University of Windsor

Scholarship at UWindsor

Electronic Theses and Dissertations | Theses, Dissertations, and Major Papers

1-1-2006

XML Schema subtyping.

Yun Li
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation

Li, Yun, "XML Schema subtyping." (2006). *Electronic Theses and Dissertations*. 7136.
https://scholar.uwindsor.ca/etd/7136

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

XML Schema Subtyping

by

Yun Li

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2006

© 2006 Yun Li

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Library and
Archives Canada

Published Heritage
Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et
Archives Canada

Direction du
Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-42327-1
Our file Notre référence
ISBN: 978-0-494-42327-1

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

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

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

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.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ABSTRACT

An XML Schema is a grammar of an XML language. It defines a set of instance XML documents that are valid sentences in this language. An XML Schema S is a subtype of another XML Schema T if the set of instances of S is a subset of the instances of T. Since XML Schema has become a mainstream data type definition format for XML documents, its subtyping problem finds many applications in XML-centric programming and Web-service technology.

The proposed subtyping algorithm is based on Antimirov's derivation calculus (Antimirov, 1994) for regular expressions and its extensions to regular hedge expressions (Kempa & Linnemann, 2003) (Hohenadel, 2003). This thesis formalizes and rebuilds the algorithm for regular tree grammars, which is very close to the subtyping algorithm for regular hedge grammars.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACKNOWLEDGEMENTS

First of all, I would like to express my gratitude to my supervisor, Dr. Jianguo Lu, who brought me to the research area of formal language theory and XML-related technologies. He taught me the way of doing research as a serious, independent, and creative researcher. His insightful comments and encouragements always inspired me to do my best in this thesis.

I also want to thank my thesis committee members, Dr. Sang-Chul Suh, Dr. Jessica Chen, and Dr. Richard Frost, for their great support.

Finally, I am very grateful for my family whose love and patience enabled me to complete this work.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGEMENTS iv

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER

I. INTRODUCTION

1.1 Type Languages for XML Documents 1
1.2 XML Schema Subtyping 2
1.3 Thesis Overview 5

II. A FORMAL LANGUAGE FOR XML SCHEMAS

2.1 Regular Grammars and Context Free Grammars 7
2.2 Trees 10
2.3 Regular Tree Grammars and Regular Tree Languages 12
2.4 Unranked & Ranked Trees 15
2.5 Intermediate Language 17
2.6 XML Schema Mapping 22
2.7 Schema Determinism 25
2.8 Subtyping 29

III. A SUBTYPING ALGORITHM FOR XML SCHEMAS

3.1 What is a Subtyping Algorithm? 31
3.2 Main Ideas of the Subtyping Algorithm 31
3.3 Check for ε-inclusion 34
3.4 Simplification Process 36
3.5 Algorithm Rules 47
3.6 Time Complexity 52

IV. IMPLEMENTATION

4.1 XML Schema Parsing 56
4.2 Type Representations 57
4.3 Subtype Checking 59
4.4 Implicit Subtyping 64

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

V. RELATED WORK

5.1 Tree-automaton-based Subtyping 66
5.2 Subtyping for Regular Expression Types 68
5.3 Subtyping Using BDD 71
5.4 Subtyping Using Antimirov's Containment Calculus 72
5.5 Comparison 75

VI. CONCLUSIONS AND FUTURE WORK

6.1 Main Contributions 80
6.2 Limitations and Future Work 81

APPENDICES

A A Set-theoretic Observation 83
B Logfile for Example 3 86

REFERENCES 93

VITA AUCTORIS 96

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF TABLES

Table 1: Eight Type Classes in Implementation 57

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF FIGURES

Figure 1: XML Schema Supervisor1....4

Figure 2: XML Schema Supervisor2....4

Figure 3: An Instance XML Document of Supervisor1 and Its Tree Representation12

Figure 4: The Relationship Between Unranked Trees and Binary Trees....16

Figure 5: XML Schema Test....27

Figure 6: Main Ideas of the Subtyping Algorithm....32

Figure 7: Proof Tree of Example 3....46

Figure 8: XML Schema Subtyping Rules....47

Figure 9: XML Schema Subtyping Architecture....55

Figure 10: DOM XML Parser....56

Figure 11: XML Schema Built-in Data Type Hierarchy....65

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER I

INTRODUCTION

Since the World Wide Web Consortium (W3C) recommended *XML* (the e**X**tensible **M**arkup **L**anguage) as a standard in 1998, XML has received widespread attention and adoption in the computer industry. Its usage ranges from document publication to data exchange and integration on the Web. In addition, XML is the cornerstone of Web service technology, which is becoming the new standard for distributed computing. Modern Web applications and Web services generate XML documents dynamically, whose types need to be checked. In particular, applications consuming one type of XML documents may accept documents of its subtypes. Hence, the subtyping problem of various XML data types has attracted substantial research attention (Hosoya, Vouillon, & Pierce, 2000) (Tozawa & Hagiya, 2003) (Kempa & Linnemann, 2003) (Hohenadel, 2003).

1.1 Type Languages for XML Documents

An XML type describes a set of XML documents, typically expressed in terms of constraints on the structure and content of the documents of that type. To improve the safety of XML data processing, most XML-related technologies assume XML documents follow an XML type.

Since the inception of the XML specification, there have been many XML type languages defined. The original specification of XML (Bray et al., 2000) defined *DTD* (**D**ocument **T**ype **D**efinitions) as its type language, or the schema language. Since then, DTD had

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

been the most commonly used type language until the introduction of XML Schema. DTD is relatively simple and has a compact syntax. However, DTD has a few shortcomings. For example, DTD has a non-XML syntax, which means one needs separate tools, such as a parser, to let a machine understand it. In addition, DTD doesn't support namespaces easily, and provides very limited data typing. For example, it doesn't have types like *date* and *integer*. Furthermore, data types defined in DTD are for attributes only.

To overcome the limitations of DTD, a large number of alternatives have been proposed. The representative ones include *XML Schema* (Fallside & Walmsley, 2004), *DSD* (**D**ocument **S**tructure **D**escription) (Klarlund et al., 2000), *RELAX* (Murata, 2001), *RELAX NG* (Clark & Murata, 2001), and so on. Among them, XML Schema is the most popular one and has become a mainstream data-type definition format for XML documents. We investigated 3448 *WSDL* (**W**eb **S**ervice **D**efinition **L**anguage) files randomly collected from the Web and found that 3070 WSDL files contain type definitions, 3054 out of which (99.48%) use XML Schema for their datatype definitions.

Because of the popularity of XML Schema, this thesis focuses on the subtyping problem of XML Schema only.

1.2 XML Schema Subtyping

There are various notions to capture the relationship between XML data types, such as the *subsumption* relation proposed by Kuper, et al. in (Kuper & Siméon, 2001) and the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

containment relation between two languages (Tozawa & Hagiya, 2003). In this thesis, subtype refers to the set-inclusion relationship (Hosoya, Vouillon, & Pierce, 2000) (Kempa & Linnemann, 2003) (Hohenadel, 2003), i.e., XML Schema S is a subtype of another XML Schema T if the set of XML documents defined by S is a subset of the set of XML documents defined by T.

To illustrate the subtyping problem, let's consider two XML Schemas: Supervisor1 and Supervisor2, in Figure 1 and 2.

In Supervisor1, the *minOccurs* and *maxOccurs* attributes of the node labeled *supervisor* are set to 0 and 1 respectively, which means the node *supervisor* can occur 0 or 1 time within the node *supervisor*. However, Supervisor2 specifies that the node *supervisor* can occur 0, 1, or 2 times within the node *supervisor*. Obviously, Supervisor2 describes strictly more XML instances, which implies the set of instances of Supervisor1 is a subset of the set of instances of Supervisor2. Hence, Supervisor1 is a subtype of Supervisor2.

Subtyping yields a substantial degree of flexibility in XML-centric programming. An XML language supporting subtyping will allow procedures/methods applicable to one type to be safely applied to its subtypes. For example, applications that are designed to process the instance XML documents of Supervisor2 will also be able to process those of Supervisor1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
  <xs:element name = "supervisor" type = "supervisorType">
    <xs:complexType name = "supervisorType">
      <xs:sequence>
        <xs:element name = "name" type = "xs:string" />
        <xs:element name = "position" type = "xs:string" />
        <xs:element ref = "supervisor" minOccurs="0" maxOccurs="1" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 1: XML Schema Supervisor1

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
  <xs:element name = "supervisor" type = "supervisorType">
    <xs:complexType name = "supervisorType">
      <xs:sequence>
        <xs:element name = "name" type = "xs:string" />
        <xs:element name = "position" type = "xs:string" />
        <xs:element ref = "supervisor" minOccurs="0" maxOccurs="2" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 2: XML Schema Supervisor2

Subtyping becomes more important with the widespread acceptance of Web service technology. A Web service is defined in terms of its types, i.e. XML Schema. So, when searching for Web services or composing Web services, we need to compare Web services in terms of XML Schemas, i.e., subtyping of XML Schemas. For example, the subtyping facility can help us locate relevant sub-services from the Web.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

1.3 Thesis Overview

The contributions of this thesis are, in short, as follows: (1) based on formal language theory, we identify an appropriate language to model XML Schemas, and formally define the language based on regular tree grammars. (2) We propose a subtyping algorithm, which is based on Antimirov's derivation calculus (Antimirov, 1994) for regular expressions and its extensions to regular hedge expressions (Kempa & Linnemann, 2003) (Hohenadel, 2003). This thesis formalizes and rebuilds the algorithm for regular tree grammars, which is very close to the subtyping algorithm for regular hedge grammars. (3) We implement the subtyping algorithm for XML Schemas.

The remainder of this thesis is organized as follows. In the next chapter, we will give the definitions of those basic notations that we use in our work, e.g., trees, regular tree grammars, and subtyping. Also, we will describe how to model XML Schemas by regular tree grammars. Chapter 3 will present the main ideas of our subtyping calculus, which was originally conceived by Antimirov (Antimirov, 1994). This chapter contains a detailed description of all extensions and modifications added to the original calculus. A detailed description of the implementation of the subtyping algorithm will be given in Chapter 4. In Chapter 5, we will discuss related work and indicate what current technical challenges are in this field. Importantly, we will compare our work with other related work at length. Chapter 6 will conclude the main contributions of our research work and then discuss the limitations of our subtyping system. Finally, we will address some

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

aspects, which at this time cannot be discussed within the context of this thesis, or which will become subject to optimization of the subtyping algorithm in the future works.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER II

A FORMAL LANGUAGE FOR XML SCHEMAS

To study the relationships between XML Schemas, first of all, we need to model XML Schemas using a formal language. It is crucial to provide a formal model for XML Schemas in order to facilitate efficient implementations of subtyping algorithms. Towards this goal, in this thesis, we propose to use formal language theory, especially regular tree grammar theory, as such a framework for XML Schemas.

To understand why we need to use regular tree grammars to model XML Schemas, we first introduce two grammars that are closely related to regular tree grammars.

2.1 Regular Grammars and Context Free Grammars

Regular Grammars and context free grammars are the type 3 and type 2 grammars of Chomsky Hierarchy (Hopcroft & Ullman, 1979), respectively.

According to (Hopcroft & Ullman, 1979), all productions of a regular grammar are of the following forms:

$n \rightarrow t$

or $n \rightarrow tn_1$ or $n \rightarrow n_1t$ but not both

or $n \rightarrow \varepsilon$

where $n, n_1 \in N$ (denoting a non-terminal set), and t is a string of terminals $\in \Sigma$ (denoting

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

a terminal set).

It is impossible to use productions of the above forms to derive recursive trees when the recursion occurs in the content model (i.e. the order and structure of the children of a tree node). For example, XML Schema Supervisor1 defines a set of trees derived from the following productions

Supervisor → *supervisor*(Name, Position, Supervisor)

Supervisor → *supervisor*(Name, Position)

If we consider parentheses as terminal symbols, there are terminal symbols (in lower-case) before and after the non-terminal symbol *Supervisor* on the right-hand side of the first production. Such a production can not be replaced by a production of form either $n \rightarrow tn_1$ or $n \rightarrow n_1t$ as defined in regular grammars. Therefore, the expressiveness of XML Schemas is beyond that of regular grammars.

Regular expressions correspond to regular grammars. They are just different ways to express the same thing, except that regular expressions are more concise. Like regular grammars, regular expressions are not expressive enough to model XML Schemas, either.

A context-free grammar (Hopcroft & Ullman, 1979) is more expressive than regular grammars or regular expressions. It allows a sequence of terminals or non-terminals on the right-hand side of a production rule. Since XML became the standard data format for

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the Web in 1998, context-free grammars have been increasingly important for XML and XML type languages such as DTD and XML Schema. Many early proposed type formalisms for XML data types (Hosoya, Vouillon, & Pierce, 2000) (Kempa & Linnemann, 2003) were based on context-free grammars.

Although context free grammars are expressive enough to model XML Schemas, the decision problem for the inclusion-checking between context-free languages is undecidable (Hopcroft & Ullman, 1979, Theorem 8.12). Syntactic restrictions have to be imposed to reduce the power of context free grammars so that the types represented by such grammars correspond to regular tree languages. These restrictions require, for any production of a context free grammar, a recursive non-terminal to occur only in the tail position and to be preceded by a *non-nullable* type expression on the right-hand side (Hosoya, Vouillon, & Pierce, 2000) (Kempa & Linnemann, 2003) (Hohenadel, 2003). A type expression is non-nullable if the language denoted by this type expression does not contain the empty string. Thus, these restrictions ensure the regularity.

To guarantee enough expressiveness and avoid the above syntactic restrictions, regular tree grammars are commonly used to model XML Schemas (Murata, Lee, Mani, & Kawaguchi, 2005). This thesis follows this approach and uses regular tree grammars in the subtyping algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2 Trees

As an XML Schema describes a set of XML documents and an XML document can be viewed as a tree, we first define what trees are in our work.

Following the definitions from (Comon et al., 2002), a ranked alphabet Σ is a finite nonempty set of symbols, each symbol of which has a unique nonnegative arity (or rank), denoting the number of its children. The ranked alphabet Σ is partitioned into disjoint sets, i.e., $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \ldots \cup \Sigma_k$ where 0, 1,..., k are nonnegative integers and Σ_m denotes the set of symbols of arity m. Elements of Σ_k are called k-ary symbols. In particular, elements in Σ_0 are constants.

A leaf alphabet X is an ordinary finite alphabet. It is disjoint from the ranked alphabet Σ considered in a given context.

<u>**Definition 1**</u>: **Terms and Trees** (Gècseg & Steinby, 1997)

Let Σ denote a finite set of operation symbols and X as a set of variables. The set $T_\Sigma(X)$ of Σ–terms with variables in X, is defined inductively as the smallest set T of strings such that:

(1) $X \subseteq T$, and

(2) $l\,(t_1, \ldots, t_m) \in T$ whenever $m \geq 0$, $l \in \Sigma_m$ and $t_1, \ldots, t_m \in T$.

Term $c(\,)$ is simplified as c when $c \in \Sigma_0$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A *ΣX–tree* is an Σ–term with variables in X. Thus, the set $T_\Sigma(X)$ is the set of all ΣX–trees.

Many XML documents can be represented by ΣX–trees. In this view, the root and inner nodes (labelled with a symbol from a ranked alphabet Σ) of a ΣX–tree correspond to elements which determine the structure of an XML document, while the leaf nodes (labelled with a symbol from a leaf alphabet X) provide data contents. For example, in Figure 3, XML document A is an instance of XML Schema Supervisor1 and can be represented by the ΣX–tree below it, where Σ = {*supervisor*, *name*, *position*} and X = {*Mary*, *secretary*, *John*, *manager*, *Zackery*, *director*}.

In the domain of subtype-checking, we are only interested in the structure of a set of XML documents, rather than in actual data values. In such cases, we ignore the leaf alphabet X, and thus, an XML document can be adequately represented by a tree over a finite alphabet Σ. Such finite labelled ordered trees are called *Σ–trees* (Neven, 2002).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

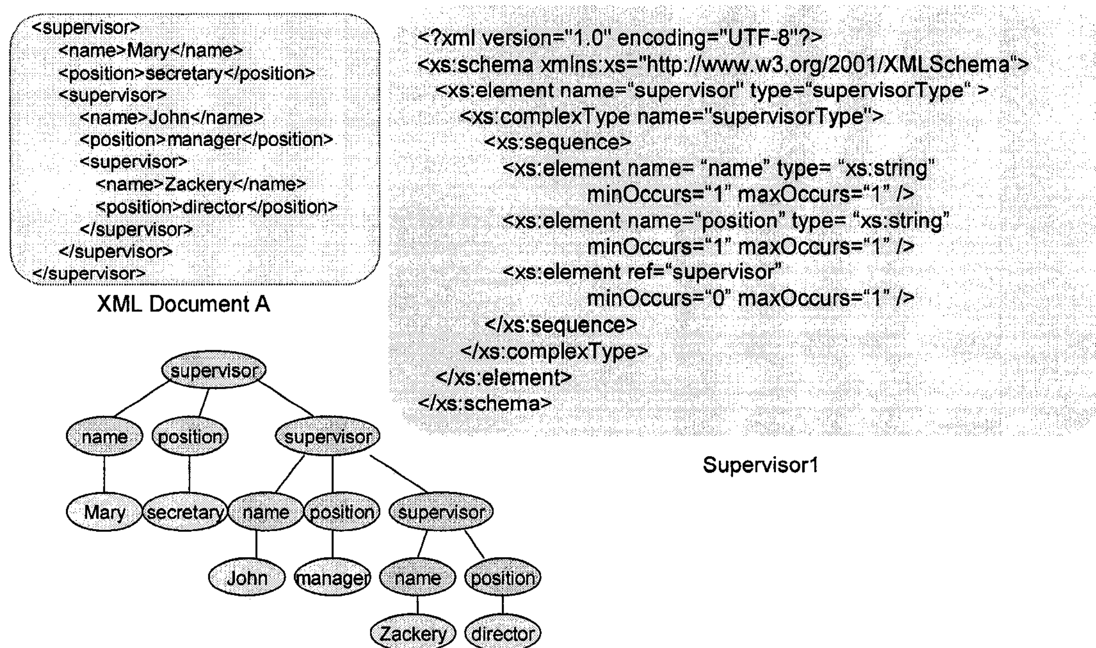

Figure 3: An instance XML document of Supervisor1 and its tree representation

2.3 Regular Tree Grammars and Regular Tree Languages

An XML Schema defines a set of Σ–trees. The formal language for XML Schema should be defined over Σ–trees. Tree grammars generate such trees and thus they are appropriate to model XML schemas. In our work, we capture XML Schemas by a class of tree grammars called *regular tree grammars*.

The formal definitions (shown below) of regular tree grammars and regular tree languages (below) are given in (Comon et al., 1999) and (Gècseg & Steinby, 1997). Please note that we do not consider the set of variables X in this thesis.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Definition 2: Regular Tree Grammar

A regular tree grammar is defined by a system $G = (\Sigma, N, P, S)$ where

- Σ is a ranked alphabet or a set of terminal symbols;
- N is a finite nonempty set of non-terminal symbols and $N \cap \Sigma = \emptyset$ is assumed;
- P is a finite set of productions of form $n \rightarrow t$, where $n \in N$ and $t \in T_\Sigma(N)$;
- S is the start symbol and $S \in N$.

Regular tree grammars have two main differences from other classes of tree grammars:

- In a regular tree grammar, all non-terminal symbols have arity 0, while other tree grammars allow non-terminals of arity greater than 0;
- A tree grammar has a set of production rules of form $t_1 \rightarrow t_2$ where t_1, t_2 are trees defined over a terminal set Σ and a non-terminal set N. Additionally, t_1 contains at least one non-terminal. On the other hand, in any production of a regular tree grammar, only a single non-terminal is allowed on its left-hand-side. That is, the form of productions is $n \rightarrow t$ where $n \in N$ and t is a tree over $\Sigma \cup N$.

A regular tree grammar is used to derive trees from the start symbol S, using the corresponding *derivation relations* which can be defined simply by interpreting the productions of a regular tree grammar as the rewrite rules of a term rewriting system. That is, we replace a non-terminal A by the right-hand-side a of a rule $A \rightarrow a$. We use the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

notation $\Rightarrow_G$ to denote the one-step derivation relation of a regular tree grammar G and the notation $\Rightarrow_G^*$ to denote the general derivability relation of G.

A regular tree language, denoted by $L(G)$, is the language generated by a regular tree grammar G. It is a set T_Σ of Σ–trees defined as:

$$L(G) = \{t \in T_\Sigma \mid s \Rightarrow_G^* t\}$$

Example 1: (regular tree grammar *G*1) A regular tree grammar that represents XML Schema Supervisor1 can be defined as $G_1 = (N, \Sigma, P, S)$, where

N = {Supervisor}

Σ = {*supervisor*, *name*, *position*, *string*}

S = Supervisor

P = { Supervisor → *supervisor*(*name*(*string*), *position*(*string*), Supervisor)

Supervisor → *supervisor*(*name*(*string*), *position*(*string*)) }

The regular tree language generated by G_1, denoted by $L(G1)$, is a set of trees, i.e.,

$L(G_1)$ = { *supervisor*(*name*(*string*), *position*(*string*)),

supervisor(*name*(*string*), *position*(*string*),

supervisor(*name*(*string*), *position*(*string*)))

……

}

The derivations of the two simplest trees in $L(G_1)$ are as follows:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$\text{Supervisor} \Rightarrow_G \textit{supervisor}(\textit{name}(\textit{string}), \textit{position}(\textit{string}))$$

$$\begin{aligned}\text{Supervisor} &\Rightarrow_G \textit{supervisor}(\textit{name}(\textit{string}), \textit{position}(\textit{string}), \text{Supervisor}) \\ &\Rightarrow_G \textit{supervisor}(\textit{name}(\textit{string}), \textit{position}(\textit{string}), \\ &\qquad \textit{supervisor}(\textit{name}(\textit{string}), \textit{position}(\textit{string})))\end{aligned}$$

A regular tree grammar $G = (N, \Sigma, P, S)$ is in *normal form* (Gècseg & Steinby, 1997) or called *normalized* (Comon et al., 1999) if each production of G is of form $n \rightarrow c$ or $n \rightarrow l(n_1,\ldots, n_m)$ where $n, n_1,\ldots, n_m \in N$, $c \in \Sigma_0$, $l \in \Sigma_m$, and $m > 0$. According to (Comon et al., 1999, Proposition 3, p51), any regular tree grammar can be transformed into a normalized regular tree grammar. In the rest of the thesis, wherever we say regular tree grammars, we mean normalized regular tree grammars.

2.4 Unranked & Ranked Trees

An *unranked tree* is an Σ–tree where nodes can have an arbitrary number of children. In other words, there is no fixed rank (or arity) associated with a label of an unranked tree. It is allowable for an XML Schema to define unranked trees. For example, in XML Schema Supervisor1, the *supervisor* node of the tree (as shown in Figure 3) has either two or three children. Hence, that tree is unranked.

In Definition 2, a regular tree grammar $G = (N, \Sigma, P, S)$ is defined over a ranked alphabet Σ. That is, the right-hand side of a production has either a constant (i.e. a terminal

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

without children) or a terminal $\in \Sigma_m (m > 0)$ followed by m number of children. Hence, a question comes up when we try to model XML Schemas that define sets of unranked trees, i.e., whether regular tree grammars are able to define unranked trees?

Neven's work (Neven, 2002) on the relationship between unranked tree automata and ranked tree automata answers the question positively. In his paper, Neven claims that any unranked tree can be encoded into a binary tree where all non-leaf nodes have exactly two children in several ways. In Figure 4, we illustrate one such possibility.

unranked tree (a) binary tree (b)

Figure 4: The relationship between unranked trees and binary trees

The unranked tree (a) in Figure 4 is a short form of the tree in Figure 3 after deleting the data values. It can be converted into the binary tree (b) in Figure 4 by using some

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

encoding technique. Specifically, in the encoding, the leftmost child of a node remains the first child (i.e. left child) of this node and the other children of this node are encoded into the right descendants of the left child. Whenever a node doesn't have the left or right child, the symbol # is inserted as a placeholder.

After such encoding, the unranked tree (a) is converted into a ranked tree where all non-leaf nodes have a fixed arity of 2. Then we can use a regular tree grammar to define such binary trees.

2.5 Intermediate Language

Using binary tree encoding, we see that unranked trees can be defined by regular tree grammars. However, the definition of regular tree grammar (i.e. Definition 2) should be slightly modified in order to deal with unrankedness.

Definition 3: **Intermediate Representation** (Lee, Mani, & Murata, 2000)

- A regular tree grammar is defined by a system $G = (\Sigma, N_1, N_2, P_1, P_2, S)$ where Σ, N_1, and N_2 are pairwise disjoint, and
- Σ is a set of terminal symbols;
- N_1 is a finite nonempty set of non-terminal symbols used for deriving trees;
- N_2 is a finite set of non-terminal symbols used for specifying content models (i.e. the orders and structures of the children of tree nodes);
- P_1 is a finite nonempty set of production rules of form $n \rightarrow l(A)$, where $n \in N_1$, $l \in$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Σ, and $A \in N_2$;

- P_2 is a finite set of production rules of form $A \rightarrow r$, where $A \in N_2$ and r is a regular expression over N_1;
- S is the start symbol and $S \in N_1$.

The right-hand side of a production rule in a regular tree grammar conforming to Definition 3 is referred to as a *type expression* in this thesis.

Please note that regular expressions are introduced in the above definition of regular tree grammar to specify the orders and structures of the children of tree nodes. By using regular expression operators such as the Kleene star (*), concatenation (·), and alternation (|), unrankedness is introduced into the definition of regular tree grammars (see Definition 3). Thus, an XML Schema, which defines a set of trees (either ranked or unranked), can be represented by a regular tree grammar as defined in Definition 3. The definition of regular expressions is given in Definition 4.

Definition 4: Regular Expressions (Hopcroft & Ullman, 1979)

Let N be an alphabet. The regular expressions over N are defined recursively as follows:

- $\emptyset$ is a regular expression and denotes the empty set;
- ε (i.e. the empty string) is a regular expression and denotes the set $\{\varepsilon\}$;
- n ($\in N$) is a regular expression and denotes the set $\{n\}$;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- If r and s are regular expressions specifying the languages L(r) and L(s), then **r·s** is a regular expression and denotes the set $\{\alpha \cdot \beta \mid \alpha \text{ is in L(r) and } \beta \text{ is in L(s)}\}$;
- If r and s are regular expressions specifying the languages L(r) and L(s), then **r | s** is a regular expression and denotes the set L(r) ∪ L(s);
- If r is a regular expression specifying the language L(r), then the Kleene star --- **r*** is a regular expression and denotes the smallest superset of L(r) that contains ε and is closed under string concatenation, i.e., the set of all strings that can be made by concatenating zero or more strings in L(r).

To illustrate the mapping between XML Schemas and regular tree grammars, we give another example below.

Example 2: (regular tree grammar *G2*) According to Definition 3, a regular tree grammar representing XML Schema Supervisor1 can be defined as $G_2 = (\Sigma, N_1, N_2, P_1, P_2, S)$, where

Σ = {*supervisor*, *name*, *position*, *string*}

N_1 = {Supervisor, Name, Position, String}

N_2 = {SupervisorType, NameType, PositionType}

S = Supervisor

P_1 = { Supervisor → *supervisor*(SupervisorType)

Name → *name*(NameType)

Position → *position*(PositionType)

String → *string* }

P_2 = { SupervisorType → Name · Position · (Supervisor | ε)

NameType → String

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

PositionType → String }

Please note that grammar G_2 can be rewritten into a regular tree grammar whose production rules have the form as in Definition 2. In other words, the forms $n \rightarrow l(A)$ and $A \rightarrow r$ as defined in Definition 3 can be just viewed as the short forms of $n \rightarrow l(n_1,\ldots, n_m)$ as defined in Definition 2.

Theorem 1: A grammar *G*' in Definition 3 can be transformed into an equivalent grammar *G* in Definition 2.

Proof: The transformation is defined recursively on the structure of the definition of grammars in Definition 3.

1) The terminal set Σ in *G* is mapped to the set Σ in *G*' and the arity of any terminals in Σ in *G* that is greater than 1 is set to 1 in Σ in *G*' (i.e., the arity of *supervisor* is changed to 1);
2) All non-terminals in the set *N* in *G* are included in the set N_1 in *G*'. In addition, new non-terminals (i.e., Name, Position and String) are added in N_1 to facilitate deriving trees from inner and leaf nodes;
3) New non-terminals ending with "Type" (i.e., SupervisorType, NameType, and PositionType) are added in the new set N_2 in *G*'. These non-terminals specify the content models of tree nodes labelled with *supervisor*, *name*, and *position*, respectively;
4) The start symbol *S* in *G* is mapped to the start symbol *S* in *G*';

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5) The productions of form $n \rightarrow l(n_1,\ldots, n_m)$ (where $n, n_1,\ldots, n_m \in N$, $l \in \Sigma_m$, and m ≥ 0) of P in G are transformed into productions of form $n \rightarrow l(A)$ and $A \rightarrow r$ (where $n \in N_1$, $l \in \Sigma$, and $A \in N_2$). The new set P_1 in G' contains all productions of form $n \rightarrow l(A)$ and the new set P_2 in G' contains all productions of form $A \rightarrow r$.

The form $n \rightarrow l(A)$ in G' conforms to the form $n \rightarrow l(n_1,\ldots, n_m)$ in G. We prove (see below) that $A \rightarrow r$ in G' can be transformed into the form $n \rightarrow l(n_1,\ldots, n_m)$ in G by using the definition of regular expressions (see Definition 4).

- If r is the empty set $\emptyset$, the empty string ε, or any symbol n in N_1, the form $A \rightarrow r$ obviously consistent with the form in G.
- Assume $n_1 \rightarrow a(A_1)$ and $n_2 \rightarrow b(A_2)$, where $A_1, A_2 \in N_2$, $n_1, n_2 \in N_1$, $a, b \in \Sigma$.

  If r is the concatenation of $n_1 \cdot n_2$, using binary tree encoding described in (Neven, 2002), we get

$$
\begin{aligned}
&A \rightarrow n_1 \cdot n_2 \\
&n_1 \rightarrow a(A_1) \\
&n_2 \rightarrow b(A_2) \\
\Rightarrow \quad & \\
&A \rightarrow a(A_1, b(A_2, \#)) \qquad (2\text{-}1)
\end{aligned}
$$

  If r is the alternation of $n_1|n_2$, using binary tree encoding described in (Neven, 2002), we get

$$
\begin{aligned}
&A \rightarrow n_1 \mid n_2 \\
&n_1 \rightarrow a(A_1) \\
&n_2 \rightarrow b(A_2) \\
\Rightarrow \quad & \\
&A \rightarrow a(A_1) \qquad (2\text{-}2)
\end{aligned}
$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$A \rightarrow b(A_2) \qquad (2\text{-}3)$$

If r is n_1*, using binary tree encoding described in (Neven, 2002), we get

$$A \rightarrow n_1^*$$
$$n_1 \rightarrow a(A_1)$$

$\Rightarrow$

$$A \rightarrow \varepsilon \qquad (2\text{-}4)$$
$$A \rightarrow a(A_1, \#) \qquad (2\text{-}5)$$
$$A \rightarrow a(A_1, a(A_1, A)) \qquad (2\text{-}6)$$

Form (2-1)-(2-6) are consistent with the form $n \rightarrow l(n_1,\ldots, n_m)$ in G.

Therefore, a grammar G' in Definition 3 can be transformed into a grammar G in Definition 2, and G' is equivalent to G. Q.E.D.

2.6 XML Schema Mapping

To facilitate XML Schema mapping, we partition the terminal set Σ into two disjoint sets Σ_0 and Σ_m (m > 0), i.e. $\Sigma = \Sigma_0 \cup \Sigma_m$. The set Σ_m corresponds to the set of element names (or tag names) and the constants in Σ_0 correspond to the built-in simple types defined in the XML Schema Recommendation (Fallside & Walmsley, 2004). The set N_1 contains those non-terminals that we add in production rules. The non-terminal set N_2 corresponds to the set of type names. When anonymous types are encountered in an XML Schema, we introduce new non-terminals (in the set N_2) and new production rules to facilitate XML Schema mapping based on our modeling language. In this thesis, we assume that every XML Schema in our problem domain always has a root element S, whose production is of form $n \rightarrow l(A)$. An element declaration defines a production rule of form $n \rightarrow l(A)$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

where $n \in N_1$, $l \in \Sigma$ corresponds to the assigned element name, and $A \in N_2$ corresponds to the assigned type name for this element. A complex or simple type-definition defines a production rule without terminals, i.e., $A \rightarrow r$. The regular expression on the right-hand side of such a production rule ($\in P_2$) corresponds to the content model of a type.

In addition, we also extend the notion e* in the definition of regular expressions to e{n, m}, in order to denote an occurrence of at least n and at most m times of iteration of a regular expression e connected by concatenation, where n is a non-negative integer and m is either a non-negative integer (n ≤ m) or the string of value *undefined*. The values of n and m directly correspond to the values of the attributes *minOccurs* and *maxOccurs* in an element declaration, respectively. The Kleene star (*) and other commonly-used operators for regular expressions are redefined (Hohenadel, 2003) as follows:

$$e^* = e\{0, \text{undefined}\}$$

$$e+ = e\{1, \text{undefined}\}$$

$$e? = e\{0, 1\}$$

Like other research work on schema subtyping (Kempa & Linnemann, 2003) (Hohenadel, 2003), the intermediate language (see Definition 3) can not model all the features of XML Schemas. We explain that in the following.

1. **Attributes**. We do not consider attributes in this thesis. How to compare two types with different attribute types? One possible method proposed by (Kempa &

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Linnemann, 2003) to solve this problem may be to consider an attribute as a special "child" of the corresponding node.

2. **Namespaces**. Namespaces are an important issue in the context of XML Schema. However, in order to simplify XML Schema mapping, this thesis ignores namespaces.

3. **Content Types**. There are four types of contents for complex types: *simple*, *element*, *mixed*, and *empty*. An element that has *simple* content contains only character data and attributes. An element that has *element* content contains child elements, but no character data content. If an element has both child elements and character data content, it has *mixed* content. If an element does not have any content (just attributes), it has *empty* content. In this thesis, we only consider elements with *simple* and *element* contents.

4. **Model groups**. In XML Schema, content models are defined using a combination of model groups, element declarations or references, and wildcards. There are three kinds of model groups: *sequence*, *choice*, and *all*. The *sequence* model group requires that the child elements appear in the order specified. The *choice* model group allows any one of child elements to appear. The *all* model group requires that all the child elements appear 0 or 1 times, in any order. These groups can be nested, and may occur multiple times, allowing you to create sophisticated content models. For now, we only consider two kinds of model groups: *sequence* and *choice*.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5. **Include and import**. An XML Schema can be composed of one or more other XML Schemas. One way (but not the only way) to compose schemas is through the mechanisms *include* and *import* provided in the XML Schema Recommendation (Fallside & Walmsley, 2004). *Include* is used when the other schema(s) has the same target namespace as the main schema. *Import* is used when the other schema document has a different target namespace. In the future, we will work on how to deal with the *import* and *include* mechanisms in XML Schema mapping.

6. **List and union types**. Most simple types in XML Schemas are atomic types, which mean they contain values that are indivisible. Besides atomic types, there are also two other varieties of simple types: *list* and *union* types. *List* types have values that are whitespace-separated list of atomic values, such as *<availableSizes>10 large 2</availableSizes>* (Walmsley, 2002). *Union* types may have values that are either atomic values or list values. In this thesis, we do not consider *list* and *union* types.

7. **Other features**. Some features provided by the XML Schema Recommendation are not an integral part of every XML Schema. These features include reusable groups, identity constraints, substitution groups, and redefinition.

2.7 Schema Determinism

The paper (Murata, Lee, Mani, & Kawaguchi, 2005) defines two restricted classes of regular tree grammars: *local tree grammars* and *single-type tree grammars.* A local tree grammar is a regular tree grammar without *competing non-terminals* (Murata, Lee, Mani,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

& Kawaguchi, 2005). Two different non-terminals compete with each other if their productions share the same terminal on the right-hand side. This class of regular tree grammars roughly corresponds to DTD (Murata, Lee, Mani, & Kawaguchi, 2005).

A single-type tree grammar is such a regular tree grammar that for each production rule of form $A \rightarrow r$, the non-terminals appearing in regular expression r do not compete with each other. We see that a single-type tree grammar is less restricted than a local tree grammar because it allows the existence of competing non-terminals in different content model.

Like XML 1.0, XML Schema requires that content models be *deterministic* (Walmsley, 2002). That is, a schema processor, as it makes its way through the children of an instance element, must be able to find only one branch of the content model that is applicable, without having to look ahead to the rest of the children. According to this specification, the expressiveness of XML Schema should be within that of single-type grammars. However, in some cases, it is beyond the expressiveness of single-type tree grammars. For example, element wildcards supported by the XML Schema Recommendation, which are represented by the *any* elements, allow elements without specifying tag names. This feature doubtless increases flexibility as to what elements may appear in a content model. However, it may lead to non-single-type schemas. Let's consider the following XML Schema borrowed from (Murata, Lee, Mani, & Kawaguchi, 2005).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
  <xs:element name = "test" >
    <xs:complexType>
      <xs:sequence>
        <xs:any namespace = "##any" processContents = "strict" />
        <xs:element name = "foo" type = "xs:integer" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name = "foo" type = "xs:string" />
</xs:schema>
```

Figure 5: XML Schema Test

The XML Schema Recommendation allows two element declarations with the same element names, as long as they are in different scopes. In XML Schema Test, the *foo* element of type *integer* is scoped to the complex type within which it is declared, while the *foo* element of type *string* is global-scoped. Although these two element declarations share the same element name, they are in different scopes and thus they are allowable in XML Schemas.

In XML Schema Test, the value of the *namespace* attribute in the *any* element declaration (in bold) is *##any*, which means the replacement element can be in any namespace whatsoever, or be in no namespace. So, one of the possible replacement elements is the *foo* element of type *string*. In such a case, the content model of XML Schema Test is non-deterministic because the processor, if it first encounters a child *foo*, will not know whether it should validate it against the *foo* declaration of type *string*, or the *foo* declaration of type *integer*.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We represent XML Schema Test by a regular tree grammar $G_3 = (\Sigma, N_1, N_2, P_1, P_2, S)$ where

Σ = {*test*, *foo*, *integer*, *string*}

N_1 = {Test, Any, Foo, String, Integer}

N_2 = {TestType, FooType}

S = Test

P_1 = { Test → *test*(TestType)

Any →*foo*(FooType1)

Foo →*foo*(FooType2)

Integer → *integer*

String → *string* }

P_2 = { TestType → Any · Foo

FooType1→ String

FooType2 → Integer }

We see that G_3 is not a single-type tree grammar because the non-terminals Any and Foo compete with each other. That is, their production rules (in bold) have different non-terminals on the left-hand side, but share the same terminal on the right-hand side. Moreover, these non-terminals appear in the right-hand-side regular expression of the following production

TestType → Any · Foo

In terms of expressive power, any local tree language is a single-type tree language and any single-type tree language is a regular tree language ((Murata, Lee, Mani, &

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Kawaguchi, 2005, Lemma 2.1). Most XML Schemas can be represented by single-type tree grammars. However, if the content model of an XML Schema is non-deterministic, only a regular tree grammar can model this XML Schema.

2.8 Subtyping

Based on the definitions of regular tree grammars and regular tree languages (see Definition 3), we formally define subtyping below.

Definition 5 (subtyping)

Given two XML Schemas R and S, R is a subtype of S (denoted by R <: S) if $L(R) \subseteq L(S)$, where L(R) and L(S) are regular tree languages generated by the regular tree grammars representing R and S, respectively.

As XML Schemas describe sets of XML documents, the subtyping problem between XML Schemas is reduced to the set-inclusion problem of sets of XML documents. This concept of inclusion-subtyping corresponds to one of the XML Schema derivation mechanisms: *type restriction*. As its name implies, type restriction means restricting the valid content of either a simple type or a complex type (i.e. *base type*) to define a new one. Specifically, the XML Schema Recommendation provides twelve facets (e.g. *minlength*, *maxlength*, *pattern*, *enumeration*, etc.) for users to specify a valid range of values, to constrain the length and precision of values, to enumerate a list of valid values, or to specify a regular expression that valid values must match. All instances of a new type derived by restriction are valid against its base type. In other words, the set of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

instances consistent with the restricted type is a subset of the set of instances of its corresponding base type. In terms of the concept of set-inclusion subtyping (Definition 5), a restricted type is a subtype of its base type.

However, this concept of set-inclusion subtyping is not sufficient to support another important XML Schema mechanism, namely, *type extension*, which yields a "subclass" relationship by adding additional child elements and/or attributes to the tail of a "superclass" type, thus extending the content of the "superclass" type. This is quite similar to inheritance in object-oriented languages. Obviously, instances of an extended type are not valid against its base type any more, since new elements and/or attributes are added, and vice versa. For example, given a type *person* we may define another type *employee* where the instances of *employee* have the same elements as the instances of *person,* except for the augmentation with a new child element named *employeeNumber*. Like what is done in object-oriented processing, we are able to apply all the methods/procedures for type *person* to the instances of type *employee*. However, since neither of the instance sets of type *employee* and type *person* is a subset of the other, the relationship between these "subclass" and "superclass" types can not be described by set-inclusion subtyping. In the paper (Bry et al., 2004), a notion of *extension-subtyping* is proposed to deal with XML Schema's type extension. However, we will not go further about this kind of subtyping here, because it is beyond the scope of this thesis.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER III

A SUBTYPING ALGORITHM FOR XML SCHEMAS

3.1 What is a Subtyping Algorithm?

The subtyping problem for XML Schema is often reduced to the inclusion problem for regular expressions or regular tree languages (Hosoya, Vouillon, & Pierce, 2000) (Kempa & Linnemann, 2003) (Hohenadel, 2003). Since we model XML Schemas based on regular tree grammars, checking the subtype relationship between two regular tree languages is the main task in this thesis.

An algorithm that aims to check for $L(r) \subseteq L(s)$ is called a *subtyping algorithm*, which lies at the core of XML-centric programming language implementations. The input of such a subtyping algorithm is a subtype relationship statement $r <: s$, which is called *regular inequality* (Antimirov, 1994). The output of a subtyping algorithm is either *true* or *false*, in accordance to the truth value of the input subtype relationship statement.

3.2 Main Ideas of the Subtyping Algorithm

Based on Antimirov's derivation calculus (Antimirov, 1994) for regular expressions and its extensions to regular hedge expressions (Kempa & Linnemann, 2003) (Hohenadel, 2003), we formalize and rebuild the algorithm for regular tree grammars, which is very close to that for regular hedge grammars. Additionally, we add some heuristics in the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

algorithm.

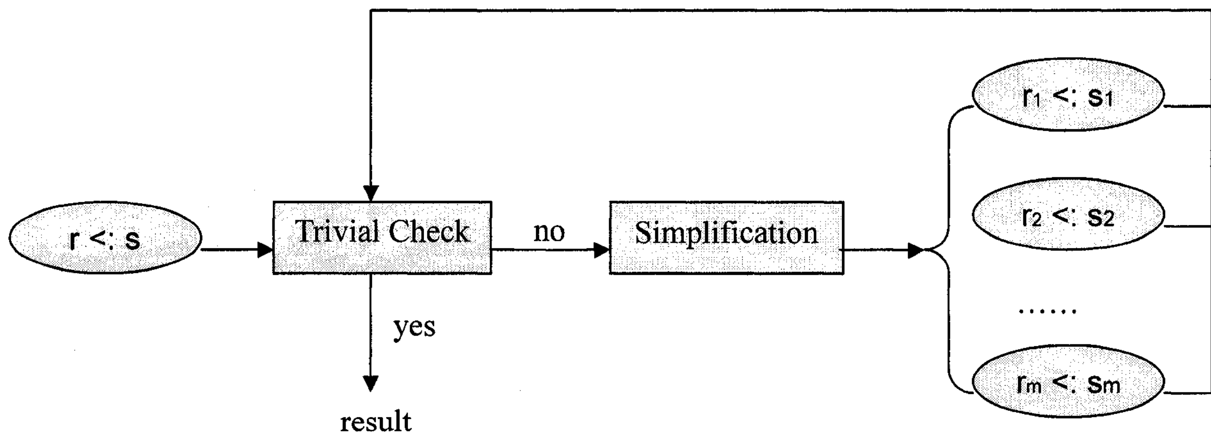

Figure 6: Main ideas of the subtyping algorithm

The basic idea of the subtyping algorithm (shown in Figure 6) is as follows: given two XML Schemas, if they can be trivially checked, the algorithm stops and returns the result immediately. Otherwise, it recursively simplifies the corresponding type expressions of the input XML Schemas, until they are simple enough to perform a trivial check.

For every invalid regular inequality, there exists at least one reduced regular inequality which is *trivially inconsistent* (Antimirov, 1994). A regular inequality r <: s is trivially inconsistent if the language generated by r contains the empty type ε and the language generated by s doesn't. If no such a trivially-inconsistent regular inequality is encountered, then after a finite number of derivation steps, the system ends up with a state that all reduced inequalities are already processed in previous derivation steps (i.e.,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

no new inequality is reduced any more). In this case, the original regular inequality is assumed to be *true*. This is a standard procedure in subtyping algorithms of recursive types (Hosoya, Vouillon, & Pierce, 2005) and (Kempa & Linnemann, 2003). The correctness and termination proofs given by the paper (Hosoya, Vouillon, & Pierce, 2005) explain that if there is no trivial inconsistency encountered in recursions, the algorithm can eventually end up with the state that it cannot produce any new reduced regular inequality, and then the input regular inequality holds.

So, subtyping algorithms of recursive types, including ours, do not prove subtyping directly. Instead, these algorithms keep track of already-treated inequalities in an environment variable σ, which is a set of inequalities and is empty at the beginning of the algorithms. Each time before a regular inequality is subtype-checked, the system checks whether this inequality is already in the environment σ. If yes, the inequality is proven to be *true*; otherwise, the inequality is put into the environment σ before the system simplifies it. Next time when the system encounters the same inequality again, it stops and returns *true*. Thus, the termination of subtyping algorithms is ensured.

From Figure 6, we see, to (dis)prove r <: s where r and s are the type representations of two given XML Schema, there are two steps: (1) trivial check; (2) simplification. The process of simplification or derivation leads to a number of simpler regular inequalities. The algorithm recursively calls steps (1) and (2) on those regular inequalities. We will discuss these two steps at length in the following subsections.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.3 Check for ε-inclusion

Checking for ε-inclusion of a type expression e means to check whether the regular tree language L(e) contains the empty string ε, i.e. $\varepsilon \in L(\mathrm{e})$.

Given a regular inequality r <: s, if the language represented by r contains ε, but the language represented by s doesn't, i.e.

$$\varepsilon \in L(\mathrm{r}) \wedge \varepsilon \notin L(\mathrm{s})$$

one can easily infer $L(\mathrm{r}) \not\subset L(\mathrm{s})$. Then, according to the formal definition of subtyping (see Definition 5), the given inequality r <: s doesn't hold. This situation, i.e., $\varepsilon \in L(\mathrm{r}) \wedge \varepsilon \notin L(\mathrm{s})$, is called a *trivial inconsistency* (Antimirov, 1994), which is a special case of ε-inclusion check.

If a trivial inconsistency occurs, the algorithm returns *false* immediately; otherwise, the input inequality r <: s is recursively simplified (or reduced), until all reduced inequalities are simple enough to perform a trivial check.

To check for ε-inclusion in the implementation of the subtypng algorithm, we redefine the function *Nullable* below, which was originally defined in (Hohenadel, 2003), by replacing its original expressions with our type expressions as the argument. Since Hohenadel's formal framework for XML Schema is different from ours, these two

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

expressions are different. The *Nullable* function returns *true* if a type expression e is *nullable*, i.e., $\varepsilon \in L(e)$; otherwise, *false*.

$$Nullable(\emptyset) = \text{false} \quad \text{(N1)}$$

$$Nullable(\varepsilon) = \text{true} \quad \text{(N2)}$$

$$Nullable(\ c) = \text{false, where } c \in \Sigma_0 \quad \text{(N3)}$$

$$Nullable(\ l(A)\ ) = \text{false, where } A \in N_2,\ l \in \Sigma \quad \text{(N4)}$$

$$Nullable(e_1 \cdot e_2) = Nullable(e_1) \wedge Nullable(e_2) \quad \text{(N5)}$$

$$Nullable(e_1 \mid e_2) = Nullable(e_1) \vee Nullable(e_2) \quad \text{(N6)}$$

$$Nullable(e\{n,m\}) = \text{true}, \quad \text{if } n = 0 \quad \text{(N7a)}$$

$$\mid Nullable(e), \quad \text{otherwise} \quad \text{(N7b)}$$

To illustrate the derivation process, we introduce a concrete example here.

Example 3: To check if R <: S holds, where

R → *a*(*a*(R)) | string

S → *a*(S) | string

After unfolding the non-terminal R and S by replacing them with their corresponding right-hand-side type expressions, the original inequality R <: S is rewritten as:

a(*a*(R)) | string <: *a*(S) | string

To check the above inequality is equal to check the following two inequalities:

(1) *a*(*a*(R)) <: *a*(S) | string

(2) string <: *a*(S) | string

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The second inequality is trivially true according to set theory and the definition of subtyping (see Definition 5). Thus, we focus on checking the first inequality in the following computations.

According to rule N4, we get

$$Nullable(a(a(\mathrm{R}))) = \text{false}$$

So, the inequality $a(a(R))$ <: $a(S)|string$ is not trivially inconsistent. We need to simplify it.

3.4 Simplification Process

XML Schemas define the tree structures of sets of XML documents. The tree structures, as defined in Definition 3 (i.e. $\mathrm{S} \rightarrow l(A)$), are derived from the root elements of XML Schemas. So, to compare two XML Schemas, we actually compare two trees derived from their root elements. From this point of view, we roughly explain how the simplification process works below.

Our subtyping algorithm adopts a top-down checking approach to fulfill this task. That is, the algorithm first checks the roots of two trees that represent two given XML Schemas. If the two roots have different labels (i.e. element names in our problem domain), then the algorithm stops and return false. In such a case, we think that there is no subtyping relationship between the two input XML Schemas. If the roots share the same element

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

name, the algorithm proceeds to check the content models of the roots. This check, augmented with the check for siblings, is repeated recursively on the inner nodes. For leaf nodes, the algorithm only needs to check their labels and siblings because leaf nodes do not have any children (or content model). Briefly speaking, our algorithm does the subtype-check for two trees dependently on the three dimensions of a node of a tree: the label, the content model and the siblings.

To make the algorithm recognize these three parts, we introduce the concept of linear forms (Antimirov, 1994). A linear form is a pair consisting of the label of a leaf node or the label of a non-leaf node followed by its content model as the first component and the siblings of the node as the second component. The set of linear forms of a tree node (denoted by a type expression) contains all possible permutations of the label, content model and siblings that the node can have.

If two tree nodes have the same label, the system only needs to check the content model and siblings of these nodes. These two parts are represented by the rest of the linear form after extracting the label. The label to be extracted is called a *leading name* (Kempa & Linnemann, 2003). The rest part, i.e., the pair only consisting of the content model and siblings, is called a *partial derivative* (Antimirov, 1996) of the original expression. Since a leading name is extracted from a linear form of the original expression, the resulted partial derivative is simpler than the original expression. The subtype-check is called recursively on partial derivatives of two type expressions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To reduce (or simplify) a regular inequality r <: s, we have to compute:

(1) the linear forms of type expressions r and s

(2) the leading names of r

(3) the partial derivatives of r and s

(4) the partial derivatives of r <: s

3.4.1 Linear Forms

Following the definition of *linear forms of a regular term* (Antimirov, 1994), Hohenadel (Hohenadel, 2003) defines the set of linear forms of an expression e, denoted by *lf*(e), as a set of pairs consisting of the leftmost expression in e as the first component and the remainder of e as the second component. For example, let $a, b\ c \in \Sigma_0$ and $l \in \Sigma_1$,

$$lf(a) = \{\langle a, \varepsilon\rangle\}$$

$$lf(l(a)) = \{\langle l(a), \varepsilon\rangle\}$$

$$lf(l(a) \cdot b) = \{\langle l(a), b\rangle\}$$

$$lf((a|\ b)\cdot c) = \{\langle a, c\rangle, \langle b, c\rangle\}$$

Intuitively, the set of linear forms of an expression actually represents the permutation of all possible sequences in the language specified by the expression.

To compute the *linear forms* of a type expression, we modify the *lf* function (Hohenadel, 2003) by taking our type expressions rather than *regular expressions* (Hohenadel, 2003) as the input. The function *lf* is redefined recursively by the following equations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$lf(\emptyset) = \emptyset \qquad \text{(LF1)}$$

$$lf(\varepsilon) = \emptyset \qquad \text{(LF2)}$$

$$lf(c) = \{<c, \varepsilon>\}, \text{ where } c \in \Sigma_0 \qquad \text{(LF3)}$$

$$lf(l(A)) = \{<l(A), \varepsilon>\}, \text{ where } A \in N_2, l \in \Sigma \qquad \text{(LF4)}$$

$$lf(e_1 \cdot e_2) = lf(e_1) \odot e_2, \text{ if } Nullable(e_1) = \text{false} \qquad \text{(LF5a)}$$

$$\mid lf(e_1) \odot e_2 \cup lf(e_2), \text{ otherwise} \qquad \text{(LF5b)}$$

$$lf(e_1 \mid e_2) = lf(e_1) \cup lf(e_2) \qquad \text{(LF6)}$$

$$lf(e\{n, m\}) = lf(e) \odot e\{n, m\}, \text{ if m is "undefined"} \qquad \text{(LF7a)}$$

$$\mid lf(e) \odot e\{n, m\text{-}1\}, \text{ otherwise} \qquad \text{(LF7b)}$$

The computation of linear forms involves a binary concatenation operation $\odot$, which takes a set of linear forms and a type expression as its arguments, and returns another set of linear forms. It is an extension of concatenation to linear forms and its definition (see below) is borrowed from Antimirov (Antimirov, 1994).

For any set of linear forms l, l' and any type expression x, t, p, excluding $\emptyset$ and ε,

$$l \odot \emptyset = \emptyset \qquad \text{(LF8)}$$

$$\emptyset \odot t = \emptyset \qquad \text{(LF9)}$$

$$l \odot \varepsilon = l \qquad \text{(LF10)}$$

$$\{<x, p>\} \odot t = \{<x, p \cdot t>\} \qquad \text{(LF11)}$$

$$(l \cup l') \odot t = (l \odot t) \cup (l' \odot t) \qquad \text{(LF12)}$$

Let's go back to Example 3. According to the definition of linear forms, we get

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$
\begin{aligned}
lf(a(a(\mathrm{R}))) &= \{\langle a(a(\mathrm{R})), \varepsilon\rangle\} && \text{by rule LF4}\\
lf(a(\mathrm{S}) \mid \mathrm{string}) &= lf(a(\mathrm{S})) \cup lf(\mathrm{string}) && \text{by rule LF6}\\
&= \{\langle a(\mathrm{S}), \varepsilon\rangle\} \cup \{\langle \mathrm{string}, \varepsilon\rangle\} && \text{by rule LF4 and LF3}\\
&= \{\langle a(\mathrm{S}), \varepsilon\rangle, \langle \mathrm{string}, \varepsilon\rangle\} && \text{by set theory}
\end{aligned}
$$

3.4.2 Function First

According to the definition of regular tree grammars (Definition 3), a leading name of a type expression can be an element name or a built-in simple type. This is the part to be extracted from the linear forms of this type expression.

Please note that a type expression may have more than one leading names. For example, the leading names of type expression $e_1|e_2$ should include the leftmost constants of e_1 and e_2. Another case is the concatenation $e_1 \cdot e_2$, where e_1 is nullable, the leading names of $e_1 \cdot e_2$ should include the leading name of e_2.

To compute the leading name(s) in a type expression, we use the *First* function, which takes a type expression as input and returns a set of leftmost terminals. The definition of *First* is given below (Aho, Sethi & Ullman, 1988).

1. If $n \stackrel{*}{\Rightarrow}_G c$ where $c \in \Sigma_0$, $First(n) = \{c\}$;
2. If $n \stackrel{*}{\Rightarrow}_G c\alpha$ where $c \in \Sigma_0$ and is a sequence of symbols from Σ and N, $First(n) = \{c\}$. If $n \stackrel{*}{\Rightarrow}_G \varepsilon$, add ε to $First(n)$;
3. If $n \stackrel{*}{\Rightarrow}_G \alpha_1\alpha_2\ldots\alpha_m$ where $\alpha_1, \alpha_2, \ldots, \alpha_m \in \Sigma_0$, add $First(\alpha_1)$ to $First(n)$. If α_1

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$\Rightarrow^*_G \varepsilon$, add *First*($\alpha_2$) to *First*($n$). If $\alpha_1 \Rightarrow^*_G \varepsilon$ and $\alpha_2 \Rightarrow^*_G \varepsilon$, add *First*($\alpha_3$) to *First*($n$), and so on.

Please note that this function is only applied on the left-hand-side type expression of a given regular inequality. In Example 3, *First*($a(a(\mathrm{R}))$) = $\{a\}$.

3.4.3 Partial Derivatives of a Type Expression

After extracting a leading name $w \in First(e)/\{\varepsilon\}$ from the linear forms of a type expression e, the remainder of the linear forms, i.e., the set of partial derivatives of e w.r.t. w is denoted by $\partial_w(e)$. The set of partial derivatives of a type expression represents the reduced representation of the original type expression.

As the linear forms of a type expression is a set of pairs, the partial derivatives of the type expression is also a set of pairs after extracting a leading name. However, the first component of a pair in $\partial_w(e)$ is the content of the leading name, instead of the leftmost type expression. We modify the definition of partial derivatives (Hohenadel, 2003) as follows.

$$\partial_w(e) = \{< cn(e_1, w), e_2> \mid <e_1, e_2> \in lf(e), \text{ and } cn(e_1, w) \neq \emptyset\}$$

In the above definition, we remove the original condition $e_2 \neq \emptyset$ and add $cn(e_1, w) \neq \emptyset$. The reason why we don't need $e_2 \neq \emptyset$ is the empty set can not be the second component of a type pair according to the definition of linear form (see Section 3.4.1).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

As defined in (Hohenadel, 2003), the definition of partial derivatives involves a function called *cn*, which is only applied on the first components of linear forms. According to the definition of linear forms, only two type expressions p and $l(A)$ can be the first component of a pair of linear forms, where $p \in \Sigma_0$, $A \in N_2$ and $l \in \Sigma$. So, the function *cn* takes type expressions of form p or $l(A)$ as an argument. As the computation of the content of a given leading name w in a type expression should also depend on w, our modified function *cn* requires two arguments, instead of just one in (Hohenadel, 2003). The function *cn* is redefined as follows:

$$cn(p, w) = \varepsilon, \text{ if } p \in \Sigma_0 \text{ and } p = w$$
$$\mid \varnothing, \text{ otherwise}$$

$$cn(l(A), w) = A, \text{ if } l = w$$
$$\mid \varnothing, \text{ otherwise}$$

As the *cn* function may return the empty set ∅, to avoid its appearing as the first component, we add the condition $cn(e_1, w) \neq \varnothing$ in our definition of partial derivatives of a type expression.

Let's go back to Example 3. The leading name of type expression $a(a(\mathrm{R}))$ is $\{a\}$. After extracting the leading name a from the linear forms of $a(a(\mathrm{R}))$ and $a(\mathrm{S})|\text{string}$, the partial derivatives of $a(a(\mathrm{R}))$ and $a(\mathrm{S}) \mid \text{string}$ w.r.t. the leading name a are given as follows:

$$\partial_a(a(a(R))) = \{\langle a(\mathrm{R}), \varepsilon\rangle\}$$
$$\partial_a(a(S)) = \{\langle \mathrm{S}, \varepsilon\rangle\}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Please note that the original concept of partial derivatives proposed by Antimirov (Antimirov, 1996) is based on regular expressions. So, the partial derivatives of a regular expression were defined as a set of regular expressions, instead of a set of pairs. As we know, an ordinary regular expression denotes sequences of letters. For any letter in such a sequence, we only need to consider its siblings in addition to its label. However, a node of an XML tree has not only siblings and a label, but also a content model denoting the order and structure of its child elements. Therefore, Anitmirov's definition of partial derivatives must be modified to make it applicable to XML types. Specifically, the two components of a pair in partial derivatives correspond to the two dimensions of a tree node: the parent-child dimension (i.e. the content model) and the sibling dimension, respectively.

3.4.4 Partial Derivatives of a Regular Inequality

According to the definition of partial derivatives of a type expression, given a regular inequality (r <: t), the partial derivatives of r and t w.r.t. a leading name $w \in First(r)/\{\varepsilon\}$ are given as follows:

$$\partial_w(r) = \{<c_1, p_1>, <c_2, p_2>, \ldots, <c_n, p_n>\}$$

$$\partial_w(t) = \{<c_1', p_1'>, <c_2', p_2'>, \ldots, <c_m', p_m'>\}$$

where $<c_i, p_i>$ is one of partial derivatives of r w.r.t. w, $i = 1, \ldots, n$, and $<c_j', p_j'>$ is one of partial derivatives of t w.r.t. w, $j = 1, \ldots, m$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Antimirov (Antimirov, 1994) defines the *partial derivatives of a regular inequality* (r <: t) as follows:

$$\partial_w(r <: t) = \{p <: q \mid p \in \partial_w(r) \text{ and } q = \Sigma\partial_w(t)\} \quad (3\text{-}1)$$

where $\Sigma\partial_w(t)$, called *derivatives* (Antimirov, 1994) of t w.r.t. *w*, is the union of all partial derivatives of t w.r.t. *w*, i.e., $\Sigma\partial_w(t) = <c_1', p_1'> \mid <c_2', p_2'> \mid \ldots \mid <c_m', p_m'>$.

According to this definition, the equation (3-1) can be extended as follows:

$$\begin{aligned}\partial_w(r <: t) = \{&<c_1, p_1> \;<:\; <c_1', p_1'> \mid <c_2', p_2'> \mid \ldots \mid <c_m', p_m'>,\\ &<c_2, p_2> \;<:\; <c_1', p_1'> \mid <c_2', p_2'> \mid \ldots \mid <c_m', p_m'>,\\ &\ldots\ldots\\ &<c_n, p_n> \;<:\; <c_1', p_1'> \mid <c_2', p_2'> \mid \ldots \mid <c_m', p_m'>\}\end{aligned} \quad (3\text{-}2)$$

We see that $\partial_w(r <: t)$ contains *n* regular inequalities, each of which has one partial derivative of *r* on the right hand side and the union of *m* partial derivatives of *t* on the left hand side, where *n* is the number of partial derivatives of *r* w.r.t. *w* and *m* is the number of partial derivatives of *t* w.r.t. *w*.

For example, the partial derivatives of *a*(*a*(*R*)) <: *a*(*S*)|string in Example 3 is as follows:

$$\partial_a(a(a(R)) <: a(S) \mid string) = \{< a(R), \varepsilon > \;<:\; < S, \varepsilon >\} \quad (3\text{-}3)$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To further simplify the computation of partial derivatives of a regular inequality (3-3), we borrow the set-theoretic observation proposed by Hosoya et al. (Hosoya, Vouillon, & Pierce, 2005), which we explain in Appendix A.

According to Hosoya et al.'s observation, we transform the inequality

$$< a(R), \varepsilon > \ <: \ <S, \varepsilon>$$

into the following Boolean set consisting of two clauses (3-4) and (3-5), each of which contains two inequalities connected by the Boolean operator OR ($\vee$).

$$\{\ a(\mathrm{R}) <: \varnothing \quad \vee \quad \varepsilon <: \varepsilon, \qquad (3\text{-}4)$$
$$a(\mathrm{R}) <: \mathrm{S} \quad \vee \quad \varepsilon <: \varnothing\ \} \qquad (3\text{-}5)$$

We can see that Hosoya et al.'s observation reduces the subset relation on Cartesian products to a subset relation on sets, and thus simplify the computation. As a result, we get a number of simpler regular inequalities reduced from the original one. In the next step, we recursively call the simplification process to check those reduced inequalities. If all clauses in the Boolean set are evaluated to be *true*, then the original inequality holds.

In our example, clause (3-4) is trivially true because $\varepsilon <: \varepsilon$ always holds. To evaluate clause (3-5), we need to call the derivation process again to check the inequality a(R) <: S. Similarly, after unfolding S by production rule S $\rightarrow$ a(S)|string and calling the subtyping algorithm again, the following regular inequalities are reduced from a(R) <: S.

$$\{\ \mathrm{R} <: \varnothing \quad \vee \quad \varepsilon <: \varepsilon, \qquad (3\text{-}6)$$
$$\mathrm{R} <: \mathrm{S} \quad \vee \quad \varepsilon <: \varnothing\ \} \qquad (3\text{-}7)$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Clause (3-6) is true because of ε <: ε. Clause (3-7) is also true because R <: S is the given input and is already in the set σ. So, a(R) <: S is true. Since both Clause (3-4) and (3-5) are true, $\partial_a(a(a(R)) <: a(S) \mid string)$ holds and thus $a(a$(R)) <: a(S)|string holds. Therefore, we prove R <: S. Figure 7 gives the proof tree of this example.

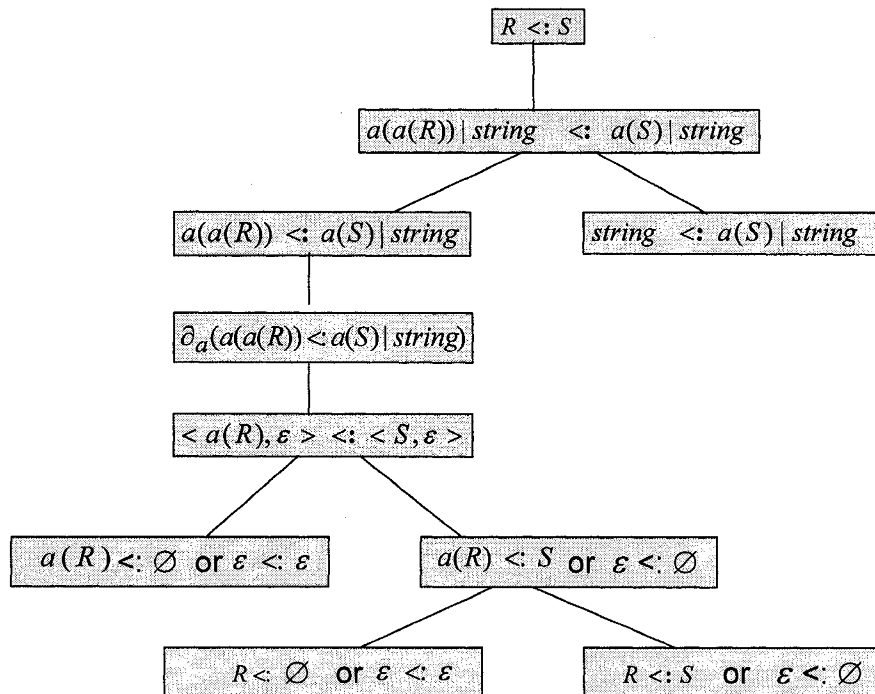

Figure 7: Proof Tree of Example 3

The logfile for Example 3 is given in Appendix B.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.5 Algorithm Rules

The subtyping algorithm for XML Schema is expressed by the rules in Figure 8.

(Terminate Rule) $$\frac{}{r <: s}, \text{if } (r <: s) \in \sigma$$

(Disprove Rule) $$\frac{}{\neg(r <: s)}, \text{if } \varepsilon \in L(r) \wedge \varepsilon \notin L(s)$$

(Derive Rule) $$\frac{\sigma \cup \{r <: s\} \mid- \partial_W(r <: s)}{\sigma \mid- r <: s}, \textit{if } \varepsilon \notin L(r)$$

(Derive Rule) $$\frac{\sigma \cup \{r <: s\} \mid- \partial_W(r <: s)}{\sigma \mid- r <: s}, \text{if } \varepsilon \in L(r) \wedge \varepsilon \in L(s)$$

(Unfold Rule) $$\frac{\sigma \cup \{R <: S\} \mid- r <: s}{\sigma \mid- R <: S}, \textit{if } R \to r, S \to s$$

(Unfold Rule) $$\frac{\sigma \cup \{R <: s\} \mid- r <: s}{\sigma \mid- R <: s}, \textit{if } R \to r$$

(Unfold Rule) $$\frac{\sigma \cup \{r <: S\} \mid- r <: s}{\sigma \mid- r <: S}, \textit{if } S \to s$$

(Disjunction Rule) $$\frac{\sigma \mid- r_1 <: s, \ \sigma \mid- r_2 <: s}{\sigma \mid- (r_1 \mid r_2) <: s}$$

Figure 8: XML Schema Subtyping Rules

To (dis)prove r <: s where r and s are the type expressions of two given XML Schema, we first apply the *Termination* rule to check whether it is already in the environment σ. Please note the set σ is empty at the beginning of the algorithm. If r <: s is in σ, we immediately prove r <: s. Otherwise, we check whether there exists a trivial inconsistency in r <: s, i.e., $\varepsilon \in L(r) \wedge \varepsilon \notin L(s)$. If the system encounters a trivial inconsistency,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

according to the *Disprove* rule, the system returns false immediately. In other cases, i.e., $(\varepsilon \in L(r) \wedge \varepsilon \in L(s))$ or $\varepsilon \notin L(r)$, we add the inequality r <: s to the environment σ and then apply the simplification process (described in Section 3.4) to it. In the two *Derive* rules in Figure 8, we see that the problem to check r <: s is reduced to the problem to check its partial derivatives $\partial_w(r <: s)$, which is a set of simplified regular inequalities after retrieving a leading name w ($\in \Sigma$) from both sides of the original inequality r <: s . We see that the *Terminate* rule and *Disprove* rule ensure the termination of the algorithm, as well as the avoidance of repeated checks of the same inequality. The *Derive* rules are for recursively applying the simplification process to those regular inequalities that fail at rules *Termination* and *Disprove*.

The remaining rules in Figure 8 depend on what input type expressions are like. If the right-hand side, or the left-hand side of a regular inequality, or both, are non-terminals (denoting by *R*, *S*), then according to the three *Unfold* rules in Figure 8, we simply interpret the productions for those non-terminals as the term rewriting rules. That is, we unfold a non-terminal by replacing it with the right-hand-side type expression of its production rule.

The *Disjunction* rule handles the case where the left-hand side is the union of two type expressions r_1 and r_2. We generate two sub-goals in such cases. The intuition behind this rule is the set-theoretic fact that

$$L(r_1) \cup L(r_2) \subseteq L(s) \text{ iff } L(r_1) \subseteq L(s) \text{ and } L(r_2) \subseteq L(s).$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

We give a complete example below to show how to apply the rules in Figure 8 to do subtype-checking.

Example 4: Suppose we want to check if $R <: S$, where

$$R \rightarrow (a(a(R)) \cdot string) \mid string$$

$$S \rightarrow ((a(S) \mid b(S)) \cdot string) \mid string$$

Assume that $\sigma = \varnothing$ at the beginning of the algorithm.

Step (1): R and S are non-terminals, by the *Unfold* rule $\dfrac{\sigma \cup \{R <: S\} \mid\!- r <: s}{\sigma \mid\!- R <: S}$, if $R \rightarrow r, S \rightarrow s$

$$\varnothing \mid\!- R <: S$$

$$\Leftrightarrow \{R <: S\} \mid\!- (a(a(R)) \cdot string) \mid string <: ((a(S) \mid b(S)) \cdot string) \mid string$$

Step (2): by the disjunction rule $\dfrac{\sigma \mid\!- r_1 <: s, \sigma \mid\!- r_2 <: s}{\sigma \mid\!- (r_1 \mid r_2) <: s}$

$$\varnothing \mid\!- R <: S$$

$$\Leftrightarrow \{R <: S\} \mid\!- (a(a(R)) \cdot string) \mid string <: ((a(S) \mid b(S)) \cdot string) \mid string$$

$$\Leftrightarrow \{R <: S\} \mid\!- a(a(R)) \cdot string <: ((a(S) \mid b(S)) \cdot string) \mid string$$

$$\wedge$$

$$\{R <: S\} \mid\!- string <: ((a(S) \mid b(S)) \cdot string) \mid string$$

Step (2-1): check for $\{R <: S\} \mid\!- (a(a(R)) \cdot string) \mid string <: ((a(S) \mid b(S)) \cdot string) \mid string$

According to rules N3, N4 and N5, we get *Nullable*($a(a(R)) \cdot string$) = false.

By the *Derive* rule $\dfrac{\sigma \cup \{r <: s) \mid\!- \partial_w (r <: s)}{\sigma \mid\!- r <: s}$, *if* $\varepsilon \notin L(r), w \in ln(r)$, we compute the partial derivatives of the regular inequality as follows.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$lf(\ a(a(R))\cdot string\ ) = \{< a(a(R)), string >\}$$

$$lf(\ ((a(S)\,|\,b(S))\cdot string)\,|\,string\ ) = \{< a(S), string >, < b(S), string >, < string\ , \varepsilon >\}$$

According to the definition of leading names, we get

$$First(\ a(a(R))\cdot string\ ) = \{a\}$$

So

$$\partial_a(a(a(R))\cdot string) = \{< a(R), string >\}$$

$$\partial_a(((a(S)\,|\,b(S))\cdot string)\,|\,string) = \{< S, string >\}$$

According to the definition of partial derivatives of a regular inequality, we get

$$\partial_a(a(a(R))\cdot string <: ((a(S)\,|\,b(S))\cdot string)\,|\,string) = < a(R), string > <: < S, string >$$

By Hosoya's set-theoretic observation (see Appendix A),

$$< a(R), string > <: < S, string >$$

is equal to

$$\begin{array}{c} a(R) <: \varnothing \quad \vee \quad string <: string \\ \wedge \\ a(R) <: S \quad \vee \quad string <: \varnothing \end{array}$$

Since $string <: string$ is trivially *true* and $string <: \varnothing$ is trivially *false*, in the next step we need to recursively call the main method for subtype-checking on the regular inequality $a(R) <: S$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$\{R <: S\} \mid - a(a(R)) \cdot string <: ((a(S) \mid b(S)) \cdot string) \mid string$$

$$\Longleftrightarrow \quad \{R <: S, a(a(R)) \cdot string <: ((a(S) \mid b(S)) \cdot string) \mid string\} \mid - a(R) <: S$$

By the *Unfold* rule $\frac{\sigma \cup \{r <: S\} \mid -r <: s}{\sigma \mid -r <: S}$, if $S \to s$

$$\{R <: S\} \mid - a(a(R)) \cdot string <: ((a(S) \mid b(S)) \cdot string) \mid string$$

$$\Longleftrightarrow \quad \{R <: S, a(a(R)) \cdot string <: ((a(S) \mid b(S)) \cdot string) \mid string\} \mid - a(R) <: S$$

$$\Longleftrightarrow$$

$$\{R <: S, a(a(R)) \cdot string <: ((a(S) \mid b(S)) \cdot string) \mid string\} \mid - a(R) <: ((a(S) \mid b(S)) \cdot string) \mid string$$

Since *Nullable*($a(R)$) = false,

by the *Derive* rule $\frac{\sigma \cup \{r <: s) \mid -\partial_w (r <: s)}{\sigma \mid -r <: s}$, *if* $\varepsilon \notin L(r), w \in First(r)/\{\varepsilon\}$

we compute the partial derivatives of the regular inequality as follows.

$$lf(\, a(R)\,) = \{< a(R), \varepsilon >\}$$

$$lf(\, ((a(S) \mid b(S)) \cdot string) \mid string\,) = \{< a(S), string >, < b(S), string >, < string, \varepsilon >\}$$

According to the definition of leading names, we get

$$First(\, a(R)\,) = \{a\}$$

So

$$\partial_a(a(R)) = \{< R, \varepsilon >\}$$

$$\partial_a(((a(S) \mid b(S)) \cdot string) \mid string) = \{< S, string >\}$$

According to the partial derivatives of a regular inequality, we get

$$\partial_a(a(R) <: ((a(S) \mid b(S)) \cdot string) \mid string) = < R, \varepsilon > <: < S, string >$$

By Hosoya's set-theoretic observation (Appendix A), we know

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$< R, \varepsilon > <: < S, string >$$

is equal to

$$\begin{array}{ccc} R <: \varnothing & \vee & \varepsilon <: string \\ \wedge & & \\ R <: S & \vee & \varepsilon <: \varnothing \end{array}$$

Since $\varepsilon <: string$ is trivially *true* and $\varepsilon <: \varnothing$ is trivially *false*, we get

$$\{R <: S\} \mid - a(a(R)) \cdot string <: ((a(S) \mid b(S)) \cdot string) \mid string$$

$$\Leftrightarrow \{R <: S, a(a(R)) \cdot string <: ((a(S) \mid b(S)) \cdot string) \mid string\} \mid - a(R) <: S$$

$$\Leftrightarrow$$

$$\{R <: S, a(a(R)) \cdot string <: ((a(S) \mid b(S)) \cdot string) \mid string\} \mid - a(R) <: ((a(S) \mid b(S)) \cdot string) \mid string$$

$$\Leftrightarrow \{R <: S, a(a(R)) \cdot string <: ((a(S) \mid b(S)) \cdot string) \mid string\} \mid - R <: S$$

Since the original inequality $R <: S$ is already in the environment σ, by the *Terminate* rule $\frac{}{r <: s}$, if $r <: s \in \sigma$, $R <: S$ holds. Q.E.D.

3.6 Time Complexity

The simplification process makes the subtype-checking for a single reduced regular inequality at a lower cost; however, it leads to an increased number of regular inequalities in need of check, which increases the time complexity of the subtyping algorithm.

Given an input inequality r <: s, if it is already in the environment σ or it is trivially inconsistent, it takes constant time to dis(prove) it. Otherwise, we have to compute the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

partial derivatives of this inequality w.r.t. any leading name $w \in ln(r)$ as follows.

$$\partial_w(r <: t) = \{<c_1, p_1> \;<:\; <c_1', p_1'> \mid <c_2', p_2'> \mid \ldots \mid <c_m', p_m'>, \tag{3-8}$$
$$<c_2, p_2> \;<:\; <c_1', p_1'> \mid <c_2', p_2'> \mid \ldots \mid <c_m', p_m'>,$$
$$\ldots \ldots$$
$$<c_n, p_n> \;<:\; <c_1', p_1'> \mid <c_2', p_2'> \mid \ldots \mid <c_m', p_m'>\}$$

where n denotes the number of partial derivatives of the L.H.S. expression r, i.e., $<c_1, p_1>$, $<c_2, p_2>$, …, $<c_n, p_n>$, m denotes the number of partial derivatives of t, i.e., $<c_1', p_1'>$, $<c_2'$, $p_2'>$, …, $<c_m', p_m'>$. The partial derivatives of the regular inequality $r <: t$ is a set consisting of n inequalities of form

$$<A, B> \;<:\; <C_1, D_1> \mid <C_2, D_2> \mid \ldots \mid <C_m, D_m> \tag{3-9}$$

According to the set-theoretic observation (explained in Appendix A), to test the inequality (3-9) is equally to test: for each subset I of {1, 2, …, m},

$$(A <: |_{i \in I}\, C_i) \vee (B <: |_{j \in \bar{I}}\, D_j)$$

where $\bar{I}$ is the complement of I, i.e., $\{1, 2, \ldots, m\} \setminus I$.

For example, when m = 3, to check

$$<A, B> \;<:\; <C_1, D_1> \mid <C_2, D_2> \mid <C_3, D_3>$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

is equally to check the set:

$A <: \emptyset$	$\vee$	$B <: C_1 \mid C_2 \mid C_3$	$I = \emptyset \subseteq \{1, 2, 3\}$
$A <: C_1$	$\vee$	$B <: C_2 \mid C_3$	$I = \{1\} \subseteq \{1, 2, 3\}$
$A <: C_2$	$\vee$	$B <: C_1 \mid C_3$	$I = \{2\} \subseteq \{1, 2, 3\}$
$A <: C_3$	$\vee$	$B <: C_1 \mid C_2$	$I = \{3\} \subseteq \{1, 2, 3\}$
$A <: C_1 \mid C_2$	$\vee$	$B <: C_3$	$I = \{1, 2\} \subseteq \{1, 2, 3\}$
$A <: C_1 \mid C_3$	$\vee$	$B <: C_2$	$I = \{1, 3\} \subseteq \{1, 2, 3\}$
$A <: C_2 \mid C_3$	$\vee$	$B <: C_1$	$I = \{2, 3\} \subseteq \{1, 2, 3\}$
$A <: C_1 \mid C_2 \mid C_3$	$\vee$	$B <: \emptyset$	$I = \{1, 2, 3\} \subseteq \{1, 2, 3\}$

The above set consists of 2^3 clauses, each of which has two regular inequalities connected by "$\vee$". So, the total number of inequalities reduced from (3-9) is 2^{m+1}. As the partial derivatives of the regular inequality $r <: t$ w.r.t. a leading name is a set consisting of n inequalities of form (3-9) (see (3-8)), for each leading name $w \in ln(r)$, the total number of regular inequalities reduced from $r <: t$ is $n \times 2^{m+1}$.

In the worst case, we may have to check all of the reduced inequalities. So, there may be an exponential blow-up incurred by considering all the subsets of $\{1, \ldots, m\}$ until the algorithm dis(prove) the regular inequality $r <: t$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER IV

IMPLEMENTATION

The implementation architecture of the subtyping algorithm is shown in Figure 9.

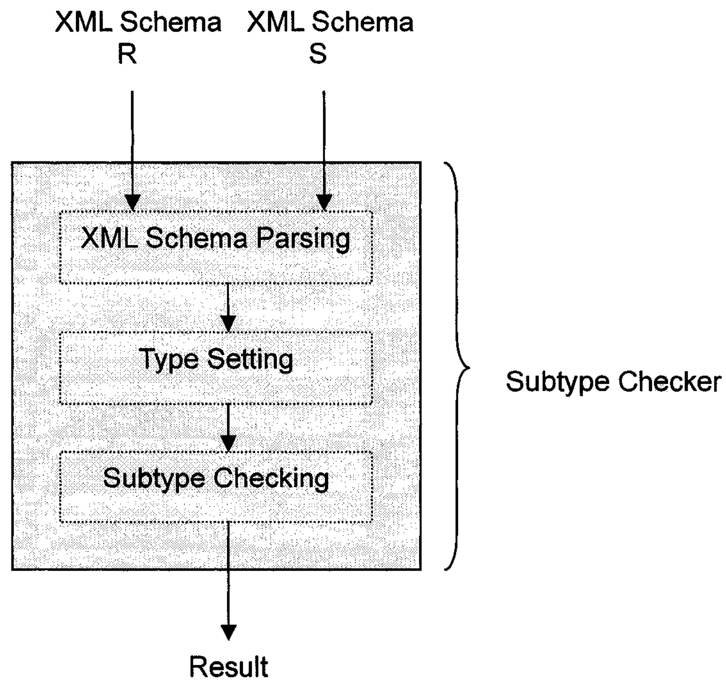

Figure 9: XML Schema subtyping architecture

The subtype checker for XML Schema consists of the following three steps:

1. Parsing input XML Schemas;
2. Converting schemas to their internal representations based on regular tree grammars;
3. Checking the subtype relationship on the intermediate representations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.1 XML Schema Parsing

Any XML Schema is XML-syntax based, which means that an XML Schema is basically an XML document. As an XML document is just a text file about data, in order to recognize the elements/attributes and their types defined in an XML Schema, we first need an XML parser to process it. The parser we've chosen in our implementation is Apache's *Xerces* (The, 2001). *Xerces* is one of the most popular XML parser that supports the W3C's XML Schema Recommendation version 1.0 and DOM (the Document Object Model) (W3C, 2004).

When we use *Xerces* to parse an XML Schema, a *DOM tree* is constructed. A DOM tree is a document object representation of a parsed XML Schema. It contains all of the elements of this XML document. By using the interfaces defined in the DOM APIs, we can access any node of a DOM tree and get the elements/attributes declared in the parsed XML Schema, as well as their assigned types. The entire process is illustrated in Figure 10.

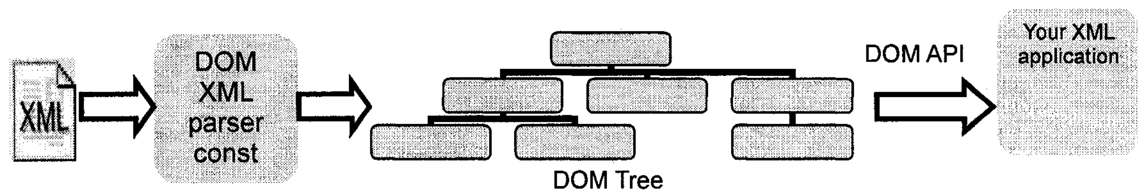

Figure 10: DOM XML Parser

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.2 Type Representations

After parsing the input XML Schemas with *Xerces* and get elements and their assigned types, we represent these XML Schemas by regular tree grammars. Following the definition of regular tree grammars (see Definition 3), we classify all of the possible type expressions of a regular tree grammar into the following eight types (Table 1). All elements and types defined in an XML Schema are represented by those eight types in the implementation of the subtyping algorithm.

Type Name	Type Expression	Description	Example
NoneType	$\emptyset$	it denotes the empty set	$\emptyset$
EmptyType	ε	It denotes the empty sequence	ε
PrimitiveType	c	It represent all XML Schema built-in simple types	*string*, *integer*, *byte*, etc.
NodeType	$l\,(A)$	A is a non-terminal representing the type name of a node with label $l \in \Sigma$	*supervisor* (supervisorType)
NamedType	n	$n \in N_1$ denote a tree node	Supervisor
ConcatenationType	$e_1 \cdot e_2$	e_1 and e_2 are regular expressions on the non-terminal set N_1	Name·Position
AlternationType	$e_1 \mid e_2$	e_1 and e_2 are regular expressions on the non-terminal set N_1	Phone | Email
IterationType	e{n,m}	e is a regular expression over N_1 n and m are non-negative integers ($n \le m$) In some cases, m may be the string of value "unbounded".	Grade{0, 10} Grade{0, unbounded}

Table 1: Eight type classes in implementation

Each type in Table 1 corresponds to a subclass of an abstract superclass called REType. Class REType contains two protected members (child1 and child 2 of type REType).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Classes NoneType and EmptyType do not use any of these two members. Classes PrimitiveType, NodeType, NamedType, and IterationType use only one of the two members. Classes ConcatenationType and AlternationType use both members because concatenation (·) and alternation (|) are binary operators. The following lists the main methods in Class REType. These methods are abstract, and thus must be implemented in its subclasses.

```
public abstract boolean isNullable();
public abstract String leadingNames();
public abstract String lf();
public abstract String pd(String w);
```

The *isNullable* method represents the implementation of function *Nullable* (see Section 3.3). It returns *true* if the type instance on which it is called is nullable; otherwise *false*.

The *leadingNames* method implements function *First* (see Section 3.4.2). It returns a string consisting of all leading names of the type instance.

The *lf* method implements the *lf* function (see Section .3.4.1). It returns a string consisting of pairs, each of which has the leftmost expression of the type instance as the first component and the remainder as the second component.

The *pd* method computes the partial derivatives of the type instance w.r.t. a leading name *w*. It returns a string consisting of pairs, each of which has the content of *w* as its first component and the second component is the same as that in the linear forms of the type

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

instance.

In the implementation, every element declaration or type definition in input XML Schemas is represented by an object of one of the eight subclasses. As we assume that every XML Schema in our work has a root element, we use the type instance for the root of an XML Schema to represent the document type, i.e., XML Schema. Therefore, to subtype-check two XML Schemas is equally to subtype-check two root elements.

4.3 Subtype Checking

Given two XML Schemas R and S, let r and s be the root type expressions of R and S, respectively. Then, the original call for the main method for subtype-checking (i.e. the *check* method in our work), is made on the regular inequality r <: s. The purpose of the *check* method is to check the subtyping relationship between two type expressions. Its inputs are two types defined in Table 1 and the environment σ. Please note that the set σ is empty at the beginning of the subtyping algorithm. The output of the *check* method is *true* if r <: s; otherwise, it is *false*. The pseudo-code for this method is given below and it is similar to that of the XOBE and Pathfinder subtyping algorithms (Kempa & Linnemann, 2003) and (Hohenadel, 2003), except for lines 12-31 where we add some heuristics to speed up the algorithm. Specifically, the system

- returns *true* immediately when checking whether a type is a subtype of the union of itself and other any type, i.e. X <: X|Y where X, Y are type expressions;

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- returns *true* when the partial derivatives of both sides of a regular inequality are the same w.r.t. any leading name of the left-hand-side type expression, i.e., $\partial_w(r) \equiv \partial_w(s)$ where r <: s is a regular inequality and $w \in First(r)/\{\varepsilon\}$.
- stops computing the partial derivatives of a regular inequality w.r.t. a leading name when the set of partial derivatives of the right-hand-side expression w.r.t. this leading name is empty;
- prunes the recursive call on the second inequality of a clause in the set of partial derivatives of a regular inequality when the first inequality holds.

We will discuss these heuristics in detail as we explain the pseudo-code line by line.

```
      boolean check(r <:s, σ ) {
1         if ( r <: s ∈ σ V r ≡ s V r ≡ ∅)
2               return TRUE;
3         if (s ≡ ∅ V (nullable(r) Λ ¬nullable(s))
4               return FALSE;
5         if(r ≡ ε)
6               return nullable(s);
7         lnSet := getLeadingNames(r);
8         if (lnSet ≡ ∅)
9               return TRUE;
10        else if (s ≡ ε)
11              return FALSE;
12        flag := TRUE;
13        pdSet := ∅;
14        foreach (w ∈ lnSet/{ε}) do {
15              if ( ∂_w(r) ≠ ∂_w(s) ) {
```


Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
16              flag := FALSE;
17              if (∂w(s) ≠ ∅)
18                  pdSet := pdSet ∪ ∂w(r <: s) ;
19          }
20      }
21      if(flag)
22          return TRUE;
23      if(pdSet ≡ ∅)
24          return FALSE;
25      else {
26          result := TRUE;
27          σ := σ ∪ {r <: s};
28          foreach ((r1 <: s1) ∨ (r2 <: s2) ∈ pdSet) do {
29              if(¬check(r1 <: s1))
30                  result := result ∧ check(r2 <: s2);
31          }
32          return result;
33      }
    }
```

According to the *Disprove* and *Termination* rule in Figure 8, given a regular inequality r <: s, the subtyping algorithm ends up either with *false* when a trivial inconsistency, i.e., *nullable*(r) $\wedge$ $\neg$*nullable*(s), is encountered, or with *true* when r <: s is already in the set σ. In the implementation, in order to shorten the path to the result in many cases, we add in line 1-4 the following trivial cases not yet discussed in Chapter 3.

r <: s is true if r $\equiv$ s (TC1)

r <: s is true if r $\equiv$ $\emptyset$ (TC2)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$r <: s \text{ is false if } s \equiv \emptyset \text{ and } r \neq \emptyset \qquad \text{(TC3)}$$

The intuitions behind TC1, TC2, and TC3 are based on set theory and the formal definition of subtyping given in Definition 5. Obviously, a set is a subset of itself. So, if the derivation process faces an inequality r <: r, immediately the Boolean value *true* is derived. Similarly, the empty set ∅ is a subset of any set and any set except for ∅ is not a subset of ∅. Then, we get TC2 and TC3.

Another trivial case is induced by the presence of ε as the left-hand-side type of an inequality, i.e., ε <: s. Obviously, the result of this inequality depends on *nullable*(s) (see lines 5-6).

If the left-hand-side type is ∅, ε, or nothing else, then it has no leading names. This case is described, in line 7, lnSet ≡ ∅. As we already check whether the left-hand-side type is ε in lines 5-6, in lines 8-9, lnSet ≡ ∅ if and only if r ≡ ∅. According to TC2, the system returns *true*. If lnSet is not empty (i.e., r is neither ∅ nor ε) and the right-hand-side type is ε, then the system returns *false* (see lines 10-11).

Lines 12-20 compute the partial derivatives of the regular inequality r <: s w.r.t. each leading name in the set lnSet. The result is stored in the set pdSet. As we discussed in Chapter 3, to compute the partial derivatives $\partial_w(r <: s)$ of a regular inequality r <: s w.r.t. a leading name w, we need to compute the partial derivatives of both sides of type expressions w.r.t. w, i.e., $\partial_w(r)$ and $\partial_w(s)$. If $\partial_w(r) \equiv \partial_w(s)$ for any leading name w in the set lnSet, then the value of a Boolean variable *flag* is *true* and thus in lines 21-22, the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

system returns *true*. In such a case, we don't need to recursively check the reduced inequalities in the set of $\partial_w(r <: s)$. For example, the following regular inequality

$$string <: string \mid integer$$

holds because lnSet = {string}, and

$$\partial_{string}(string) = \{<\varepsilon, \varepsilon>\}$$

$$\partial_{string}(string \mid integer) = \{<\varepsilon, \varepsilon>\}$$

If the set of partial derivatives of the right-hand-side type s w.r.t. a leading name *w* is empty, i.e., $\partial_w(s) \equiv \emptyset$, no derivation w.r.t. this leading name is possible and therefore we stop computing the partial derivatives of r <: s w.r.t. this leading name and continue the loop with the next leading name in the set lnSet. For example, the following regular inequality

$$integer <: string$$

evaluate trivially to *false* because the leading name of the left-hand-side type expression, i.e., *integer*, doesn't occur in the right-hand-side type and thus the set of partial derivatives of the right-hand-side type expression w.r.t. the leading name *integer* is empty, i.e. $\partial_{integer}(string) \equiv \emptyset$. In such a case, the set pdSet is empty too. According to lines 23-24, *false* is returned.

If all of trivial checks we discussed above fail, the original regular inequality r <: s

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

completely processed and considered as "previously analyzed" by being added to the environment σ. According to the modified definition of partial derivatives of a regular inequality (see Chapter 3) and Hosoya et al.'s set-theoretic observation (see Appendix A), each element of the set pdSet is a clause consisting of two simpler inequalities connected by "or". In lines 28-29, the *check* method will be recursively called on the reduced inequalities in the set pdSet. The result of *true* will be returned if and only if all clauses in the set pdSet are evaluated to be *true*.

4.4 Implicit Subtyping

There are 44 built-in simple types defined in XML Schema. The derivation relationships among all these built-in types form the type hierarchy in Figure 11 (Biron & Malhotra, 2001).

From Figure 11, we see that except for 19 built-in primitive types, the rest simple built-in types are derived, either by restriction or by list, from primitive types or other simple built-in types. As we discussed in Section 2.5, the restriction derivation of a simple or complex type leads to an implicit subtyping relationship between the base type and the restricted type. Therefore, the subtyping relationships caused by restriction (not derived by list) among those built-in simple types are established in the implementation of our subtyping algorithm. One point that is worth to mention is that the extension derivation, unlike restriction, doesn't imply a subtyping relationship.

From the set-inclusion definition of subtyping (Definition 5), we know that subtyping is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

transitive. That means if a <: b and b <: c, then a <: c holds. Any subtyping relationship that satisfies the transitivity property of subtyping can be recognized by our subtyping algorithm. For example, from the built-in data type hierarchy (in Figure 8), we know that integer is a subtype of decimal and nonNegativeInteger is a subtype of integer. Then the subtyping between nonNegativeInteger and decimal also holds.

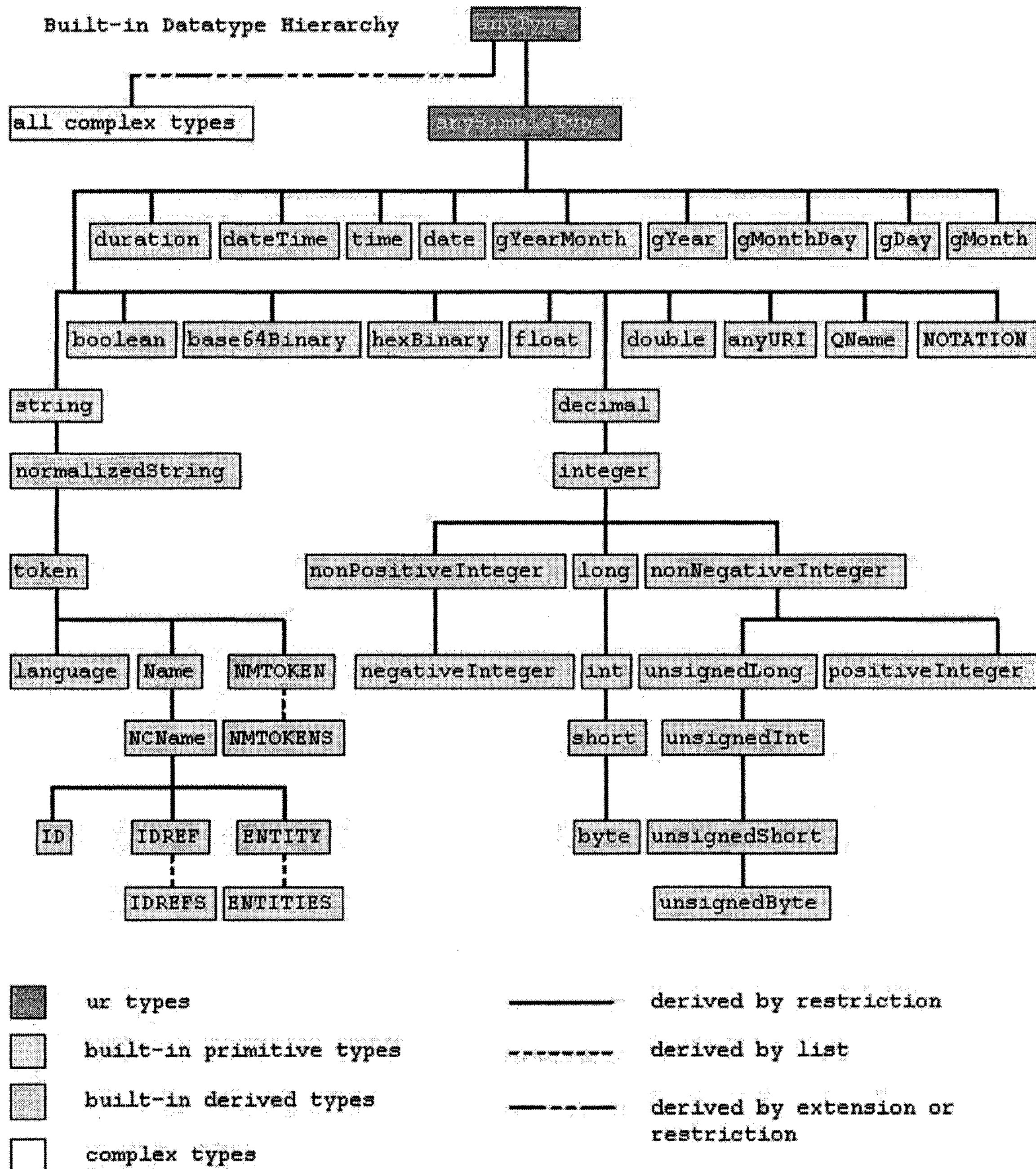

Figure 11: XML Schema built-in data type hierarchy

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER V

RELATED WORK

Many approaches have been proposed for the subtyping problem in the domain of XML processing in the past a few years. Roughly, we classified them into the following two groups. Our approach belongs to the second one.

5.1 Tree-automaton-based Subtyping

As the subtyping problem for XML schema can be reduced to the set-inclusion problem between two regular tree languages, a classical approach for testing the inclusion of tree languages is to construct tree automata for two XML schemas, and then use tree-automata techniques (Comon et al., 1999) to check the inclusion relationship between these tree automata. Specifically, it works as follows:

a) Construct tree automata *Ar* and *As*, accepting *L(r)* and *L(s)* respectively.

b) Compute the complement $\overline{As}$ of *As*, by constructing a deterministic automaton *As*' from *As* using a *subset* construction, and then making it total (if it is partial) and exchanging final and non-final states of *As*'. This step is quite expensive.

c) Take the intersection of *Ar* and $\overline{As}$, using a *product* construction. This is a highly expensive operation because the number of states of the new tree automata $A = (Ar \cap \overline{As})$ is generally exponential in the number of states of *As*.

d) Test the emptiness of *A*. If *A* only accepts ε, i.e., $L(A) \equiv \{\varepsilon\}$, which means that no final state of *A* is reachable given any tree, then $Ar \subseteq As$ holds and thus r <: s is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

evaluated to be *true*; otherwise, the system returns *false*. (according to the set-theoretic fact that $Ar \subseteq As$ iff $Ar \cap \overline{As}$ = empty)

Using this approach, the subtyping problem, which is reduced to the inclusion problem between tree automata, is known to be decidable but EXPTIME-hard (Seidl, 1999). The reason is that this approach requires many expensive manipulations of tree automata, which usually cost $O(2^n)$ in the worst case, where n is the number of states of a tree automaton. A non-deterministic tree automaton for a practical XML schema usually has $10^2 - 10^3$ states. So, this approach may cause an exponential blow-up. For example, the complexity of converting a non-deterministic tree automaton into a deterministic tree automaton using the subset construction algorithm is exponential in the states of the resulted deterministic tree automaton.

However, as the expressive power and attractive mathematical properties of tree automaton make itself a natural basis for type systems for tree-structured data (such as XML documents), many early and later research work in the field of subtype-checking is based on tree automata. Those representative research work includes Hosoya et al.'s subtyping algorithm using regular type expressions (Hosoya, Vouillon, & Pierce, 2000) and (Hosoya, Vouillon, & Pierce, 2005), and the XML schema containment checker proposed by Tozawa & Hagiya (Tozawa & Hagiya, 2003).

Hosoya et al. implemented the classical subtyping algorithm in the early prototype implementation of a statically typed programming language called *XDuce* (Hosoya &

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Pierce, 2000). However, a problem came up when they checked the subtyping relationship between two type representations that involve a large degree of sharing. For example, considering the following regular inequality:

$$a^*, b^* <: (a \mid b)^*$$

We can see that both the R.H.S. type and the L.H.S. type share the same alphabet $\{a, b\}$. The L.H.S. type denotes a set of ordered sequences consisting of any number of ***a***s followed by any number of ***b***s, while the R.H.S. type denotes a set of unordered sequences consisting of any number of ***a***s and any number of ***b***s in any order. Obviously, the set denoted by the L.H.S. type is a subset of the set denoted by the R.H.S. type. Therefore, it should be easy to prove that the L.H.S. type is a subtype of the R.H.S. type, no matter what and how big types ***a*** and ***b*** may be. In practice, it is seldom necessary to explore all the states of tree automata as we do in the classical approach.

5.2 Subtyping for Regular Expression Types

To solve the problem, Hosoya et al. proposed a new set-inclusion subtyping algorithm (Hosoya, Vouillon, & Pierce, 2000) and (Hosoya, Vouillon, & Pierce, 2005). Unlike the classical algorithms based on determinization of tree automata, the proposed algorithm checks the inclusion relation by a top-down traversal of the original type expressions. It works as follows: given a pair of types, it checks matching of the top-most type constructors, proceeds to the subcomponents of the types, and repeats the same check recursively until it reaches leaves that require only trivial checks. The main contribution of this top-down algorithm is that it enables many simple optimizations in the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

implementation of a subtyping algorithm. Thus, it decreases the high complexity in many typical cases. In particular, it exploits the reflexivity property in those cases where the input types being compared have a large degree of sharing in their representations. For example, they use $a <: a$, $b <: b$ to prune large parts of the subtype checking for the inequality $a^*, b^* <: (a \mid b)^*$.

In addition, Hosoya et al. (Hosoya, Vouillon, & Pierce, 2000) first proposed the concept of *regular expression types* and defined regular expression types as a natural generalization of DTDs, describing structures of XML documents using regular expression operators, i.e., *, ?, |. Furthermore, they formalize the connection of regular expression types to tree automata. In their work, a tree automaton is a finite mapping from type states to the internal form of regular expression types. Therefore, regular expression types directly correspond to tree automata.

Hosoya et al. claim that their algorithm can be viewed as a variant of Aiken and Murphy's set-inclusion constraint solver (Aiken & Murphy, 1991). However, the domains of these two algorithms are different. Hosoya et al.'s algorithm is applied to a type system for XML processing, while Aiken and Murphy's algorithm focuses on program analysis for optimization. So, several modifications and optimizations have been added to Aiken and Murphy's algorithm before it is applied to check XML schema subtyping. First of all, types in Aiken & Murphy's algorithm can contain free variables and the goal of this algorithm is to obtain a substitution for the variables that satisfies the given set-constraints. Hosoya et al. removed the rules related to free variables from their algorithm. Secondly, Hosoya et al.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

argue that their algorithm is complete, and give the proof of completeness as well as the proofs of soundness and termination of their algorithm in the paper (Hosoya, Vouillon, & Pierce, 2005). Their algorithm, thus, can generate comprehensible error messages in case of type-checking failure, while the completeness is not critical in Aiken and Murphy's algorithm. In addition, they add the notion of *subtagging* to support subtyping between types with different labels. For example, we can have the subtype relation: *student*[Tel*] <: *person*[Tel*], based on such a declaration that the tag *student* is subtag of *person*, i.e. ***subtag*** *student* <: *person*. This feature goes beyond the expressive power of DTDs, but similar to the "substitution groups" mechanism in XML Schema. Last but not less important, Hosoya et al. (Hosoya, Vouillon, & Pierce, 2005) added a number of optimization in the implementation of their subtype-checker. These optimizations are specialized to the subtyping problem that arises in practice in the domain of XML processing. For example, they use set-theoretic observations (see Appendix A) to overcome the difficulties produced by subtype checking the "untagged" union types where the components of a union may have the same outermost label.

Hosoya et al. present the results of some preliminary measurements of their algorithm's practical effects in the paper (Hosoya, Vouillon, & Pierce, 2005). The authors claim that they tried their method on many practical cases and it can check subtyping quite efficiently (less than one second on XDuce applications that involve fairly large types, such as the full DTD for XHTML documents).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.3 Subtyping Using BDD

Another subtyping algorithm worth notice is the XML schema containment checker proposed by Tozawa & Hagiya (Tozawa & Hagiya, 2003). This approach adopts Hosoya et al.'s algorithm (in Section 5.2) to convert XML schemata into non-deterministic tree automata (NTAs) and then uses *semi-implicit* techniques to perform the determinization of NTAs. Briefly speaking, Tozawa & Hagiya's semi-implicit technique means that each subset of the state set of a NTA is encoded by a *binary decision diagram* (BDD) (Bryant, 1986), whereas *implicit* techniques (Clarke, Grumberg & Peled, 1999) usually encode the state set of a NTA with a single BDD. With their semi-implicit technique, Tozawa & Hagiya don't use the expensive operations on tree automata, such as the complement and intersection operations, to explicitly decide whether $L(A) \subseteq L(B)$ holds, or not, but rather they use BDD operations to perform the determination of two NTAs A and B. Tozawa & Hagiya claim that semi-implicit techniques are not used in previous work on the language containment-checking and their algorithm based on these techniques is efficient and can answer problems that cannot be solved by previously known algorithms. They also claim that their technique can directly be applied to the type systems of schemas that can easily be transformed into NTAs, such as regular expression types, RELAX and DTDs. The correctness proof of their algorithm is given in (Tozawa & Hagiya, 2003).

Although both Tozawa et al.'s algorithm and Hosoya et al's algorithm model XML schema by (binary) tree automata, they are quite different in the fact that Hosoya et al's algorithm does not use BDD and is explicit, i.e. it uses set operations on types, which are essentially Boolean operations on tree automata.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To get a better comparison, Tozawa et al. implemented their "semi-implicit" algorithm and the classical algorithm (Comon et al., 1999). Also, they re-implemented Hosoya et al.'s subtyping algorithm in Java (since it was originally implemented in O'Caml). The researchers claim that the result of applying these three algorithms on three experimental examples and one real-world XHTML example shows that their BDD-based algorithm performed well in general, while both Hosoya et al.'s algorithm and the classical algorithm caused blow-up in two test examples. Among these algorithms, the classical algorithm performed the worst. Tozawa et al. applied their subtyping algorithm in the development of a typed XML processing language called *XML Processing Plus Plus*, which is released from IBM alphaWorks. Interested readers can find details at the web site http://www.alphaworks.ibm.com/tech/xmlprocessingplusplus.

5.4 Subtyping Using Antimirov's Containment Calculus

Instead of modeling XML schemas by tree automata, an alternative approach adopts a purely algebraic decision procedure to solve the subtyping problem for XML Schema, without constructing tree automata. That seems to be an interesting contribution since highly-expensive manipulations of tree automata are avoided. This approach uses Antimorov's derivation calculus (Antimirov, 1994) to recursively simplify the type representations of two XML schemas, until they are simple enough to perform a trivial check. The representative work in this branch is Kempa & Linnemann's subtyping algorithm for XML objects (XOBE) (Kempa & Linnemann, 2003) and Hohenadel's subtyping algorithm for the type system of XQuery and XML Schema (Hohenadel, 2003).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To cope with type checking in the *XOBE* project of the University of Lübeck (Kempa & Linnemann, 2003), Antimirov's algorithm for checking subtyping between regular expressions was modified to be applicable for subtype-checking between XML Schemas. One big modification in the XOBE-version subtyping algorithm is that in XOBE, an XML Schema is represented internally by a *regular hedge grammar* and types defined in this schema are represented by *regular hedge expressions,* instead of regular expressions as in Antimirov's algorithm. In their work (Kempa & Linnemann, 2003), a regular hedge grammar G is defined as a tuple (T, N, s, P), where T is a set of terminal symbols (consisting simple types names B and elements names E), N is a set of non-terminal symbols (consisting names of groups and complex types), s is the start expression and P is a set of production rules of form $n \rightarrow r$ with $n \in N$ and r is a regular hedge expression over $T \cup N$. The production rules in the set P have to fulfill two constraints: (1) recursive non-terminals may appear in tail positions only; (2) recursive non-terminals must be preceded by at least one regular hedge expression which does not contain the empty hedge ε. These two constraints ensure regularity.

Regular hedge expressions, in their work, are defined recursively as follows:

- the empty set
- the empty hedge ε
- any simple type name $b \in B$
- any complex type name $n \in N$
- e[r], where e$\in E$ is an element name and r is a regular hedge expression
- r, s where r and s are regular hedge expressions
- r | s where r and s are regular hedge expressions

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Kleene star (r*) where r is a regular hedge expression

The syntax is quite similar to the syntax of the external form of regular expression types. The formal definitions of regular hedge grammar and regular hedge expressions can be found in (Kempa & Linnemann, 2003).

Checking the subtype relationship between two regular hedge expressions is the main task in type checking of XOBE programs. Kempa & Linnemann extend Antimirov algorithm to the regular hedge expressions and the regular hedge grammar case. Kempa & Linnemann (Kempa & Linnemann, 2003) describe their subtyping algorithm as "compared to standard subtyping based on regular tree automata which involve the computation of automata intersection and automata complement, our algorithm is more efficient. Although our algorithm has a potential exponential inefficiency as the automata procedure, there are cases where our algorithm is exponentially faster." They also present some preliminary performance measurements of three XOBE programs using XHTML or WML schema and claim that their subtyping algorithm runs at acceptable speed for these applications.

In addition to XOBE, Antimirov's algorithm was later refined and applied to the construction of a compiler for XQuery, which was a part of the "*Pathfinder*" project at the University of Konstanz (Hohenadel, 2003). The basic idea of Pathfinder version of subtyping algorithm is quite similar to the XOBE one, except that the author avoids to use regular hedge grammars and regular hedge expressions in modeling the type system of XQuery and XML Schema. In the context of his work, types are represented by the so-called *regular expressions*, the concept of which is different from the formal definition of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

regular expressions based on formal language theory.

Although the construction of tree automata is avoided, the subtyping algorithms based on Antimirov's calculus, including ours, still cause a high degree of time complexity in the worst case. The reason is that the simplification process leads to a lower degree of time complexity for checking a single type pair; but to an increased (may be exponential blow-up in the worst case) number of type pairs derived from the original type expressions which are in need of subtyping check.

5.5 Comparison

In our survey of related work in the field of subtype-checking, we find that the subtyping problem for XML schemas seems to be able to be divided into two sub-problems. One is how to define formalism for XML Schema. The other is how to develop an efficient algorithm to check the subtyping relationship for XML Schemas in real-world practice. The first sub-problem focuses on the expressive power of a type formalism (or type representation). That is, appropriate type formalism should be sufficiently expressive for modeling XML schemas. Additionally, the subtyping problem based on the type formalism should be decidable. The second sub-problem emphasizes the efficiency of a subtyping algorithm. In this section, we compare our work with other research work in the same field from these two points of view.

Many early proposed type formalisms for specifying XML Schema or DTD, e.g. *Regular Expression Types* (Hosoya, Vouillon, & Pierce, 2000), *Regular Hedge Expressions* (Kempa

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

& Linnemann, 2003), "*Regular Expressions*" (Hohenadel, 2003), are based on context-free grammars but restricted by adding some syntactic conditions to ensure regularity (see Section 2.1). These conditions are necessary since the decision problem for inclusion between context-free languages denoted by context-free grammars is undecidable (Hopcroft & Ullman, 1979, Theorem 8.12). They need to impose additional restrictions to reduce the power of context free grammars so that the type formalisms for XML schemas correspond to regular tree languages. To illustrate this, we borrow an example from (Hohenadel, 2003). Suppose that type T1 is defined by recursion in the following production:

$$\mathrm{T1} \rightarrow \mathrm{T1} \cdot \mathrm{integer} \mid \varepsilon$$

The derivation of T1 will lead to an endless recursion as follows.

$lf(\mathrm{T1}) = lf(\mathrm{T1}\cdot \mathrm{integer} \mid \varepsilon)$ by the *Unfold* rule

$= lf(\mathrm{T1}\cdot \mathrm{integer}) \cup lf(\varepsilon)$ by rule LF6 in Section 3.4.1

$= lf(\mathrm{T1}\cdot \mathrm{integer})$ by rule LF2 in Section 3.4.1, $lf(\varepsilon)=\varnothing$

$= (lf(\mathrm{T1}) \odot \mathrm{integer}) \cup lf(\mathrm{integer})$ since T1 is nullable, by rule LF5b

$= (lf(\mathrm{T1}\cdot \mathrm{integer} \mid \varepsilon) \odot \mathrm{integer}) \cup lf(\mathrm{integer})$ by the *Unfold* rule

$= (((lf(\mathrm{T1}) \odot \mathrm{integer}) \cup lf(\mathrm{integer})) \odot \mathrm{integer}) \cup lf(\mathrm{integer})$

$= (((lf(\mathrm{T1}\cdot \mathrm{integer} \mid \varepsilon) \odot \mathrm{integer}) \cup lf(\mathrm{integer})) \odot \mathrm{integer}) \cup lf(\mathrm{integer})$

......

Because the production of T1 contains recursive occurrences of this type at the beginning of the right-hand-side expression, an endless application of the *lf* rules to any of these

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

recursive occurrences is unavoidable.

The formal framework we proposed for XML Schema is based on regular tree grammars. The inclusion problem between regular tree languages is known to be decidable (Seidl, 1999). So, we don't need any constraints on recursive types as in the Hosoya et al.'s algorithm (Hosoya, Vouillon, & Pierce, 2000), XOBE (Kempa & Linnemann, 2003) and Pathfinder subtyping algorithms (Hohenadel, 2003). Moreover, the technique of representing XML Schemas based on regular tree grammars provides a formal framework for XML Schema using formal language theory. That is, this framework helps to describe, compare XML Schemas in a rigorous manner (e.g., check for equivalence or subtype relationship), and facilitates the implementation of a subtyping algorithm. Many previous modeling languages for XML schemas are not based on formal language theory. Some of them are not rigorously defined; others are lack of sufficient expressiveness in a larger problem domain.

As we mentioned before, the formal system for XML Schema subtyping we propose in this thesis is based on Antimirov's derivation calculus. Like other work based on the same calculus (Kempa & Linnemann, 2003) (Hohenadel, 2003), our subtyping algorithm has the following advantages and disadvantages in comparison with tree-automata-based approaches (Hosoya, Vouillon, & Pierce, 2000) (Tozawa & Hagiya, 2003):

Advantages:

- Avoiding the construction of tree automata, and thus the algorithm is simple;
- Decreasing the time complexity to check a single regular inequality.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Disadvantage:

- Increasing the number of regular inequalities in need of subtype-checking. In the worst case, there exists an exponential blow-up in the number of partial derivatives of the right-hand-side type of a regular inequality.

Antimirov's derivation calculus provides a purely algebraic decision procedure to solve the subtyping problem. Thus, in comparison with those algorithms based on tree automata, the algorithms based on Antimirov's approach, including ours, seem simpler than the classical one in typical cases of current XML processing, though there is still a high complexity in the worst case. To make it clear, let's look at the following example. Suppose we want to check

$$X <: X|Y$$

where X → $l(a)$ and Y → $(a|b)^*a(a|b)^{n-1}$. Here, we use $(a|b)^{n-1}$ as a shorthand for (n-1) times of concatenation of $(a|b)$. Similarly, $(a|b)^*$ denotes 0 or more times of concatenation of $(a|b)$.

Using the classical approach, we need to construct tree automata for X and X|Y. It is known that the minimal deterministic finite automaton for Y has 2^n states (Perrin, 1990). So, it cost $O(2^m)$ where $m=2^n$ to check the subtyping relationship based on operations of tree automata. Obviously, this approach is very expensive.

However, using Antimirov's approach, only four simpler regular inequalities (listed below) reduced from the input inequality X <: X|Y and it takes constant time to check each of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

them.

$$a <: \emptyset \quad \vee \quad \varepsilon <: \varepsilon$$
$$a <: a \quad \vee \quad \varepsilon <: \emptyset$$

Among those algorithms based on Antimirov's approach, our subtyping algorithm is very similar to the XOBE and Pathfinder subtyping algorithms. We share the basic ideas (originally conceived by Antimirov) to check and simplify a regular inequality. Therefore, the computations of linear forms of type expressions, leading names of type expressions, partial derivatives of type expressions, and partial derivatives of regular inequalities are similar. However, since the formal frameworks proposed for modeling XML Schema in the XOBE and Pathfinder subtyping algorithms are different from ours, we make some modifications to make Antimirov's calculus suitable for our type formalism. Also, we redefine some basic concepts, and adopt rigorously-defined concepts and functions in standard textbooks, e.g., the definition of partial derivatives of a type expression and the *First* function. In the implementation of the subtyping algorithm, we add some heuristics not presented in the XOBE and Pathfinder subtyping algorithms, to speed up the subtype-check for XML Schemas.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

6.1 Main Contributions

The main contributions of this thesis in the field of XML Schema subtyping are given as follows.

- We identify the appropriate language to model XML Schema, and formally define the language based on Regular Tree Grammar. In the past, there have been substantial studies on the formal models of XML Schema or DTD, including Regular Expression Types (Hosoya, Vouillon, & Pierce, 2000), Regular Hedge Expressions (Kempa & Linnemann, 2003), and so on. However, some of them are not rigorously defined. Others lack sufficient expressiveness to model XML Schemas. The technique of representing XML Schemas based on regular tree grammars provides a formal framework for XML Schema using formal language theory. Hence, this framework helps to describe, compare XML Schemas in a rigorous manner (e.g., check for equivalence or subtype relationship), and facilitates the implementation of the subtyping algorithm.

- We present a formal system for the subtype-checking of our language, which is based on Antimirov's derivation calculus (Antimirov, 1994) for regular expressions and its extensions to regular hedge expressions (Kempa & Linnemann, 2003) (Hohenadel, 2003). This thesis formalizes and rebuilds the algorithm for regular tree grammars, which is very close to that for regular hedge grammars.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- We add modifications and heuristics so that the subtyping algorithm for XML Schema. Although a high complexity is required to check subtyping between XML schemas in the worst case (because the number of simplified type pairs needed to check may be exponential in the number of partial derivatives of the right-hand-side type of a regular inequality), by choosing appropriate representations and applying a few domain-specific heuristics, it is expected to improve the time complexity in those typical cases that we encounter most often in XML processing.

- We have completed the implementation of the subtyping algorithm for XML Schema. It performs well in the subtype checking for simple XML Schemas, containing user-defined simple and complex types, types derived by restriction and extension, and all built-in simple data types defined in XML Schema.

6.2 Limitations and Future Work

Comparing to DTD, XML Schema has many advanced features, such as namespaces, reusable groups, identity constraints, substitution groups, redefinition, and so on. However, none of the existing schema modeling languages, including ours, can capture all the features of XML Schema (see details in Section 2.5). Currently, we are conducting a survey of existing XML Schemas on the Web in order to identify scarcely-used features. In this way, we can tailor our modeling language to what is needed in practice.

The efficiency of the subtyping algorithm is anther important motivation in our future

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

work. Although subtyping algorithms have been implemented in the compiler construction of typed languages (such as XQuery, XDuce, etc.), there are still many technical challenges in this area. The main difficulty that we have to face in this field is that the decision problem of subtyping for XML types is algorithmically difficult because a high complexity (EXPTIME) is generally required to check subtyping between XML schemas in the worst case. Since the subtyping algorithm is not efficient, it is not able to search subtypes from a large data set. Hence we will extend the subtyping facility with more efficient searching methods borrowed from information retrieval techniques, in order to build an effective schema search engine.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDICES

APPENDIX A

A Set-theoretic Observation

The following explains the set-theoretic observation proposed by Hosoya et al. (Hosoya, Vouillon, & Pierce, 2005).

Suppose we want to check the following subtyping relationship:

$$\langle A, B\rangle <: \langle C_1, D_1\rangle \mid \langle C_2, D_2\rangle \mid \langle C_3, D_3\rangle \qquad \text{(A-1)}$$

In general, a cross product $X \times Y$ is equal to $\langle X \times \tau\rangle \cap \langle \tau \times Y\rangle$, where τ is the maximal type denoting the set of all ground types. Then, $\langle C_1, D_1\rangle \mid \langle C_2, D_2\rangle \mid \langle C_3, D_3\rangle$ is equal to:

$$\begin{aligned} &\langle C_1,\tau\rangle \cap \langle \tau, D_1\rangle \\ \mid\ &\langle C_2,\tau\rangle \cap \langle \tau, D_2\rangle \\ \mid\ &\langle C_3,\tau\rangle \cap \langle \tau, D_3\rangle \end{aligned} \qquad \text{(A-2)}$$

Using distributivity of intersections over unions, we turn the disjunctive form of (A-2) to the following conjunctive form:

$$\begin{aligned} &\langle C_1,\tau\rangle \mid \langle C_2,\tau\rangle \mid \langle C_3,\tau\rangle \\ \cap\ &\langle \tau, D_1\rangle \mid \langle C_2,\tau\rangle \mid \langle C_3,\tau\rangle \\ \cap\ &\langle C_1,\tau\rangle \mid \langle \tau, D_2\rangle \mid \langle C_3,\tau\rangle \\ \cap\ &\langle C_1,\tau\rangle \mid \langle C_2,\tau\rangle \mid \langle \tau, D_3\rangle \\ &\ldots\ldots \end{aligned} \qquad \ldots\ldots\ \text{(A-3)}$$

In each clause of (A-3), if C_i appears, then the corresponding argument D_i does not

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

appear, and vice versa. So a short form for one clause of (A-3) is as follows:

$$<|_{i \in I} C_i, \tau> \mid <\tau, |_{j \in \bar{I}} D_j> \quad \text{(A-4)}$$

where I is a subset of {1, 2, 3} and $\bar{I}$ is the complement of I, i.e., $\{1, 2, 3\} \setminus I$. Since the conjunctive form (A-3) is the intersection of clauses of form (A-4) for all subset I of {1, 2, 3}, the inequality of (A-1) can be rewritten as, for each subset I of {1, 2, 3}:

$$<A, B> \;<:\; <|_{i \in I} C_i, \tau> \mid <\tau, |_{j \in \bar{I}} D_j> \quad \text{(A-5)}$$

Let $C = |_{i \in I} C_i$ and $D = |_{j \in \bar{I}} D_j$, then inequality (A-5) is transformed into:

$$<A, B> \;<:\; <C, \tau> \mid <\tau, D> \quad \text{(A-6)}$$

It suffices to test the following two inequalities:

$$(A <: C) \lor (B <: D) \quad \text{(A-7)}$$

To prove this, suppose $<A, B> \;<:\; <C, \tau> \mid <\tau, D>$ holds and the negation of (A-7) is true, i.e., neither A <: C nor B <: D hold. We can find a tree $t_1 \in L(A)$ but $t_1 \notin L(C)$, and another tree $t_2 \in L(B)$ but $t_2 \notin L(D)$. Thus, $<t_1, t_2> \in L(<A, B>)$. However, neither $<t_1, t_2> \in L(<C, \tau>)$ nor $<t_1, t_2> \in L(<\tau, D>)$. Then $L(<A, B>) \not\subset L(<C, \tau>)$ and $L(<A, B>) \not\subset L(<\tau, D>)$. That is, $L(<A, B>) \not\subset L(<C, \tau>) \mid L(<\tau, D>)$. So, $<A, B> \;<:\; <C, \tau> \mid <\tau, D>$ doesn't hold. That contracts the assumption. The other direction, i.e., if A <: C or

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$B <: D$, then $\langle A, B\rangle <: \langle C, \tau\rangle \mid \langle \tau, D\rangle$, is obviously true. Therefore, $\langle A, B\rangle <: \langle C, \tau\rangle \mid \langle \tau, D\rangle \Leftrightarrow (A <: C) \vee (B <: D)$. Q.E.D.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX B

Logfile for Example 3

```
input inequality:   t <: r

-------------------------- Call 0 Starts----------------------------

Check:   t <: r

(1) t:a[s]|string
(1) s:a[t]
(1) r:a[r]|string

type1: This is NamedType.
        Type name: t
        Type definition: a[s]|string

type2: This is NamedType.
        Type name: r
        Type definition: a[r]|string

Unfold type1 and type2.
Check: a[s]|string   <:  a[r]|string

-------------------------- Call 1 Starts----------------------------

Check:   a[s]|string <: a[r]|string

(1) s:a[t]
(1) t:a[s]|string
(1) r:a[r]|string

type1: This is AlternationType.
        child1:  This is NodeType.
        nodeName:  a
        Subtree:  This is NamedType.
        Type name: s
        Type definition: a[t]

        child2:  This is PrimitiveType.
        Simple type:  string

type2: This is AlternationType.
        child1:  This is NodeType.
        nodeName:  a
        Subtree:  This is NamedType.
        Type name: r
        Type definition: a[r]|string

        child2:  This is PrimitiveType.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        Simple type:  string


Result:
  (1)check:   a[s]  <:  a[r]|string
&
  (2)check:   string  <:  a[r]|string


-------------------------- Call 2 Starts----------------------------

Check:   a[s] <: a[r]|string

(1) s:a[t]
(1) t:a[s]|string
(1) r:a[r]|string

type1: This is NodeType.
        nodeName:  a
        Subtree:  This is NamedType.
        Type name: s
        Type definition: a[t]


type2: This is AlternationType.
        child1:  This is NodeType.
        nodeName:  a
        Subtree:  This is NamedType.
        Type name: r
        Type definition: a[r]|string

        child2:  This is PrimitiveType.
        Simple type:  string

The leadingName(s) of type1 are: a

Linear form of type1: <a[s], empty>

Linear form of type2: <a[r], empty>  <string, empty>

Partial derivatives of type1 by the leading name "a": <s, empty>

Partial derivatives of type2 by the leading name "a": <r, empty>

The derivation produces 2x2 simpler inequalities:
(1)s <: none    or    empty <: empty
(2)s <: r    or    empty <: none

-------------------------- Call 3 Starts----------------------------

Check:   s <: none

(1) s:a[t]
(1) t:a[s]|string

type1: This is NamedType.
        Type name: s
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        Type definition: a[t]

type2: This is NoneType.

Processing ends when type2 is NoneType.

Result:   false

-------------------------- Call 3 Ends----------------------------


-------------------------- Call 4 Starts----------------------------

Check:   empty <: empty


type1: This is EmptyType.

type2: This is EmptyType.

Processing ends when type1 and type2 are structurally equal.

Result:   true

-------------------------- Call 4 Ends----------------------------


-------------------------- Call 5 Starts----------------------------

Check:   s <: r

(1) s:a[t]
(1) t:a[s]|string
(1) r:a[r]|string

type1: This is NamedType.
        Type name: s
        Type definition: a[t]

type2: This is NamedType.
        Type name: r
        Type definition: a[r]|string

Unfold type1 and type2.
Check: a[t]   <:  a[r]|string

-------------------------- Call 6 Starts----------------------------

Check:   a[t] <: a[r]|string

(1) t:a[s]|string
(1) s:a[t]
(1) r:a[r]|string

type1: This is NodeType.
        nodeName:  a
        Subtree:  This is NamedType.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
        Type name: t
        Type definition: a[s]|string


type2: This is AlternationType.
        child1:  This is NodeType.
        nodeName:  a
        Subtree:  This is NamedType.
        Type name: r
        Type definition: a[r]|string

        child2:  This is PrimitiveType.
        Simple type:  string

The leadingName(s) of type1 are: a

Linear form of type1: <a[t], empty>

Linear form of type2: <a[r], empty>  <string, empty>

Partial derivatives of type1 by the leading name "a": <t, empty>

Partial derivatives of type2 by the leading name "a": <r, empty>

The derivation produces 2x2 simpler inequalities:
(1)t <: none    or    empty <: empty
(2)t <: r    or    empty <: none

-------------------------- Call 7 Starts----------------------------

Check:   t <: none

(1) t:a[s]|string
(1) s:a[t]

type1: This is NamedType.
        Type name: t
        Type definition: a[s]|string

type2: This is NoneType.

Processing ends when type2 is NoneType.

Result:   false

-------------------------- Call 7 Ends----------------------------


-------------------------- Call 8 Starts----------------------------

Check:   empty <: empty


type1: This is EmptyType.

type2: This is EmptyType.
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Processing ends when type1 and type2 are structurally equal.

Result: true

-------------------------- Call 8 Ends----------------------------

-------------------------- Call 9 Starts----------------------------

Check: t <: r

(1) t:a[s]|string
(1) s:a[t]
(1) r:a[r]|string

type1: This is NamedType.
Type name: t
Type definition: a[s]|string

type2: This is NamedType.
Type name: r
Type definition: a[r]|string

Assumption set:
(1) t<:r
(2) a[s]|string<:a[r]|string
(3) a[s]<:a[r]|string
(4) s<:r
(5) a[t]<:a[r]|string
Processing ends when t<:r is already in the set of assumption or can be derived from the transitivity property of subtyping.

Result: true

-------------------------- Call 9 Ends----------------------------

-------------------------- Call 10 Starts----------------------------

Check: empty <: none

type1: This is EmptyType.

type2: This is NoneType.

Processing ends when type2 is NoneType.

Result: false

-------------------------- Call 10 Ends----------------------------

Result: true

-------------------------- Call 6 Ends----------------------------

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Result: true

-------------------------- Call 5 Ends----------------------------

-------------------------- Call 11 Starts----------------------------

Check:   empty <: none


type1: This is EmptyType.

type2: This is NoneType.

Processing ends when type2 is NoneType.

Result:   false

-------------------------- Call 11 Ends----------------------------


Result: true

-------------------------- Call 2 Ends----------------------------


-------------------------- Call 12 Starts----------------------------

Check:   string <: a[r]|string

(1) r:a[r]|string

type1: This is PrimitiveType.
        Simple type:  string

type2: This is AlternationType.
        child1:  This is NodeType.
        nodeName:  a
        Subtree:  This is NamedType.
        Type name: r
        Type definition: a[r]|string

        child2:  This is PrimitiveType.
        Simple type:  string

The leadingName(s) of type1 are: string

Linear form of type1: <string, empty>

Linear form of type2: <a[r], empty>  <string, empty>

Partial derivatives of type1 by the leading name "string": <empty,
empty>

Partial derivatives of type2 by the leading name "string": <empty,
empty>
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Processing ends when the partial derivatives of type1 and type2 by the leading name "string" are identical.

Result: true

-------------------------- Call 12 Ends----------------------------

Result: true

-------------------------- Call 1 Ends----------------------------

Result: true

-------------------------- Call 0 Ends----------------------------

Final result: true

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

REFERENCES

Aho, A.V., Sethi, R., & Ullman, J.D. (1988). Compilers: Principles, Techniques, and Tools. Addison-Wesley.

Aiken, A., & Murphy, B. R. (1991). Implementing regular tree expressions. *Proceedings of the fifth ACM conference on functional programming languages and computer architecture,* 427-447.

Antimirov, V. (1994). Rewriting regular inequalities (extended abstract). *Fundamentals of Computation Theory, Lecture Notes in Computer Science, 965*, 116-125.

Antimirov, V. (1996). Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science, 155*, 291-319.

Biron, P. V., & Malhotra, A. (Eds). (2001). XML Schema Part 2: Datatypes. W3C Recommendation, May. http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., & Maler, E. (2000). Extensible markup language (XML^{TM}). http://www.w3.org/XML/.

Bryant , R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions and Computers*, C-35(8), 677-691.

Clark, J., & Murata, M. (2001). RELAX NG.. http://www.relaxng.org.

Clarke, E. M., Grumberg, O., & Peled, D. A. (1999). Model checking. MIT press.

Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S. & Tommasi M. (2002). Tree automata techniques and applications. Draft book. http://www.grappa.univ-lille3.fr/tata.

Fallside, D. C., & Walmsley, P. (Eds). (2004). XML Schema Part 0: Primer Second Edition. W3C Recommendation. http://www.w3.org/TR/2004/REC-xmlschema-0-

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

20041028/.

Gècseg, F., & Steinby, M. (1997) Tree languages, in: Rozenberg, G., Salomaa, A. (Eds), Handbook of Formal Language (vol. 3), Words, B., Springer-Verlag, Berlin Heidelberg, 1-68.

Hohenadel, S. A. (2003). Subtyping for regular tree types: A Java-based Implementation. Master's Thesis, University of Konstanz, Germany.

Hopcroft, J. E., & Ullman, J. D. (1979). Introduction to automata theory, languages, and computation. Addison-Wesley.

Hosoya, H., & Pierce, B. C. (2000). XDuce: A typed XML processing language (preliminary report). *Proceedings of the 3rd International Workshop on the Web and Databases (WebDB2000)*, Lecture Notes in Computer Science, *1997*, 226-244. Also appeared in *ACM Transactions on Internet Technology* (2003), 3(2), 117-148.

Hosoya, H., Vouillon, J., & Pierce, B. C. (2000). Regular expression types for XML. *Proceedings of the fifth ACM SIGPLAN international conference on functional programming ICFP '00, 35*(9) 11-22.

Hosoya, H., Vouillon, J., & Pierce, B. C. (2005). Regular expression types for XML. *ACM transactions on programming languages and systems (TOPLAS), 27*(1) 46-90.

Kempa, M., & Linnemann, V. (2003). Type checking in XOBE. *Proceedings of datenbanksysteme für business, technologie und web, BTW '03, LNI,* 227-246.

Klarlund, N., Mller, A., & Schwartzbach, M. I. (2000). DSD: A schema language for XML. http://www.brics.dk/DSD/.

Kuper, G. M., & Siméon, J. (2001). Subsumption for XML Types. *Proceedings of the 8th International Conference on Database Theory.* Lecture Notes in Computer Science, 1973, 331-345.

Lee, D., Mani, M., & Murata, M. (2000). Reasoning about XML schema languages using

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

formal language theory. Technical report, IBM Almaden Research Center, RJ#10197, Log#95071. http://www.cs.ucla.edu/~dongwon/paper/.

Murata, M. (2001). RELAX (REgular LAnguage description for XML). http://www.xml.gr.jp/relax/.

Murata, M., Lee, D., Mani, M., & Kawaguchi, K. (2005). Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology (TOIT), 5*(4), 660-704.

Neven, F. (2002). Automata theory for XML researchers. *ACM SIGMOD Record, 31*(3), 39-46.

Perrin, D. (1990). Finite automata. In J. van Leeuwen, A. Meyer, M. Nivat, M. Paterson, and D. Perrin, editors, *Handbook of Theoretical Computer Science*, vol.(B). Elsevier Science Publishers, Amsterdam; and MIT Press.

Seidl, H. (1990). Deciding equivalence of finite tree automata. *SIAM Journal on Computing, 19*(3), 424-437.

The Apache XML Project. (2001). Xerces Java Parser (version 1.4.4). http://xml.apache.org/xerces-j/index.html.

Tozawa, A., & Hagiya, M. (2003). XML Schema containment checking based on semi-implicit techniques. *Implementation and Application of Automata, 8th International Conference, CIAA 2003, Santa Barbara, California, USA, July 16-18.* Lecture Notes on Computer Science, 2759, 213-225.

Walmsley, P. (2002). Definitive XML Schema. Prentice Hall PTR. http://www.datypic.com/books/DefXMLSchema.

W3C. (2004) Document Object Model (DOM) technical reports. http://www.w3.org/DOM.DOMTR.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

VITA AUCTORIS

NAME	Yun Li
PLACE OF BIRTH	Kunming, Yunnan, China
YEAR OF BIRTH	1970
EDUCATION	Shanghai Jiao Tong University, Shanghai, China 1988-1992 B.Eng.
	University of Windsor, Windsor, Ontario 2002-2004 B.C.S.
	University of Windsor, Windsor, Ontario 2004-2006 M.S.C.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.