical conflicts. Finally, we introduce our implementation and show a merge template XML file that can specify how to combine two input XML documents into one result XML document, build semantic connections between these two documents, and support recursive processing and merging of XML elements.
Acknowledgements

First, I would like to gratefully thank Dr. Mengchi Liu, my supervisor, for his guidance, many suggestions, encouragement, and patience. Without his direction and insights, this thesis would not have become a reality.

Also, I am grateful to Dr. Tomoyuki Yamakami for guiding and helping me during my graduate studies.

I wish to thank the members of my committee, Dr. Iluju Kiringa and Dr. Michael Weiss for reading my thesis and for their valuable suggestions.

I would like to express special thanks to many people who have assisted me in many ways. They all have my sincere gratitude. I am indebted to Baoxian Xiao for his help and encouragement. Donald Christopher Belway deserves particular credit for proofreading my thesis. I thank Dr. Shijun Li and Chunsheng Liu for proofreading parts of this thesis and for their suggestions and help. I am grateful to Zhihong Li for all of her prompt assistance. I would also like to thank Qiong Ruan, Feng Zheng, HaiBo Wang, Lan Yang, Zhuowen Wang, Wenjie Tong, Xiafei Jia, and Tao Fu for their help. I am also grateful to Wanzhen Yu, Jinping Zhang, Qinghong Song, Xia Sun, Diping Wen, Xuemei Chen, Hongqi Huang, and Honglin Liao for their encouragement and help.

I would like to thank my daughter Xuliang Li for making me laugh and happy. My final acknowledgment must go to my husband Jintang Li for his support, understanding, and encouragement.
# Contents

Abstract .......................................................................................................................... iii

Acknowledgements .......................................................................................................... iv

List of Figures .................................................................................................................. ix

1 Introduction .................................................................................................................. 1
  1.1 Motivation ................................................................................................................. 1
  1.2 Objectives ................................................................................................................. 2
  1.3 Outline of the Thesis ............................................................................................... 3

2 Background .................................................................................................................. 5
  2.1 XML—the eXtensible Markup Language ................................................................. 5
  2.2 Parts of an XML Document .................................................................................... 7
  2.3 DTD and XML Schema ......................................................................................... 8
  2.4 Related Supporting Standards for XML ................................................................. 13
  2.5 Overview of XSLT ................................................................................................. 14

3 Related Work ................................................................................................................. 17
  3.1 Semantic Integration of XML Data ......................................................................... 17
  3.2 Systems for Processing XML or XML Streams .................................................... 18
3.3 Merging or Integration of XML Data with Similar or Identical Structures . . 19
3.4 Deep Union Operator .................................................. 19
3.5 Merge Templates .......................................................... 20
3.6 Instance Identification .................................................. 20

4 Our Approach ................................................................. 22

4.1 The Merging Operation .................................................. 23
   4.1.1 Definition of the Merging Operation ...................... 23
   4.1.2 An Example of Merging of XML Documents .......... 25
4.2 The Algorithm for the Merging Operation ...................... 31
4.3 Mechanism for Identifying XML Instances ..................... 34
4.4 The Generated XML Document $F_{LOJ}$ ......................... 35
4.5 The Semantic Associations between Two XML Documents to be Merged . 39
   4.5.1 The Assumptions about Two XML Documents .......... 39
   4.5.2 Notions ............................................................... 40
4.6 Merging XML Elements and Handling Typical Conflicts .......... 42
   4.6.1 Typical Conflicts .................................................. 42
   4.6.2 Merging XML Elements and Handling Typical Conflicts .... 44
4.7 The Resulting XML Document of the Merging Operation .......... 54
4.8 Features of Our Approach ............................................. 59

5 Implementation Issues .................................................... 61

5.1 The Requirements of the Merging System ......................... 61
5.2 A Merge Template XML File .......................................... 62
5.3 Using XSLT and Java to Implement the Prototype ............. 65
5.4 Additional Example ..................................................... 68
   5.4.1 Instance Identification .......................................... 68
   5.4.2 Element Merging and Conflict Resolution .............. 74
6 Conclusion and Future Work

6.1 Conclusion ......................................................... 81
6.2 Future Work ......................................................... 82

Bibliography .............................................................. 84
List of Figures

2.1 Indicators and their meanings. ........................................... 9
2.2 Default values. .......................................................... 10
2.3 An example XSLT stylesheet. ........................................... 16

4.1 XML document $F_1$ in Example 5. .................................. 26
4.2 XML document $F_2$ in Example 5. .................................. 27
4.3 DTD file $D_1$ in Example 5. ....................................... 27
4.4 DTD file $D_2$ in Example 5. ....................................... 28
4.5 The result single XML document $F_3$ in Example 5. .......... 29
4.6 $F_{LOJ}$ in Example 5. ............................................... 30
4.7 Element $x$ and element $y$ in $F_1$ ................................... 37
4.8 The relationship between $c_1$ and $c_2$ ............................. 47
4.9 Procedure and typical conflicts they handle ...................... 54

5.1 An example merge template XML file. .............................. 62
5.2 XSLT stylesheet for procedure $LeftOuterJoin$ (1) ............. 66
5.3 XSLT stylesheet for procedure $LeftOuterJoin$ (2) ............. 67
5.4 XML document $F_1$ in Example 6. ................................. 69
5.5 XML document $F_2$ in Example 6. ................................. 70
5.6 XML Schema file $S_1$ for $F_1$ in Example 6 .................... 71
5.7 XML Schema file $S_2$ for $F_2$ in Example 6 .................... 72
5.8  The merge template XML file in Example 6.     73
5.9  \( F_{LOJ} \) in Example 6.                        75
5.10 The result single XML document \( F_3 \) in Example 6.  76
Chapter 1

Introduction

1.1 Motivation

XML (eXtensible Markup Language) is fast emerging as the dominant standard for data representation and exchange over the Internet. It provides a flexible syntax for representing semi-structured data.

With various tools available, data on the Web can be transformed into XML. However, information about real world objects may spread over heterogeneous XML documents and may be incomplete. For example, several XML documents on the Web may describe the same object, such as a person or an organization. To get complete information, we need to merge these heterogeneous XML documents.

A lot of research in semantic integration of XML data has been conducted [3, 8, 31, 32]. Moreover, many systems for processing XML or XML streams are developed [11, 17, 18, 26, 28]. However, a merging operation is not available in any of these works or systems. A lot of research in merging or integration of XML data has been done [22, 23, 24, 25, 36, 37], but all these works deal with merging or integration of XML data that has similar or
identical structures. Therefore, there is a need to merge heterogeneous XML documents, or XML documents that have different structures.

When merging heterogeneous XML documents, several research issues arise. Merging depends directly on the structures of the input XML documents. However, XML data extracted from the Web may be irregular. In addition, it is critical to identify XML data instances representing the same real world object when merging XML documents, but each XML document has specific elements and/or attributes to identify objects. For example, an XML document about students’ grades has a student number to identify each student. For an XML document consisting of names and phone numbers of professors, the name of a professor cannot identify a professor if two professors with the same name exist. Furthermore, conflicts may emerge when merging these XML documents. Therefore, how to deal with the heterogeneous structures of XML documents, systematically identify XML data instances, properly detect and solve conflicts, and effectively merge the appropriate XML elements from different XML sources to obtain complete information is a challenge.

In this thesis, we present a new approach to merging XML documents to address the above issues.

Bertino et al. point out that data integration involves reconciliation at data model level, data schema level, and data instance level [6]. We mainly focus on reconciliation at data instance level to merge XML documents that have different structures.

1.2 Objectives

Our main goal is to propose an approach to semantic merging of XML documents that have different structures. More specifically, the objective of the work presented is to define a merging operation that can merge two XML documents that have different structures,
design an algorithm and a set of procedures for this operation, present a mechanism to identify XML instances when merging, explore the semantic associations between two XML documents to be merged, propose a method for merging XML elements and handling conflicts, and implement a prototype to merge XML documents.

1.3 Outline of the Thesis

The rest of the thesis is organized as follows:

- Chapter 2 introduces background knowledge that is important to understanding this thesis.

- Chapter 3 gives an overview of related work. We first outline research works on semantic integration of XML data, and concisely describe several systems for processing XML or XML streams. Then, we give a summary of research works on merging or integration of XML data with similar or identical structures. Subsequently, we briefly introduce a Deep Union operator over semi-structured data. After that, we introduce a research work that uses Merge Templates for merging of XML documents, and compare it to the work of this thesis. At the end of this chapter, we focus on instance identification, which is critical to data integration.

- Chapter 4 presents our approach to merging XML documents. In this chapter, a merging operation that is similar to a full outer join in relational algebra is defined, and an algorithm and a set of procedures for this operation are presented. In addition, we study the mechanism for identifying XML instances and the semantic associations between two XML documents to be merged. We also demonstrate the method for merging XML elements and handling typical conflicts, and examine the XML
documents that this algorithm produces. We illustrate our approach with a running example that is used throughout this chapter.

- Chapter 5 discusses implementation issues. We have implemented a prototype to merge XML documents based on the approach presented in Chapter 4. In this chapter, we first introduce the requirements of our XML merging system. Then, we present a merge template XML file that can specify how to combine two input XML documents into one result XML document, build semantic connections between these two documents, and support recursive processing and merging of XML elements. After that, we discuss how to use XSLT and Java to implement the prototype. Subsequently, we give another example of merging of XML documents to demonstrate the functions of the prototype.

- Chapter 6 concludes this thesis and points out future work.
Chapter 2

Background

2.1 XML— the eXtensible Markup Language

XML (eXtensible Markup Language) is a markup language designed to store, carry, and exchange data [5, 38]. It allows developers to define their own customized tags for almost every type of document.

In order to present the benefits of XML, we need to mention its two predecessors: SGML (Standard Generalized Markup Language) [34] and HTML (Hyper Text Markup Language) [9, 45].

SGML is the parent language of all markup languages. It is an international standard for describing documents. It separates content from presentation and format. HTML is a markup language written in SGML. It is used to capture and publish content on Web pages. Both SGML and HTML are very successful markup languages, but they have serious short-comings. Although SGML is very powerful, it is very complex, especially for the everyday uses of the Web. HTML consists of a predefined set of tags, and it is used primarily for defining how content is to be displayed. In 1996 the World Wide Web Consortium (W3C)
began to define a new markup language with the power and extensibility of SGML and with the simplicity of HTML. Finally, in 1998, the W3C approved Version 1.0 of the XML specification, and XML, a new markup language, was born.

XML is a subset of SGML and the goal of XML is “to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML” [38].

XML has many advantages over SGML and HTML. Compared to SGML, XML is much simpler, but it is still a meta-language and offers most of the power of SGML. Compared with HTML, XML has many advantages. First, XML allows document authors to create their own tag sets, elements, attributes, and document structures to suit their individual needs. The meaning of these tags is defined in a separate document, either a DTD (Document Type Definition) [38] (DTDs are part of the W3C’s XML 1.0 recommendation) or an XML Schema [42, 43, 44]. We introduce DTD and XML Schema in Section 2.3. Second, XML separates content from presentation. Third, XML documents can be validated against DTDs or XML schemas. Finally, XML makes information more accessible and re-usable because XML’s flexible markup can be used by any XML software.

Data has different forms: structured data and unstructured data. Structured data fits well a predefined schema and it is usually provided by databases. Unstructured data cannot be decomposed into standard components and it is usually stored as files, graphics, and a variety of other formats. Semi-structured data is in an intermediate format between structured data and unstructured data. “Semi-structured data is data that is neither raw data, nor very strictly typed as in conventional database systems” [1]. Unlike structured data, semi-structured data need not conform to a fixed schema. “It has been shown that XML is well suited for representing semi-structured data” [10].
2.2 Parts of an XML Document

An element is a unit of XML data. It consists of a start tag, an end tag, and the information between the tags, which is referred to as the content. Every XML document consists of a single element, which is called the root element.

An element can have simple content, element content, mixed content, or empty content. An element has simple content if it only contains text. An element has element content if it only contains other elements. An element has mixed content if it contains both text and other elements. An element has empty content if it contains no content.

An element can also have attributes. An attribute is usually used to describe a characteristic or property of an element. An attribute is represented as name-value pair that occurs inside the start-tag after the element name.

An element that is contained inside another element is a child of the element that contains it. The element that contains the child is referred to as the parent. Two elements that are contained inside the same parent are considered siblings.

An XML document has a hierarchical structure, and represents a tree structure. In this tree structure, an occurrence of an element is a node. For instance, the root element of an XML document is called the root node in the tree structure of this XML document. Any node that is hierarchically above a node in the tree structure of an XML document is the ancestor of that node. Any node which is hierarchically below a node in the tree structure of an XML document is the descendant of that node.

Example 1. The following is an XML document named people.xml. In people.xml, the root element is People element. This root element has four Person child elements. Each Person element has an attribute PersonName, and three child elements: Telephone, Address, and Age.
<People>

<Person PersonName="M. Smith">
  <Telephone>1-613-543-2600 Ext:1035</Telephone>
  <Address>100 Main Street</Address>
  <Age>35</Age>
</Person>

<Person PersonName="Mary Lee">
  <Telephone>(613)543-2600 Ext:9999</Telephone>
  <Address>200 Broadway Street</Address>
  <Age>30</Age>
</Person>

<Person PersonName="Mary Lee">
  <Telephone>(613)543-2600 Ext:8888</Telephone>
  <Address>200 Broadway Street</Address>
  <Age>30</Age>
</Person>

<Person PersonName="John Bell">
  <Telephone>(234)543-2700 Ext:1234</Telephone>
  <Address>150 King Street</Address>
  <Age>40</Age>
</Person>
</People>

2.3 DTD and XML Schema

There are two categories of XML documents: well-formed and valid. A document is well-formed if it obeys the syntax of XML. An XML document is said to be valid, if it has a DTD (Document Type Definition) or an XML Schema associated with it and if it complies
with the DTD or XML Schema.

The Document Type Definition (DTD) defines the valid syntax of a class of XML documents. That is, it lists a number of element names, which elements can appear in combination with other ones, what attributes are available for each element type, etc. [29].

In DTD, an element is declared using an element declaration. An element with simple content is declared with #CDATA or #PCDATA. CDATA means character data and PCDATA means parsed character data. If an element has child elements, the child elements must be enumerated in the same order as they appear in the DTD.

Moreover, we can specify how many times an element can appear in an XML document. The cardinality of an element defines how many times the element may occur in the XML document. Figure 2.1 shows the indicators for cardinality and their meanings.

<table>
<thead>
<tr>
<th>Indicators</th>
<th>Meanings</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>occur exactly 1 time</td>
</tr>
<tr>
<td>*</td>
<td>0 or more times</td>
</tr>
<tr>
<td>+</td>
<td>1 or more times</td>
</tr>
<tr>
<td>?</td>
<td>exactly 0 or 1 time</td>
</tr>
</tbody>
</table>

Figure 2.1: Indicators and their meanings.

In DTD, an attribute is declared using an ATTLIST declaration. An attribute declaration specifies the type and default value of an attribute. An attribute can be character data (CDATA), unique ID data (ID), ID of another element (IDREF), a variable that represents some common text (ENTITY), and so on. Figure 2.2 presents the default values of attributes.

**Example 2.** The following is the DTD for people.xml presented in Example 1. It declares that the root element of people.xml is *People* element. This root element contains zero or more *Person* elements. Each *Person* element has a *Telephone* child element, an *Address*
<table>
<thead>
<tr>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#DEFAULT value</td>
<td>If there is no value for the attribute in the XML document, the value specified in the DTD will be used.</td>
</tr>
<tr>
<td>#FIXED value</td>
<td>The attribute value is fixed.</td>
</tr>
<tr>
<td>#IMPLIED</td>
<td>The value does not have to be supplied in the XML document.</td>
</tr>
<tr>
<td>#REQUIRED</td>
<td>The value has to be supplied in the XML document.</td>
</tr>
</tbody>
</table>

Figure 2.2: Default values.

child element, and an *Age* child element. The type of *Telephone* element, *Address* element, and *Age* element is #PCDATA. Moreover, each *Person* element has an attribute named *PersonName*, which contains character data (CDATA), and this attribute is required.

```xml
<!DOCTYPE People [ 
  <!ELEMENT People (Person*)>  
  <!ELEMENT Person (Telephone, Address, Age)>  
  <!ATTLIST Person PersonName CDATA #REQUIRED>  
  <!ELEMENT Telephone (#PCDATA)>  
  <!ELEMENT Address (#PCDATA)>  
  <!ELEMENT Age (#PCDATA)>  
]> 
```

XML Schema provides a means for defining the structure, content and semantics of XML documents. XML Schema is an alternative to DTD. Moreover, XML Schema is intended as a replacement for DTD.

XML Schema defines two kinds of elements: simple elements and complex elements.

A simple element contains only text and no other elements and attributes. A simple element can be described directly by specifying its name and type.
A complex element contains other elements (child element) and/or attributes. Each child element of a complex element must be declared individually. Moreover, one of the order indicators, <all>, <choice>, and <sequence>, must be used to specify how these child elements should occur in the complex element. The <all> indicator designates that each child element must occur once and the child elements can appear in any order. The <choice> indicator designates that any child element from the only two child elements can occur. The <sequence> indicator specifies an ordered sequence of child elements. An attribute declaration specifies the name and data type of an attribute of a complex element. Attribute declarations should follow the declarations for child elements of a complex element.

In XML Schemas, we can define the cardinality of an element (the number of the occurrences of an element) by occurrence indicators, such as maxOccurs (the maximum number of occurrences), and minOccurs (the minimum number of occurrence of an element).

Furthermore, XML Schemas support data types and namespaces. In DTDs, we can only specify data types for textual data (PCDATA and CDATA). On the other hand, XML schemas support more complex textual and numeric data types. An XML namespace is a collection of names that are specified as legal elements or attributes within an XML document. A namespace URI (Universal Resource Identifier) is a URI that identifies an XML namespace and it is called the namespace name.

XML Schemas are intended as a replacement for DTDs because they provide the ability to specify data types for attribute values and element content, support namespaces, and are XML documents themselves.

Example 3. The following is the XML Schema for people.xml presented in Example 1. The root element of this XML Schema is schema element. The first line:

<xmlns:xs="http://www.w3.org/2001/XMLSchema"> specifies that the elements and data types used in the schema come from namespace “http://www.w3.org/2001/XMLSchema".
This schema defines two complex elements: *People* element and *Person* element, and three simple elements: *Telephone* element, *Address* element, and *Age* element. Here maxOccurs is set to unbounded which means that *People* element can contain unlimited number of *Person* elements. Each *Person* element has a *Telephone* child element, an *Address* child element, and an *Age* child element. The type of *Telephone* element and *Address* element is xs:string and the type of *Age* element is xs:integer. Both xs:string and xs:integer are built-in data types. In addition, an attribute element in this XML Schema defines a *PersonName* attribute for *Person* element. Use attribute of this attribute element explicitly specifies that *PersonName* is a required attribute. Note that every element in this schema has the prefix xs:, which is associated with the W3C XML Schema namespace. It is obvious that this XML Schema is more verbose than the corresponding DTD presented in Example 2.

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="People">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Person" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Telephone" type="xs:string"/>
              <xs:element name="Address" type="xs:string"/>
              <xs:element name="Age" type="xs:integer"/>
            </xs:sequence>
            <xs:attribute name="PersonName" type="xs:string" use="required"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
2.4 Related Supporting Standards for XML

XML can use the standards applied to HTML, such as Cascading Style Sheets (CSS) and HyperText Transfer Protocol (HTTP). In addition, the W3C working groups have developed additional supporting standards for XML, including Extensible Style Language (XSL) [39], XQuery [47], XML-Linking (XLink) [46], XPointer [48], etc.

Because a document author can choose any tag names for an XML document, a browser does not necessarily know how to display the elements in an XML document. XSL (Extensible Style Language) is the language that tells the browser how to display the various elements in an XML document. XSL consists of three parts: XSLT (Extensible Style Language Transformations) [40], XSL FO (XSL Formatting Objects), and XPath [41]. XSLT is a language for transforming XML documents into other XML documents. We give an overview of XSLT in Section 2.5. XSL FO is an XML vocabulary for specifying formatting. XPath is a language for addressing parts of an XML document. In XPath, a location path selects a set of nodes. There are two kinds of location paths: relative location paths and absolute location paths. A relative location path specifies a path relative to the context node (the context node is where the addressing starts). An absolute location path starts with /. This / by itself selects the root node of the document.

XQuery is a language for retrieving data items from XML-formatted documents. XQuery is expected to do for XML documents and XML databases what SQL (Structured Query Language) does for relational databases. XLink and XPointer are languages used to link XML documents to each other.
2.5 Overview of XSLT

XSLT, eXtensible Stylesheet Language Transformations, is a rule-based language for transforming an XML document into another XML document. It has emerged as the principal means for processing XML. The XSLT specification is maintained by the W3C and is one of the most exciting technologies to come out of the XML family.

XSLT is a template-driven environment for defining transformations between XML documents. An XSLT stylesheet contains template rules that are matched against nodes in the input document. When a match is detected, the rule’s template is instantiated and the result is added to the result tree.

XSLT is heavily affected by the design of functional programming languages [35]. In fact, XSLT is a functional programming language for XML documents. In XSLT stylesheets, we can define variables, and realize branching and loops. Also, we can call XSLT functions. In addition, we can define named templates and call named templates with parameters. Moreover, templates can be called recursively.

Furthermore, XSLT specification defines a general mechanism for calling extension functions written in any languages [19].

Example 4. The XSLT stylesheet shown in Figure 2.3 capitalizes all the element names and attribute names in any XML document. That is, this XSLT stylesheet can work with any source XML document. For XML document people.xml presented in Example 1, it generates an XML document Cpeople.xml as follows.

```xml
<PEOPLE>
    <PERSON PERSONNAME="M. Smith">
        <TELEPHONE>1-613-543-2600 Ext:1035</TELEPHONE>
        <ADDRESS>100 Main Street</ADDRESS>
    </PERSON>
</PEOPLE>
```
<AGE>35</AGE>

</PERSON>

<PERSON PERSONNAME="Mary Lee">

<Telephone>(613) 543-2600 Ext:9999</Telephone>

<Address>200 Broadway Street</Address>

<Age>30</Age>

</PERSON>

<PERSON PERSONNAME="Mary Lee">

<Telephone>(613) 543-2600 Ext:8888</Telephone>

<Address>200 Broadway Street</Address>

<Age>30</Age>

</PERSON>

<PERSON PERSONNAME="John Bell">

<Telephone>(234) 543-2700 Ext:1234</Telephone>

<Address>150 King Street</Address>

<Age>40</Age>

</PERSON>

</PEOPLE>
Figure 2.3: An example XSLT stylesheet.
Chapter 3

Related Work

This chapter discusses related work. Section 3.1 outlines research works on semantic integration of XML data. Section 3.2 concisely describes several systems for processing XML or XML streams. Section 3.3 reviews research works on merging or integration of XML data with similar or identical structures. Section 3.4 describes briefly a Deep Union operator over semi-structured data. Section 3.5 introduces a research work that uses Merge Templates for merging of XML documents, and includes a comparison between this work and our work presented. Section 5.4.1 summarizes various methods for dealing with instance identification.

3.1 Semantic Integration of XML Data

A lot of research in semantic integration of XML data has been conducted [3, 8, 31, 32]. Ahmed describes the formalisms for representing ontologies in the LDAP-based integration system [3]. This integration system can coherently integrate source data, schemata discrepancies, and semantic information under a common framework. Castano et al. propose a semantic approach to integration of heterogeneous XML data by building a domain ontology [8]. In this domain ontology, DTDs of heterogeneous XML data sources and
their contents are represented by a mediation scheme that is an XML-based description of the relevant concepts, relationships, and properties featuring heterogeneous DTDs. In the Castano work, integration of heterogeneous XML data is done without altering the original XML data. Rodriguez-Gianolli et al. present a framework that can provide a tool to integrate DTDs into a common conceptual schema [31, 32]. This tool can be used semi-automatically to produce a conceptual schema from several XML DTDs. It allows the user to enhance and refine the schematic knowledge derived from DTDs with domain expertise, and parses XML documents to populate the conceptual schema.

In this thesis, we mainly focus on reconciliation at data instance level and define the merging operation to both syntactically and semantically combine two XML documents into one single XML document. This operation is not available in any of the above works.

3.2 Systems for Processing XML or XML Streams

Several systems for processing XML or XML streams are developed [11, 17, 18, 26, 28].

YAT uses XML in data integration and allows XML documents to be materialized from relational databases [11]. Lore is a semi-structured data repository that builds a database system to query and index XML data [26].

The Niagara Internet Query System focuses on providing query capabilities for XML documents and can handle infinite streams [28]. It finds relevant XML documents to a given query by using a collaboration between the Niagara XML-QL query processor and the Niagara “text-in-context” XML search engine. The Tukwila data integration system accepts queries in a subset of the XQuery language, gets XML data from URI (Uniform Resource Identifier) sources specified in the query, and executes the XQuery over them [17, 18]. The Tukwila XML engine can return query results even as it is still reading data from the source(s).

18
The above systems are mainly used to query XML data while the merging operation defined in this thesis aggregates XML data.

3.3 Merging or Integration of XML Data with Similar or Identical Structures

A lot of research in merging or integration of XML data that has similar or identical structures has been done [22, 23, 24, 25]. A data model for semi-structured data is introduced and an integration operator is defined in [23, 24]. This integration operator can integrate similarly structured XML data. It extends union and join, and deals with null/unknown and inconsistent values. Lindholm designs a 3-way merging algorithm for XML files that comply with an identical DTD [22]. The author designs the algorithm by dividing the problem into tree matching and merging of matched trees. Manger presents an algorithm for merging SGML or XML files that are valid instances of the same but arbitrary DTD [25]. The mechanism proposed in this thesis can merge two XML documents that have different structures.

3.4 Deep Union Operator

Buneman et al. introduce a Deep Union operator that is used to combine edge-labelled trees based on a semi-structured data model with complex labels [7]. This very limited model requires that each tree node must have a key. The merging operation defined in this thesis does not impose this restriction on the XML data to be merged.
3.5 Merge Templates

Merge Templates that specify how to recursively combine two XML documents are introduced by Tufte et al. [36, 37]. Tufte et al. define a binary operation, called Merge, over XML documents. This Merge operation can take two similarly structured XML documents and piece them together to generate one result XML document. The merging process is controlled by a Merge Template that specifies how to recursively combine two XML documents into a single XML document.

Our work is different from their work with respect to several aspects. First, the Merge operation proposed by Tufte et al. combines two similarly structured XML documents to create aggregates over streams of XML fragments while the merging operation defined in this thesis can merge two XML documents that have different structures. Second, the mechanism proposed in this thesis combines Skolem function and Boolean functions defined for designers to identify XML instances when merging. Third, the method for merging XML elements and handling typical conflicts is proposed in this thesis. Finally, the merge template XML file proposed in this thesis is easy to create, and it can support recursive processing and merging of XML elements.

3.6 Instance Identification

When merging XML documents, it is critical to identify XML data instances representing the same object of the real world. Albert [4] uses the terms instance identification, object identification, or entity identification to refer to this problem.

In the past several years, the problem of instance identification has been investigated [2, 4, 12, 13, 14, 16, 20, 21, 27, 30, 49, 51]. These papers propose various methods to deal with this problem. A universal key is used in [4, 21]. Ahmed et al. and Reddy et al. let
a user specify key equivalence [2, 30]. Common attributes are used in [49, 51]. Queries for a user are defined by Gogolla et al. [14]. Lim et al. define the union of keys of the data sources [20]. Domain specific heuristics are proposed in [13, 16]. Smith-Waterman distance is presented by Monge [27]. Statistical information retrieval is introduced by Cohen [12]. However, these works deal with databases and support typed data. Therefore, it is hard to apply them directly on semi-structured data.

Skolem function is introduced in [15]. It returns a value for an object as the identifier of this object. Saccol et al. present a proposal for instance identification [33]. This proposal is based on Skolem function and is implemented by XSLT (eXtensible Stylesheet Language Transformations). The mechanism presented in this thesis combines Skolem function and Boolean functions defined for designers [33] to identify XML instances.
Chapter 4

Our Approach

In this chapter, we present our approach to merging XML documents. Section 4.1 defines a merging operation that is similar to a full outer join in relational algebra. It can merge two XML documents that have different structures. Section 4.2 presents the algorithm for this operation. Section 4.3 studies the mechanism for identifying XML instances. Section 4.4 examines one of the XML documents that this algorithm produces. Section 4.5 explores the semantic associations between the two XML documents to be merged and introduces semantically identical attributes and elements, semantically corresponding elements, and semantically corresponding attribute and element. Section 4.6 demonstrates the method for merging XML elements and handling typical conflicts. Section 4.7 explores the result XML document for the merging operation. Section 4.8 sums up the features of our approach. In addition, we present a set of procedures of the algorithm for the merging operation in this chapter.
4.1 The Merging Operation

4.1.1 Definition of the Merging Operation

The merging operation to be defined can merge two XML documents that have different structures, and create one single XML document. We assume that two XML documents to be merged share many tag names and also have some tags with different tag names. We also assume that two tags that share the same tag name in these two documents describe the same kind of objects in the real world but their corresponding elements may have different structures. More details of these assumptions about two XML documents to be merged are given in Section 4.5.1.

This merging operation can be formally represented as:

\[ F_3 = \text{merging} \,(F_1, F_2) \text{ on } (D_1, D_2, P_1, P_2, E_1, E_2) \]

where \( F_1 \) and \( F_2 \) are two input XML documents to be merged and \( F_3 \) is the merged XML document; \( D_1 \) and \( D_2 \) are the DTDs (Document Type Definition) of \( F_1 \) and \( F_2 \); \( P_1 \) and \( P_2 \) are absolute location paths in XPath (paths for short) that designate the elements to be merged in \( F_1 \) and \( F_2 \) respectively; \( E_1 \) and \( E_2 \) are Boolean expressions that are used to control the merging of the XML elements in \( F_1 \) and \( F_2 \).

Each path \( P_1 \) or \( P_2 \) consists of several element names in the corresponding input XML document separated by \(/\). The order of these element names is significant. The first one is the name of the root element of the corresponding XML document and the last one indicates the name of the elements to be merged. Moreover, each pair of consecutive element names in a path is associated with a pair of a parent and a child in the corresponding XML document.

Boolean expression \( E_1 \) is used to identify XML instances when merging \( F_1 \) and \( F_2 \). Also, it is used for merging of XML elements and handling conflicts. It consists of k
relational expressions connected by Boolean operator $\wedge$ with the the following form ($k \geq 1$):

$$(d_{1_1} = d_{2_1}) \land (d_{1_2} = d_{2_2}) \land \cdots \land (d_{1_k} = d_{2_k})$$

Let $e_1$ be one of the elements in $F_1$ whose path is $P_1$ and $e_2$ one of the elements in $F_2$ whose path is $P_2$. In $E_1$, $d_{1_i}$ and $d_{2_i}$ ($1 \leq i \leq k$) can be an attribute, a descendant, an attribute of an ancestor of $e_1$ and $e_2$, or any XML data related to $e_1$ and $e_2$, respectively. $E_1$, the logical conjunction of $k$ relational expressions, determines if $e_1$ and $e_2$ describe the same object in the real world when merging $F_1$ and $F_2$. As long as $E_1$ is true, $e_1$ in $F_1$ and $e_2$ in $F_2$ describe the same object and they are merged. We say that $e_1$ in $F_1$ and $e_2$ in $F_2$ are matching elements when they describe the same object.

Boolean expression $E_2$ is used to determine if $e_2$ in $F_2$ that does not have any matching $e_1$ in $F_1$ should be incorporated into the result XML document $F_3$. It consists of $l$ relational expressions connected by Boolean operator $\wedge$ ($k \geq l \geq 0$). Note that every relational expression in $E_2$ is one of the $k$ relational expressions in $E_1$. When $l = 0$, every $e_2$ that does not have a matching $e_1$ in $F_1$ is incorporated into the result XML document $F_3$.

In relational algebra, a natural join can join together two relations based on their common attributes. A natural join of relations $R_1$ and $R_2$ discards the tuples in $R_1$ or $R_2$ that do not have matching tuples in the other relation in the result relation, but in some cases this lost data holds useful information and preserving non-matching rows is important. On the other hand, an outer join retains the information that would have been lost from the result of natural join, replacing missing data with nulls. There are three kinds of outer join: left outer join, right outer join, and full outer join. A full outer join extracts the matching rows of the two tables and preserves non-matching rows from both tables. Analogously, the merging operation defined above merges XML documents $F_1$ and $F_2$ that have different structures and creates one single XML document $F_3$. It merges $e_1$ in $F_1$ and its matching $e_2$ in $F_2$ according to $D_1$, $D_2$, and $E_1$. It incorporates every modified non-matching $e_1$
in $F_1$ and some modified non-matching $e_2$ elements in $F_2$ based on $D_1$, $D_2$, $E_1$, and $E_2$. Moreover, it incorporates the elements in $F_1$ that do not need merging.

### 4.1.2 An Example of Merging of XML Documents

In this section, we give an example of merging of XML documents, which is used as a running example throughout this thesis.

**Example 5.** The two input XML documents $F_1$ and $F_2$ in Figures 4.1 and 4.2 have different structures. They describe professors by different elements: *Professor* elements in $F_1$ and *Person* elements in $F_2$. $F_1$, $F_2$, $D_1$, $D_2$ are shown in Figures 4.1, 4.2, 4.3, and 4.4 respectively. $P_1$, $P_2$, $E_1$, and $E_2$ are as follows.

- $P_1 = /\text{University/Department/Professors/Professor}$.
- $P_2 = /\text{WhiteBook/Person}$.
- $E_1 = (\text{:: Department/@DName} = \text{WorkInfo/AcademicUnit}) \land
  (\text{@Name} = \text{PersonName}) \land (\text{ContactInfo/Telephone} = \text{ContactInfo/Phone})$.
- $E_2 = (\text{:: Department/@DName} = \text{WorkInfo/AcademicUnit})$.

$F_1$ and $F_2$ are merged into $F_3$ by the merging operation defined above. The result XML document $F_3$ is shown in Figure 4.5. In addition, an XML document $F_{LOJ}$ is produced from $F_1$ and $F_2$ and it is presented in Figure 4.6. We examine XML document $F_{LOJ}$ in Section 4.4.

The above $P_1$ and $P_2$ specify that *Professor* elements in $F_1$ whose path is

$/\text{University/Department/Professors/Professor}$ and *Person* elements in $F_2$ whose path is

$/\text{WhiteBook/Person}$ are merged into the result XML document $F_3$. So, for this example, $e_1$ is any *Professor* element in $F_1$ and $e_2$ is any *Person* element in $F_2$.

In the above $E_1$, ::Department/@DName represents the attribute DName of the ancestor Department of Professor in $F_1$ and WorkInfo/AcademicUnit denotes the child AcademicUnit of the child WorkInfo of Person in $F_2$ (:: and @ denote an ancestor and an attribute respectively). According to $E_1$, a *Professor* element in $F_1$ and a *Person* element in $F_2$ describe the same professor and are merged into a *Professor* element in $F_3$ if the value of the
Figure 4.1: XML document \( F_1 \) in Example 5.
<WhiteBook>
  <Person Title="Professor">
    <PersonName>Alice Wang</PersonName>
    <WorkInfo>
      <Campus>Main Campus</Campus>
      <AcademicUnit>Computer Science</AcademicUnit>
    </WorkInfo>
    <ContactInfo>
      <Email>alicewang@edu.ca</Email>
      <Phone>1-613-666-2600 Ext:1234</Phone>
    </ContactInfo>
  </Person>
  <Person Title="Professor">
    <PersonName>Tina Taylor</PersonName>
    <WorkInfo>
      <Campus>New Campus</Campus>
      <AcademicUnit>Mathematics</AcademicUnit>
    </WorkInfo>
    <ContactInfo>
      <Email>tinataylor@edu.ca</Email>
      <Phone>1-613-666-2600 Ext:4321</Phone>
    </ContactInfo>
  </Person>
  <Person Title="Assistant Professor">
    <PersonName>Ann Smith</PersonName>
    <WorkInfo>
      <Campus>New Campus</Campus>
      <AcademicUnit>Mathematics</AcademicUnit>
    </WorkInfo>
    <ContactInfo>
      <Email>annsmith@edu.ca</Email>
      <Phone>1-613-666-2600 Ext:3333</Phone>
    </ContactInfo>
  </Person>
</WhiteBook>

Figure 4.2: XML document $F_2$ in Example 5.

<!DOCTYPE University [
  <!ELEMENT University (Department*)>
  <!ELEMENT Department (Professors, Courses)>  
  <!ATTLIST Department DName CDATA #REQUIRED>
  <!ELEMENT Professors (Professor*)>
  <!ELEMENT Professor (Website, ContactInfo)>  
  <!ATTLIST Professor Name CDATA #REQUIRED Title CDATA #REQUIRED>
  <!ELEMENT Website (#PCDATA)>  
  <!ELEMENT ContactInfo (Telephone, Office)>  
  <!ELEMENT Telephone (#PCDATA)>  
  <!ELEMENT Office (#PCDATA)>  
  <!ELEMENT Courses (Course*)>
  <!ELEMENT Course (#PCDATA)>  
]>  

Figure 4.3: DTD file $D_1$ in Example 5.

27
attribute $DName$ of the ancestor $Department$ of a $Professor$ is the same as the content of the descendant $AcademicUnit$ of a $Person$, the value of the attribute $Name$ of a $Professor$ is the same as the content of the child $PersonName$ of a $Person$, and the content of the descendant $Telephone$ of a $Professor$ is the same as that of the descendant $Phone$ of a $Person$. Note that the attribute $Name$ cannot identify a $Professor$ element in $F_1$ because a $Department$ may have two $Professor$ descendants that have the same value for the attribute $Name$.

We can see that the $Professor$ element in $F_1$ that has “Computer Science” as the attribute $DName$ of the ancestor $Department$, “Alice Wang” as the attribute $Name$, and “1-613-666-2600 Ext:1234” as the descendant $Telephone$ and the $Person$ element in $F_2$ that has “Computer Science” as the descendant $AcademicUnit$, “Alice Wang” as the child $PersonName$, and “1-613-666-2600 Ext:1234” as the descendant $Phone$ are matching elements because this $Professor$ element in $F_1$ and this $Person$ element in $F_2$ make Boolean expression $E_1$ true. Also, the $Professor$ element in $F_1$ that has “Mathematics” as the attribute $DName$ of the ancestor $Department$, “Tina Taylor” as the attribute $Name$, and “1-613-666-2600 ext:4321” as the descendant $Telephone$ and the $Person$ element in $F_2$ that has “Mathematics” as the descendant $AcademicUnit$, “Tina Taylor” as the child $PersonName$, and “1-613-666-2600 ext:4321” as the descendant $Phone$ are matching elements.
Figure 4.5: The result single XML document $F_3$ in Example 5.
Figure 4.6: \( F_{LOJ} \) in Example 5.
According to the above Boolean expression $E_2$, a non-matching Person in $F_2$ will be incorporated into $F_3$ if there exists a Department in $F_1$ that has an attribute DName whose value is the same as the content of the child AcademicUnit of this non-matching Person. On the other hand, if no Department in $F_1$ has an attribute DName whose value is the same as the content of the child AcademicUnit of a non-matching Person in $F_2$, this non-matching Person cannot be incorporated into $F_3$ because there does not exist an element in $F_1$ that can have this Person as a descendant.

Conflicts may emerge when merging $F_1$ and $F_2$ into $F_3$. For example, the Professor element in $F_1$ with “Tina Taylor” as attribute Name and the Person element in $F_2$ with “Tina Taylor” as child element PersonName are a pair of matching elements. This pair of matching elements are merged into the Professor element in $F_3$ that has “Tina Taylor” as attribute Name. It has attribute Title with value “Assistant Professor|Professor” because Professor in $F_1$ has an attribute Title with value “Assistant Professor” but the matching Person in $F_2$ has an attribute Title with value “Professor”. “Assistant Professor|Professor” indicates a conflict between the Title attribute of Professor element and the Title attribute of the matching Person element. It also means that it is not clear which attribute value is the correct one. More details of conflict resolution are given in Section 4.6.

4.2 The Algorithm for the Merging Operation

Before we present the algorithm for this merging operation, we introduce some terms that are used in this algorithm.

Path $\alpha$ is the prefix path of path $\beta$ if $\alpha$ is the left part of $\beta$ or $\alpha$ is equal to $\beta$. For example, /Books is the prefix path of /Books/Book, but /Class/Instructors is not the prefix path of /Class/Students because they have different second element names: Instructors and Students. It is obvious that the path of any ancestor of an element is the prefix path of the path
of this element. Path $\gamma$ is the parent path of path $\delta$ if $\gamma$ is the prefix path of $\delta$ and $\delta$ contains one more element name than $\gamma$. For Example 5, $/\text{University/Department/Professors}$ is the parent path of $/\text{University/Department/Professors/Professor}$.

The algorithm for the merging operation is as follows.

Algorithm $\text{xmlmerge}$

Input: $F_1$, $F_2$, $D_1$, $D_2$, $P_1$, $P_2$, $E_1$, and $E_2$.

Output: $F_3$.

$r_{F_1} :=$ the root element of $F_1$;

call $\text{LeftOuterJoin} (F_1, F_2, r_{F_1}, D_1, D_2, P_1, P_2, E_1, F_{LOJ})$;

/* $F_{LOJ}$ is the XML document generated by procedure $\text{LeftOuterJoin}$ */

$r_{F_{LOJ}} :=$ the root element of $F_{LOJ}$;

call $\text{FullOuterJoin} (F_{LOJ}, F_1, F_2, r_{F_{LOJ}}, D_1, D_2, P_1, P_2, E_1, E_2, F_3)$

End of algorithm $\text{xmlmerge}$

Algorithm $\text{xmlmerge}$ merges $F_1$ with DTD $D_1$ and $F_2$ with DTD $D_2$ based on $P_1$, $P_2$, $E_1$, and $E_2$, and generates an XML document $F_3$, which contains every element merged from $e_1$ in $F_1$ and its matching $e_2$ in $F_2$, each modified non-matching $e_1$ in $F_1$, some modified non-matching $e_2$ elements ($e_1$ is one of the elements in $F_1$ whose path is $P_1$ and $e_2$ is one of the elements in $F_2$ whose path is $P_2$), and the elements in $F_1$ that do not need merging.

Algorithm $\text{xmlmerge}$ calls two recursive procedures $\text{LeftOuterJoin}$ and $\text{FullOuterJoin}$. We present procedure $\text{FullOuterJoin}$ in Section 4.7.

Procedure $\text{LeftOuterJoin}$ is as follows.

Procedure 1. $\text{LeftOuterJoin}$

Procedure $\text{LeftOuterJoin} (F_1, F_2, e_{F_1}, D_1, D_2, P_1, P_2, E_1, F_{LOJ})$

if the path of $e_{F_1}$ in $F_1$ is not equal to $P_1$
then
output the start tag of $e_{F_1}$ to $F_{LOJ}$;
output all the attributes of $e_{F_1}$ to $F_{LOJ}$;

for each child element $c$ of $e_{F_1}$$\quad$
if the path of $c$ is the prefix path of $P_1$$\quad$
then call $LeftOuterJoin$ ($F_1$, $F_2$, $c$, $D_1$, $D_2$, $P_1$, $P_2$, $E_1$, $F_{LOJ}$)$\quad$
else copy $c$ to $F_{LOJ}$ $\quad$
output the end tag of $e_{F_1}$ to $F_{LOJ}$ else $\quad$
if $e_{F_1}$ has a matching element $e_{F_2}$ in $F_2$$\quad$
then $\quad$
output the start tag of $e_{F_1}$ to $F_{LOJ}$; $\quad$
for every attribute $a_1$ of $e_{F_1}$ call $processa1$ ($e_{F_2}$, $a_1$, $D_1$, $D_2$, $E_1$, $F_{LOJ}$); $\quad$
for every attribute $a_2$ of $e_{F_2}$ call $processa2$ ($e_{F_1}$, $a_2$, $D_1$, $D_2$, $E_1$, $F_{LOJ}$); $\quad$
for every child element $c_1$ of $e_{F_1}$ call $processc1$ ($e_{F_1}$, $e_{F_2}$, $e_{F_1}$, $c_1$, $D_1$, $D_2$, $E_1$, $F_{LOJ}$); $\quad$
for every child element $c_2$ of $e_{F_2}$ call $processc2$ ($e_{F_1}$, $e_{F_2}$, $c_2$, $D_1$, $D_2$, $E_1$, $F_{LOJ}$); $\quad$
output the end tag of $e_{F_1}$ to $F_{LOJ}$ else $\quad$
output the start tag of $e_{F_1}$ to $F_{LOJ}$; $\quad$
output all the attributes of $e_{F_1}$ to $F_{LOJ}$; $\quad$
for every child element $c$ of $e_{F_1}$ $\quad$
if $c$ has a semantically corresponding attribute $a$ that is an attribute of $e_2$$\quad$
then $\quad$
output an attribute to $F_{LOJ}$ whose attribute name is that of $a$ and $\quad$
whose value is the content of $c$; $\quad$
for every child element $c$ of $e_{F_1}$$\quad$
if $c$ does not have a semantically corresponding attribute that is an attribute of $e_2$$\quad$
then copy $c$ to $F_{LOJ}$; $\quad$
output the end tag of $e_{F_1}$ to $F_{LOJ}$; End of procedure $LeftOuterJoin$

Based on $P_1$, $P_2$, $D_1$, $D_2$, and $E_1$, procedure $LeftOuterJoin$ merges $e_1$ in $F_1$ and its matching $e_2$ in $F_2$ and resolves conflicts by calling procedures $processa1$, $processa2$, $processc1$, and $processc2$. It also incorporates every non-matching $e_1$ in $F_1$. Moreover, it incorporates the elements in $F_1$ that do not need merging. As a result, XML document $F_{LOJ}$ that procedure $LeftOuterJoin$ produces contains every element merged from $e_1$ in $F_1$ and its matching $e_2$ in $F_2$, every modified non-matching $e_1$ in $F_1$, and the elements in $F_1$ that do not need merging.
We use Example 5 to explain the main functions of procedures process1 and process2. Procedure LeftOuterJoin merges Professor element in $F_1$ and its matching Person element in $F_2$ into Professor element in XML document $F_{LOJ}$. For each attribute of Professor element in $F_1$, procedure process1 decides how this attribute is merged into the attributes of Professor element in $F_{LOJ}$, and for each attribute of Person element in $F_2$, procedure process2 decides if and how this attribute is merged into the attributes of Professor element in $F_{LOJ}$. For instance, the attribute Title of Professor element in $F_1$ and the attribute Title of the matching Person element in $F_2$ are merged into the attribute Title of Professor element in $F_{LOJ}$. The Professor element that has “Tina Taylor” as attribute Name in $F_{LOJ}$ has attribute Title with value “Assistant Professor” because the value of the attribute Title of the Professor in $F_1$ is “Assistant Professor” but the value of the attribute Title of the matching Person in $F_2$ is “Professor”. “Assistant Professor” indicates a conflict between the attribute Title of Professor element and the attribute Title of the matching Person element. It also means that it is not clear which attribute value is the correct one.

We explain procedure LeftOuterJoin in Section 4.4 and procedures process1, process2, processc1, and processc2 in Section 4.6.

4.3 Mechanism for Identifying XML Instances

In this section, we study the mechanism for identifying XML instances presented by algorithm xmlmerge.

A Skolem function returns a value for an object as the identifier of this object. Although no Skolem function is directly calculated, the computation of Boolean expression $E_1$ has the equivalent effects as a Skolem function does. That is, a Skolem function, which concatenates the content or value of $d_1, d_{12}, \ldots$, and $d_{1k}$ in $F_1$, or the content or value of $d_2, \ldots$,
\(d_2, \ldots, d_{2n}\) in \(F_2\), and returns this concatenated value for an object as the identifier of this object, is constructed. As long as two identifiers for two objects described in \(F_1\) and \(F_2\) are equivalent, these two objects are the same object.

For Example 5, the constructed Skolem function concatenates the attribute \(DName\) of the ancestor \(Department\), the attribute \(Name\), and the descendant \(Telephone\) of a \(Professor\) element in \(F_1\), or the descendant \(AcademicUnit\), the child \(PersonName\), and the descendant \(Phone\) of a \(Person\) element in \(F_2\) and returns this concatenated value for an object as the identifier. As long as two identifiers for two objects in \(F_1\) and \(F_2\) returned by this Skolem function are the same, these two objects refer to the same object in the real world and the corresponding \(Professor\) element in \(F_1\) and \(Person\) element in \(F_2\) are merged.

More details of the mechanism for identifying XML instances are included in Chapter 5.

4.4 The Generated XML Document \(F_{LOJ}\)

In this section, we examine XML document \(F_{LOJ}\) that procedure \(LeftOuterJoin\) produces.

In \(LeftOuterJoin (F_1, F_2, e_{F_1}, D_1, D_2, P_1, P_2, E_1, F_{LOJ})\), \(e_{F_1}\) is the currently processed element in \(F_1\) and it always has the property: \(e_{F_1}\) is one of the elements in \(F_1\) that need merging, or \(e_{F_1}\) does not need merging but some of the descendants of \(e_{F_1}\) need merging.

At first, \(e_{F_1}\) is the root element of \(F_1\) and it has the above property.

When \(LeftOuterJoin\) processes \(e_{F_1}\), it compares the path of \(e_{F_1}\) in \(F_1\) with \(P_1\) to determine if this \(e_{F_1}\) is one of the elements that need merging.

If the path of \(e_{F_1}\) in \(F_1\) is not \(P_1\), it outputs a child element of \(e_{F_1}\) to \(F_{LOJ}\) when the path of this child element is not the prefix path of \(P_1\), or it recursively calls itself for a child element of \(e_{F_1}\) when the path of this child element is the prefix path of \(P_1\). This
means that \textit{LeftOuterJoin} only recursively calls itself for elements whose path is the prefix path of \( P_1 \). Based on the definition of prefix path in Section 4.2 and the characteristics of \( P_1 \) mentioned in Section 4.1.1, elements whose path is the prefix path of \( P_1 \) are elements that need merging or they have descendants that require merging. Therefore, for procedure \textit{LeftOuterJoin}, the currently processed element \( e_{F_1} \) always has the above property.

If the path of \( e_{F_1} \) in \( F_1 \) is equal to \( P_1 \), \( e_{F_1} \) is \( e_1 \) that needs merging (\( e_1 \) is one of the elements in \( F_1 \) whose path is \( P_1 \)). \textit{LeftOuterJoin} does not recursively call itself any more and merges \( e_{F_1} \) in \( F_1 \) and its matching element \( e_{F_2} \) in \( F_2 \) into \( F_{LOJ} \) if \( e_{F_1} \) has a matching element in \( F_2 \).

In the following, we examine if and how procedure \textit{LeftOuterJoin} incorporates each element in \( F_1 \) into \( F_{LOJ} \).

Assume that \( y \) is an element in \( F_1 \) that needs merging, and \( x \) is an element in \( F_1 \) that does not need merging and \( x \) is not a descendant of \( y \). So, the path of \( y \) in \( F_1 \) should be \( P_1 \) (i.e. \( y \) is \( e_1 \)) and the path of \( x \) in \( F_1 \) should be different from \( P_1 \).

We consider the relationship between \( x \) and \( y \) in \( F_1 \). There are four cases:

1. \( x \) is an ancestor of \( y \).
2. \( x \) is a sibling of an ancestor of \( y \).
3. \( x \) and \( y \) are siblings.
4. \( x \) is a descendant of a sibling of \( y \).

We illustrate the above four cases in Figure 4.7.

For (1), if \( x \) is not the parent of \( y \), let \( a \) be a child of \( x \) and an ancestor of \( y \). When procedure \textit{LeftOuterJoin} processes \( x \), it copies the start tag of \( x \) and all the attributes of \( x \) into \( F_{LOJ} \). Then, it copies the siblings of \( a \) whose element names are not the same as the element name of \( a \) into \( F_{LOJ} \), and calls itself recursively for \( a \) because the path of \( a \) is the prefix path of \( P_1 \), or calls itself recursively for each \( a \) if the cardinality of \( a \) is greater than
Figure 4.7: Element \( x \) and element \( y \) in \( F_1 \)

1. Note that it keeps the order of the children of \( x \) in \( F_1 \). Finally, it copies the end tag of \( x \). If \( x \) is the parent of \( y \), when \texttt{LeftOuterJoin} processes \( x \), it outputs the start tag of \( x \) and all the attributes of \( x \) into \( F_{LOJ} \). Then, it copies the siblings of \( y \) whose element names are not the same as the element name of \( y \), and calls itself for \( y \) because the path of \( y \) is \( P_1 \), or calls itself for each \( y \) if the cardinality of \( y \) is greater than 1. Also, it copies the end tag of \( x \). When \texttt{LeftOuterJoin} calls itself for \( y \), it merges \( y \) and its matching element in \( F_2 \) into \( F_{LOJ} \) if \( y \) has a matching element in \( F_2 \), or it modifies \( y \) and incorporates this modified \( y \) into \( F_{LOJ} \) if \( y \) does not have a matching element in \( F_2 \). Note that \( y \) is a non-matching \( e_1 \) when \( y \) does not have a matching element in \( F_2 \). \texttt{LeftOuterJoin} modifies each non-matching \( e_1 \) to make each non-matching \( e_1 \) obey the structure of the element in \( F_{LOJ} \) that is merged from \( e_1 \) and its matching \( e_2 \). For Example 5, \texttt{Department} element is an ancestor of \texttt{Professor} element. It is incorporated in \( F_{LOJ} \) and its \texttt{Professor} descendant is merged with \texttt{Person} element in \( F_2 \).
For (2), let the ancestor of \( y \) be \( b \). Also, assume that the parent of \( x \) and \( b \) is \( p \). When \texttt{LeftOuterJoin} processes \( p \), it copies the start tag of \( p \) and all the attributes of \( p \). Then, it copies the siblings of \( b \) whose element names are not the same as the element name of \( b \), including \( x \), and recursively calls itself for \( b \) because the path of \( b \) is the prefix path of \( P_1 \), or calls itself recursively for each \( b \) if the cardinality of \( b \) is greater than 1. Finally, it copies the end tag of \( p \). In Example 5, \texttt{Courses} element is a sibling of an ancestor of \texttt{Professor} element in \( F_1 \) and the content, the structure, and the path of \texttt{Courses} element remain unchanged in \( F_{LOJ} \).

For (3), let the parent of \( x \) and \( y \) be \( q \). When \texttt{LeftOuterJoin} processes \( q \), it copies the start tag of \( q \) and all the attributes of \( q \). Then, it copies the siblings of \( y \) whose element names are not the same as the element name of \( y \), including \( x \), and calls itself for \( y \) because the path of \( y \) is \( P_1 \), or calls itself for each \( y \) if the cardinality of \( y \) is greater than 1. Finally, it copies the end tag of \( q \).

For (4), assume that the ancestor of \( x \) that is a sibling of \( y \) is \( c \). In addition, let the parent of \( c \) and \( y \) be \( r \). When \texttt{LeftOuterJoin} processes \( r \), it copies the start tag of \( r \) and all the attributes of \( r \). Then, it copies the siblings of \( y \) whose element names are not the same as the element name of \( y \), including \( c \), and calls itself for \( y \) because the path of \( y \) is \( P_1 \), or calls itself for each \( y \) if the cardinality of \( y \) is greater than 1. Finally, it copies the end tag of \( r \). Note that \( x \) is copied to \( F_{LOJ} \) because \( x \) is a descendant element of \( c \) and \( c \) is copied to \( F_{LOJ} \).

To sum up, procedure \texttt{LeftOuterJoin} merges \( y \) in \( F_1 \) and its matching element in \( F_2 \) into \( F_{LOJ} \). If \( y \) does not have a matching element, \( y \) is incorporated into \( F_{LOJ} \). In addition, procedure \texttt{LeftOuterJoin} copies \( x \) in \( F_1 \) to \( F_{LOJ} \) (i.e. \( x \) keeps its structure and content in \( F_{LOJ} \)) when \( x \) is not an ancestor of \( y \), or incorporates \( x \) into \( F_{LOJ} \) and merges descendant \( y \) of \( x \) with the matching element of \( y \) in \( F_2 \) when \( x \) is an ancestor of \( y \).

We examine XML document \( F_{LOJ} \) presented in Figure 4.6 in Example 5. \( F_{LOJ} \) consists
of every element merged from Professor element in $F_1$ and its matching Person element in $F_2$: the merged Professor element in $F_{LOJ}$ that has “Alice Wang” as the attribute Name and “1-613-666-2600 Ext:1234” as the descendant Telephone and the merged Professor element in $F_{LOJ}$ that has “Tina Taylor” as the attribute Name and “1-613-666-2600 ext:4321” as the descendant Telephone. $F_{LOJ}$ also incorporates each non-matching $e_1$. The Professor in $F_1$ that has “Alice Wang” as the attribute Name and “613-666-2600 Ext:8888” as the descendant Telephone is in $F_{LOJ}$. Moreover, $F_{LOJ}$ incorporates the elements in $F_1$ that do not need merging. For example, Courses elements and Department elements are in $F_{LOJ}$.

4.5 The Semantic Associations between Two XML Documents to be Merged

There should exist semantic associations between the two XML documents to be merged since there is a need to merge these two XML documents. More specifically, some elements in these two XML documents should describe the same kinds of objects in the real world. In this section, We explore the semantic associations between these two XML documents and introduce notions such as semantically identical elements, semantically corresponding elements, semantically identical attributes, and semantically corresponding attribute and element to describe the relationship between related elements in $F_1$ and $F_2$.

4.5.1 The Assumptions about Two XML Documents

We rephrase the assumptions about $F_1$ and $F_2$ mentioned at the beginning of Section 4.1.1.
(1) $F_1$ and $F_2$ share many tag names and also have some tags with different tag names.
(2) Two tags that share the same tag name in $F_1$ and $F_2$ describe the same kind of objects in the real world, but the corresponding elements can have the same structure or have different structures.
(3) For tags with different tag names in $F_1$ and $F_2$, some of them can still describe the same kinds of objects. In this case, Boolean expression $E_1$ indicates that they describe the same kinds of objects.

(4) For two tags in $F_1$ and $F_2$ that describe the same kind of objects, the corresponding elements have the same cardinality.

(5) For two elements whose tags describe the same kind of objects in $F_1$ and $F_2$, their two attributes have the same attribute type and the same default value if these two attributes have the same attribute name in $F_1$ and $F_2$.

4.5.2 Notions

Elements whose tags describe the same kind of objects in $F_1$ and $F_2$ can be classified into two categories: semantically identical elements and semantically corresponding elements.

We say that two elements in $F_1$ and $F_2$ are semantically identical elements if their tags describe the same kind of objects and they have the same structure.

Assume that Number element is described as

```xml
<!ELEMENT Number (#PCDATA)>
```

in two XML documents to be merged. These two Number elements are semantically identical. Two semantically identical elements can have different element names. In this case, Boolean expression $E_1$ indicates that they describe the same kind of objects. In Example 5, the descendant Telephone of Professor element in $F_1$ and the descendant Phone of Person element in $F_2$ are semantically identical elements because $(ContactInfo/TelPhone = ContactInfo/Phone)$ is one of the relational expressions in Boolean Expression $E_1$ and both Telephone and Phone are specified as parsed character data elements.
For a pair of elements with the same tag name in $F_1$ and $F_2$, they express the same kind of objects but may have different structures. Two elements in $F_1$ and $F_2$ are *semantically corresponding elements* if their tags describe the same kind of objects but they have different structures. We consider Example 5. */ContactInfo* element in $F_1$ and */ContactInfo* element in $F_2$ both describe contact information, but they have different structures. They are semantically corresponding elements. Also, two semantically corresponding elements can have different element names. In this case, Boolean expression $E_1$ indicates that they describe the same kind of objects. We still consider Example 5. The ancestor */Department* of */Professor* element in $F_1$ and the descendant */AcademicUnit* of */Person* element in $F_2$ are semantically corresponding elements because ($\text{Department/@DName = WorkInfo/AcademicUnit}$) is one of the relational expressions in Boolean Expression $E_1$. In addition, $e_1$ in $F_1$ and $e_2$ in $F_2$ should be semantically corresponding elements because they actually express the same kind of objects and they describe the same object in the real world if they make Boolean expression $E_1$ true. For Example 5, a */Professor* element in $F_1$ and a */Person* element in $F_2$ are semantically corresponding elements since they both describe professors.

Two attributes in $F_1$ and $F_2$ are said to be *semantically identical attributes* if they have the same name, and one is an attribute of an element in $F_1$ and the other is an attribute of the semantically identical or corresponding element of this element in $F_2$. In Example 5, the attribute */Title* of */Professor* element in $F_1$ and the attribute */Title* of */Person* element in $F_2$ are semantically identical attributes. Also, two semantically identical attributes can have different names. In this case, they are specified as semantically identical attributes in Boolean expression $E_1$.

Moreover, an attribute in one XML document to be merged can have a semantically corresponding element in another XML document to be merged. An attribute and an element are *a pair of semantically corresponding attribute and element* if the name of this
attribute is the same as the name of this element, this attribute is an attribute of element \( g \) in one XML document to be merged and this element is a child element of the semantically corresponding element of element \( g \) in another XML document to be merged, this attribute is a required attribute and of type CDATA, and this element is specified as a parsed character data element with cardinality 1 and it does not have any attribute. Also, the attribute name and element name of a pair of semantically corresponding attribute and element can be different. In this case, Boolean expression \( E_1 \) indicates that they are a pair of semantically corresponding attribute and element. For Example 5, the attribute \textit{Name} of \textit{Professor} element in \( F_1 \) and the child element \textit{PersonName} of \textit{Person} element in \( F_2 \) are a pair of semantically corresponding attribute and element because \textit{Professor} in \( F_1 \) and \textit{Person} in \( F_2 \) are semantically corresponding elements and \((@Name = PersonName)\) is specified in Boolean expression \( E_1 \).

### 4.6 Merging XML Elements and Handling Typical Conflicts

In this section, we see how procedure \textit{LeftOuterJoin} merges XML elements and handles conflicts. By examining four procedures \textit{processa1}, \textit{processa2}, \textit{processc1}, and \textit{processc2} that \textit{LeftOuterJoin} calls to merge \( e_1 \) in \( F_1 \) and its matching \( e_2 \) in \( F_2 \) and resolve conflicts, we demonstrate the method for merging XML elements and handling typical conflicts presented by algorithm \textit{xmlmerge}.

#### 4.6.1 Typical Conflicts

If \( e_{F_1} \) has a matching element \( e_{F_2} \) in \( F_2 \), procedure \textit{LeftOuterJoin} merges \( e_{F_1} \) in \( F_1 \) and \( e_{F_2} \) in \( F_2 \) into an element in \( F_{LOJ} \). Note that \( e_{F_1} \) in \( F_1 \) is \( e_1 \) and \( e_{F_2} \) in \( F_2 \) is the matching \( e_2 \) of \( e_{F_1} \). Let \( a_1 \) be an attribute of \( e_{F_1} \) and \( a_2 \) an attribute of \( e_{F_2} \). Let \( c_1 \) be a child element of \( e_{F_1} \)
and \( c_2 \) a child element of \( e_{F_2} \). Conflicts may emerge when merging \( F_1 \) and \( F_2 \) into \( F_{LOJ} \). Typical conflicts are:

- Conflicts between \( a_1 \) and \( a_2 \).

- Conflicts between \( a_1 \) and \( c_2 \).

- Conflicts between \( c_1 \) and \( a_2 \).

- Conflicts between \( c_1 \) or a descendant of \( c_1 \) and \( c_2 \) or a descendant of \( c_2 \) (i.e. conflicts between a descendant of \( e_{F_1} \) and a descendant of \( e_{F_2} \)).

- Conflicts between \( c_2 \) or a descendant of \( c_2 \) and an ancestor of \( e_{F_1} \) (i.e. conflicts between a descendant of \( e_{F_2} \) and an ancestor of \( e_{F_1} \)).

In Example 5, the attribute \( \text{Title} \) of \( \text{Professor} \) element in \( F_1 \) and the attribute \( \text{Title} \) of \( \text{Person} \) element in \( F_2 \) are semantically identical attributes. We can see that the title of Tina Taylor in \( F_1 \) is “Assistant Professor” but the title of Tina Taylor in \( F_2 \) is “Professor”. This is a conflict between \( a_1 \) and \( a_2 \). The child element \( \text{AcademicUnit} \) of the child element \( \text{WorkInfo} \) of \( \text{Person} \) element in \( F_2 \) and the ancestor \( \text{Department} \) of \( \text{Professor} \) element in \( F_1 \) are semantically corresponding elements because \( (: \text{Department/}@DName = \text{WorkInfo/AcademicUnit}) \) is one of the relational expressions in Boolean Expression \( E_1 \). If the child \( \text{WorkInfo of Person} \) element in \( F_2 \) is simply merged into \( \text{Professor} \) element in \( F_{LOJ} \), \( \text{AcademicUnit} \) element in \( F_{LOJ} \) has a \( \text{Department} \) ancestor that is a semantically corresponding element of this \( \text{AcademicUnit} \) element. This is a conflict between a descendant of \( c_2 \) and an ancestor of \( e_{F_1} \).
4.6.2 Merging XML Elements and Handling Typical Conflicts

When merging $e_{F_1}$ in $F_1$ and its matching $e_{F_2}$ in $F_2$, if attribute $a_1$ of $e_{F_1}$ has neither a semantically identical attribute that is an attribute of $e_{F_2}$ nor any semantically corresponding element that is a child element of $e_{F_2}$, $a_1$ is included into the element in $F_{LOJ}$ that is merged from $e_{F_1}$ and its matching $e_{F_2}$ as an attribute. If attribute $a_1$ of $e_{F_1}$ in $F_1$ has a semantically identical attribute that is an attribute of $e_{F_2}$ in $F_2$, $a_1$ and its semantically identical attribute should be merged into one attribute. That is, the element in $F_{LOJ}$ that is merged from $e_{F_1}$ and its matching $e_{F_2}$ should have only one attribute to reflect $a_1$ and its semantically identical attribute. If $a_1$ and its semantically identical attribute are consistent with each other, redundancy is eliminated by merging them into one attribute; otherwise, a conflict is indicated in the merged one attribute. Similarly, if attribute $a_1$ of $e_{F_1}$ in $F_1$ has a semantically corresponding element that is a child element of $e_{F_2}$ in $F_2$, $a_1$ and its semantically corresponding element should be merged into one attribute.

Procedure processal accomplishes the above tasks.

For each attribute $a_1$ of $e_{F_1}$, LeftOuterJoin calls procedure processal to decide if $a_1$ has a semantically identical attribute that is an attribute of $e_{F_2}$ in $F_2$ to solve conflicts between $a_1$ and any $a_2$, determine if $a_1$ has a semantically corresponding element that is a child element of $e_{F_2}$ in $F_2$ to solve conflicts between $a_1$ and any $c_2$, and decide how $a_1$ is merged into $F_{LOJ}$. If two semantically identical attributes $a_1$ of $e_{F_1}$ and $a_2$ of $e_{F_2}$ have inconsistent values value(a1) and value(a2), procedure processal outputs value(a1)|value(a2) as the value of attribute $a_1$ of the merged element in $F_{LOJ}$ (this merged element in $F_{LOJ}$ is the element in $F_{LOJ}$ that is merged from $e_{F_1}$ and its matching $e_{F_2}$). This value(a1)|value(a2) is an or-value and it implies it is not clear which one is the correct one [23].

We present procedure processal in the following.

Procedure 2. processal

Procedure processal ($e_{F_2}$, $a_1$, $D_2$, $E_1$, $F_{LOJ}$)
if $a_1$ and attribute $a_2$ of $e_{F_2}$ are semantically identical attributes
then
  compare the value of $a_1$ and the value of $a_2$;
  if there is a conflict between $a_1$ and $a_2$
    then
      output an attribute whose attribute name is that of $a_1$ and whose value is
      \[
      \text{value}(a_1) \| \text{value}(a_2) \text{ to } F_{LOJ}
      \]
      /* \text{value}(a_1) \text{ and } \text{value}(a_2) \text{ represent the value of } a_1 \text{ and}
      \text{the value of } a_2 \text{ respectively} */
    else
      output $a_1$ to $F_{LOJ}$
  else
    output $a_1$ to $F_{LOJ}$
else
  if $a_1$ and child element $c_2$ of $e_{F_2}$ are a pair of semantically corresponding attribute
  and element
    then
      compare the value of $a_1$ and the content of $c_2$;
      if there is a conflict between $a_1$ and $c_2$
        then
          output an attribute whose attribute name is that of $a_1$ and whose value is
          \[
          \text{value}(a_1) \| \text{content}(c_2) \text{ to } F_{LOJ}
          \]
          /* content($c_2$) is the content of $c_2$ */
        else
          output $a_1$ to $F_{LOJ}$
      else
        output $a_1$ to $F_{LOJ}$

End of procedure processo1

Analogously, if attribute $a_2$ of $e_{F_2}$ in $F_2$ has a semantically corresponding element that
is a child element of $e_{F_1}$ in $F_1$, $a_2$ and its semantically corresponding element should be
merged into one attribute. Procedure processo2 carries out this task.

For each attribute $a_2$ of $e_{F_2}$, LeftOuterJoin calls procedure processo2 to check if $a_2$ has
been processed by processo1, determine if $a_2$ has a semantically corresponding element
that is a child element of $e_{F_1}$ in $F_1$ to solve conflicts between $a_2$ and any $c_1$, and decide if
and how $a_2$ is merged into $e_{F_1}$ element in $F_{LOJ}$.

We present procedure processo2 in the following.
Procedure 3. processa2

Procedure processa2 ($e_{F_1}, a_2, D_1, E_1, F_{LOJ}$)

if $a_2$ and attribute of $e_{F_1}$ are not semantically identical attributes
then
  if $a_2$ and child element $c_1$ of $e_{F_1}$ are a pair of semantically corresponding attribute and element
  then
    compare the value of $a_2$ and the content of $c_1$;
    if there is a conflict between $a_2$ and $c_1$
    then
      output an attribute whose attribute name is that of $a_2$ and whose value is $value(a_2)|content(c_1)$ to $F_{LOJ}$
  else
    output $a_2$ to $F_{LOJ}$
else output $a_2$ to $F_{LOJ}$

End of procedure processa2

We consider Example 5. The attribute Name of Professor element in $F_1$ and the child element PersonName of Person element in $F_2$ are a pair of semantically corresponding attribute and element. They are consistent with each other and they are merged into the Name attribute of Professor element in $F_{LOJ}$. The attribute Title of Professor element in $F_1$ and the attribute Title of Person element in $F_2$ are semantically identical attributes and they are merged into the Title attribute of Professor element in $F_{LOJ}$. $F_{LOJ}$ reports a conflict for the Title attribute: the title of Tina Taylor is “Assistant Professor|Professor”.

When merging $e_{F_1}$ in $F_1$ and its matching $e_{F_2}$ in $F_2$, if a child element $c_1$ of $e_{F_1}$ has neither a semantically corresponding attribute that is an attribute of $e_{F_2}$ nor any semantically corresponding or identical element that is a descendant of $e_{F_2}$ in $F_2$, $c_1$ is included into the merged element in $F_{LOJ}$ as a child element. If a descendant of $e_{F_1}$ and a descendant of $e_{F_2}$ are semantically corresponding or identical elements, they should be merged when merging $e_{F_1}$ and its matching $e_{F_2}$.

The relationship between a descendant of $e_{F_1}$ and a descendant of $e_{F_2}$ is illustrated in Figure 4.8. Recall that $c_1$ is a child element of $e_{F_1}$ and $c_2$ is a child element of $e_{F_2}$. In
this figure, (c) shows that no correspondence of an element and its semantically identical or corresponding element between \( c_1 \) or any descendant of \( c_1 \) and \( c_2 \) or any descendant of \( c_2 \) is found: \( c_1 \) and \( c_2 \) are unrelated.

![Diagram](image)

**Figure 4.8:** The relationship between \( c_1 \) and \( c_2 \)

Assume that \( d_1 \) is a descendant of \( e_{F_1} \), \( d_2 \) is a descendant of \( e_{F_2} \), and \( d_1 \) and \( d_2 \) are semantically corresponding or identical elements. Based on the assumptions about two XML document to be merged, \( d_1 \) and \( d_2 \) have the same cardinality. If the cardinality of \( d_1 \) or \( d_2 \) is not greater than one, \( d_1 \) and \( d_2 \) are merged into an element and conflicts between them are detected and reported. Otherwise, \( d_1 \) and \( d_2 \) cannot be simply merged into an element. When \( d_1 \) and \( d_2 \) describe the same object, they are merged into an element; when \( d_1 \) and \( d_2 \) describe different objects, both \( d_1 \) and \( d_2 \) are incorporated into the merged element in \( F_{LOJ} \). Moreover, when \( d_1 \) and \( d_2 \) are semantically corresponding elements that
represent the same object, if $d_2$ has some attributes and/or descendants that $d_1$ does not have, an element that has both the attributes and descendants of $d_1$ and the extra attributes and/or descendants of $d_2$ is incorporated into the merged element in $F_{LOJ}$.

Recursive procedures $processc1$ carries out the tasks described above. For each child element $c_1$ of $e_{F_1}$, $LeftOuterJoin$ calls procedure $processc1$ to decide if and how $c_1$ should be merged into $e_{F_1}$ element in $F_{LOJ}$ and detect and solve conflicts between $c_1$ or a descendant of $c_1$ and any $c_2$ or a descendant of $c_2$.

We give a more detailed description of procedure $processc1$. For each child element $c_1$ of $e_{F_1}$, $processc1$ first checks if $c_1$ has a semantically corresponding attribute that is an attribute of $e_{F_2}$ based on the DTDs of $F_1$ and $F_2$ and Boolean expression $E_1$. Then, it determines if $c_1$ has any semantically identical or corresponding element(s) that is (are) a descendant(s) of $e_{F_2}$ according to the DTDs of $F_1$ and $F_2$ and Boolean expression $E_1$. If it does not find any semantically identical or corresponding element(s) of $c_1$, it outputs $c_1$ when $c_1$ has no child elements, or outputs the start tag of $c_1$ and all attributes of $c_1$, calls itself recursively for each child element $j_1$ of $c_1$ to decide how $j_1$ should be merged into $F_{LOJ}$ and detect and solve conflicts between $j_1$ and any $c_2$, and outputs the end tag of $c_1$ when $c_1$ has child elements. Otherwise, $processc1$ takes the cardinality of $c_1$ into consideration, merges $c_1$ and its semantically identical or corresponding element(s) that is (are) a descendant(s) of $e_{F_2}$, and reports conflicts if there are any conflicts.

Procedure $processc1$ is presented below.

**Procedure 4. processc1**

Procedure $processc1$($e_{F_1}$, $e_{F_2}$, $p_{c_1}$, $c_1$, $D_1$, $D_2$, $E_1$, $F_{LOJ}$)

if ($c_1$ is not specified in $E_1$) or ($c_1$ is specified in $E_1$ and $c_1$ makes $E_1$ true with $e_{F_2}$)

then

if ($c_1$ is a child element of $e_{F_1}$ and $c_1$ does not have any semantically corresponding attribute that is an attribute of $e_{F_2}$) or ($c_1$ is not a child element of $e_{F_1}$)

then

$S_{id} :=$ all the semantically identical elements of $c_1$ that are descendants of $e_{F_2}$;

if $S_{id}$ is empty

48
then
\[ S_{co} := \text{all the semantically corresponding elements of } c_1 \text{ that are descendants of } e_{F_2}; \]
if \( S_{co} \) is empty
then
if \( c_1 \) has child elements
then
output the start tag of \( c_1 \) to \( F_{LOJ} \);
output all the attributes of \( c_1 \) to \( F_{LOJ} \);
for every child element \( j_1 \) of \( c_1 \)
\[ \text{call } processc1 \left( e_{F_1}, e_{F_2}, c_1, j_1, D_1, D_2, E_1, F_{LOJ} \right); \]
output the end tag of \( c_1 \) to \( F_{LOJ} \)
else
output \( c_1 \) to \( F_{LOJ} \)
else
if the cardinality of \( c_1 \leq 1 \)
then
compare \( c_1 \) and the only element \( e_{co} \) in \( S_{co} \);
if there are conflicts between \( c_1 \) and \( e_{co} \)
then
output \( c_1 \) and report conflicts between \( c_1 \) and \( e_{co} \)
else
if \( e_{co} \) has some attributes and/or descendants that \( c_1 \) does not have
then
output an element whose element name is that of \( c_1 \) and that has both
the attributes and descendants of \( c_1 \) and the extra attributes and/or
descendants of \( e_{co} \) to \( F_{LOJ} \)
else
output \( c_1 \) to \( F_{LOJ} \)
else
if \( c_1 \) is the first \( c_1 \) child element of \( p_{c_1} \) among all \( c_1 \) child elements
of \( p_{c_1} \)
then
\[ S_{c_1} := \text{all } c_1 \text{ child elements of } p_{c_1}; \]
\[ S_{coNotInS_{c_1}} := \text{all elements in } S_{co} \text{ that do not have corresponding elements in } S_{c_1}; \]
\(/!* \text{an element in } S_{co} \text{ and its corresponding element in } S_{c_1} \text{ describe the same object} */!*
if an element in \( S_{co} \) has some attributes and/or descendants that \( c_1 \) does not have
then
\[ S_{c_1InS_{co}} := \text{all elements in } S_{c_1} \text{ that have corresponding elements in } S_{co}; \]
\[ S_{c_1NotInS_{co}} := \text{all elements in } S_{c_1} \text{ that do not have corresponding elements in } S_{co}; \]
\( S_{coInSc_1} := \) all elements in \( S_{co} \) that have corresponding elements in \( S_{c_1} \);
output every element in \( S_{c_1NotInSc_1} \) to \( F_{LOJ} \);
for each element \( e_{c_1InSc_1} \) in \( S_{c_1InSc_1} \)
let \( e_{coInSc_1} \) be the corresponding element of \( e_{c_1InSc_1} \) in \( S_{coInSc_1} \);
\text{ /* } e_{coInSc_1} \text{ and } e_{c_1InSc_1} \text{ describe the same object */ }
output an element whose element name is that of \( e_{c_1InSc_1} \) and that has both
the attributes and descendants of \( e_{c_1InSc_1} \) and the extra attributes and/or
descendants of \( e_{coInSc_1} \) to \( F_{LOJ} \)
for every element \( e_{coNotInSc_1} \) in \( S_{coNotInSc_1} \)
according to \( E_1 \) change some tag names of \( e_{coNotInSc_1} \) if any;
output this modified \( e_{coNotInSc_1} \) to \( F_{LOJ} \)
else
output each element in \( S_{c_1} \) to \( F_{LOJ} \);
for every element \( e_{coNotInSc_1} \) in \( S_{coNotInSc_1} \)
according to \( E_1 \) change some tag names of \( e_{coNotInSc_1} \) if any;
output this modified \( e_{coNotInSc_1} \) to \( F_{LOJ} \)
else
if the cardinality of \( c_1 \leq 1 \)
then
compare \( c_1 \) and the only element \( e_{id} \) in \( S_{id} \);
if there are conflicts between \( c_1 \) and \( e_{id} \)
then output \( c_1 \) and report conflicts between \( c_1 \) and \( e_{id} \)
else output \( c_1 \)
else
if \( c_1 \) is the first \( c_1 \) child element of \( p_{c_1} \) among all \( c_1 \) child elements
of \( p_{c_1} \)
then
\( S_{c_1} := \) all \( c_1 \) child elements of \( p_{c_1} \);
\( S_{idNotInSc_1} := \) all elements in \( S_{id} \) that do not have corresponding elements in \( S_{c_1} \);
\text{ /* } an element in \( S_{id} \) and its corresponding element in \( S_{c_1} \\
\text{ describe the same object */ }
output each element in \( S_{c_1} \) to \( F_{LOJ} \);
for every element \( e_{idNotInSc_1} \) in \( S_{idNotInSc_1} \)
according to \( E_1 \) change some tag names of \( e_{idNotInSc_1} \) if any;
output this modified \( e_{idNotInSc_1} \) to \( F_{LOJ} \)

End of procedure processC1

Let us consider Example 5 again. We examine if and how each child element of a
Professor element in $F_1$ is merged into the merged Professor element in $F_{LOJ}$. The child element Website of Professor element in $F_1$ does not have any semantically corresponding attribute that is an attribute of Person element in $F_2$ and it does not have any semantically identical or corresponding element that is a descendant of Person element in $F_2$ either. Moreover, it does not have any child elements. Therefore, it is included into $F_{LOJ}$ as a child of the merged Professor element. The child element ContactInfo of Professor element in $F_1$ and the child element ContactInfo of Person element in $F_2$ are semantically corresponding elements with cardinality 1 and there are no conflicts between them. The child ContactInfo of Person has a child Email that the child ContactInfo of Professor does not have. Therefore, ContactInfo that has Telephone, Office, and Email child elements is incorporated into $F_{LOJ}$ as a child of the merged Professor.

When merging $e_{F_1}$ in $F_1$ and its matching $e_{F_2}$ in $F_2$, if a child element $c_2$ of $e_{F_2}$ has neither a semantically corresponding attribute that is an attribute of $e_{F_1}$ nor any semantically corresponding or identical element that is a descendant or ancestor of $e_{F_1}$, $c_2$ is included into the merged element in $F_{LOJ}$ as a child element. A descendant of $e_{F_2}$ may be a semantically corresponding or identical element of an ancestor of $e_{F_1}$. Assume that descendant $d$ of $e_{F_2}$ has a semantically corresponding or identical element that is an ancestor of $e_{F_1}$ in $F_1$. To deal with $d$, two solutions are possible. One is to simply include $d$ into the merged element in $F_{LOJ}$. This results in a typical conflict: a conflict between $c_2$ or a descendant of $c_2$ and an ancestor of $e_{F_1}$. Another is to simply exclude $d$. This also has a problem: if $d$ contains some descendants that are not semantically corresponding or identical elements of any ancestors of $e_{F_1}$, the information about these descendants of $d$ is lost in the merged element in $F_{LOJ}$. It is appropriate to reconcile these two opposing solutions by modifying $d$ and incorporating this modified $d$ into the merged element in $F_{LOJ}$.

Recursive procedures processc2 is responsible for completing the above tasks. For each child element $c_2$ of $e_{F_2}$, LeftOuterJoin calls procedure processc2 to decide if and how $c_2$
should be merged into $e_{F_1}$ element in $F_{LOJ}$ and detect and solve conflicts between $c_2$ or a descendant of $c_2$ and an ancestor of $e_{F_1}$.

We provide a more detailed description of procedure $processc2$. For any child element $c_2$ of $e_{F_2}$, $processc2$ first checks if procedures $processa1$ and $processc1$ have already processed $c_2$. Then, it determines whether there is an ancestor element of $e_{F_1}$ that is a semantically identical or corresponding element of $c_2$ based on the analysis of the DTDs of $F_1$ and $F_2$ and Boolean expression $E_1$. If there exists an ancestor of $e_{F_1}$ that is a semantically identical or corresponding element of $c_2$, $processc2$ excludes $c_2$ from $F_{LOJ}$ when $c_2$ has no child elements, or excludes the start tag of $c_2$, all attributes of $c_2$, and the end tag of $c_2$, and calls itself for each child element $j_2$ of $c_2$ to decide if and how $j_2$ should be merged into $F_{LOJ}$ and detect and solve conflicts between $j_2$ and any ancestor of $e_{F_1}$ when $c_2$ has child elements. Otherwise, it outputs $c_2$ when $c_2$ has no child elements, or outputs the start tag of $c_2$ and all attributes of $c_2$, calls itself recursively for each child element $j_2$ of $c_2$ to decide if and how $j_2$ should be merged into $F_{LOJ}$ and detect and solve conflicts between $j_2$ and any ancestor of $e_{F_1}$, and outputs the end tag of $c_2$ when $c_2$ has child elements.

Procedure $processc2$ is as follows.

**Procedure 5. processc2**

Procedure $processc2$ ($e_{F_1}, e_{F_2}, c_2, D_1, D_2, E_1, F_{LOJ}$)

if ($c_2$ is not specified in $E_1$) or ($c_2$ is specified in $E_1$ and $c_2$ makes $E_1$ true with $e_{F_1}$)
then
if ($c_2$ is a child element of $e_{F_2}$ and $c_2$ does not have any semantically corresponding attribute that is an attribute of $e_{F_1}$) or ($c_2$ is not a child element of $e_{F_2}$)
then
if $c_2$ is not a semantically identical or corresponding element of any descendant of $e_{F_1}$
then
if $c_2$ has a semantically identical or corresponding element that is an ancestor of $e_{F_1}$
then
if $c_2$ has child elements
then
for every child element $j_2$ of $c_2$
call $processc2$ ($e_{F_1}, e_{F_2}, j_2, D_1, D_2, E_1, F_{LOJ}$)
else
if $c_2$ has child elements
then
output the start tag of $c_2$ to $F_{LOJ}$;
output all the attributes of $c_2$ to $F_{LOJ}$;
for every child element $j_2$ of $c_2$
  call processc2 ($e_{F_1}, e_{F_2}, j_2, D_1, D_2, E_1, F_{LOJ}$);
output the end tag of $c_2$ to $F_{LOJ}$
else
output $c_2$ to $F_{LOJ}$
End of procedure processc2

Let us now examine Example 5 to see if and how each child element of $Person$ element in $F_2$ is merged into $Professor$ element in $F_{LOJ}$. The child element $PersonName$ of $Person$ element in $F_2$ is a semantically corresponding element of the attribute $Name$ of $Professor$ element in $F_1$ and it has already been processed by procedure processal. The child $WorkInfo$ has not been processed by procedures processal and processc1 because it does not have any semantically corresponding attribute that is an attribute of $Professor$ element in $F_1$ and it does not have any semantically identical or corresponding elements that are descendants of $Professor$ element in $F_1$ either. In addition, it has no semantically identical or corresponding element that is an ancestor of $Professor$ element in $F_1$. It has $Campus$ and $AcademicUnit$ child elements. Procedure processc2 calls itself for $Campus$ child and $AcademicUnit$ child. $Campus$ does not have any semantically identical or corresponding element that is a descendant or an ancestor of $Professor$ in $F_1$, but $AcademicUnit$ and the ancestor $Department$ of $Professor$ in $F_1$ are semantically corresponding elements. Consequently, $WorkInfo$ that has only $Campus$ child element is incorporated into $F_3$ as a child element of the merged $Professor$ element. The child element $ContactInfo$ of $Person$ element in $F_2$ is a semantically corresponding element of the child element $ContactInfo$ of $Professor$ element in $F_1$ and it has already been processed by processc1.
Figure 4.9 describes the corresponding typical conflicts each procedure presented above handles.

<table>
<thead>
<tr>
<th>Procedures</th>
<th>Typical Conflicts</th>
</tr>
</thead>
<tbody>
<tr>
<td>processa1</td>
<td>conflicts between $a_1$ and $a_2$ and conflicts between $a_1$ and $c_2$</td>
</tr>
<tr>
<td>processa2</td>
<td>conflicts between $c_1$ and $a_2$</td>
</tr>
<tr>
<td>processes1</td>
<td>conflicts between a descendant of $e_{F_1}$ and a descendant of $e_{F_2}$</td>
</tr>
<tr>
<td>processes2</td>
<td>conflicts between a descendant of $e_{F_3}$ and an ancestor of $e_{F_1}$</td>
</tr>
</tbody>
</table>

Figure 4.9: Procedure and typical conflicts they handle.

### 4.7 The Resulting XML Document of the Merging Operation

In this section, we examine XML document $F_3$ that procedure `FullOuterJoin` produces. $F_3$ is the resulting XML document of the merging operation.

Procedure `FullOuterJoin` incorporates every element in $F_{LOJ}$ into XML document $F_3$, and modifies some non-matching $e_2$ elements and inserts the modified non-matching $e_2$ elements into $F_3$ as child elements of some elements whose path is the parent path of $P_1$ ($P_1$ and $P_2$ are paths that designate the elements to be merged in $F_1$ and $F_2$ respectively, and $e_1$ is one of the elements in $F_1$ whose path is $P_1$ and $e_2$ is one of the elements in $F_2$ whose path is $P_2$). That is, XML document $F_3$ that procedure `FullOuterJoin` produces consists of every element merged from $e_1$ in $F_1$ and its matching $e_2$ in $F_2$, each modified non-matching $e_1$ in $F_1$, some modified non-matching $e_2$ elements in $F_2$, and the elements in $F_1$ that do not need merging.

`FullOuterJoin` modifies some non-matching $e_2$ elements for two reasons. One is to resolve conflicts between a descendant of non-matching $e_2$ and an ancestor of $e_1$. The other is to make the non-matching $e_2$ elements obey the structure of the element in $F_{LOJ}$ that is merged from $e_1$ and its matching $e_2$. 

54
In $\text{FullOuterJoin} \ (F_{LOJ}, F_1, F_2, e_{FLOJ}, D_1, D_2, P_1, P_2, E_1, E_2, F_3)$, at first, $e_{FLOJ}$ is the root element of $F_{LOJ}$. When the path of $e_{FLOJ}$ in $F_{LOJ}$ is the parent path of $P_1$, $\text{FullOuterJoin}$ modifies some non-matching $e_2$ elements that can make $E_2$ true with this $e_{FLOJ}$, and includes the modified non-matching $e_2$ elements into $F_3$ as the child elements of this $e_{FLOJ}$. Note that this $e_{FLOJ}$ and all its ancestors are incorporated into $F_3$. Let $e_{non}$ be a non-matching $e_2$. If $e_{non}$ makes $E_2$ true with this $e_{FLOJ}$, $\text{FullOuterJoin}$ changes the tag name of $e_{non}$ to the tag name of $e_1$ in $F_1$, changes some attribute names of $e_{non}$, converts some child elements of $e_{non}$ to attributes of $e_{non}$, and for every child element $c_{non}$ of $e_{non}$ calls recursive procedure $\text{procescn}$ to solve conflicts between $c_{non}$ and any ancestor of $e_1$ and make $c_{non}$ obey the structure of the merged element in $F_{LOJ}$.

Consider Example 5, because the non-matching $\text{Person}$ element in $F_2$ that has “Ann Smith” as the child $\text{PersonName}$ and the $\text{Professors}$ element that has the ancestor $\text{Department}$ that has attribute $\text{DName}$ with value “Mathematics” make $E_2$ true, this non-matching $\text{Person}$ is modified and included into $F_3$ as a child element of this $\text{Professors}$ element. For this non-matching $\text{Person}$, procedure $\text{FullOuterJoin}$ changes its tag name to $\text{Professor}$, converts the child $\text{PersonName}$ into the attribute $\text{Name}$ because the attribute $\text{Name}$ of $\text{Professor}$ in $F_1$ and the child $\text{PersonName}$ of $\text{Person}$ in $F_2$ are a pair of semantically corresponding attribute and element, and calls procedure $\text{procescn}$ for the child element $\text{WorkInfo}$ and the child element $\text{ContactInfo}$.

Procedure $\text{FullOuterJoin}$ is as follows.

**Procedure 6. FullOuterJoin**

Procedure $\text{FullOuterJoin} \ (F_{LOJ}, F_1, F_2, e_{FLOJ}, D_1, D_2, P_1, P_2, E_1, E_2, F_3)$

if the path of $e_{FLOJ}$ in $F_{LOJ}$ is not the parent path of $P_1$
then
output the start tag of $e_{FLOJ}$ to $F_3$;
output all the attributes of $e_{FLOJ}$ to $F_3$;
for every child element $c_{LOJ}$ of $e_{FLOJ}$
if the path of $c_{LOJ}$ is the prefix path of the parent path of $P_1$
then call $\text{FullOuterJoin} \ (F_{LOJ}, F_1, F_2, c_{LOJ}, D_1, D_2, P_1, P_2, E_1, E_2, F_3)$
else output \( c_{LOJ} \) to \( F_3 \);
output the end tag of \( e_{FLOJ} \) to \( F_3 \);
else
output the start tag of \( e_{FLOJ} \) to \( F_3 \);
output all the attributes of \( e_{FLOJ} \) to \( F_3 \);
if \( e_{FLOJ} \) has child elements
then
for every child element \( c_{LOJ} \) of \( e_{FLOJ} \)
if the path of \( c_{LOJ} \) is not \( P_1 \)
then
output \( c_{LOJ} \) to \( F_3 \)
else
if \( c_{LOJ} \) is the first \( c_{LOJ} \) child element of \( e_{FLOJ} \)
then
output all \( c_{LOJ} \) child elements of \( e_{FLOJ} \) to \( F_3 \);
\( S_{non} := \) all non-matching \( e_2 \) elements in \( F_2 \);
for every \( e_{non} \) in \( S_{non} \) /* \( e_{non} \) is a non-matching \( e_2 \) */
if \( e_{FLOJ} \) and \( e_{non} \) make \( E_2 \) true
then
output the start tag of \( e_1 \) in \( F_1 \) to \( F_3 \);
for every attribute \( \omega \) of \( e_{non} \)
if \( \omega \) has a semantically identical attribute \( \omega' \) that is an attribute of \( e_1 \)
then
output an attribute whose attribute name is that of \( \omega' \) and whose value is that of \( \omega \) to \( F_3 \)
else
output \( \omega \) to \( F_3 \);
for every child element \( c_{non} \) of \( e_{non} \)
if \( c_{non} \) has a semantically corresponding attribute \( \rho \) that is an attribute of \( e_1 \)
then
output an attribute whose attribute name is that of \( \rho \) and whose value is the content of \( c_{non} \) to \( F_3 \);
for every child element \( c_{non} \) of \( e_{non} \) call \textit{processcn} \((e_{non}, c_{non}, D_1, D_2, E_1, F_3)\);
else
\( S_{non} := \) all non-matching \( e_2 \) elements in \( F_2 \);
for every \( e_{non} \) in \( S_{non} \) /* \( e_{non} \) is a non-matching \( e_2 \) */
if \( e_{FLOJ} \) and \( e_{non} \) make \( E_2 \) true
then
output the start tag of \( e_1 \) in \( F_1 \) to \( F_3 \);
for every attribute $\omega$ of $e_{\text{non}}$
if $\omega$ has a semantically identical attribute $\omega'$ that is an attribute of $e_1$
then output an attribute whose attribute name is that of $\omega'$ and whose value
is that of $\omega$ to $F_3$;
else output $\omega$ to $F_3$;
for every child element $c_{\text{non}}$ of $e_{\text{non}}$
if $c_{\text{non}}$ has a semantically corresponding attribute $\rho$ that is an attribute of $e_1$
then output an attribute whose attribute name is that of $\rho$ and whose value is the content
of $c_{\text{non}}$ to $F_3$;
for every child element $c_{\text{non}}$ of $e_{\text{non}}$
call $\text{processen}(e_{\text{non}}, c_{\text{non}}, F_1, P_1, D_1, D_2, E_1, F_3)$;
output the end tag of $e_1$ in $F_1$ to $F_3$;
output the end tag of $e_F\text{LOJ}$ to $F_3$;

End of procedure $\text{FullOuterJoin}$

In $\text{processen}(e_{\text{non}}, \theta, F_1, P_1, D_1, D_2, E_1, F_3)$, $e_{\text{non}}$ is a non-matching $e_2$ that can be
incorporated into $F_3$, and $\theta$ is a descendant of $e_{\text{non}}$. Procedure $\text{processen}$ determines if $\theta$ has a semantically identical or corresponding element that is an ancestor of $e_1$ to solve
conflicts between $\theta$ and any ancestor of $e_1$, and decides if $\theta$ has a semantically identical or corresponding element that is a descendant of $e_1$ to make $\theta$ obey the structure of the
element merged from $e_1$ and its matching $e_2$ in $F_{\text{LOJ}}$.

Procedure $\text{processen}$ is presented below.

**Procedure 7. $\text{processen}$**

Procedure $\text{processen}(e_{\text{non}}, \theta, F_1, P_1, D_1, D_2, E_1, F_3)$
if ($\theta$ is not specified in $E_1$) or
($\theta$ is specified in $E_1$ and $\theta$ makes $E_1$ false with each $e_1$ in $F_1$ )
then
if ($\theta$ is a child element of $e_{\text{non}}$ and $\theta$ does not have a semantically corresponding
attribute that is an attribute of $e_1$) or ($\theta$ is not a child element of $e_{\text{non}}$)
then
if $\theta$ has a semantically identical or corresponding element that is an ancestor of $e_1$
then
if $\theta$ has child elements
then

57
for every child element $\sigma$ of $\theta$
call $processcn$ ($e_{non}, \sigma, F_1, P_1, D_1, D_2, E_1, F_3$)
else
if $\theta$ has a semantically identical or corresponding element $\theta'$ that is a descendant of $e_1$
then
output the start tag of $\theta'$ to $F_3$;
for every attribute $\mu$ of $\theta$
if $\mu$ has a semantically identical attribute $\mu'$ that is an attribute of $\theta'$
then output an attribute whose attribute name is that of $\mu'$ and whose value is
that of $\mu$ to $F_3$
else output $\mu$ to $F_3$;
if $\theta$ has child elements
then
for every child element $\sigma$ of $\theta$
call $processcn$ ($e_{non}, \sigma, F_1, P_1, D_1, D_2, E_1, F_3$)
else
output the content of $\theta$ to $F_3$;
output the end tag of $\theta'$ to $F_3$
else
if $\theta$ has child elements
then
output the start tag of $\theta$ to $F_3$;
output all the attributes of $\theta$ to $F_3$;
for every $\sigma$ child element of $\theta$
call $processcn$ ($e_{non}, \sigma, F_1, P_1, D_1, D_2, E_1, F_3$)
output the end tag of $\theta$ to $F_3$
else
output $\theta$ to $F_3$
End of procedure $processcn$

Let us examine Example 5. The child element $WorkInfo$ of the non-matching $Person$ does not have any semantically identical or corresponding element that is an ancestor or a descendant of $Professor$ element in $F_1$. $WorkInfo$ has $Campus$ and $AcademicUnit$ child elements. When procedure $processcn$ processes the child element $WorkInfo$ of this non-matching $Person$, it recursively calls itself for the child $Campus$ and the child $AcademicUnit$. $Campus$ does not have any semantically identical or corresponding element
that is an ancestor or a descendant of Professor in \( F_1 \), but AcademicUnit and the ancestor Department of Professor in \( F_1 \) are semantically corresponding elements. So, the child element WorkInfo that has only one child Campus is included into the Professor element in \( F_3 \) that has the attribute Name with value “Ann Smith”. The child element ContactInfo of the non-matching Person and the child element ContactInfo of Professor element in \( F_1 \) are semantically corresponding elements. When procedure processcn processes the child element ContactInfo of the non-matching Person, it recursively calls itself for each child of ContactInfo. ContactInfo has two child elements: the child Email and the child Phone. The child Email and the descendant Email of Professor are semantically identical elements. This child Email of the child ContactInfo of the non-matching Person is incorporated into \( F_3 \). The child Phone and the descendant Telephone of Professor in \( F_1 \) are semantically identical elements. So, procedure processcn changes the tag name of Phone to the tag name of Telephone and outputs this modified child Phone into \( F_3 \). That is, the child ContactInfo that has the child Email and the child Telephone is incorporated into the Professor element in \( F_3 \) that has Name attribute with value “Ann Smith”.

To sum up, the non-matching Person in \( F_2 \) that has “Ann Smith” as the child Person-Name is modified and incorporated into \( F_3 \) as the Professor element that has the attribute Name with value “Ann Smith”. This Professor element in \( F_3 \) shown in Figure 4.5 is the only element that is not included in \( F_{LOJ} \) presented in Figure 4.6. That is, the difference between \( F_3 \) and \( F_{LOJ} \) is that \( F_3 \) contains some modified non-matching \( e_2 \) elements but \( F_{LOJ} \) does not.

### 4.8 Features of Our Approach

In this section, we outline the features of our approach.

The merging operation we have defined is:
\[ F_3: = \text{merging} (F_1, F_2) \text{ on } (D_1, D_2, P_1, P_2, E_1, E_2). \]

It is obvious that any two paths in \( F_1 \) and \( F_2 \) can be assigned to \( P_1 \) and \( P_2 \). In other words, algorithm \textit{xmlmerge} can merge elements from two XML documents that have different structures at any level.

The roles that \( F_1, F_2, \) and \( F_3 \) play are different. All XML data in \( F_1 \) is useful while \( F_2 \) only contains some needed XML data. \( F_3 \), the output of algorithm \textit{xmlmerge}, incorporates all the data in \( F_1 \) and some data in \( F_2 \). If some XML data in another XML document needs to be merged into this \( F_3 \), \textit{xmlmerge} can merge this \( F_3 \) and this XML document to produce a new \( F_3 \). That is, \( F_3 \) acts as an accumulator.

Some XML processing systems flatten XML data, process it, and finally construct the XML result. In contrast, our system directly deals with XML data without flattening it. Our approach to merging of XML documents can be used for processes involving combining XML fragments and documents. It can avoid the flattening costs and improve performance.
Chapter 5

Implementation Issues

We have implemented a prototype to merge XML documents using XSLT and Java. Most functions presented in this thesis are supported.

In this chapter, we discuss implementation issues. In Section 5.1, we introduce the requirements of our XML merging system. In Section 5.2, we present a merge template XML file that can specify how to combine two input XML documents into one resulting XML document, build semantic connections between these two documents, and support recursive processing and merging of XML elements. In Section 5.3, we discuss how to use XSLT and Java to implement the prototype. In Section 5.4, we give another example of merging of XML documents to demonstrate the functions of the prototype.

5.1 The Requirements of the Merging System

1. Functionality requirements
   (a) The system should merge two XML documents that have different structures and generates one single XML document.
   (b) The system must provide an interface for users and other programs.
   (c) The system should prompt users to set up conditions for merging: paths $P_1$ and
2. Performance requirements
   The system should generate accurate results for merging of XML documents and provide fast processing speed.

5.2 A Merge Template XML File

In our implementation, a merge template XML file is used to express conditions for merging: paths \( P_1 \) and \( P_2 \) and Boolean expressions \( E_1 \) and \( E_2 \). It is used to build semantic connections between XML elements of the two input XML documents, effectively specify how to combine these two XML documents into a result XML document, and support recursive processing and merging of XML elements.

```xml
<MergeTemplate>
  <P1>
    <Path Name="University"/>
    <Path Name="Department"/>
    <Path Name="Professors"/>
    <Path Name="Professor"/>
  </P1>
  <P2>
    <Path Name="WhiteBook"/>
    <Path Name="Person"/>
  </P2>
  <E1>
    <Factor Name1="::Department/@DName" Name2="WorkInfo/AcademicUnit"
      InE2="Yes"/>
    <Factor Name1="@Name" Name2="PersonName" Function="sameName"/>
    <Factor Name1="ContactInfo/Telephone" Name2="ContactInfo/Phone"
      Function="samePhone"/>
  </E1>
</MergeTemplate>
```

Figure 5.1: An example merge template XML file.
Figure 5.1 shows an example merge template file for Example 5 in Chapter 4. In this merge template file, MergeTemplate element is the root element, which has three child elements: P1, P2, and E1. P1 indicates the path of elements to be merged in F1 shown in Figure 4.1. P2 indicates the path of elements to be merged in F2 shown in Figure 4.2. E1 gives the information for identifying XML instances and handling typical conflicts.

Recall that P1 and P2 are paths that designate the elements to be merged in F1 and F2 respectively. Each path P1 or P2 consists of several element names in the corresponding input XML document separated by /. In this example merge template file shown in Figure 5.1, each element name in Path P1 is specified in the attribute Name of each child Path of P1 element and each element name in Path P2 is specified in the attribute Name of each child Path of P2 element. This example merge template file specifies that Professor elements whose path is /University/Department/Professors/Professor in F1 shown in Figure 4.1 and Person elements whose path is /WhiteBook/Person in F2 shown in Figure 4.2 are merged into F3 shown in Figure 4.5.

The order of the element names in P1 or P2 is important. The first one is the name of the root element of the corresponding XML document and the last one indicates the name of the elements to be merged. Moreover, each pair of consecutive element names in a path is associated with a pair of a parent and a child in the corresponding XML document. Furthermore, the child element in each pair of a parent and a child in F1 associated with a pair of consecutive element names in P1 is the only kind child that needs merging or has descendants that require merging. All these characteristics are used to support recursive processing of XML elements in F1 and merging of designated elements in F1 and F2.

We take procedure LeftOuterJoin as an example to examine how this merge template XML file supports recursive processing and merging of XML elements. In our implementation, a set variable s is created. Its initial value is the set of all Path child elements of P1 element. We assume s has ordered elements in it. This order is the same as the order of
Path child elements of P1. The number of elements in s decreases as elements in F1 are being processed. Note that at first the value of the attribute Name of the first element in s is the name of the root element of F1 and this root element is currently being processed by LeftOuterJoin. In fact, the value of the attribute Name of the first element in s is always the same as the name of the XML element being processed in F1. This association and the number of elements in s are used to control and guide recursively processing of XML elements in F1 and merging of designated elements in F1 and F2. More specifically, the number of elements in s is used to determine if recursion ends. If s only has one element, the recursion ends and merging is performed. When performing merging, LeftOuterJoin calls four procedures: processa1, processa2, processc1, and processc2 to merge e1 in F1 and its matching e2 in F2 and resolve conflicts.

Each Factor child of E1 element describes a conditional expression in E1 and a Factor that has an InE2 attribute with value “Yes” also describes a conditional expression in E2. In Chapter 4, for simplicity, we assume that d1, in F1 and d2, in F2 (1 ≤ i ≤ k) specified in E1 have the same representations. In fact, they may have different formats. In our implementation, we use user-defined Boolean functions to solve this problem. Consequently, the mechanism presented in this thesis combines Skolem function and user-defined Boolean functions to identify XML instances. In Figure 5.1, two user-defined Boolean functions: samename and samephone are specified by Function attributes of two Factor child elements of E1 element. These two Boolean functions are included in an enumeration that consists of all the defined Boolean functions for identifying XML instances. Boolean function samename (name1, name2) returns true if name1 and name2 are two names that actually refer to the same name although they may have different representations. For example, samename (“Ann Smith”, “Smith, Ann”) returns true. Boolean function samephone (phone1, phone2) returns true if phone1 and phone2 represent the same phone number although they can have different formats. For instance, samephone (“1-(613)666-2600 Ext: 1234”, “666-2600 Ext: 1234”) returns true because “1-(613)666-2600 Ext: 1234” and “666-2600 Ext:
1234" actually represent the same phone number in Ottawa, Canada.

There are two reasons why we use an XML file to represent conditions for the merging operation defined in this thesis. First, the mechanism presented deals with XML documents directly and effectively and it is natural to use another XML file for merging of XML documents. Second, this merge template XML file can effectively specify how to combine two input XML documents into one single XML document and support recursive processing and merging of XML elements.

For brevity, we use DTDs in this thesis. In fact, in our implementation we use XML Schemas instead of XML DTDs. The reason is that an XML Schema is also an XML document and the mechanism presented in this thesis can deal with an XML document directly and effectively.

5.3 Using XSLT and Java to Implement the Prototype

We have implemented a prototype to merge XML documents using XSLT and Java. In this section, we discuss how to use XSLT and Java to implement the prototype.

In an XSLT stylesheet, we can define named templates and call them with proper parameters. Moreover, templates can be called recursively. The effect of named templates is analogous to procedures or subroutines in other programming languages.

Figures 5.2 and 5.3 show an XSLT stylesheet for procedure LeftOuterJoin. In this stylesheet, loj is a named template. It calls itself recursively. It has eight parameters: NameF1, NameF2, NodeSetFile1, NameS1, NameS2, PathSet1, PathSet2, and E1Set. NameF1 and NameF2 are the file names of F1 and F2. At first, NodeSetFile1 is a set that contains only one node: the root node of F1. NameS1 and NameS2 are the file names of the Schemas of F1 and F2. At first, PathSet1 is the set of child nodes of the child P1 of MergeTemplate in
Figure 5.2: XSLT stylesheet for procedure LeftOuterJoin (1)
the merge template file. \textit{PathSet2} is the set of child nodes of the child \textit{P2} of \textit{MergeTemplate}. \textit{E1Set} is the set of child nodes of the child \textit{E1} of \textit{MergeTemplate}. Template \textit{loj} outputs an XML document, which is XML document \(F_{LOJ}\) generated by procedure \textit{LeftOuterJoin}.

Template \textit{loj} calls another named template: \textit{ElementMerging}. \textit{ElementMerging} merges \(e_1\) and its matching \(e_2\), and modifies non-matching \(e_1\), and outputs the element merged from \(e_1\) and its matching \(e_2\) and the modified non-matching \(e_1\) into \(F_{LOJ}\).

Although XSLT is very powerful for XML documents processing, XSLT has some disadvantages. String manipulation is cumbersome to express in XSLT. Also, XSLT cannot support a user interaction directly. As we all know, Java can deal with strings very efficiently and can support a user interaction very well. XSLT specification defines a general mechanism for calling extension functions written in any languages. In our implementation, we make use of this mechanism to invoke code written in Java in our stylesheets to manipulate strings and support user interaction. For example, we use Java to implement user-defined Boolean functions to identity XML instances when merging.
The namespace declaration xmlns:java=http://xml.apache.org/xalan/java lets us call user-defined Java methods in XSLT stylesheets.

5.4 Additional Example

Example 6. The two input XML documents $F_1$ and $F_2$ have different structures. They describe employees by different elements: Employee elements in $F_1$ and Person elements in $F_2$. $F_1$ and $F_2$ are merged into $F_3$ by the merging operation defined in this thesis. $F_1$, $F_2$, and $F_3$ are shown in Figures 5.4, 5.5, and 5.10 respectively. XML Schemas $S_1$ for $F_1$ and $S_2$ for $F_2$ are shown in Figures 5.6 and 5.7 respectively. The merge template file is presented in Figure 5.8. Another generated XML document $F_{LOJ}$ is shown in Figure 5.9. $P_1$, $P_2$, $E_1$, and $E_2$ are as follows.

$P_1 = /Factory/Department/Employees/Employee$.

$P_2 = /FactoryInfo/People/Person$.

$E_1 = (:: Department/@DName = WorkIn/Unit) \land (\text{sameName}(Name, @PName) = true)$.

$E_2 = (:: Department/@DName = WorkIn/Unit)$.

5.4.1 Instance Identification

According to the above $P_1$ and $P_2$, Employee elements in $F_1$ whose path is

$/Factory/Department/Employees/Employee$ and Person elements in $F_2$ whose path is

$/FactoryInfo/People/Person$ are merged into the result XML document $F_3$. Thus, for this example, $e_1$ is any Employee element in $F_1$ and $e_2$ is any Person element in $F_2$.

In the above $E_1$, sameName (name1, name2) is a user-defined Boolean function, which returns true if name1 and name2 are two names that actually refer to the same name although they may have different representations.

According to $E_1$ (symbols :: and @ denote an ancestor and an attribute respectively), an Employee element in $F_1$ and a Person element in $F_2$ describe the same employee and are merged into an Employee element in $F_3$ if the value of the attribute DName of the ancestor
<Factory Name="Red Cap">
  <Department DName="Production">
    <Employees>
      <Employee>
        <Name>Smith, Paul</Name>
        <Age>35</Age>
        <Contact>
          <Phone>5555</Phone>
          <Phone>1111</Phone>
          <Address>
            <Number>78</Number>
            <Street>Main Street</Street>
          </Address>
          <Email>paul@redcap.ca</Email>
        </Contact>
      </Employee>
    </Employees>
  </Department>
  <Department DName="Sales">
    <Employees>
      <Employee>
        <Name>Shirley Green</Name>
        <Age>31</Age>
        <Contact>
          <Phone>7777</Phone>
          <Address>
            <Number>556</Number>
            <Street>Broadway</Street>
          </Address>
          <Email>shirley@redcap.ca</Email>
        </Contact>
      </Employee>
      <Employee>
        <Name>Paul Smith</Name>
        <Age>40</Age>
        <Contact>
          <Phone>8888</Phone>
          <Address>
            <Number>10</Number>
            <Street>Prince Road</Street>
          </Address>
          <Email>paul2@redcap.ca</Email>
        </Contact>
      </Employee>
    </Employees>
  </Department>
</Factory>

Figure 5.4: XML document $F_1$ in Example 6.
Figure 5.5: XML document $F_2$ in Example 6.
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Factory">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Department" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Employees">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Employee" minOccurs="0" maxOccurs="unbounded">
                      <xs:complexType>
                        <xs:sequence>
                          <xs:element name="Name" type="xs:string"/>
                          <xs:element name="Age" type="xs:integer"/>
                          <xs:element name="Contact"/>
                        </xs:complexType>
                      <xs:sequence>
                        <xs:element name="Phone" type="xs:string" maxOccurs="unbounded"/>
                        <xs:element name="Address"/>
                        <xs:sequence>
                          <xs:element name="Number" type="xs:string"/>
                          <xs:element name="Street" type="xs:string"/>
                        </xs:sequence>
                        <xs:element>
                        <xs:element name="Email" type="xs:string"/>
                        </xs:element>
                      </xs:sequence>
                    </xs:complexType>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:complexType>
          </xs:sequence>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>

Figure 5.6: XML Schema file $S_1$ for $F_1$ in Example 6
Figure 5.7: XML Schema file $S_2$ for $F_2$ in Example 6
<MergeTemplate>
  <P1>
    <Path Name="Factory"/>
    <Path Name="Department"/>
    <Path Name="Employees"/>
    <Path Name="Employee"/>
  </P1>
  <P1>
    <Path Name="FactoryInfo"/>
    <Path Name="People"/>
    <Path Name="Person"/>
  </P2>
  <E1>
    <Factor Name1="::Department/@DName" Name2="WorkIn/Unit" InE2="Yes"/>
    <Factor Name1="Name" Name2="@PName" Function="samename"/>
  </E1>
</MergeTemplate>

Figure 5.8: The merge template XML file in Example 6.

Department of an Employee is the same as the content of the descendant Unit of a Person, and the content of the child Name of an Employee and the value of the attribute PName of a Person make user-defined Boolean function samename true. Note that the child Name of an Employee element cannot identify an Employee element in $F_1$ because two Department elements may have Employee descendants that have the same content for the child Name.

The Skolem function, which concatenates the attribute DName of the ancestor Department and the child Name of an Employee element in $F_1$, or the descendant Unit and the attribute PName of a Person element in $F_2$, is constructed. Moreover, this Skolem function produces one unique instance identification for the combination of the name of a department or a unit and a person’s name with different representations because the user-defined Boolean function samename (name1, name2) returns true when name1 and name2 are two names that actually refer to the same name and have different representations. For this
example, the Skolem function returns one unique instance identification for both the combination of "Smith, Paul" and "Production" and the combination of "Paul Smith" and "Production". As a result, the mechanism presented combines Skolem function and user-defined Boolean functions to identify XML instances. As long as two identifiers for two objects in $F_1$ and $F_2$ generated by this Skolem function are the same, these two objects refer to the same object and the corresponding Employee element in $F_1$ and Person element in $F_2$ are merged.

Based on $E_1$, Employee elements in $F_1$ and Person elements in $F_2$ can be divided into four groups: matching Employee elements, non-matching Employee elements, matching Person elements, and non-matching Person elements. The Employee element in $F_1$ that has "Smith, Paul" as the child Name and "Production" as the attribute DName of the ancestor Department and the Person element in $F_2$ that has "Paul Smith" as the attribute PName and "Production" as the descendant Unit are a pair of matching elements. The Employee element in $F_1$ that has "Shirley Green" as the child Name and "Sales" as the attribute DName of the ancestor Department and the Person element in $F_2$ that has "Shirley Green" as the attribute PName and "Sales" as the descendant Unit are another pair of matching elements. The Employee element in $F_1$ that has "Paul Smith" as the child Name and "Sales" as the attribute DName of the ancestor Department is a non-matching Employee. The Person element in $F_2$ that has "Alice Bush" as the attribute PName and "Production" as the descendant Unit is a non-matching Person.

5.4.2 Element Merging and Conflict Resolution

In this section, we see how algorithm xmlmerge merges elements and resolve conflicts in Example 6. Also, we examine XML documents $F_{LOJ}$ and $F_3$ that algorithm xmlmerge produces.

- Merging of Employee Elements and Their Matching Person Elements
Figure 5.9: $F_{LOJ}$ in Example 6.
Figure 5.10: The result single XML document $F_3$ in Example 6.
The first pair of matching Employee element in $F_1$ and Person element in $F_2$ are merged into the Employee element in $F_{LOJ}$ that has “Paul Smith” as the attribute PName and “Production” as the attribute DName of the ancestor Department.

The child element Name of Employee element in $F_1$ and the attribute PName of Person element in $F_2$ are semantically corresponding element and attribute because Employee element in $F_1$ and Person element in $F_2$ are semantically corresponding elements and \((\text{sameName}(\text{Name}, @\text{PName})= \text{true})\) is one of the relational expressions in Boolean expression $E_1$. The child element Name of Employee element in $F_1$ and the attribute PName of Person element in $F_2$ are combined into the attribute PName of the merged Employee element in $F_{LOJ}$.

The child element Age of Employee element in $F_1$ and the child element Age of Person element in $F_2$ are semantically identical elements because they have the same tag name and they have the same structure. The cardinality of Age is 1, so the child element Age of Employee and the child element Age of Person are combined into the Age child element of the merged Employee element in $F_{LOJ}$. No conflicts between the child element Age of Employee element and the child element Age of Person element are reported since they are consistent with each other.

The child element Contact of Employee element does not have a semantically corresponding or identical element that is a descendant of Person element. It contains Phone, Address, Email child elements. The child element Phone of Contact and the child element Phone of Person are semantically identical elements. The cardinality of Phone is greater than 1, so the child element Phone of the child element Contact and the child element Phone of Person are usually fused into the Phone child elements of the child Contact of the merged Employee element in $F_{LOJ}$. Every child element Phone of the child element Contact of Employee is contained in the child Contact of the merged Employee element in $F_{LOJ}$. On the other hand,
for any Phone child of Person, it can be incorporated into the child Contact of the merged Employee element in F_{LOJ} only if it has no corresponding Phone child of the child Contact of Employee in F_1. We can see that the only phone child of Person, <Phone>1111</Phone>, has its corresponding Phone child of the child Contact of Employee in F_1. This corresponding Phone child element of the child Contact of Employee in F_1 is <Phone>1111</Phone>. Therefore, the child Contact of the merged Employee element in F_{LOJ} just contains Phone child elements of the child Contact of Employee element in F_1. The child Address of the child Contact of Employee and the child Address of Person are semantically corresponding elements because they have the same tag name but they have different structures. The cardinality of Address is 1, so the child Address of the child Contact of Employee and the child Address of Person are merged into the child Address of the child Contact of the merged Employee element in F_{LOJ}. There are no conflicts between the child Address of Contact and the child Address of Person, and the child Address of Person has a PostCode child that the child Address of Contact does not have, and as a result, this child PostCode is added to the child Address of the child Contact of the merged Employee element in F_{LOJ}. The child Email of the child Contact of Employee does not have a semantically corresponding or identical element that is a descendant of Person element. This child Email is embodied in the child Contact of the merged Employee element in F_{LOJ}.

The child WorkIn of Person element in F_2 does not have any semantically corresponding or identical element that is a descendant or an ancestor of Employee element in F_1. It has Factory, Unit, and Group child elements. The ancestor Factory of Employee element in F_1 and the child Factory of the child WorkIn of Person in F_2 are semantically corresponding elements since they have the same tag name and different structures. Also, the ancestor Department of Employee and the child Unit of the child WorkIn of Person in F_2 are semantically corresponding elements because
(:Department/@DName = WorkIn/Unit) is specified in Boolean expression \( E_1 \). Consequently, WorkIn that has only child element Group is included into the merged Employee element in \( F_{LOJ} \) as a child element.

The child Position of Person element in \( F_2 \) does not have any semantically corresponding or identical element that is a descendant or an ancestor of Employee element in \( F_1 \). It is contained in the merged Employee element in \( F_{LOJ} \).

The second pair of matching Employee element in \( F_1 \) and Person element in \( F_2 \) are merged into the Employee element in \( F_{LOJ} \) that has “Shirley Green” as the attribute PName and “Sales” as the attribute DName of the ancestor Department. This merged Employee element in \( F_{LOJ} \) indicates a conflict between the child Age of Employee element in \( F_1 \) and the child Age of Person element in \( F_2 \): the child Age of the merged Employee element in \( F_{LOJ} \) has content “31|35”. Person element has a child Phone, <Phone>2222</Phone>, which does not have a corresponding child Phone of the child Contact of Employee element in \( F_1 \). So, <Phone>2222</Phone> is incorporated into the child Contact of the merged Employee element in \( F_{LOJ} \).

- Modification of the Non-Matching Employee and Non-Matching Person

\( F_1 \) contains only one non-matching Employee element: the Employee element that has “Paul Smith” as the child Name and “Sales” as the attribute DName of the ancestor Department. This non-matching Employee element is modified and incorporated into \( F_{LOJ} \). Its child Name is changed to attribute PName in order to obey the structure of the merged Employee element in \( F_{LOJ} \).

\( F_2 \) also contains only one non-matching Person: the Person element in \( F_2 \) that has “Alice Bush” as the attribute PName and “Production” as the descendant Unit. Because this non-matching Person element in \( F_2 \) and the Employees element in \( F_{LOJ} \) that has the ancestor Department that has the attribute DName with value “Production” make Boolean expression \( E_2 \) true, this non-matching Person is modified and
embodied into $F_3$ as a child element of this $Employees$ element. Note that each $Employees$ element and its ancestors in $F_{LOJ}$ are incorporated into $F_3$.

This non-matching $Person$ element in $F_2$ is modified to obey the structure of the merged $Employee$ element in $F_{LOJ}$. The element name of this non-matching $Person$ element is changed to that of the merged $Employee$ element in $F_{LOJ}$. The child $WorkIn$ of this non-matching $Person$ element contains $Factory$, $Unit$, and $Group$ child elements. As the child $Factory$ and the ancestor $Factory$ of $Employee$ element in $F_1$ are semantically corresponding elements and the child $Unit$ and the ancestor $Department$ of $Employee$ in $F_1$ are semantically corresponding elements, $WorkIn$ that has only child element $Group$ is incorporated into the $Employee$ element in $F_3$ that has the attribute $PName$ with value “Alice Bush”. The remaining of the child elements of this non-matching $Person$ element are incorporated into $F_3$.

- The Generated XML Documents $F_{LOJ}$ and $F_3$

The generated XML document $F_{LOJ}$ shown in Figure 5.9 contains every $Employee$ element merged from $Employee$ in $F_1$ and its matching $Person$ in $F_2$, every modified non-matching $Employee$ in $F_1$, and the elements in $F_1$ that do not need merging. For instance, $Factory$, $Department$, and $Employees$ elements in $F_1$ do not need merging and they are incorporated in $F_{LOJ}$.

The resulting XML document $F_3$ is composed of every $Employee$ element merged from $Employee$ in $F_1$ and its matching $Person$ in $F_2$, each modified non-matching $Employee$ in $F_1$, some modified non-matching $Person$ elements in $F_2$ (in Example 6, there is only one non-matching $Person$ element in $F_2$ and it is incorporated into $F_3$), and the elements in $F_1$ that do not need merging.
Chapter 6

Conclusion and Future Work

6.1 Conclusion

The contributions of this thesis are as follows:

- We define a merging operation over XML documents that is similar to a full outer join in relational algebra. It can merge two XML documents that have different structures.

- We design an algorithm and a set of procedures for this merging operation.

- We present a mechanism that combines Skolem function and Boolean functions defined for designers to identify XML instances when merging.

- We explore the semantic associations between two XML documents to be merged and introduce *semantically identical attributes and elements*, *semantically corresponding elements*, and *semantically corresponding attribute and element*.

- We propose a method for merging XML elements and handling typical conflicts.

- We present a merge template XML file that can effectively specify how to combine two input XML documents into one result XML document, build semantic connections between these two documents, and support recursive processing and merging.
of XML elements.

A paper from this thesis has been accepted for presentation and publication by an international conference: the 23rd International Conference on Conceptual Modeling (ER2004) [50].

Our approach differs from state-of-art XML documents merging systems in several aspects. First, the merging operation defined can merge two XML documents that have different structures. Second, the mechanism proposed combines Skolem function and Boolean functions defined for designers to identify XML instances when merging. Third, the method for merging XML elements and handling typical conflicts is proposed in this thesis. Finally, the merge template XML file proposed can specify how to combine two input XML documents into one result XML document, build semantic connections between these two documents, and support recursive processing and merging of XML elements.

6.2 Future Work

In this thesis, we do not discuss XML elements that have mixed contents. We intend to extend our approach to merge XML documents that contain elements that have mixed contents.

The prototype we implemented cannot merge XML documents over the network. We plan to extend this prototype to support merging of XML documents over the network.

The algorithm for the merging operation can only generate $F_3$ from $F_1$, $F_2$, $D_1$, $D_2$, $P_1$, $P_2$, $E_1$, and $E_2$. We plan to design an algorithm to produce both $F_3$ and its DTD $D_3$. $D_3$ should be very similar to $D_1$ except that the element declaration for the element merged from $e_1$ and its matching $e_2$, non-matching $e_1$, and non-matching $e_2$ in $F_3$ is different from the element declaration for $e_1$ in $F_1$.  

82
Our approach has some limitations. First, it may be difficult to set up Boolean expressions $E_1$ and $E_2$ when the set of XML documents for merging is large. Second, our approach is based on the assumptions about two XML documents to be merged, but these assumptions do not apply if two XML documents have very different tag names. We aim to investigate how to solve these problems.

In addition, we intend to explore other possible operations over XML documents, such as intersection, difference, etc. which will be studied in future work.
Bibliography


89