

Institut National Polytechnique de Grenoble

# Logics for XML

Pierre GENEVÈS

Thesis presented in partial fulfillment of the requirements for the degree of Ph.D. in Computer Science and Software Systems from the Institut National Polytechnique de Grenoble. Dissertation prepared at the Institut National de Recherche en Informatique et Automatique, Montbonnot, France. Thesis defended on the 4<sup>th</sup> of December 2006.

Board of examiners:

Giorgio GHELLI	Referee
Denis LUGIEZ	Referee
Makoto MURATA	Referee
Christine COLLET	Examiner
Vincent QUINT	Ph.D. advisor
Nabil LAYAÏDA	Ph.D. co-advisor



# Abstract

---

This thesis describes the theoretical and practical foundations of a system for the static analysis of XML processing languages. The system relies on a fixpoint temporal logic with converse, derived from the  $\mu$ -calculus, where models are finite trees. This calculus is expressive enough to capture regular tree types along with multi-directional navigation in trees, while having a single exponential time complexity. Specifically the decidability of the logic is proved in time  $2^{O(n)}$  where  $n$  is the size of the input formula.

Major XML concepts are linearly translated into the logic: XPath navigation and node selection semantics, and regular tree languages (which include DTDs and XML Schemas). Based on these embeddings, several problems of major importance in XML applications are reduced to satisfiability of the logic. These problems include XPath containment, emptiness, equivalence, overlap, coverage, in the presence or absence of regular tree type constraints, and the static type-checking of an annotated query.

The focus is then given to a sound and complete algorithm for deciding the logic, along with a detailed complexity analysis, and crucial implementation techniques for building an effective solver. Practical experiments using a full implementation of the system are presented. The system appears to be efficient in practice for several realistic scenarios.

The main application of this work is a new class of static analyzers for programming languages using both XPath expressions and XML type annotations (input and output). Such analyzers allow to ensure at compile-time valuable properties such as type-safety and optimizations, for safer and more efficient XML processing.



# Résumé

---

Cette thèse présente les fondements théoriques et pratiques d'un système pour l'analyse statique de langages manipulant des documents et données XML. Le système s'appuie sur une logique temporelle de point fixe avec programmes inverses, dérivée du  $\mu$ -calcul modal, dans laquelle les modèles sont des arbres finis. Cette logique est suffisamment expressive pour prendre en compte les langages réguliers d'arbres ainsi que la navigation multidirectionnelle dans les arbres, tout en ayant une complexité simplement exponentielle. Plus précisément, la décidabilité de cette logique est prouvée en temps  $2^{O(n)}$  où  $n$  est la taille de la formule dont le statut de vérité est déterminé.

Les principaux concepts de XML sont traduits linéairement dans cette logique. Ces concepts incluent la navigation et la sémantique de sélection de noeuds du langage de requêtes XPath, ainsi que les langages de schémas (incluant DTD et XML Schema). Grâce à ces traductions, les problèmes d'importance majeure dans les applications XML sont réduits à la satisfaisabilité de la logique. Ces problèmes incluent notamment l'inclusion, la satisfaisabilité, l'équivalence, l'intersection, le recouvrement des requêtes, en présence ou en l'absence de contraintes régulières d'arbres, et le typage statique d'une requête annotée.

Un algorithme correct et complet pour décider la logique est proposé, accompagné d'une analyse détaillée de sa complexité computationnelle, et des techniques d'implantation cruciales pour la réalisation d'un solveur efficace en pratique. Des expérimentations avec l'implantation complète du système sont présentées. Le système apparaît efficace et utilisable en pratique sur plusieurs scénarios réalistes.

La principale application de ce travail est une nouvelle classe d'analyseurs statiques pour les langages de programmation utilisant des requêtes XPath et des types réguliers d'arbres. De tels analyseurs permettent de s'assurer, au moment de la compilation, de propriétés importantes comme le typage correct des programmes ou leur optimisation, pour un traitement plus sûr et plus efficace des données XML.



# Preface

---

This manuscript presents my research work done at the *Institut National de Recherche en Informatique et Automatique* (INRIA Rhône-Alpes, France), from November 2003 to September 2006, within the WAM research project. The work was supported by a personal Ph.D. grant from the french ministry for research (*Ministère délégué à la Recherche*).

This manuscript focuses on presenting the main results obtained for the static analysis of XML specifications using logical formalisms. The list of the main articles and communications I have authored or co-authored during my Ph.D. thesis follows. Some results are not presented in this manuscript (in particular, during this period I have spent several months at IBM T.J. Watson Research Center, New York, United States, working on scalable runtime XML processing architectures, for which I received an IBM invention achievement award).

## Main Publications

- [1] Pierre Genevès, Nabil Layaïda, and Alan Schmitt. Efficient Static Analysis of XML Paths and Types. To appear in *PLDI'07: Proceedings of the 2007 ACM Conference on Programming Language Design and Implementation*, San Diego, California, USA, June 2007. ACM Press.
- [2] Pierre Genevès and Nabil Layaïda. A system for the static analysis of XPath. *ACM Transactions on Information Systems (TOIS)*, 24(4), October 2006.
- [3] Pierre Genevès and Nabil Layaïda. Deciding XPath containment with MSO. To appear in *Elsevier Data & Knowledge Engineering (DKE)*, 2007.
- [4] Pierre Genevès and Nabil Layaïda. Comparing XML Path Expressions. In *DocEng'06: Proceedings of the 2006 ACM Symposium on Document Engineering*, pages 65–74, Amsterdam, The Netherlands, October 2006. ACM Press.
- [5] Pierre Genevès and Kristoffer Høgsbro Rose. Compiling XPath for streaming access policy. In *DocEng '05: Proceedings of the 2005 ACM*

*Symposium on Document Engineering*, pages 52–54, Bristol, UK, November 2005. ACM Press.

- [6] Pierre Genevès and Jean-Yves Vion-Dury. Logic-based XPath optimization. In *DocEng'04: Proceedings of the 2004 ACM Symposium on Document Engineering*, pages 211–219, Milwaukee, Wisconsin, USA, October 2004. ACM Press.
- [7] Pierre Genevès and Jean-Yves Vion-Dury. XPath formal semantics and beyond: A Coq-based approach. In *TPHOLS '04: Emerging Trends Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics*, pages 181–198, Salt Lake City, Utah, USA, August 2004. University Of Utah.
- [8] Kristoffer Høgsbro Rose and Pierre Genevès. Optimization of XPath expressions for evaluation upon streaming XML data, *IBM Research Patent Filed*, May 2004. This patent was awarded an invention achievement award, given by Samuel J. Palmisano (chairman of IBM Corporation) in July 2004.

# Acknowledgements

---

I would like to take this opportunity to thank the many people who have contributed either directly or indirectly to the development of this thesis.

My acknowledgements first go to Vincent Quint, who accepted me as a PhD candidate in his team, and provided me with a high quality research environment. I also thank him for always having had confidence in me, and letting me freely choose my research directions and the way of investigating them. In his team I have met my co-advisor and friend Nabil Layaida with whom I have enjoyed sharing many happy moments doing research. I am deeply indebted to Nabil for his availability, continued support, and, further, for having given me the taste of research.

Giorgio Ghelli, Denis Lugiez, and Makoto Murata honoured me by accepting the role of referee for this dissertation. I would like to thank them for accepting this task. I also thank Christine Collet for accepting the role of examiner for this dissertation.

I am grateful to Alan Schmitt for his insights in the enjoyable collaboration from which the two final chapters of this dissertation benefitted; and since he is very pleasant to work with. I would like to thank Benjamin C. Pierce who was at the origin of my meeting and subsequent collaboration with Alan, following our discussion during a visit at INRIA.

I would like to thank Bob Schloss for giving me several opportunities to join his research team at IBM Watson. I am grateful to Kristoffer H. Rose for being my mentor during my summers spent there.

I also thank Akihiko Tozawa for fruitful discussions by email, Frédéric Lang for helpful discussions, and Jean-Yves Vion-Dury for kindly introducing me to research during my first year.

Finally, I would like to thank all the people from the B aisle of the INRIA building in Montbonnot, who are definitely responsible for the sympathetic and enthusiastic research atmosphere at INRIA Rhône Alpes.

On a more personal note, I would like to thank my parents and my brother for their unconditional support. I also thank my friends for their continuing support and everything else. Thank you all.



# Table of Contents

<b>Title Page</b>	<b>i</b>
<b>Abstract</b>	<b>iii</b>
<b>Preface</b>	<b>vii</b>
<b>Acknowledgements</b>	<b>ix</b>
<b>Table of Contents</b>	<b>xi</b>
<b>List of Figures</b>	<b>xv</b>
<b>List of Notations</b>	<b>xvii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation and Objectives . . . . .	1
1.1.1 XML Documents and Schemas . . . . .	1
1.1.2 XPath . . . . .	3
1.1.3 Static Type-Checking . . . . .	5
1.1.4 Research Challenges . . . . .	5
1.2 Overview of this Dissertation . . . . .	6
1.2.1 Applications . . . . .	6
1.2.2 Outline . . . . .	6
<b>2 Foundations of XML Processing</b>	<b>11</b>
2.1 Trees and Tree Types . . . . .	11
2.1.1 Finite Trees and Hedges . . . . .	11
2.1.2 Schema Languages and Regular Tree Types . . . . .	12
2.1.3 Binary Tree Types . . . . .	15
2.1.4 Finite Tree Automata . . . . .	15
2.2 Queries . . . . .	20
2.2.1 Syntax of XPath Expressions . . . . .	21
2.2.2 XPath Denotational Semantics . . . . .	21
2.3 Logical Formalisms: Two Yardsticks . . . . .	23
2.4 First Order Logic . . . . .	24
2.5 Monadic Second-Order Logic . . . . .	25
2.5.1 Preliminary Definitions . . . . .	25
2.5.2 WS2S Formulas . . . . .	26
2.5.3 WS2S Semantics . . . . .	27

TABLE OF CONTENTS

---

2.5.4	Equivalence of WS2S and FTA	27
2.5.5	From Formulas to Automata	28
2.5.6	WS2S Complexity	30
2.6	Temporal Logics	32
2.6.1	FO Relatives	32
2.6.2	MSO Relatives	32
2.7	Systems for XML Type-Checking	33
2.7.1	Formulations of the Static Validation Problem	33
2.7.2	Inverse Type Inference with Tree Transducers	33
2.7.3	XDuce, CDuce, Xstatic	34
2.7.4	Symbolic XML Schema Containment	35
2.7.5	XJ	35
2.7.6	Approximated Approaches for XSLT	35
2.7.7	Path Correctness for $\mu$ XQ Queries	36
2.8	The Spatial Logic Perspective	36
2.8.1	The Sheaves Logic	37
<b>3</b>	<b>Monadic Second-Order Logic for XML</b>	<b>41</b>
3.1	Introduction	41
3.2	Representation of XML Trees	41
3.3	Interpretation of XPath Queries	42
3.3.1	Navigation and Recursion	43
3.3.2	Logical Composition of Steps	44
3.3.3	Formulating XPath Containment	46
3.3.4	Soundness and Completeness	47
3.4	Complexity Analysis and Optimization	49
3.4.1	Optimization Based on Guided Tree Automata	50
3.5	Implementation and Experiments	55
3.6	Outcome	56
<b>4</b>	<b>XML and the Modal <math>\mu</math>-Calculus</b>	<b>59</b>
4.1	Introduction	59
4.2	The $\mu$ -Calculus	59
4.3	Kripke Structures and XML Trees	62
4.4	XPath Embedding	63
4.4.1	Logical Interpretation of Axes	63
4.4.2	Logical Interpretation of Expressions	64
4.4.3	Correctness and Complexity	66
4.5	Translation of Regular Tree Languages	69
4.6	Solving XML Decision Problems	70
4.7	Complexity Analysis and Implementation Principles	72
4.8	Outcome	74
<b>5</b>	<b>A Fixpoint Modal Logic with Converse for XML</b>	<b>77</b>
5.1	Introduction	77
5.2	Focused Trees	77
5.3	Formulas of the Logic	78
5.4	Translations of XML Concepts	83
5.4.1	XPath Embedding	84
5.4.2	Embedding Regular Tree Languages	85

---

<b>6</b>	<b>Satisfiability-Testing Algorithm</b>	<b>87</b>
6.1	Introduction	87
6.2	Preliminary Definitions	87
6.3	The Algorithm	89
6.3.1	Example Run of the Algorithm	90
6.4	Correctness and Complexity	91
6.5	Implementation Techniques	96
6.5.1	Implicit Representation of Sets of $\psi$ -Types	96
6.5.2	Satisfying Model Reconstruction	98
6.5.3	Conjunctive Partitioning and Early Quantification	98
6.5.4	BDD Variable Ordering	99
6.6	Typing Applications and Experimental Results	99
6.6.1	Experimental Results	100
6.7	Outcome	105
<b>7</b>	<b>Conclusion</b>	<b>107</b>
7.1	Summary of the Main Contributions	107
7.2	Perspectives	107
7.2.1	Further Optimizations of the Logical Solver	107
7.2.2	Pushing the XPath Decidability Envelope Further	108
7.2.3	Enhancing the Translation of Regular Tree Types	108
7.2.4	Efficiently Supporting Attributes and Data Values	108
7.2.5	Query Optimization	109
7.2.6	Query Evaluation via Model-Checking	109
7.2.7	Application to the Static Analysis of Transformations	109
	<b>Bibliography</b>	<b>111</b>
	<b>Computational Complexity for Logical Satisfiability Dealt With in this Dissertation</b>	<b>123</b>
	<b>Résumé étendu</b>	<b>125</b>



# List of Figures

1.1	Sample Tree of a Well-Formed Document. . . . .	2
1.2	XPath Axes Partition from Context Node. . . . .	4
2.1	Unranked and Binary Tree Representations. . . . .	12
2.2	Relative Expressiveness of Schema Languages. . . . .	13
2.3	Binarization of Tree Types. . . . .	16
2.4	A Sample DTD. . . . .	16
2.5	Sample Context-Free Tree Type Expression. . . . .	17
2.6	Sample Binary Tree Type Expression. . . . .	17
2.7	A Sample NFTA $(Q, Q_f, \Gamma)$ . . . . .	19
2.8	XPath Abstract Syntax. . . . .	22
2.9	Representations of a Satisfying Interpretation of $X \subseteq Y$ . . . . .	29
3.1	Sample XML Tree in MONA WS2S Syntax. . . . .	43
3.2	Translating XPath into WS2S. . . . .	45
3.3	WS2S Translation of a Sample XPath in MONA Syntax. . . . .	46
3.4	Sample WS2S Formula for XPath Containment in MONA Syntax. . . . .	48
3.5	WS2S Translation of $e_3$ in MONA Syntax. . . . .	51
3.6	Depth Levels in the Unranked and Binary Cases. . . . .	51
3.7	Computation of the Depth Levels of Nodes Selected by a Path. . . . .	53
3.8	Translating XPath into WS2S with Restricted Variable Scopes. . . . .	54
3.9	Optimized WS2S Translation of $e_3$ in MONA Syntax. . . . .	54
3.10	Statistics on Intermediate Automata for a Containment Check. . . . .	56
4.1	Semantics of the $\mu$ -Calculus. . . . .	61
4.2	Dualities for Negation Normal Form. . . . .	61
4.3	Translation of XPath Axes. . . . .	64
4.4	Translation of Expressions and Paths. . . . .	65
4.5	XPath Translation Example. . . . .	65
4.6	Translation of Qualifiers. . . . .	66
5.1	Logic formulas . . . . .	79
5.2	Interpretation of formulas . . . . .	79
5.3	Cycle-free formulas . . . . .	80
5.4	Example of Back and Forth XPath Navigation Translation. . . . .	85
6.1	Truth Assignment of a Formula . . . . .	89
6.2	Operations used by the Algorithm. . . . .	90
6.3	Run of the Algorithm for Checking Emptiness of <code>self::b/parent::a</code> . . . . .	91

## LIST OF FIGURES

---

6.4	Partial Satisfiability . . . . .	92
6.5	Satisfiability Relation . . . . .	95
6.6	Queries Taken from the XPathmark Benchmark. . . . .	101
6.7	Results for Instances Found in Research Papers. . . . .	102
6.8	Results for Instances with Horizontal Navigation. . . . .	103
6.9	Queries Used in the Presence of DTDs. . . . .	103
1	Exemple: arbre d'un document bien-formé. . . . .	126
2	Partition des axes depuis un nœud de contexte. . . . .	129

# Notations

---

Symbol	Description	Page
$::=$	Definition of an abstract syntax	
$2^X$	Powerset of $X$	
$\mathbb{N}$	Natural integers	
$\Sigma$	Alphabet of node labels	(11)
$\mathcal{H}_\Sigma$	Labeled hedges	(11)
$\mathcal{T}_\Sigma^n$	Labeled unranked trees	(11)
$\mathcal{T}_\Sigma^2$	Labeled binary trees	(11)
$\beta(\cdot)$	Mapping from hedges to binary trees	(12)
$\beta^{-1}(\cdot)$	Mapping from binary trees to hedges	(12)
$\mathcal{L}_{\text{cft}}$	Context-free tree type expressions	(13)
$\llbracket \cdot \rrbracket_\theta$	Denotational semantics of tree types	(13)
$\mathcal{L}_{\text{rt}}$	Regular tree type expressions	(14)
$\mathcal{L}_{\text{dtd}}$	DTD tree type expressions	(15)
$\mathcal{L}_{\text{bt}}$	Binary tree type expressions	(15)
$\mathcal{B}(\cdot)$	Mapping from unranked to binary tree types	(15)
$\mathcal{S}_e \llbracket \cdot \rrbracket \cdot$	Denotational semantics of XPath expressions	(21)
$\mathcal{L}_{\text{XPath}}$	XPath expressions	(22)
$\text{Dom}(t)$	Domain of a tree (set of nodes)	(23)
$\prec_{\text{ch}}$	Child relation between tree nodes	(23)
$\prec_{\text{sb}}$	Sibling relation between tree nodes	(23)
$\prec_{\text{ch}}^*$	Transitive-reflexive closure of $\prec_{\text{ch}}$	(24)

LIST OF NOTATIONS

---

$\prec_{sb}^*$	Transitive-reflexive closure of $\prec_{sb}$	(24)
$\tilde{t}$	Tuple representation of the structure $t$	(26)
$X_\sigma$	Characteristic set of the label $\sigma$	(26)
$\mathcal{L}_{ws2s}$	WS2S formulas	(26)
$\mathcal{L}(\varphi)$	Language defined by the formula $\varphi$	(27)
$\tilde{\tilde{t}}$	Matricial representation of the structure $t$	(27)
$\mathcal{A}[[\varphi]]$	Tree automaton corresponding to $\varphi$	(28)
$\perp$	Termination symbol	(28)
$\hat{t}$	Tree representation of the structure $t$	(28)
$\mathcal{W}_e[[\cdot]]:$	WS2S Interpretation of XPath	(44)
$L_e[[\cdot]].$	Calculation of a set of depth levels	(52)
$\mathcal{W}'_e[[\cdot]]$	Optimized WS2S Interpretation of XPath	(52)
$\Xi$	Signature for the $\mu$ -calculus	(59)
$Prop$	Set of atomic propositions	(59)
$Var$	Set of propositional variables	(59)
$FProg$	Set of atomic programs	(59)
$\bar{a}$	Converse of an atomic program	(60)
$Prog$	Set of programs	(60)
$\mathcal{L}_\mu^{\text{full}}$	Set of $\mu$ -calculus formulas	(60)
$[[\varphi]]:$	Interpretation of $\mu$ -calculus formula $\varphi$	(60)
$\varphi_{\text{root}}$	$\mu$ -formula satisfied at a root	(62)
$\varphi_{\text{ft}}$	$\mu$ -formula ensuring structure finiteness	(63)
$[[\cdot]]$	Translation of tree types into $\mu$ -calculus	(69)
$\sigma^\circ$	Node with unknown context mark	(78)
$\mathcal{F}$	Set of finite focused trees	(78)
$[[\varphi]].$	Interpretation of the formula $\varphi$	(79)
$unf(\varphi)$	Finite unfolding of a formula $\varphi$	(81)
$\mathcal{S}_e[[\cdot]].$	XPath interpretation as focused tree sets	(83)
$[[\cdot]]$	Translation of tree types into $\mathcal{L}_\mu$	(85)
$\text{exp}(\varphi)$	Unwinding of a formula $\varphi$	(87)
$\text{cl}(\psi)$	Fisher-Ladner closure of $\psi$	(87)
$\Sigma(\psi)$	Set of atomic propositions for $\psi$	(88)
$\text{cl}^*(\psi)$	Extended Fisher-Ladner closure of $\psi$	(88)

---

$\text{Lean}(\psi)$	Lean of $\psi$	(88)
$\text{Typ}(\psi)$	Set of $\psi$ -types	(88)
$\dot{\in}$	Truth assignment relation	(88)
$\Delta_a(\cdot, \cdot)$	Compatibility relation for two $\psi$ -types	(88)
$X^i$	Set of triples after $i$ iterations	(90)
$T^i$	Set of $\psi$ -types after $i$ iterations	(90)
$\llbracket \cdot \rrbracket$	Partial satisfiability of a formula	(91)
$\varphi_c(t)$	Most constrained formula for a $\psi$ -type $t$	(91)
$\rho$	Path (concatenation of modalities)	(91)
$\cdot \Vdash_\rho \cdot$	Satisfiability relation	(94)
$\vec{t}$	Bit-vector representation of $t$	(97)
$\chi_T$	Characteristic function of a set $T$	(97)



# Introduction

---

## 1.1 Motivation and Objectives

This work was initially motivated by the need for efficient static type checkers for XML processing languages. Such programming languages use schemas [Fallside and Walmsley, 2004] and XPath [Clark and DeRose, 1999] queries as first class language constructs. Current examples of these languages include the W3C recommendation XSLT [Clark, 1999] for the transformation of XML documents, and the forthcoming XQuery [Boag et al., 2006] recommendation for querying XML databases. Providing such languages with decidable and efficient static type systems has been one of the major research challenges over the last decade, notably gathering the programming language, database theory, structured documents, and theoretical computer science communities. This work follows the research effort initiated in [Murata, 1996, Tozawa, 2001, Milo et al., 2003, Hosoya and Pierce, 2003].

This work resulted in the design of a new logic of finite trees adapted for XML, and its decision procedure, presented in this dissertation. The logical solver has been implemented as the core of a system for the general static analysis and type-checking of XML specifications. The system can be used as a component of static analyzers for programming languages manipulating both XPath expressions and XML type annotations.

This dissertation presents the theoretical investigations that led to the foundations of this new logic of finite trees, along with the algorithmic bases and implementation principles on which the logical solver relies. These discoveries are applied to the resolution of XML type-checking problems, which are embedded in the logic. Solved problems include static typing of XPath in the presence of regular tree type constraints.

### 1.1.1 XML Documents and Schemas

*Extensible Markup Language* (XML) [Bray et al., 2004] is a text file format for representing tree structures in a standard form.

The whole structure of an XML document, if we abstract over less important details, is a tree of variable arity, in which nodes (also called *elements* in the XML jargon) are labeled, leaves of the tree are text nodes, and the ordering

## 1. INTRODUCTION

---

between children of a node is significant. XML can be seen as a concrete syntax for describing such tree structures using mark-up texts. An example of an XML document is as follows:

```
<plant>
  <category>Vascular</category>
  <tissue>
    <name>Phloem</name>
    <def>The phloem is a living tissue that carries organic
      nutrients to all parts of the plant where needed.</def>
    <note>In trees, the phloem is part of the bark.</note>
  </tissue>
</plant>
```

An element is described by a pair of an opening tag `< ... >` and an closing tag `< /... >`, between which the element content is inserted. In the previous example, “plant”, “category”, “tissue”, “name”, “def”, and “note” are labels (*tag names* in the XML jargon).

The XML specification does not fix a priori the set of allowed labels in an XML document nor it defines any semantics for labels. Only well-formedness conditions are defined in particular to ensure proper nesting of elements, which allows to consider XML documents as trees. For instance, Figure 1.1 gives a more visual tree representation of the previous well-formed sample XML document.

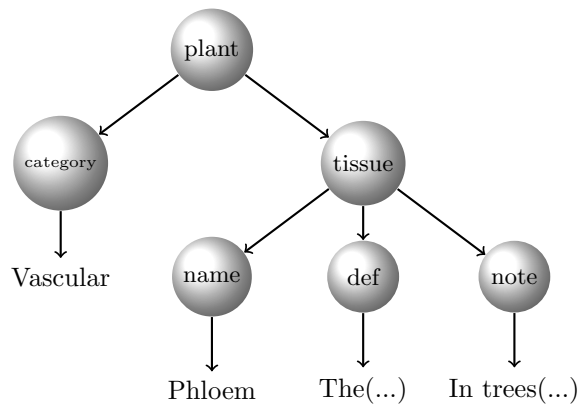


Figure 1.1: Sample Tree of a Well-Formed Document.

The set of labels occurring in an XML document is determined by *schemas* that can freely be defined by users. A *schema* (also called an *XML type*) is a description of constraints on the structure of documents such as allowed labels and their possible nesting structures. A schema thus defines a class of XML documents. Two levels of correctness can therefore be distinguished for XML documents:

- *well-formedness* which applies to documents that obey the necessary and sufficient syntactic condition (defined by the XML specification) for being interpreted as trees;

- *validity* which applies to documents that conform to the additional constraints described by a given schema.

The validity of a document implies its well-formedness since the schema describes constraints on the tree and not on the text representation of the XML document.

Each application can define its own data format by defining schemas, at a higher abstract level (tree structures). In that sense, XML is often said to be a metalanguage or a “format for data formats”.

Separating the two levels of correctness allows applications to share generic software tools for manipulating well-formed XML documents (parsers, editors, query and transformation tools...). These tools all implement the same syntactic conventions defined by the XML specification (such as the way of including comments, external fragments, special characters...). XML thus allows a first level of processing on an XML document as soon as it is well-formed, without making the additional and much stronger hypothesis that it is valid w.r.t to some schema. This genericity is one of XML strengths. As a consequence, we have seen unprecedented speed and range in the adoption of XML. A large number of schemas have been defined and are actually widely used in practice, for instance: XHTML (the XML version of HTML), SVG (for vector graphics), SMIL (for synchronized multimedia documents), MathML (for mathematical formulas), SOAP (for remote procedure calls), XBRL (for financial information), FIX (for securities transactions), SMD (for music), X3D (for 3D modeling) and CML (for chemical structures).

### 1.1.2 XPath

XPath [Clark and DeRose, 1999, Berglund et al., 2006] has been introduced by the W3C as the standard query language for addressing and retrieving information in XML documents. It allows to navigate in XML trees and return a set of matching nodes. As such, XPath forms the essence of XML data access.

In their simplest form XPath expressions look like “directory navigation paths”. For example, the XPath expression

/book/chapter/section

navigates from the root of a document (designated by the leading slash “/”) through the top-level “book” nodes, to their “chapter” child nodes, and on to their child nodes named “section”. The result of the evaluation of the entire expression is the set of all the “section” nodes that can be reached in this manner. Furthermore, at each step in the navigation the selected nodes can be filtered using qualifiers. A qualifier is a boolean expression between brackets that can test the existence or absence of paths. So if we ask for

/book/chapter/section[citation]

then the result is *all* “section” elements that have a least one child element named “citation”. The situation becomes more interesting when combined with XPath’s capability of searching along “axes” other than the shown “children of” axis. Indeed the above XPath is a shorthand for

/child::book/child::chapter/child::section[child::citation]

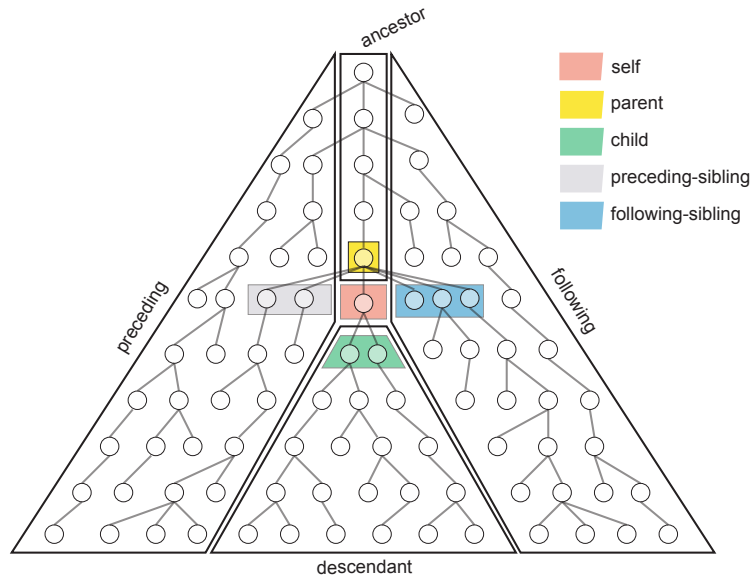


Figure 1.2: XPath Axes Partition from Context Node.

where it is made explicit that each *path step* is meant to search the “child” axis containing all children of the nodes selected at previous step. If we instead asked for

```
/child::book/descendant::*[child::citation]
```

then the last step selects nodes of any kind that are among the descendants of the top element “book” and have a “citation” sub-element. One may also use other axes such as “preceding-sibling” for navigating backward through nodes of the same parent, or “ancestor” for navigating upward recursively (see Figure 1.2). *Document order* is defined as the order in which a depth-first tree traversal visits nodes. Axes that perform navigation in reverse document order are called *reverse axes* (or alternatively *backward* or *upward* axes in the literature).

Previous examples are *absolute* XPath expressions as they start with a “/” which refers to the root. The meaning of a *relative* expression (without the leading “/”) is defined with respect to a *context node* in the tree. The *context node* simply refers to the tree node from which navigation starts. Starting from a particular context node in a tree, every other nodes can easily be reached: XPath axes define a partitioning of a tree from any context node. Figure 1.2 illustrates this on a sample tree. More informal details on the complete XPath standard can be found in the W3C specification [Clark and DeRose, 1999].

XPath is increasingly popular due to its expressive power and its compact syntax. These two advantages have given XPath a central role both in other key XML specifications and XML applications. It is used in XQuery [Boag et al., 2006] as a core query language; in XSLT [Clark, 1999] as node selector in the transformations; in XML Schema [Fallside and Walmsley, 2004] to define keys; in XLink [DeRose et al., 2001] and XPointer [DeRose et al., 2002] to reference portions of XML data. XPath is also used in many applications

such as update languages [Sur et al., 2004] and access control [Fan et al., 2004].

### 1.1.3 Static Type-Checking

XML applications most commonly use schemas for performing validation (also called *dynamic type-checking*). Validation consists in using a schema validator that analyzes a particular XML document w.r.t a given schema in order to ensure that the document actually conforms to the expectations of the application.

In practice however XML documents are often generated dynamically by some program. Typically, programs that manipulate XML first access data (possibly conforming to an available schema) using XPath expressions, and then build and return an output XML document intended to conform to a given schema.

An ambitious approach is the *static type-checking* of these programs, which consists in ensuring at compile-time that invalid documents can never arise as outputs of XML processing code. A static type checker analyzes a program, possibly in conjunction with schemas that describe its input and output (depending whether such schemas are available). The problem's difficulty is a function of the language in which the program and the schemas are expressed.

Schema languages have been extensively studied and are now well understood as subsets of regular tree languages [Murata et al., 2005]. However, although many attempts have been made for better understanding static type-checking techniques, in particular through the design of domain specific languages [Hosoya and Pierce, 2003], no approach is effectively able to deal with XPath, which nevertheless remains the essence of XML navigation and data access.

### 1.1.4 Research Challenges

The reason for the limitations of existing approaches is the difficulty of XPath static analysis. It is known that the static analysis of the complete XPath standard is undecidable. Importance and range of applications nevertheless motivate research questions: what is the largest XPath fragment with decidable static analysis? Which fragments can be effectively decided in practice? How to determine if an XPath expression is satisfiable on any of the XML trees defined by a given schema? How to know if two XPath queries will always yield the same result when evaluated on a document valid w.r.t. a given schema? Does the result of an XPath expression over a valid document always conform to another schema? Is there an algorithm able to answer these questions in an efficient way so that it can be used in practice?

One source of difficulty for such an algorithm is that it needs to check properties on a possibly infinite quantification over a set of trees. A variety of factors furthermore contribute to its complexity such as the operators allowed in XPath queries and the combination of them (cf. Chapter 2.2). A consequence of these difficulties is that such research questions are still open.

## 1.2 Overview of this Dissertation

This dissertation starts from the idea that for deciding XML problems, two issues must be addressed. First, identify an appropriate logic with sufficient expressiveness to capture both regular tree types and XPath style navigation and node selection semantics. Second, solve efficiently the satisfiability problem which allows to test if a given formula of the logic admits a satisfying XML document as a model.

### 1.2.1 Applications

The main application of this work is the static analysis of programs manipulating XML data and documents. This dissertation provides the necessary foundations and system implementations for solving the major XML decision problems that naturally arise from such static analyses.

The most basic decision problem for a query language is the emptiness check [Benedikt et al., 2005]: whether or not an expression yields a non-empty result. XPath emptiness is important for optimization of host languages implementations: for instance, if one can decide at compile time that a query is not satisfiable then subsequent bound computations can be avoided.

Another basic decision problem is the XPath equivalence problem: whether or not two queries always return the same result. It is important for reformulation and optimization of the query itself [Genevès and Vion-Dury, 2004], which aim at enforcing operational properties while preserving semantic equivalence [Abiteboul and Vianu, 1999, Levin and Pierce, 2005].

The most critical problem for the type-checking of XML transformations is XPath containment: whether or not, for any tree, the result of a particular query is included in the result of another one. It is required for the control-flow analysis of XSLT [Møller et al., 2005]. It is also needed for checking integrity constraints [Fallside and Walmsley, 2004], and for checking access control in XML security applications [Fan et al., 2004].

Other decision problems needed in applications include for example XPath overlap (whether two expressions select common nodes) and coverage (whether nodes selected by an expression are always contained in the union of the results selected by several other expressions).

This dissertation effectively solves these problems in the presence, or absence, of XML type constraints such as DTDs [Bray et al., 2004] or XML Schemas [Fallside and Walmsley, 2004]. This makes possible to ensure valuable properties (such as type-safety and optimizations) at compile-time, toward safer and more efficient runtime XML processing. Results presented in this dissertation thus notably open promising perspectives for the effective static analysis of XML transformations.

### 1.2.2 Outline

The first part of this dissertation is dedicated to state-of-the-art related tools and techniques. Chapter 2 introduces some known theoretical foundations and formalisms used in the remaining of this dissertation, while progressively introducing related work.

In a second part, Chapter 3 and Chapter 4 conduct preliminary investigations with known logics in the context of XML. Specifically, Chapter 3 studies to which extent monadic second order logic can be used in practice, despite its high complexity, for solving XML static analysis problems such as XPath containment. Chapter 4 introduces the  $\mu$ -calculus as a powerful replacement for monadic second order logic, and studies its use for XML reasoning.

Based on the lessons learned from these investigations, the third part of this dissertation presents the final contribution. Chapter 5 proposes a logic of finite trees specifically designed for XML. Chapter 6 describes a proposed algorithm for testing the satisfiability of the logic, along with implementation techniques. Finally, Chapter 7 concludes this dissertation and gives several perspectives.



## State of the Art



# Foundations of XML Processing

---

In this chapter, some known theoretical foundations and formalisms used in the following chapters of this dissertation are introduced. State of the art related work is presented as underlying concepts are progressively introduced.

## 2.1 Trees and Tree Types

This section introduces the formal models of XML documents and schemas most often considered in the literature as well as in Chapters 2, 3, and 4 of this dissertation <sup>1</sup>.

### 2.1.1 Finite Trees and Hedges

An XML document can be seen as a finite ordered and labeled tree of unbounded depth and arity. Since there is no a priori bound on the number of children of a node; such a tree is therefore *unranked* [Neven, 2002b]. Tree nodes are labeled with symbols taken from a countably infinite alphabet  $\Sigma$ . There is a straightforward isomorphism between sequences of unranked trees and binary trees [Hosoya and Pierce, 2003, Neven, 2002b]. In order to describe it, trees are first formally defined. An unranked tree is defined as  $\sigma(h)$  where  $\sigma \in \Sigma$  and  $h$  is a hedge, i.e. a sequence of unranked trees, defined as follows:

$$\mathcal{H}_\Sigma \ni h ::= \begin{array}{ll} \text{hedge} & \\ \sigma(h), h' & \text{non-empty sequence of trees} \\ | & \\ () & \text{empty sequence} \end{array}$$

The set of unranked trees is denoted by  $\mathcal{T}_\Sigma^n$ . A binary tree  $t$  is either a  $\sigma$ -labeled root of two subtrees ( $\sigma \in \Sigma$ ) or the empty tree:

$$\mathcal{T}_\Sigma^2 \ni t ::= \begin{array}{ll} \text{binary tree} & \\ \sigma(t, t') & \text{node} \\ | & \\ \epsilon & \text{empty tree} \end{array}$$

---

<sup>1</sup>Chapter 5 elaborates further on this model by introducing *focused* trees.

Unranked trees are translated into binary trees with the following function  $\beta(\cdot)$ :

$$\begin{aligned}\beta(\cdot) &: \mathcal{H}_\Sigma \rightarrow \mathcal{T}_\Sigma^2 \\ \beta(\sigma(h), h') &\stackrel{\text{def}}{=} \sigma(\beta(h), \beta(h')) \\ \beta(()) &\stackrel{\text{def}}{=} \epsilon\end{aligned}$$

The inverse translation function  $\beta^{-1}(\cdot)$  converts a binary tree into a sequence of unranked trees:

$$\begin{aligned}\beta^{-1}(\cdot) &: \mathcal{T}_\Sigma^2 \rightarrow \mathcal{H}_\Sigma \\ \beta^{-1}(\sigma(t, t')) &\stackrel{\text{def}}{=} \sigma(\beta^{-1}(t), \beta^{-1}(t')) \\ \beta^{-1}(\epsilon) &\stackrel{\text{def}}{=} ()\end{aligned}$$

For example, Figure 2.1 illustrates how the sample tree  $a(b, c, d)$  is mapped to its binary representation  $a(b(\epsilon, c(\epsilon, d(\epsilon, \epsilon))), \epsilon)$  and vice-versa.

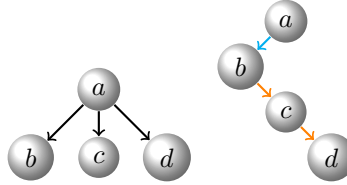


Figure 2.1: Unranked and Binary Tree Representations.

Note that the translation of a single unranked tree results in a binary tree of the form  $\sigma(t, \epsilon)$ . Reciprocally, the inverse translation of such a binary tree always yields a single unranked tree. When modeling XML, it is therefore possible to focus on binary trees of the form  $\sigma(t, \epsilon)$ , without loss of generality. The following section presents how this isomorphism between binary and unranked trees also extends to tree types. Such binary mappings allow to simplify formal notations used in the remaining.

### 2.1.2 Schema Languages and Regular Tree Types

Schemas describe structural constraints for XML documents. There are many formalisms (called *schema languages*) for specifying schemas (or “types”). For instance: DTD, which is part of the XML specification [Bray et al., 2004], XML Schema (W3C) [Fallside and Walmsley, 2004], and RELAX NG (OASIS/ISO) [Clark and Murata, 2001] are actively used by various applications. Each schema language has different constraint mechanisms and different expressivenesses. A detailed characterization of each schema language can be found in [Murata et al., 2005]. No current schema language goes beyond the expressive power of regular tree languages. From an XML point of view, regular tree types form a strict superset of standards such as XML Schemas and DTDs (cf. Figure 2.2). Therefore, in this dissertation, regular tree languages are considered as the general mechanism for typing XML documents.

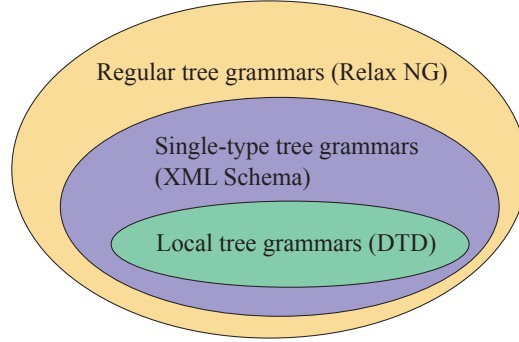


Figure 2.2: Relative Expressiveness of Schema Languages.

A tree type expression  $T$  is syntactically defined as follows:

$\mathcal{L}_{\text{cft}} \ni T ::=$		context-free tree type expression
	$\emptyset$	empty set of trees
	$()$	empty sequence
	$X$	variable
	$l[T]$	label
	$T_1, T_2$	sequence
	$T_1 \mid T_2$	disjunction
	$\text{let } \overline{X_i.T_i} \text{ in } T$	$n$ -ary binder

where  $l \in \Sigma$  and  $X \in TVar$  assuming that  $TVar$  is a countably infinite set of type variables. Abbreviated type expressions can be defined as follows:

$$\begin{aligned}
 T^? &\stackrel{\text{def}}{=} () \mid T \\
 T^* &\stackrel{\text{def}}{=} \text{let } X.T \text{ in } T, X \mid () \\
 T^+ &\stackrel{\text{def}}{=} T, T^*
 \end{aligned}$$

Given an environment  $\theta$  of type variable bindings, the semantics of tree types is given by the denotation function  $\llbracket \cdot \rrbracket_\theta$ :

$$\begin{aligned}
 \llbracket \cdot \rrbracket_\theta &: \mathcal{L}_{\text{cft}} \times (TVar \rightarrow 2^{\mathcal{T}^\Sigma}) \rightarrow 2^{\mathcal{T}^\Sigma} \\
 \llbracket \emptyset \rrbracket_\theta &\stackrel{\text{def}}{=} \emptyset \\
 \llbracket () \rrbracket_\theta &\stackrel{\text{def}}{=} \{()\} \\
 \llbracket X \rrbracket_\theta &\stackrel{\text{def}}{=} \theta(X) \\
 \llbracket l[T] \rrbracket_\theta &\stackrel{\text{def}}{=} \{l'(t) \mid l' \prec l \wedge t \in \llbracket T \rrbracket_\theta\} \\
 \llbracket T_1, T_2 \rrbracket_\theta &\stackrel{\text{def}}{=} \{t_1, t_2 \mid t_1 \in \llbracket T_1 \rrbracket_\theta \wedge t_2 \in \llbracket T_2 \rrbracket_\theta\} \\
 \llbracket T_1 \mid T_2 \rrbracket_\theta &\stackrel{\text{def}}{=} \llbracket T_1 \rrbracket_\theta \cup \llbracket T_2 \rrbracket_\theta \\
 \llbracket \text{let } \overline{X_i.T_i} \text{ in } T \rrbracket_\theta &\stackrel{\text{def}}{=} \llbracket T \rrbracket_{lfp(S)}
 \end{aligned}$$

where  $\prec$  is a global subtagging relation: a reflexive and transitive relation on labels<sup>2</sup>, and  $S(\theta') = \theta[X_i \mapsto \llbracket T_i \rrbracket_{\theta'}]_{i \geq 1}$ . Note that each function  $S$  is monotone according to the ordering  $\subseteq$  on  $TVar \rightarrow 2^{\Sigma^n}$ , and thus has a least fixpoint  $lfp(S)$ .

Types as defined above actually correspond to arbitrary context-free tree types, for which the decision problem for inclusion is known to be undecidable [Hopcroft et al., 2000]. An additional restriction is imposed to reduce the expressive power of considered types so that they correspond to regular tree languages. The restriction (also used in [Hosoya et al., 2005b]) consists in a simple syntactic condition that allows unguarded (i.e. not enclosed by a label) recursive uses of variables, but restricts them to tail positions<sup>3</sup>. This condition ensures regularity, and the resulting class of regular tree languages is denoted  $\mathcal{L}_{rt}$ .

## Document Type Definitions

This subsection further details the connection between regular tree types and the widely used DTD standard. As they are defined in the W3C recommendation, DTDs [Bray et al., 2004] are local tree grammars<sup>4</sup>, which are strictly less expressive than regular tree types. In the XML terminology, a type expression is called the *content model*. DTD content models are described by the following syntax:

$T ::=$		DTD tree type expression
	$l$	label
	$ $	
	$T_1 \mid T_2$	disjunction
	$ $	
	$T_1, T_2$	sequence
	$ $	
	$T?$	optional occurrence
	$ $	
	$T^*$	zero, one or more occurrences
	$ $	
	$T^+$	one or more occurrences
	$ $	
	$()$	empty sequence

where  $l \in \Sigma$ . From the W3C specification, a DTD can be seen as a function that associates a content model to each label taken from a subset  $\Sigma'$  of  $\Sigma$ , such that  $\Sigma'$  gathers all labels used in content models. The set  $\mathcal{L}_{\text{dtd}}$  of tree types

<sup>2</sup>Subtagging goes beyond the expressive power of DTDs but a similar notion called “substitution groups” exists in XML Schemas (see [Hosoya et al., 2005b] for more details on subtagging).

<sup>3</sup>For instance the type “let  $\overline{X, Y_i \cdot a[]}, \overline{Y_i}$  in  $b[], X \mid ()X$ ” is allowed.

<sup>4</sup>A local tree grammar is a regular tree grammar without *competing* non-terminals. Two non-terminals  $A$  and  $B$  of a tree grammar are said to compete with each other if one production rule has  $A$  in its left-hand side, one production rule has  $B$  in its left-hand side, and these two rules share the same terminal symbol in the right-hand side.

described by DTDs can thus be represented as follows:

$\mathcal{L}_{\text{dtd}} \ni T ::=$		DTD tree type expression
	$l$	label
	$T_1 \mid T_2$	disjunction
	$T_1, T_2$	sequence
	$T?$	optional occurrence
	$T^*$	zero, one or more occurrences
	$T^+$	one or more occurrences
	$()$	empty sequence
	$\text{let } \overline{l_i.T_i} \text{ in } T$	$n$ -ary binder

Note that  $\mathcal{L}_{\text{dtd}} \subseteq \mathcal{L}_{\text{rt}}$  is obvious, by associating a unique type variable to each label. In the following, DTDs are therefore not distinguished from general regular tree types anymore.

### 2.1.3 Binary Tree Types

Section 2.1.1 presented a straightforward isomorphism between binary trees and sequences of unranked trees. There is also an isomorphism between unranked and binary tree types, which follows exactly the same intuition as for trees.

Binary tree types are described by the following syntax:

$\mathcal{L}_{\text{bt}} \ni T ::=$		binary tree type expression
	$\emptyset$	empty set of trees
	$()$	empty sequence
	$T_1 \mid T_2$	disjunction
	$l(X_1, X_2)$	label
	$\text{let } \overline{X_i.T_i} \text{ in } T$	$n$ -ary binder

For any type, there is an equivalent binary type, and vice-versa. The translation function  $\mathcal{B}(\cdot)$  shown on Figure 2.3 (and adapted from the one found in [Hosoya et al., 2005b]) is used to convert a type into its corresponding binary representation. The function considers the environment  $\theta : TVar \rightarrow \mathcal{L}_{\text{rt}}$  for accessing the type bound to a variable  $X_i$  by constructs of the form “let  $\overline{X_i.T_i}$  in  $T$ ”.

For example, Figure 2.4 gives a sample DTD that validates the well-formed XML document presented in Section 1.1.1 of Chapter 1. The corresponding context-free tree type expression is presented on Figure 2.5. It uses 14 type variables (preceded by a dollar sign \$ by convention). Figure 2.6 shows its translation into binary tree type syntax.

### 2.1.4 Finite Tree Automata

Tree automata are a convenient operational formalism for expressing the notion of tree languages. A language is recognizable if there exists an automaton which recognizes trees of the language. A detailed classification of tree automata and associated results on the recognizability of tree languages are presented in [Comon et al., 1997]. This section presents the most basic results on finite tree automata needed for the remaining of this dissertation.

$$\begin{aligned}
\mathcal{B}(\cdot) &: \mathcal{L}_{\text{rt}} \rightarrow \mathcal{L}_{\text{bt}} \\
\mathcal{B}(\emptyset) &\stackrel{\text{def}}{=} \emptyset \\
\mathcal{B}(\epsilon) &\stackrel{\text{def}}{=} \epsilon \\
\mathcal{B}(X) &\stackrel{\text{def}}{=} \mathcal{B}(\theta(X)) \\
\mathcal{B}(l[T]) &\stackrel{\text{def}}{=} \text{let } X_1.\mathcal{B}(T), X_2.\epsilon \text{ in } l(X_1, X_2) \\
\mathcal{B}(T_1 \mid T_2) &\stackrel{\text{def}}{=} \mathcal{B}(T_1) \mid \mathcal{B}(T_2) \\
\mathcal{B}(\text{let } \overline{X_i.T_i} \text{ in } T) &\stackrel{\text{def}}{=} \text{let } \overline{X_i.\mathcal{B}(T_i)} \text{ in } \mathcal{B}(T) \\
\mathcal{B}(\emptyset, T) &\stackrel{\text{def}}{=} \emptyset \\
\mathcal{B}(\epsilon, T) &\stackrel{\text{def}}{=} \mathcal{B}(T) \\
\mathcal{B}(X, T) &\stackrel{\text{def}}{=} \mathcal{B}(\theta(X), T) \\
\mathcal{B}(l[T_1], T_2) &\stackrel{\text{def}}{=} \text{let } X_1.\mathcal{B}(T_1), X_2.\mathcal{B}(T_2) \text{ in } l(X_1, X_2) \\
\mathcal{B}((T_1 \mid T_2), T_3) &\stackrel{\text{def}}{=} \mathcal{B}(T_1, T_3) \mid \mathcal{B}(T_2, T_3) \\
\mathcal{B}((T_1, T_2), T_3) &\stackrel{\text{def}}{=} \mathcal{B}(T_1, (T_2, T_3)) \\
\mathcal{B}(\text{let } \overline{X_i.T_i} \text{ in } T, T') &\stackrel{\text{def}}{=} \text{let } \overline{X_i.\mathcal{B}(T_i)} \text{ in } \mathcal{B}(T, T')
\end{aligned}$$

Figure 2.3: Binarization of Tree Types.

```

<!ELEMENT plant (category?, tissue*, phylogeny?)>
<!ELEMENT category (#PCDATA)>
<!ELEMENT tissue (name+, def, note?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT def (#PCDATA)>
<!ELEMENT note (#PCDATA)>
<!ELEMENT phylogeny (plant+)>

```

Figure 2.4: A Sample DTD.

**Bottom-Up Finite Tree Automata** Formally, a bottom-up non-deterministic finite tree automaton (NFTA) over an alphabet  $\Sigma$  of node labels is a tuple  $(Q, Q_f, \Gamma)$  where  $Q$  is the set of states,  $Q_f \subseteq Q$  is a set of accepting states, and  $\Gamma$  is a set of transitions. Transitions are either of the form  $q \leftarrow \sigma$  or of the form  $q'' \leftarrow \sigma(q, q')$ , depending on the arity of the symbol  $\sigma \in \Sigma$  (respectively a leaf or a binary constructor) and where  $q, q', q''$  are automaton states belonging to  $Q$ . A bottom-up NFTA starts from the leaves and moves up the tree. At each step of the execution, a state is inductively associated with each subtree. The tree is accepted if the state labeled at the root is an accepting state.

```

$Empty -> EMPTYSET
$Epsilon -> ()
$Any -> ()
$PCData -> ()
$name -> name($PCData)
$note -> note($PCData)
$1 -> $plant | $plant, $1
$phylogeny -> phylogeny($1)
$category -> category($PCData)
$def -> def($PCData)
$2 -> $name | $name, $2
$tissue -> tissue($2, $def, () | $note)
$3 -> () | $tissue, $3
$plant -> plant(() | $category, $3, () | $phylogeny)

Start symbol is $plant
14 type variables.
7 terminals.

```

Figure 2.5: Sample Context-Free Tree Type Expression.

```

$2 -> plant($1, $Epsilon) | plant($1, $2)
$7 -> EPSILON | note($Epsilon, $Epsilon)
$5 -> def($Epsilon, $7)
$3 -> name($Epsilon, $5) | name($Epsilon, $3)
$10 -> EPSILON | phylogeny($2, $Epsilon) | tissue($3, $10)
$1 -> EPSILON | phylogeny($2, $Epsilon) |
      tissue($3, $10) | category($Epsilon, $10)
$plant -> plant($1, $Epsilon)

Start symbol is $plant
7 type variables.
7 terminals.

```

Figure 2.6: Sample Binary Tree Type Expression.

**Top-Down Finite Tree Automata** There exists a symmetric counterpart of bottom-up NFTA called top-down NFTA, which correspond to the alternate direction used to recognize a tree. A top-down NFTA  $(Q, Q_i, \Gamma)$  starts at the root and moves down to the leaves. Based on a state and a current node in the tree, a new state is inductively associated with each subtree. Transitions thus have the reverse form, and  $Q_i$  is the set of initial states. The tree is accepted if every branch can be gone through this way.

**Determinism** A deterministic finite tree automaton (DFTA) is one where no two transition rules have the same left-hand side. This definition matches the intuitive idea that for an automaton to be deterministic, one and only one transition must be possible for a given node.

**Expressive Power** Top-down and bottom-up NFTA are equivalent (the transition rules are simply reversed, and the final states become the initial states). However, top-down DFTA are strictly less powerful than their deterministic bottom-up counterparts. This is because transition rules of tree automata can be seen as rewrite rules; and for top-down ones, the left-hand sides correspond to parent nodes. Consequently a deterministic top-down tree automaton will only be able to test for tree properties that are true in all branches, because the choice of the state to write into each child branch is determined at the parent node, without knowing the child branches contents.

Every bottom-up NFTA is equivalent to a bottom-up DFTA which can be obtained by the process of determinization. Determinization relies on the “subset construction” and the number of states of the equivalent DFTA can be exponential in the number of states of the given NFTA (see [Comon et al., 1997] for the detailed algorithm). In the bottom-up paradigm, since NFTA and DFTA accept the same sets of tree languages, they are usually not distinguished and simply both referred as finite tree automata (FTA).

FTA are equivalent to regular tree types and therefore have the same expressiveness.

**FTA as XML Types** Murata was the first to consider tree automata as a schema definition language [Murata, 1998]. Since then, FTA were heavily used in many research works for modeling XML types [Neven, 2002a]. In fact, the schema language Relax NG [Clark and Murata, 2001], a competitor of XML Schema [Fallside and Walmsley, 2004] (itself introduced as a replacement for DTDs [Bray et al., 2004]) is even directly inspired by FTA. A detailed comparison of these schema languages based on formal language theory is provided in [Murata et al., 2005].

As a simple example, Figure 2.7 illustrates a sample NFTA which accepts the set of trees defined by the DTD shown on Figure 2.4. The NFTA accepts the set of all binary trees  $\beta(t)$  such that the unranked tree  $t$  is validated by the DTD of Figure 2.4. Note that the NFTA can be seen as another notation for the binary tree type expression shown on Figure 2.6. More interestingly, the DFTA obtained by determinization of this NFTA can be seen as the operational validator of the DTD.

$$\begin{aligned}
 Q &= \{q_1, q_2, q_3, q_5, q_7, q_{10}, q_\epsilon, q_{\text{plant}}\} \\
 Q_f &= \{q_{\text{plant}}\} \\
 \Gamma &= \left\{ \begin{array}{ll} q_2 & \leftarrow \text{plant}(q_1, q_\epsilon) \\ q_2 & \leftarrow \text{plant}(q_1, q_2) \\ q_7 & \leftarrow \epsilon \\ q_7 & \leftarrow \text{note}(q_\epsilon, q_\epsilon) \\ q_5 & \leftarrow \text{def}(q_\epsilon, q_7) \\ q_3 & \leftarrow \text{name}(q_\epsilon, q_5) \\ q_3 & \leftarrow \text{name}(q_\epsilon, q_3) \\ q_{10} & \leftarrow \epsilon \\ q_{10} & \leftarrow \text{phylogeny}(q_2, q_\epsilon) \\ q_{10} & \leftarrow \text{tissue}(q_3, q_{10}) \\ q_1 & \leftarrow \epsilon \\ q_1 & \leftarrow \text{phylogeny}(q_2, q_\epsilon) \\ q_1 & \leftarrow \text{tissue}(q_3, q_{10}) \\ q_1 & \leftarrow \text{category}(q_\epsilon, q_{10}) \\ q_{\text{plant}} & \leftarrow \text{plant}(q_1, q_\epsilon) \end{array} \right\}
 \end{aligned}$$

 Figure 2.7: A Sample NFTA  $(Q, Q_f, \Gamma)$ .

**Closure Properties** One of the main advantages of FTA (compared to DTDs for instance) is their closure under set theoretic operations such as union, intersection, and complementation [Comon et al., 1997].

The union of two tree automata is trivially built: let  $A_1 = (Q_1, Q_{f_1}, \Gamma_1)$  and  $A_2 = (Q_2, Q_{f_2}, \Gamma_2)$  be two FTA. Since states of a FTA may be renamed without loss of generality, it is assumed that  $Q_1 \cap Q_2 = \emptyset$ . It is then straightforward to verify that  $A_1 \cup A_2 = (Q, Q_f, \Gamma)$  defined by:  $Q = Q_1 \cup Q_2$ ,  $Q_f = Q_{f_1} \cup Q_{f_2}$  and  $\Gamma = \Gamma_1 \cup \Gamma_2$ .

Similarly, the intersection of two tree automata  $A_1 = (Q_1, Q_{f_1}, \Gamma_1)$  and  $A_2 = (Q_2, Q_{f_2}, \Gamma_2)$  is simply obtained by calculating a product automaton:

$$A_1 \cap A_2 = (Q_1 \times Q_2, Q_{f_1} \times Q_{f_2}, \Gamma_1 \times \Gamma_2)$$

Complementation of a *complete* DFTA simply consists in flipping accepting and rejecting states. Note that a DFTA  $(Q, Q_f, \Gamma)$  is *complete* if and only if there is a transition  $q'' \leftarrow \sigma(q, q')$  for each  $\sigma \in \Sigma$  and  $(q, q', q'') \in Q^3$ . Completing an automaton (e.g. adding new missing states and transitions, and then possibly updating the final set of states [Comon et al., 1997]) may be required before complementing it. The complement of a FTA  $A$  is noted  $\mathcal{C}(A)$ .

**Containment for FTA** By taking advantage of these closure properties, it is possible to check the containment of two FTA  $A_1$  and  $A_2$  (determining whether the set of trees accepted by  $A_1$  is included into the set of trees accepted by  $A_2$ ) as the emptiness check of the FTA  $A_1 \cap \mathcal{C}(A_2)$ .

It can be decided in linear time whether the language accepted by a FTA is empty (see [Comon et al., 1997] for details). However, complementation requires determinization of the tree automaton, which may cause an exponential

increase of the number of states in the worst case [Comon et al., 1997]. Thus this technique has exponential time complexity. Essentially, there is no better way for checking containment between two FTA. As a result, the FTA containment problem is in EXPTIME<sup>5</sup> [Seidl, 1990].

## 2.2 Queries

Most queries used in the context of XML are either boolean or unary. Boolean queries give a yes/no answer on a tree (for instance the validation of an XML document w.r.t to a DTD is a boolean query). Unary queries select nodes from a document (for instance, finding the set of nodes selected by an XPath expression is a unary query).

Unary queries considered in this dissertation are among those defined by the powerful XPath standard introduced in Section 1.1.2. The static analysis of XPath queries is a hard problem that has recently attracted a lot of theoretical research attention. In particular, the computational complexity of the containment problem for XPath expressions has received much attention from the database community [Deutsch and Tannen, 2001, Wood, 2003, Neven and Schwentick, 2003, Schwentick, 2004, Miklau and Suciu, 2004]. The complexity of the emptiness problem for XPath expressions has also been studied in [Benedikt et al., 2005]. One source of difficulty for such decision problems is that they need to be checked on a possibly infinite quantification over a set of trees. A variety of factors also contribute to their complexity such as the operators allowed in XPath queries and the combination of them. For instance, one difficulty arises from the combination of upward and downward navigation on trees with recursion [Vardi, 1998]. Actually, when the whole XPath language is considered, decision problems such as containment and emptiness are undecidable. Therefore, in the literature, the focus was given to identifying major XPath features and studying their impact on the complexity of XPath decision problems. The distinctions between major features studied in the literature (extended from [Benedikt et al., 2005]) follow:

- positive vs. non-positive: depending whether the negation operator is considered (positive) or not (non-positive) inside qualifiers.
- downward vs. upward: depending whether queries specify downward or upward traversal of the tree, or both.
- recursive vs. non-recursive: depending whether XPath transitive closure axes (for instance “descendant” or “ancestor”) are considered or not.
- qualified vs. non-qualified: depending whether queries allow filtering qualifiers or not.
- with vs. without data values: depending whether comparisons of data values expressing joins are allowed or not.
- with vs. without counting: depending whether counting of tree nodes is allowed or not.

---

<sup>5</sup>The complexity class EXPTIME is the set of all decision problems solvable by a deterministic Turing machine in  $O(2^{p(n)})$  time, where  $p(n)$  is a polynomial function of the input size  $n$ .

Several XPath fragments combining only a few of these features have been studied: see [Schwentick, 2004] for an overview. From these results, it is known that containment and satisfiability for (reasonably) restricted XPath fragments, even without type constraints, ranges from EXPTIME to undecidable. However, techniques used for obtaining computational complexity bounds over specific subfragments do not scale when additional features are considered, and thus give no hints on how to address more realistic fragments. At the time of this dissertation, no relevant algorithm effectively able of answering realistic XPath decision problems in acceptable time and space bounds is known. XPath decision problems have been partially characterized from a strict computational complexity point of view, and remain unsolved in practice.

### 2.2.1 Syntax of XPath Expressions

In this dissertation, particular attention is paid at supporting a large XPath fragment, as realistic as possible, covering major features of the XPath standard [Clark and DeRose, 1999]. The syntax of considered XPath expressions is given on Figure 2.8. The considered XPath fragment is non-positive, both downward and upward, recursive, qualified, and also includes union and intersection. It includes all axes. This is the largest fragment considered so far in the literature. It covers all major XPath features except counting and data values. The integration of counting is kept for future work, based on related work on logics for counting [Dal-Zilio et al., 2004]. Data values are known to cause undecidability of XPath containment when combined with previous factors [Benedikt et al., 2005, Schwentick, 2004]<sup>6</sup>.

### 2.2.2 XPath Denotational Semantics

In the classical denotational semantics of paths, first given in [Wadler, 2000], the evaluation of an XPath expression over an XML document  $t$  returns a set of nodes reachable from a context node  $x$ . The denotational semantics of the considered XPath fragment (adapted from [Wadler, 2000]) is given by the formal semantics function  $\mathcal{S}_e$  which defines the set of nodes returned by expressions, starting from a context node  $x$  in the tree:

$$\begin{aligned} \mathcal{S}_e[\cdot] : \mathcal{L}_{\text{XPath}} &\rightarrow \text{Node} \rightarrow \text{Set}(\text{Node}) \\ \mathcal{S}_e[/p]x &\stackrel{\text{def}}{=} \mathcal{S}_p[p] \text{root}() \\ \mathcal{S}_e[p]x &\stackrel{\text{def}}{=} \mathcal{S}_p[p]x \\ \mathcal{S}_e[e_1 \mid e_2]x &\stackrel{\text{def}}{=} \mathcal{S}_e[e_1]x \cup \mathcal{S}_e[e_2]x \\ \mathcal{S}_e[e_1 \cap e_2]x &\stackrel{\text{def}}{=} \mathcal{S}_e[e_1]x \cap \mathcal{S}_e[e_2]x \end{aligned}$$

<sup>6</sup>Note however that the very recent work found in [Bojanczyk et al., 2006] obtained the theoretical decidability (between NEXPTIME and 3-NEXPTIME) for a limited form of data value comparison. Integration of such restricted comparisons in the considered fragment and the effective algorithm presented in Chapter 6 is one of the perspectives of this dissertation. At least an additional exponential time blow-up is however expected.

$\mathcal{L}_{XPath} \ni e ::=$	$\begin{array}{l} /p \\   \\ p \\   \\ e_1 \mid e_2 \\   \\ e_1 \cap e_2 \end{array}$	XPath expression absolute path relative path union intersection
$Path \ p ::=$	$\begin{array}{l} p_1/p_2 \\   \\ p[q] \\   \\ a::\sigma \\   \\ a::* \end{array}$	path path composition qualified path step with node test step
$Qualif \ q ::=$	$\begin{array}{l} q_1 \text{ and } q_2 \\   \\ q_1 \text{ or } q_2 \\   \\ \text{not } q \\   \\ p \end{array}$	qualifier conjunction disjunction negation path
$Axis \ a ::=$	$\begin{array}{l} \text{child} \\   \\ \text{self} \\   \\ \text{parent} \\   \\ \text{descendant} \\   \\ \text{descendant-or-self} \\   \\ \text{ancestor} \\   \\ \text{ancestor-or-self} \\   \\ \text{following-sibling} \\   \\ \text{preceding-sibling} \\   \\ \text{following} \\   \\ \text{preceding} \end{array}$	tree navigation axis (see Figure 1.2)

Figure 2.8: XPath Abstract Syntax.

The formal semantics function  $\mathcal{S}_p$  defines the set of nodes returned by paths:

$$\begin{aligned} \mathcal{S}_p[\cdot] \cdot &: Path \rightarrow Node \rightarrow \text{Set}(Node) \\ \mathcal{S}_p[p_1/p_2]x &\stackrel{\text{def}}{=} \{x_2 \mid x_1 \in \mathcal{S}_p[p_1]x \wedge x_2 \in \mathcal{S}_p[p_2]x_1\} \\ \mathcal{S}_p[p[q]]x &\stackrel{\text{def}}{=} \{x_1 \mid x_1 \in \mathcal{S}_p[p]x \wedge \mathcal{S}_q[q]x_1\} \\ \mathcal{S}_p[a::\sigma]x &\stackrel{\text{def}}{=} \{x_1 \mid x_1 \in \mathcal{S}_a[a]x \wedge \text{name}(x_1) = \sigma\} \\ \mathcal{S}_p[a::*]x &\stackrel{\text{def}}{=} \{x_1 \mid x_1 \in \mathcal{S}_a[a]x\} \end{aligned}$$

Note that the semantics of the  $p_1/p_2$  construct corresponds to composition of unary queries. In this sense, XPath is fundamentally different from regular expressions patterns a la Hosoya [Hosoya and Pierce, 2001] that rather use pattern-matching techniques. The function  $\mathcal{S}_q$  defines the semantics of qualifiers that basically state the existence or absence of one or more paths from a

context node:

$$\begin{aligned}
 \mathcal{S}_q[\cdot] &: \text{Qualifier} \rightarrow \text{Node} \rightarrow \text{Boolean} \\
 \mathcal{S}_q[q_1 \text{ and } q_2]x &\stackrel{\text{def}}{=} \mathcal{S}_q[q_1]x \wedge \mathcal{S}_q[q_2]x \\
 \mathcal{S}_q[q_1 \text{ or } q_2]x &\stackrel{\text{def}}{=} \mathcal{S}_q[q_1]x \vee \mathcal{S}_q[q_2]x \\
 \mathcal{S}_q[\text{not } q]x &\stackrel{\text{def}}{=} \neg \mathcal{S}_q[q]x \\
 \mathcal{S}_q[p]x &\stackrel{\text{def}}{=} \mathcal{S}_p[p]x \neq \emptyset
 \end{aligned}$$

The semantics of paths relies on the navigational semantics of axes, given by the function  $\mathcal{S}_a$ :

$$\begin{aligned}
 \mathcal{S}_a[\cdot] &: \text{Axis} \rightarrow \text{Node} \rightarrow \text{Set}(\text{Node}) \\
 \mathcal{S}_a[\text{child}]x &\stackrel{\text{def}}{=} \text{children}(x) \\
 \mathcal{S}_a[\text{parent}]x &\stackrel{\text{def}}{=} \text{parent}(x) \\
 \mathcal{S}_a[\text{descendant}]x &\stackrel{\text{def}}{=} \text{children}^+(x) \\
 \mathcal{S}_a[\text{ancestor}]x &\stackrel{\text{def}}{=} \text{parent}^+(x) \\
 \mathcal{S}_a[\text{self}]x &\stackrel{\text{def}}{=} \{x\} \\
 \mathcal{S}_a[\text{descendant-or-self}]x &\stackrel{\text{def}}{=} \mathcal{S}_a[\text{descendant}]x \cup \mathcal{S}_a[\text{self}]x \\
 \mathcal{S}_a[\text{ancestor-or-self}]x &\stackrel{\text{def}}{=} \mathcal{S}_a[\text{ancestor}]x \cup \mathcal{S}_a[\text{self}]x \\
 \mathcal{S}_a[\text{preceding}]x &\stackrel{\text{def}}{=} \{y \mid y \ll x\} \setminus \mathcal{S}_a[\text{ancestor}]x \\
 \mathcal{S}_a[\text{following}]x &\stackrel{\text{def}}{=} \{y \mid x \ll y\} \setminus \mathcal{S}_a[\text{descendant}]x \\
 \mathcal{S}_a[\text{following-sibling}]x &\stackrel{\text{def}}{=} \{y \mid y \in \text{child}(\text{parent}(x)) \wedge x \ll y\} \\
 \mathcal{S}_a[\text{preceding-sibling}]x &\stackrel{\text{def}}{=} \{y \mid y \in \text{child}(\text{parent}(x)) \wedge y \ll x\}
 \end{aligned}$$

Path and axis navigation (illustrated on a sample tree by Figure 1.2) relies on a few assumed primitives over the XML tree data model:  $\text{root}()$  returns the root of the tree;  $\text{children}(x)$  which returns the set of nodes which are children of the node  $x$ ;  $\text{parent}(x)$  which returns the parent node of the node  $x$ ; the relation  $\ll$  which defines the ordering:  $x \ll y$  holds if and only if the node  $x$  is before the node  $y$  in the depth-first traversal order of the  $n$ -ary XML tree; and finally  $\text{name}()$  which returns the labeling of a node.

### 2.3 Logical Formalisms: Two Yardsticks

Unranked trees defined in Section 2.1.1 can be viewed as logical structures, in the sense of mathematical logic [Ebbinghaus and Flum, 2005]. In this vision, the domain of a tree  $t$ , viewed as a structure, is the set of nodes of  $t$ , denoted by  $\text{Dom}(t)$ . Formally,  $\text{Dom}(t)$  is the subset of  $\mathbb{N}^*$  defined as follows: if  $t = \sigma(t_1, \dots, t_n)$  with  $\sigma \in \Sigma$ ,  $n \geq 0$  and  $t_1, \dots, t_n \in \mathcal{T}_\Sigma^n$ , then  $\text{Dom}(t) = \{\epsilon\} \cup \{iu \mid i \in \{1, \dots, n\}, u \in \text{Dom}(t_i)\}$ . Thus,  $\epsilon$  represents the root while  $vj$  represents the  $j^{\text{th}}$  successor of  $v$ .

A relational vocabulary  $(\prec_{\text{ch}}, \prec_{\text{sb}}, \{O_\sigma \mid \sigma \in \Sigma\})$  is often used [Neven, 2002a, Barceló and Libkin, 2005, Bojanczyk et al., 2006]. In this vocabulary, the  $O_\sigma$

are unary relation predicates. For each  $\sigma$  label in the alphabet  $\Sigma$ ,  $O_\sigma$  is the set of nodes that are labeled with  $\sigma$ . The symbols  $\prec_{\text{ch}}$  and  $\prec_{\text{sb}}$  are binary predicates. The symbol  $\prec_{\text{ch}}$  is interpreted as the child relation: the set of pairs  $(v, v \cdot i)$  where  $v, v \cdot i \in \text{Dom}(t)$ . The symbol  $\prec_{\text{sb}}$  is the sibling order: the set of pairs  $(v \cdot i, v \cdot (i + 1))$  where  $v \cdot i, v \cdot (i + 1) \in \text{Dom}(t)$ .

Classically,  $\prec_{\text{ch}}^*$  is defined as the transitive-reflexive closure of  $\prec_{\text{ch}}$  (the descendant/ancestor relationship between two nodes), and  $\prec_{\text{sb}}^*$  as the transitive-reflexive closure of  $\prec_{\text{sb}}$  (the linear ordering on siblings).

Most formalisms used in the context of XML are related to one of the two logics used over these relational structures: first-order logic, and monadic second order logic:

- first-order logic and relatives are frequently used for query languages since they nicely capture their navigational features presented in the previous Section 2.2.2.
- monadic second order logic, which extends first-order logic by quantification over sets of nodes, is one of the most expressive (yet decidable) known logic. One of its main advantages in the context of XML is its ability to fully support XML types (regular tree languages).

The next sections are dedicated to these two logical formalisms, which are used as yardsticks logics in the XML setting. First-order logic is denoted by FO, and monadic second order logic by MSO. For XML applications, the relational vocabulary contains at least the labeling predicates  $O_\sigma$  for  $\sigma \in \Sigma$ , which are thus omitted from notations in the remaining. The rest of the vocabulary is listed between brackets. For example,  $\text{MSO}[\prec_{\text{ch}}, \prec_{\text{sb}}]$  refers to the vocabulary  $(\prec_{\text{ch}}, \prec_{\text{sb}}, \{X_\sigma \mid \sigma \in \Sigma\})$ . An important distinction between MSO and FO is that  $\prec_{\text{ch}}^*$  and  $\prec_{\text{sb}}^*$  are definable from  $\prec_{\text{ch}}$  and  $\prec_{\text{sb}}$  in MSO (using second-order quantification) but not in FO.

## 2.4 First Order Logic

Over a general relational structure, FO is undecidable, while its two-variable fragment is decidable [Mortimer, 1975]. Therefore, restricting FO to its two-variable fragment, denoted  $\text{FO}^2$ , has become a classical idea when looking for decidability [Grädel and Otto, 1999]. Furthermore, since  $\prec_{\text{ch}}^*$  and  $\prec_{\text{sb}}^*$  are not definable from  $\prec_{\text{ch}}$  and  $\prec_{\text{sb}}$  in FO,  $\text{FO}^2[\prec_{\text{ch}}^*, \prec_{\text{sb}}^*]$  is generally considered.

From the work found in [Genevès and Vion-Dury, 2004] and [Marx, 2004a], it is known that XPath expressive power is close to  $\text{FO}^2[\prec_{\text{ch}}^*, \prec_{\text{sb}}^*]$  that captures its navigational behavior. Specifically, in [Genevès and Vion-Dury, 2004], a  $\text{FO}^2[\prec_{\text{ch}}^*, \prec_{\text{sb}}^*]$  interpretation of an XPath fragment is given and proven correct w.r.t. to XPath denotational semantics presented in Section 2.2.2. The work found in [Marx, 2004a] characterizes the navigational fragment of XPath (introduced as “Core XPath” in [Gottlob et al., 2005]) and shows how it can be extended in order to be complete with respect to  $\text{FO}^2[\prec_{\text{ch}}^*, \prec_{\text{sb}}^*]$ .

The very recent work found in [Bojanczyk et al., 2006] proves the decidability of  $\text{FO}^2[\prec_{\text{ch}}, \prec_{\text{sb}}, \sim]$  where  $\sim$  is a binary predicate such that  $x \sim y$  holds for two nodes if they have the same data value. A consequence is the theoretical decidability of a limited form of comparison of data values in XPath. The corresponding decision procedure is observed to be between NEXPTIME

and 3-NEXPTIME, but unfortunately the approach gives no clue for a relevant effective algorithm [Bojanczyk et al., 2006].

FO nevertheless remains a convenient formalism for obtaining decidability results or theoretical characterizations of XPath queries. However, an argument in favor of MSO is that FO and its variants do not fully capture regular tree types [Benedikt and Segoufin, 2005] which make them unsuited for dealing with XML types.

## 2.5 Monadic Second-Order Logic

MSO over trees is one of the most expressive – yet decidable – logic known. It is known since the 1960’s that MSO exactly captures regular tree types. The appropriate MSO[ $\prec_{\text{ch}}, \prec_{\text{sb}}$ ] variant over finite binary trees is named *WS2S* which stands for *weak monadic second-order logic of two successors*. WS2S was introduced in [Thatcher and Wright, 1968, Doner, 1970]. In this calculus, first-order variables range over tree nodes. Second-order variables are interpreted as finite sets of tree nodes. *Weak* means that the set variables are allowed to range only over finite sets. This is enough since XML documents have an unbounded depth but remain finite trees. *Monadic* means that quantification is only allowed over unary relations (sets), not over polyadic relations. The *two successors* refer to the left and right successors of a node in the binary tree. They are sufficient to consider general unranked XML trees without loss of generality, owing to the mapping  $\beta(\cdot)$  presented in Section 2.5.1.

This section progressively introduces WS2S in detail, and explains how it is decided through the automaton-logic connection [Thatcher and Wright, 1968, Doner, 1970] using tree automata introduced in Section 2.1.4.

### 2.5.1 Preliminary Definitions

For notation consistency purposes, by convention, 0 is used for denoting the left successor and 1 for denoting the right successor of a node in a binary tree. The definition of the domain of a finite binary tree is thus slightly updated as follows. For  $t \in \mathcal{T}_\Sigma^2$ ,  $\text{Dom}(t)$  is defined as the subset of  $\{0, 1\}$  such that if  $t = \sigma(t_0, t_1)$  with  $\sigma \in \Sigma$  and  $t_0, t_1 \in \mathcal{T}_\Sigma^2$ , then  $\text{Dom}(t) = \{\epsilon\} \cup \{iu \mid i \in \{0, 1\}, u \in \text{Dom}(t_i)\}$ .  $\epsilon$  represents the root while  $vj$  represents the  $(j + 1)^{\text{th}}$  successor of  $v$ , for  $j \in \{0, 1\}$ . A node in the binary tree is thus a finite string over the alphabet  $\{0, 1\}$ .

The notion of *characteristic sets* is now defined, which further formalizes and generalizes the  $O_\sigma$  unary predicates introduced in Section 2.3 for the labeling. A *characteristic function* of a set  $B$  is a function from  $A$  to  $\{0, 1\}$ , where  $A$  is a superset of  $B$ . It returns 1 if and only if the element of  $A$  is also an element of  $B$ :

$$\begin{aligned} B &\subseteq A \\ f : A &\rightarrow \{0, 1\} \\ \forall a \in A, f(a) &= \begin{cases} 1, & \text{if } a \in B \\ 0, & \text{if } a \notin B \end{cases} \end{aligned}$$

A *characteristic set* is a subset of a set  $A$  that contains all elements of  $A$  for which the characteristic function returns 1:

$$\begin{aligned} X_f &\subseteq A \\ X_f &= \{a \in A \mid f(a) = 1\} \end{aligned}$$

In the following, characteristic sets of interest are subsets of  $\text{Dom}(t)$ , which denote where a particular property holds in a tree. Particular attention is paid to the characteristic sets  $X_{f_\sigma}$  which denote where a particular symbol  $\sigma$  occurs. Consider for instance the binary tree  $t = a(b(\epsilon, c(\epsilon, d)), \epsilon)$  over the alphabet  $\Sigma = \{a, b, c, d\}$ . It is identified by its tuple representation  $\tilde{t} = (X_{f_a}, X_{f_b}, X_{f_c}, X_{f_d})$  where  $X_{f_\sigma}$  is the characteristic set of the symbol  $\sigma$ :

$$\begin{aligned} X_{f_a} &= \{\epsilon\} \\ X_{f_b} &= \{0\} \\ X_{f_c} &= \{01\} \\ X_{f_d} &= \{011\} \end{aligned}$$

The set  $X_{f_a} \cup X_{f_b} \cup X_{f_c} \cup X_{f_d}$  of all positions contained in characteristic sets forms a *shape*.

A node belongs to a characteristic set  $X_{f_\sigma}$  (also noted  $X_\sigma$ ) if and only if the node is labeled by  $\sigma$ . Note that in the example of Figure 2.1, one and only one symbol occurs at each position. In the general case however, there is no restriction on the content of characteristic sets. A given node may belong to several characteristic sets. In this case, a node may be labeled by several symbols. This can be used to encode other properties than XML labeling. On the opposite, a particular position may not be a member of any characteristic set. In this case, the overall structure contains a node which is not labeled by any symbol of the considered alphabet; therefore it is no longer a labeled tree on this alphabet. Chapter 3 examines how XML trees can be encoded by constraining these structures using WS2S formulas introduced in the next section.

### 2.5.2 WS2S Formulas

From a syntactic point of view, WS2S formulas can be generated by a simple core language, whose abstract syntax follows:

$\mathcal{L}_{\text{ws2s}} \ni \Phi, \Psi ::=$		formula
	$X \subseteq Y$	inclusion
	$X = Y - Z$	difference
	$X = Y.0$	first successor
	$X = Y.1$	second successor
	$\neg\Phi$	negation
	$\Phi \wedge \Psi$	conjunction
	$\exists X.\Phi$	existential quantification

where  $X$ ,  $Y$ , and  $Z$  denote arbitrary second-order variables. Other usual logical connectives can be derived as syntactic sugars of the core:

$$\begin{aligned} \Phi \vee \Psi &\stackrel{\text{def}}{=} \neg(\neg\Phi \wedge \neg\Psi) \\ \Phi \Rightarrow \Psi &\stackrel{\text{def}}{=} \neg\Phi \vee \Psi \\ \Phi \Leftrightarrow \Psi &\stackrel{\text{def}}{=} \Phi \wedge \Psi \vee \neg\Phi \wedge \neg\Psi \\ \forall X.\Phi &\stackrel{\text{def}}{=} \neg\exists X.\neg\Phi \end{aligned}$$

Note that only second order variables appear in the core. This is because first order variables can be encoded as singleton second-order variables. A notation convention is adopted for simplifying the remaining part of the chapter: first-order variables are noted in lowercase and second-order variables in uppercase.

### 2.5.3 WS2S Semantics

This section gives an interpretation of WS2S formulas as finite subsets of  $\{0, 1\}^*$ . Given a fixed main formula  $\varphi$  with  $k$  variables, its semantics is defined inductively. Let a tuple representation  $\tilde{t} = (X_1, \dots, X_k) \in (\{0, 1\}^*)^k$  be an interpretation of  $\varphi$ . The notation  $\tilde{t}(X)$  denotes the interpretation  $X_i$  (such that  $1 \leq i \leq k$ ) that  $\tilde{t}$  associates to the variable  $X$  occurring in  $\varphi$ . The semantics of  $\varphi$  is inductively defined relative to  $\tilde{t}$ . The notation  $\tilde{t} \models \varphi$  (which is read:  $\tilde{t}$  satisfies  $\varphi$ ) is used if the interpretation  $\tilde{t}$  makes  $\varphi$  true:

$$\begin{aligned} \tilde{t} \models X \subseteq Y &\text{ iff } \tilde{t}(X) \subseteq \tilde{t}(Y) \\ \tilde{t} \models X = Y - Z &\text{ iff } \tilde{t}(X) = \tilde{t}(Y) \setminus \tilde{t}(Z) \\ \tilde{t} \models X = Y.0 &\text{ iff } \tilde{t}(X) = \{p.0 \mid p \in \tilde{t}(Y)\} \\ \tilde{t} \models X = Y.1 &\text{ iff } \tilde{t}(X) = \{p.1 \mid p \in \tilde{t}(Y)\} \\ \tilde{t} \models \neg\varphi &\text{ iff } \tilde{t} \not\models \varphi \\ \tilde{t} \models \varphi_1 \wedge \varphi_2 &\text{ iff } \tilde{t} \models \varphi_1 \text{ and } \tilde{t} \models \varphi_2 \\ \tilde{t} \models \exists X.\varphi &\text{ iff } \exists I \subseteq \{0, 1\}^*, \tilde{t}[X \mapsto I] \models \varphi \end{aligned}$$

where the notation  $\tilde{t}[X \mapsto I]$  denotes the tuple representation that interprets  $X$  as  $I$  and all other variables as  $\tilde{t}$  does. Note that the two successors of a particular position always exist in WS2S.

A formula  $\varphi$  naturally defines a language  $\mathcal{L}(\varphi) = \{\tilde{t} \mid \tilde{t} \models \varphi\}$  over the alphabet  $(\{0, 1\}^*)^k$ , where  $k$  is the number of variables of  $\varphi$ .

### 2.5.4 Equivalence of WS2S and FTA

It has been known since the 1960's that the class of regular tree languages is linked to decidability questions in formal logics. In particular, WS2S is decidable through the automaton-logic connection [Thatcher and Wright, 1968, Doner, 1970], using tree automata (introduced in Section 2.1.4). In 1968, Thatcher and Wright proved the following equivalence:

**Theorem 2.5.1** ([Thatcher and Wright, 1968]) *WS2S is as expressive as finite tree automata.*

The proof works in two directions. First, it is shown that a WS2S formula can be created such that it simulates a successful run of a tree-automaton. Second, for any given WS2S formula a corresponding tree automaton can be built.

Technically, the correspondence of WS2S formulas and tree automata relies on a convenient representation that links the truth status of a formula with the recognition operated by an automaton. This representation is a matricial vision of the tuple representation described in Section 2.5.1. Let  $\tilde{t}$  be a tuple, its matricial representation  $\tilde{t}$  is indexed by variables indices and positions in

the tree. Entries of  $\tilde{t}$  correspond to values in  $\{0, 1\}$  of characteristic functions: an entry  $(v, p) = 1$  in  $\tilde{t}$  means that the position  $p$  belongs to the variable  $X_v$ .

Consider for instance the formula  $\varphi = (\exists X. \exists Y. Y = Z. 0 \wedge X = Z. 1)$  which has three variables  $X, Y$ , and  $Z$ . A typical matrix looks like:

	$\epsilon$	0	00	01	010	1
$X$	1	1	0	0	0	0
$Y$	0	1	0	1	0	0
$Z$	0	0	1	0	0	1

Note that this matrix is finite since only finite trees are considered. It furthermore allows to capture finite trees of unbounded depth. As a counterpart, there is an infinite number of matrices that define the same interpretation: any number of columns of zeros may be appended at the right end of the matrix (for positions after the end of the tree). Let  $\tilde{t}$  be the minimum matrix, without such empty suffix. Rows of the matrix are called tracks and give the interpretation of each variable, which is defined as the finite set  $\{p \mid \text{the bit for position } p \text{ in the } X_i \text{ track is } 1\}$ .

Each column of the matrix is a bit vector that indicates the membership status of a node to the variables of the formula. The automaton recognizes all the interpretations (matrices) that satisfy the formula. A line by line reading of the matrix gives the interpretation of each variable (i.e. its associated set of positions), whereas an automaton processes the matrix column by column; it transits on each bit-vector.

### 2.5.5 From Formulas to Automata

Given a particular formula, a corresponding FTA can be built in order to decide the truth status of the formula.

Let  $\varphi$  be a formula with  $k$  second-order variables. As an interpretation of  $\varphi$ , consider a tuple representation  $\tilde{t} = (X_1, \dots, X_k) \in (\{0, 1\}^*)^k$ . The tree automaton that corresponds to  $\varphi$  is noted  $\mathcal{A}[\varphi]$ .  $\mathcal{A}[\varphi]$  operates over the alphabet  $\Sigma = \{0, 1\}^k$ , and can be seen as processing  $\tilde{t}$  column by column. Note however that there is an infinite number of matrices that defines the same interpretation. On one hand, any number columns of zeros can appear at the end of the matrix. On the other hand, a column of zeros can also appear for any position in the tree, before a non-empty column, denoting that this position is not a member of any interpretation. The automaton therefore faces a problem: when recognizing a column of zeros, knowing if the recognition should stop (because the end of the tree has been reached) or continue. In other terms, the automaton needs to know the maximal depth of the tree as an additional information in order to know when to stop. To this end, a new termination symbol  $\perp$  is introduced. From a matricial point of view, this symbol appears as a component of a bit-vector whenever this component will not be 1 anymore for the remaining bit-vectors to be processed. Technically,  $\mathcal{A}[\varphi]$  recognizes the tree representation  $\hat{t}$  of  $\tilde{t}$ .  $\hat{t}$  is obtained from  $\tilde{t}$  as follows:

1. the set of positions of  $\hat{t}$  is the prefix-closure of  $X_1 \cup \dots \cup X_k$
2. leaves of  $\hat{t}$  are labeled with  $\perp^k$

3. binary constructors of the tree are labeled with an element of  $\{\perp, 0, 1\}^k$  such that the  $i^{\text{th}}$  component of a position  $p$  in  $\hat{t}$  is marked: 1 if and only if  $p \in X_i$ , 0 if and only if  $p \notin X_i$  and some extension of  $p$  is in  $X_i$ , and  $\perp$  otherwise

Note that in this tree representation,  $\perp$  appears as a component of a node label whenever no descendant node has a 1 for the same component. For example, Figure 2.9 gives the tuple, the matrix, and the tree representation of a particular satisfying interpretation of the formula  $X \subseteq Y$ .

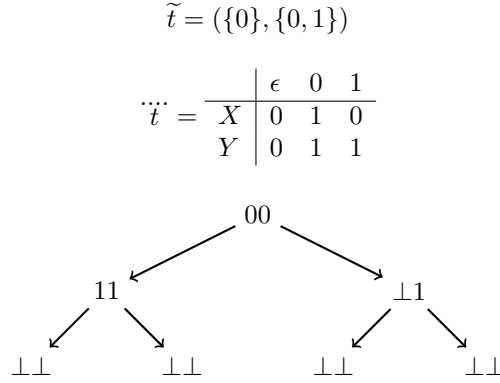


Figure 2.9: Representations of a Satisfying Interpretation of  $X \subseteq Y$

**Theorem 2.5.2** ([Thatcher and Wright, 1968, Doner, 1970]) *For every formula  $\varphi$ , there is an automaton  $\mathcal{A}[\varphi]$  such that:*

$$\tilde{t} \models \varphi \equiv \mathcal{A}[\varphi] \text{ accepts } \hat{t}$$

The automaton  $\mathcal{A}[\varphi]$  is calculated using an induction scheme. A basic bottom-up tree automaton corresponds to each atomic formula:

$$\mathcal{A}[X \subseteq Y] = \left( \left( \begin{array}{cc} q \leftarrow \perp\perp, & q \leftarrow \perp 0(q, q) \\ q \leftarrow \perp 1(q, q), & q \leftarrow 00(q, q) \\ q \leftarrow 01(q, q), & q \leftarrow 11(q, q) \end{array} \right), \{q\} \right)$$

$$\mathcal{A}[X = Y - Z] = \left( \left( \begin{array}{cc} q \leftarrow \perp\perp\perp, & q \leftarrow \perp\perp 0(q, q), \\ q \leftarrow \perp 0\perp(q, q), & q \leftarrow \perp 00(q, q), \\ q \leftarrow \perp 01(q, q), & q \leftarrow \perp 11(q, q), \\ q \leftarrow 0\perp\perp(q, q), & q \leftarrow 0\perp 0(q, q), \\ q \leftarrow 0\perp 1(q, q), & q \leftarrow 00\perp(q, q), \\ q \leftarrow 000(q, q), & q \leftarrow 001(q, q), \\ q \leftarrow 011(q, q), & q \leftarrow 11\perp(q, q), \\ q \leftarrow 110(q, q), & \end{array} \right), \{q\} \right)$$

$$\mathcal{A}[X = Y.0] = \left( \left( \begin{array}{cc} q \leftarrow \perp\perp, & q' \leftarrow 00(q, q') \\ q' \leftarrow 00(q', q) & q' \leftarrow 01(q'', q) \\ q'' \leftarrow 1\perp(q, q) & q'' \leftarrow 10(q, q) \end{array} \right), \{q'\} \right)$$

$$\mathcal{A}[\![X = Y.1]\!] = \left( \left\{ \begin{array}{ll} q \leftarrow \perp\perp, & q' \leftarrow 00(q, q') \\ q' \leftarrow 00(q', q) & q' \leftarrow 01(q, q'') \\ q'' \leftarrow 1\perp(q, q) & q'' \leftarrow 10(q, q) \end{array} \right\}, \{q'\} \right)$$

Logical connectives are then translated into automata-theoretic operations, taking advantage of the closure properties of tree automata (presented in Section 2.1.4). Formula conjunction is translated into intersection of automata:

$$\mathcal{A}[\![\varphi_1 \wedge \varphi_2]\!] = \mathcal{A}[\![\varphi_1]\!] \cap \mathcal{A}[\![\varphi_2]\!]$$

and negation is translated into automata complementation:

$$\mathcal{A}[\![\neg\varphi]\!] = \mathbb{C}(\mathcal{A}[\![\varphi]\!])$$

Existential quantification relies on projection and determinization of tree automata. The automaton  $\mathcal{A}[\![\exists X.\varphi]\!]$  is derived from  $\mathcal{A}[\![\varphi]\!]$  by projection. This means the alphabet of  $\mathcal{A}[\![\exists X.\varphi]\!]$  has to be one element smaller than the alphabet of  $\mathcal{A}[\![\varphi]\!]$ . In every tuple of  $\mathcal{A}[\![\varphi]\!]$  the X component is removed, so that its size is decreased by one. The rest of the automaton remains the same. Intuitively,  $\mathcal{A}[\![\exists X.\varphi]\!]$  acts as  $\mathcal{A}[\![\varphi]\!]$  except it is allowed to guess the bits for X. The automaton  $\mathcal{A}[\![\exists X.\varphi]\!]$  may be non-deterministic even if  $\mathcal{A}[\![\varphi]\!]$  was not [Comon et al., 1997], that is why determinization is required.

As a result, for every formula  $\varphi$  it is possible to build an automaton  $\mathcal{A}[\![\varphi]\!]$  in this manner, which defines the same language as  $\varphi$ :

$$\mathcal{L}(\mathcal{A}[\![\varphi]\!]) = \mathcal{L}(\varphi)$$

Analyzing the automaton  $\mathcal{A}[\![\varphi]\!]$  allows to decide the truth status of the formula  $\varphi$ :

- if  $\mathcal{L}(\mathcal{A}[\![\varphi]\!]) = \emptyset$  then  $\varphi$  is unsatisfiable;
- else  $\varphi$  is satisfiable. If  $\mathcal{L}(\mathbb{C}(\mathcal{A}[\![\varphi]\!])) = \emptyset$  then  $\varphi$  is always satisfiable (valid).

Possessing the full automaton corresponding to a formula is of great value, since it can be used for generating examples and counter-examples of the truth status of the formula. A relevant example (or counter-example) can be built by looking for an accepting run of the automaton (or its complement).

### 2.5.6 WS2S Complexity

Two factors have a major impact on the cost of a WS2S decision procedure:

1. the number of second-order variables in the formula
2. the number of states of the corresponding automaton (automaton size)

The number of second-order variables determines the alphabet size. More precisely, a formula with  $k$  variables is decided by an automaton operating on the alphabet  $\Sigma = \{0, 1\}^k$ . Representing the transition function  $\delta$  of such an automaton can be prohibitive. Indeed, in the worst case, the representation of a complete FTA requires  $2^k \cdot |Q|^3$  transitions where  $Q$  is the set of states of the automaton. A direct encoding with classical FTA such as the one described in Section 2.5.5 would lead to an impracticable algorithm. Modern logical

solvers represent transition functions using BDDs [Bryant, 1986] that can lead to exponential improvements [Klarlund and Møller, 2001, Tanabe et al., 2005].

As seen in Section 2.5.5, automaton construction is performed inductively by composing automata corresponding to each sub-formula. During this process, the number of states of intermediate automata may grow significantly. Automaton size depends on the nature of the automata-theoretic operation applied and the sizes of automata constructed so far. Each operation on tree automata particularly affects the size of the resulting automaton:

- Automata intersection causes a quadratic increase in automaton size in the worst case, as well as all binary WS2S connectors ( $\wedge$ ,  $\vee$ ,  $\Rightarrow$ ) that involve automata products [Klarlund et al., 2001].
- when considering deterministic complete automata, automata complementation corresponding to WS2S negation is a linear-time algorithm that consists in flipping accepting and rejecting states.
- The major source of complexity originates from automata determinization which may cause an exponential increase of the number of states in the worst case [Comon et al., 1997]. Logical quantification involves automaton projection (c.f. Section 2.5.5) which may result in a non-deterministic automaton, thus involving determinization. Hopefully, a succession of quantifications of the same type can be combined as a single projection followed by a single determinization. However, any alternation of second-order quantifiers requires a determinization, thus possibly causing an exponential increase of the automaton size.

As a consequence, the number of states of the final automaton corresponding to a formula with  $n$  quantifier alternations is in the worst case a tower of exponentials of height  $c \cdot n$  where  $c$  is some constant, and this is a lower bound [Stockmeyer and Meyer, 1973]. The translation from logical formulas to tree automata is thus *non-elementary*<sup>7</sup>:

**Theorem 2.5.3** [Meyer, 1975, Stockmeyer, 1974] *The satisfiability problem for WS2S formulas has an unbounded stack of exponentials as worst case lower bound.*

This high complexity, originating from the full construction and complementation of intermediate tree automata, is the counterpart of WS2S expressiveness and succinctness. Chapter 3 of this dissertation investigates how it is possible to deal with this complexity in practice, proposes a decision procedure for XPath containment based on WS2S along with optimizations of the WS2S decision procedure in the XML setting.

<sup>7</sup>The term *elementary* introduced by Grzegorzcyk [Grzegorzcyk, 1953] refers to functions obtained from some basic functions by operations of limited summation and limited multiplication. Consider the function *tower()* defined by:

$$\begin{cases} \textit{tower}(n, 0) = n \\ \textit{tower}(n, k + 1) = 2^{\textit{tower}(n, k)} \end{cases}$$

Grzegorzcyk has shown that every elementary function in one argument is bounded by  $\lambda n. \textit{tower}(n, c)$  for some constant  $c$ . Hence, the term *non-elementary* refers to a function that grows faster than any such function.

## 2.6 Temporal Logics

Some temporal and fixpoint logics closely related to FO and MSO have been introduced and allow to avoid explicit automata construction.

### 2.6.1 FO Relatives

For query languages, Computational Tree Logic (CTL) has been proposed in [Clarke and Emerson, 1981]. CTL is equivalent to FO over tree structures [Barceló and Libkin, 2005] and its satisfiability is in EXPTIME. The connection between XPath and FO relatives like CTL has been studied in [Marx, 2004b, Miklau and Suciu, 2004, Barceló and Libkin, 2005]. In particular, the work found in [Marx, 2004b] characterizes a subset of XPath in terms of extensions of CTL, whose satisfiability is in EXPTIME. Authors of [Miklau and Suciu, 2004] also observed that a fragment of XPath can be embedded in CTL. However, regular tree languages are not fully captured by FO [Benedikt and Segoufin, 2005]. These approaches are therefore not intended to support XML types.

In an attempt to reach more expressive power, the work that is presented in [Afanasiev et al., 2005] proposes a variant of Propositional Dynamic Logic (PDL) [Fischer and Ladner, 1979] with an EXPTIME complexity, but whose exact expressive power (as a strict subset of MSO) is still under study.

The goal of the XPath research presented so far is limited to establishing new theoretical properties and complexity bounds.

The research presented in this dissertation differs in that it seeks, in addition to the previous goals, efficient implementation techniques and concrete design that may be directly applied to XML type-checking problems involving XPath queries and regular tree types.

### 2.6.2 MSO Relatives

The propositional modal  $\mu$ -calculus introduced in [Kozen, 1983] has been shown to be as expressive as non-deterministic tree automata [Emerson and Jutla, 1991]. From [Arnold and Niwinski, 1992, Kupferman and Vardi, 1999], it is known that WS2S is exactly as expressive as the alternation-free fragment (AFMC) of the propositional modal  $\mu$ -calculus. The  $\mu$ -calculus subsumes all early logics such as CTL and PDL (see [Barceló and Libkin, 2005] for a recent survey on tree logics). The  $\mu$ -calculus is trivially closed under negation, can be extended with converse programs, and still remains decidable in EXPTIME [Vardi, 1998]. The best known complexity for the resulting logic is  $2^{O(n^4 \cdot \log n)}$  [Grädel et al., 2002]. As a counterpart of its substantially inferior complexity, it loses the succinctness of MSO. Fixpoint logics are indeed notorious for being difficult to understand, even for reasonably expert people, as pointed by [Bradfield and Stirling, 2001]. However, it is assumed in this dissertation that this is not a problem since the logic is only intended as a target for the compilation of XML concepts. As such, the  $\mu$ -calculus constitutes an interesting alternative for studying MSO-related problems. From a theoretical perspective, the AFMC with converse sounds as an appropriate logic for XML: it is expressive enough to capture a significant class of XPath decision problems, while offering an interesting balance between complexity and expressiveness.

The work found in [Tanabe et al., 2005] proposes a decision procedure for the AFMC, whose time complexity is  $2^{O(n \cdot \log n)}$ . However, models of the logic are Kripke structures (general infinite graphs), and the logic lacks the finite model property (i.e. there exist formulas which are satisfiable on Kripke structures and unsatisfiable on finite trees). In a preliminary work on XML type-checking, a logic for finite trees was presented [Tozawa, 2004], but the logic is not closed under negation.

Chapter 4 of this dissertation studies how the recent AFMC decision procedure proposed in [Tanabe et al., 2005] can be used in the context of XML. Based on the outcome of these investigations, the final Chapters 5 and 6 prove the decidability of a new logic for finite trees, derived from the  $\mu$ -calculus, in time  $2^{O(n)}$  and propose an effective algorithm for checking its satisfiability in practice.

## 2.7 Systems for XML Type-Checking

This section presents other related work on XML type-checking frameworks, which do not definitely aim at supporting XPath. Actually, none of the approach presented in this section is able to effectively deal with the expressive power of the XPath fragment considered in this dissertation (and presented in Section 2.2.1). Nevertheless, this section gathers the main approaches and ideas developed elsewhere for static type-checking in the XML setting. Although notably different, several approaches can be seen as complementary to the work proposed in this dissertation. Most techniques are based on regular tree languages and use tree automata introduced in Section 2.1.4.

### 2.7.1 Formulations of the Static Validation Problem

The paper [Audebaud and Rose, 2000] was influential in clearly defining the static validation problem. As an early attempt, it also proposes a set of typing rules to establish relationships between the input and output type of an XSLT transformation, but the method is only applicable to a tiny fragment of XSLT. The XML type-checking problem was later described in [Suciu, 2002]. A more recent survey work on the static type checkers for XML transformation languages can be found in [Møller and Schwartzbach, 2005]. The remaining part of this section presents the major known frameworks and innovations around the type-checking of XML.

### 2.7.2 Inverse Type Inference with Tree Transducers

The paper [Suciu, 2002] describes how static type-checking can be performed using forward type inference. Forward type inference refers to the ability to automatically deduce the output type of the XML document derived from the evaluation of an XML transformation. This is usually done by inference rules, and corresponding type inference algorithms are generally polynomial in the XML setting [Tozawa, 2001]. Type inference is used to do type-checking. For instance, if a program is assumed to return a type  $T_{\text{out}}$ ; once the inferred output type  $T_{\text{out}}^{\text{inf}}$  is known, type-checking can be performed by testing the inclusion  $T_{\text{out}}^{\text{inf}} \subseteq T_{\text{out}}$ . The work found in [Milo et al., 2003, Suciu, 2002] reveals

an important limitation of forward type inference in the context of XML: unfortunately, forward type inference is not complete. This is because the output type of a program may actually be a non-regular tree language that cannot be inferred. In that case, the inferred regular type is typically a larger approximation of the actual type, and the type-checker rejects the correct program, because  $T_{\text{out}}^{\text{inf}} \not\subseteq T_{\text{out}}$  (an example and details on this limitation can be found in [Suciu, 2002]).

The work found in [Milo et al., 2003] introduces the technique of inverse type inference in an attempt to overcome this problem. Inverse type inference computes the allowed input language for a so-called  $k$ -pebble transducer given its output language. The resulting algorithm has non-elementary complexity. The paper [Martens and Neven, 2003] investigates how the expressive power of tree transducers must be further restricted in order to allow a polynomial time decision algorithm. The practical relevance and usability of techniques based on tree transducers have not yet been demonstrated.

**XSLT0** The paper [Tozawa, 2001] examines a fragment of XSLT called XSLT0 which covers the structural recursion core of XSLT. It relies on inverse type inference to perform exact static validation, in the manner of [Milo et al., 2003] but with a more efficient (exponential time) algorithm. However, XSLT0 does not support XPath but only allows simple child steps in the recursion. Compiling XSLT into XSLT0 is thus possible for only the simplest transformations.

### 2.7.3 XDuce, CDuce, Xtatic

XDuce [Hosoya and Pierce, 2003] was the first domain specific programming language with type-checking of XML operations. The most essential part of the type system is the subtyping relation, which is defined by inclusion of the values represented by the types (this is also called *structural* subtyping<sup>8</sup>). The proposed algorithm for subtyping attempts to avoid the worst case exponential time complexity in practical cases. Instead of relying on tree automata determinization, it checks the inclusion relation by a top-down traversal of the original type expressions. XDuce’s algorithm builds on the previous work found in [Aiken and Murphy, 1991], and extends it with several implementation techniques. The resulting algorithm appears efficient in practice [Hosoya and Pierce, 2003]. XDuce has provided the foundation for later languages, in particular the CDuce [Benzaken et al., 2003, Frisch, 2004] and XStatic [Gapeyev and Pierce, 2003] languages. The CDuce language attempts to extend XDuce towards being a general purpose functional language. To this end, CDuce provides a more sophisticated type system featuring function types, intersection and negation types. It extends XDuce with higher-order functions, variations of pattern matching primitives, and parametric polymorphism [Hosoya et al., 2005a]. Xtatic aims at integrating the main ideas from XDuce into C#. All these languages support pattern-matching through regular expression types but not XPath. As pointed in [Colazzo et al., 2004], a major difference is that pattern-matching implements a *one-match* semantics,

---

<sup>8</sup>Structural subtyping is usually opposed to *nominal* subtyping in which type compatibility and equivalence are not determined by the type’s structure but through explicit declarations and names of the types. See [Su et al., 2002] and [Siméon and Wadler, 2003] for more details on subtyping paradigms.

i.e. every pattern, instead of collecting every matched piece of data (as in standard query languages such as XPath), only binds the first match. Although some recent work shows how to translate parts of XPath into Xtatic [Gapeyev and Pierce, 2004], the XPath fragment considered does not include reverse axes nor negation in qualifiers.

#### 2.7.4 Symbolic XML Schema Containment

The work found in [Tozawa and Hagiya, 2003] proposes a symbolic algorithm, based on binary decision diagrams [Bryant, 1986], in order to solve the containment between two XML schemas. The algorithm appears to be efficient in practice and favorably compares to the one used by XDuce. The idea of using symbolic techniques is similar to the one used in implementations presented in Chapters 4 and Chapter 6. The implicit encoding of FTA presented in [Tozawa and Hagiya, 2003] is however significantly simpler since it only considers XML types (XML types only use a simple form of tree navigation; they do not need upward nor multidirectional navigation in trees as XPath does). Nevertheless, this work was the first to reveal the interest of using implicit techniques in the context of XML. This work suggests and motivates further developments such as simplifications for particular cases of the more general symbolic techniques used in Chapters 4 and Chapter 6.

#### 2.7.5 XJ

The XJ [Harren et al., 2005] language aims at integrating XML processing closely into Java. Types are regular expressions over XML Schema declarations. The type system has two levels: regular expression operators and XML Schema declarations. A peculiarity of XJ is that subtyping on the schema level is *nominal*, i.e. type compatibility and containment is determined by explicit declarations and the name of the types (as in Java). This aspect contrasts with the structural subtyping systems used in XDuce (and in this dissertation). XJ subtyping on the regular expression level is defined as regular language inclusion on top of the schema subtyping. [Møller and Schwartzbach, 2005] argues that an inherited drawback of the underlying nominal style of subtyping is that a given XML value may be tied too closely with its schema type, which thus makes certain transformations more complex than they could be. XJ nevertheless provides an interesting experiment of integration of type-safe processing in Java, and a detailed study of nominal subtyping in the context of XML can be found in [Siméon and Wadler, 2003].

#### 2.7.6 Approximated Approaches for XSLT

Several approaches aim at proposing XSLT debugging features at compile-time by choosing to sacrifice exact decidability and to settle for pragmatic approximations instead. Among this line of work, the paper [Dong and Bailey, 2004] aims at conservatively analyzing the flow of an XSLT transformation. It uses the control-flow information to detect unreachable templates and guarantee termination. The analysis is however less precise than the more recent one found in [Møller et al., 2005]. The work [Møller et al., 2005] presents a more complete approximated technique that is able to statically detect errors in

XSLT stylesheets. Their approach could certainly benefit from using the exact algorithm proposed in Chapter 6 instead of their conservative approximation.

### 2.7.7 Path Correctness for $\mu$ XQ Queries

The work found in [Colazzo et al., 2006] proposes a sound and complete type system for ensuring path correctness for XML queries. The notion of navigation correctness is similar to the emptiness problem formulated in chapter 4.6 that can be used for detecting contradictions. The common idea is that if a subexpression of a query always yields an empty result then this should be considered as an error. The considered query language in [Colazzo et al., 2006], called  $\mu$ XQ, covers a minimal core of XQuery [Boag et al., 2006] but ignores reverse navigation. In comparison, the XPath fragment considered in this dissertation includes all axes. The algorithm presented in Chapter 6 may provide perspectives on how to extend the type system of [Colazzo et al., 2006] to deal with reverse navigation.

## 2.8 The Spatial Logic Perspective

Spatial logics are formalisms traditionally used for describing the behavior and spatial structure of concurrent systems. The main ingredient of spatial logics is an operator called composition (or separation), which usually permits reasoning over concurrent and mobile processes [Boneva and Talbot, 2005]. Spatial logics have recently been found useful in the study of semistructured data and related query languages as they allow to express properties about structures such as graphs [Cardelli et al., 2002, Dawar et al., 2004] and trees [Cardelli and Ghelli, 2004].

The work found in [Cardelli and Ghelli, 2004] proposes the TQL logic as the core of a query language for semistructured data represented as unranked trees and unordered trees. The TQL logic is based on the ambient logic [Cardelli and Gordon, 2000, Cardelli and Gordon, 2006]. It is known that TQL is more expressive than MSO since it can express some counting properties about trees that can not be defined in MSO. It has been shown that a fragment of the ambient logic contained in TQL is undecidable [Charatonik et al., 2003]. Nevertheless, decidable fragments of TQL could be useful for building type systems for semistructured data such as the one proposed in [Calcagno et al., 2003], and also for testing emptiness and containment of queries, as suggested in [Cardelli and Ghelli, 2004]. TQL thus provides an interesting foundation for further research.

The work found in [Boneva and Talbot, 2005] considers a fragment of TQL called STL and characterizes its expressiveness. STL satisfiability is shown undecidable but some syntactic restrictions over STL formulas allow to capture MSO.

The logic TL described in [Dal-Zilio et al., 2004] is also based on the ambient logic. TL can be encoded into the so-called sheaves automata proposed in [Dal-Zilio and Lugiez, 2003], whose transitions are conditioned by Presburger formulas.

The major difference between these spatial logics and the work presented in this dissertation is that spatial logics operates on *unordered* trees, whereas this dissertation considers ordered trees (cf. Section 2.1.1) such as structured

documents. On one hand, the extension of TQL's data model with ordering is an interesting and important open issue [Conforti et al., 2002]. On the other hand, extending the logic of ordered trees proposed in the Chapters 5 and 6 of this dissertation with counting constraints is also an interesting and promising perspective. These research directions can thus be seen as complementary and could certainly benefit from a reciprocal inspiration.

### 2.8.1 The Sheaves Logic

The work found in [Dal-Zilio and Lugiez, 2006] introduces a modal logic for documents called GDL, inspired from TQL, and proves the decidability of a fragment of GDL called the Sheaves logic. The Sheaves logic (SL) operates on ordered trees, and combines regularity and counting constraints. SL provides an interleaving operator for dealing with mixed ordered and unordered content. One one hand SL lacks recursion, i.e. fixpoint operators which are needed for supporting query languages (cf. Chapter 4); one the other hand SL allows to reason about numerical properties of the contents of elements, and may provide the inspiration for the integration of counting constraints in the logic presented in Chapter 5, kept for future work.



# Preliminary Investigations towards a Logic for XML



# Monadic Second-Order Logic for XML

---

## 3.1 Introduction

This chapter first investigates how MSO can be used in the context of XML, despite its non-elementary complexity<sup>1</sup>. A sound and complete decision procedure for containment of XPath queries is proposed based on MSO. Specifically, XPath queries are translated into equivalent formulas in WS2S introduced in Section 2.5.2. Using this translation, the logical formulation of the containment problem is constructed, and optimized, by taking into account XPath peculiarities. The containment formula is then decided using tree automata. When the containment relation does not hold between two XPath expressions, a counter-example XML tree is generated. A complexity analysis is provided, along with practical experiments.

**Chapter Outline** Section 3.2 presents the encoding of XML trees into WS2S. Section 3.3 explains the translation of XPath queries to logical formulas. A complexity analysis and an optimization method are given in Section 3.4. Experimental results and the outcome of this approach are respectively discussed in Sections 3.5 and 3.6.

## 3.2 Representation of XML Trees

Section 2.5.1 presented how characteristic sets can be used for describing shapes. A shape is basically a second order variable, interpreted as a set of nodes, for which particular properties hold. Using WS2S, this section now expresses additional requirements that a shape should fulfill in order to be an XML tree.

The first requirements are structural. First, in order to be a tree, a shape  $X$  must be prefix-closed, that is, for any position in the tree, any prefix of this

---

<sup>1</sup>It is well known that type inference for higher-order typed lambda calculi can have non-elementary complexity, and is nevertheless effectively used by typed functional programming languages such as those of the ML family [Henglein and Mairson, 1991].

position is also in the tree:

$$\text{PrefixClosed}(X) \stackrel{\text{def}}{=} \forall x.\forall y.((y = x.1 \vee y = x.0) \wedge y \in X) \Rightarrow x \in X$$

This ensures the shape is fully connected. Second, a predicate for the root of  $X$  is defined:

$$\text{IsRoot}(X, x) \stackrel{\text{def}}{=} x \in X \wedge \neg(\exists z.z \in X \wedge (x = z.1 \vee x = z.0))$$

In order to be a tree and not a hedge,  $X$  must have only one root with no sibling:

$$\text{SingleRoot}(X) \stackrel{\text{def}}{=} \forall x.\text{IsRoot}(X, x) \Rightarrow x.1 \notin X$$

Then, the labeling of the tree must be consistent with XML. The same symbol may appear at several locations in the tree with different arities: either as a binary constructor or as a leaf. However, one and only one symbol is associated with a position in the shape. Assume that the set of characteristic sets forms a partition:

$$\begin{aligned} \text{Partition}(X, X_1, \dots, X_n) &\stackrel{\text{def}}{=} X = \bigcup_{i=1}^n X_i \wedge \text{Disjoint}(X_1, \dots, X_n) \\ \text{Disjoint}(X_1, \dots, X_n) &\stackrel{\text{def}}{=} \bigwedge_{i \neq j} X_i \cap X_j = \emptyset \end{aligned}$$

this prevents a node to have multiple labels, but it also prevents a tree to be labeled using an infinite alphabet. The problem comes from declaring  $X = \bigcup_{i=1}^n X_i$  that prevents any other symbol to occur in the tree. Consider instead that the characteristic sets must be disjoint, then a position in the tree may not be a member of any of the considered characteristic sets. That is how labeling from an infinite alphabet is emulated. As a result, an XML tree is encoded in the following way:

$$\begin{aligned} \text{XMLTree}(X, X_1, \dots, X_n) &\stackrel{\text{def}}{=} \text{PrefixClosed}(X) \\ &\wedge \text{SingleRoot}(X) \\ &\wedge \text{Disjoint}(X_1, \dots, X_n) \\ &\wedge X \neq \emptyset \end{aligned}$$

where  $X$  is the tree (non-empty in order not to get degenerated results) and the  $X_i$ s are the characteristic sets. Figure 3.1 introduces how this is formulated in MONA Syntax [Klarlund and Møller, 2001], for the case of two characteristic sets of interest named **Xbook** and **Xcitation**. The only difference is that the shape  $X$  is declared as a global free variable named **\$** together with associated restrictions, instead of being passed as a parameter to predicates. In MONA syntax, “**var2**” is the keyword for declaring a free second-order variable; “**a111**” is the universal quantifier for first-order variables; and “**&**” and “**|**” respectively stand for the “**∧**” and “**∨**” connectives.

### 3.3 Interpretation of XPath Queries

This section explains how an XPath expression can be translated into an equivalent WS2S formula. This logical interpretation basically consists in considering a query as a relation that connects two tree nodes: a context node from which the query is applied, and a result node (selected by the query).

```

ws2s;
# Data Model
var2 $ where ~empty($)
  & (all1 x : all1 y : ((y=x.1 | y=x.0)
    & (y in $)) => x in $)
  & all1 r : (r in $ & ~(ex1 z : z in $
    & (r=z.1 | r=z.0)))
    => r.1 notin $;

# Characteristic sets
var2 Xbook, Xcitation;

# Partition
((all1 x : x in Xbook =>x notin Xcitation)
&(all1 x : x in Xcitation =>x notin Xbook));

```

Figure 3.1: Sample XML Tree in MONA WS2S Syntax.

### 3.3.1 Navigation and Recursion

As a first step toward a WS2S encoding of XPath expressions, the navigational primitives over binary trees must be expressed. Considering binary trees involves recursion for modeling the usual child relation on unranked trees (c.f. Figure 2.1 and the isomorphism between binary and unranked trees detailed in Section 2.1.1). Recursion is not available as a basic construct of WS2S. Recursion can be defined via a transitive closure formulated using second-order quantification.

The following-sibling relation is first expressed in WS2S. Consider a second-order variable  $F$  as the set of nodes of interest. The following-sibling relation is defined as an induction scheme. The base case just captures that the immediate right successor of  $x$  is effectively its first following sibling:

$$(x.1 \in F)$$

Then the induction step states that the immediate right successor of every position in  $F$  is also among the following siblings, and formulates this as a transitive closure:

$$\forall z.(z \in F \Rightarrow z.1 \in F)$$

The global requirement for a node  $y$  to be one of the following siblings of  $x$  is now formulated. The node  $y$  must belong to the set  $F$  which is closed under the following-sibling relation starting from  $x.1$ :

$$(x.1 \in F \wedge \forall z.z \in F \Rightarrow z.1 \in F) \Rightarrow y \in F$$

Note that this formula is satisfied for multiple sets  $F$ . For instance, the set of all tree nodes satisfies this implication. Actually, only the smallest set  $F$  for which the formula holds is of interest: the set which contains all and only all following siblings. A way to express this is to introduce a universal quantification over

$F$ . Indeed, ranging over all such set of nodes notably takes into account the particular case where  $F$  is minimal, i.e. the set of interest. If the global formula holds for every  $F$ ,  $y$  is also in the minimal set that contains only the following siblings of  $x$ . Therefore, the XPath “following-sibling” axis is defined as the WS2S predicate:

$$\begin{aligned} \text{followingsibling}(X, x, y) &\stackrel{\text{def}}{=} \forall F. F \subseteq X \Rightarrow \\ &((x.1 \in F \wedge \forall z. z \in F \Rightarrow z.1 \in F) \Rightarrow y \in F) \end{aligned}$$

that expresses the requirements for a node  $y$  to be a following sibling of a node  $x$  in the tree  $X$ . XPath “descendant” axis can be modeled in the same manner. The set  $D$  of interest is initialized with the left child of the context node, and is closed under both successor relations:

$$\begin{aligned} \text{descendant}(X, x, y) &\stackrel{\text{def}}{=} \forall D. D \subseteq X \Rightarrow \\ &(x.0 \in D \wedge \forall z. (z \in D \Rightarrow z.1 \in D \wedge z.0 \in D) \Rightarrow y \in D) \end{aligned}$$

Considering these two relations as navigational primitives, more complex ones can be built out of them:

$$\begin{aligned} \text{child}(X, x, y) &\stackrel{\text{def}}{=} y = x.0 \vee \text{followingsibling}(X, x.0, y) \\ \text{following}(X, x, y) &\stackrel{\text{def}}{=} \exists z. z \in X \wedge z.1 \in X \wedge \text{ancestor}(X, x, z) \\ &\quad \wedge \text{descendant}(X, z.1, y) \\ \text{self}(X, x, y) &\stackrel{\text{def}}{=} x = y \\ \text{descendantorself}(X, x, y) &\stackrel{\text{def}}{=} \text{self}(X, x, y) \vee \text{descendant}(X, x, y) \end{aligned}$$

Eventually, the other XPath axes are defined as syntactic sugars by taking advantage of XPath symmetry:

$$\begin{aligned} \text{ancestor}(X, x, y) &\stackrel{\text{def}}{=} \text{descendant}(X, y, x) \\ \text{parent}(X, x, y) &\stackrel{\text{def}}{=} \text{child}(X, y, x) \\ \text{precedingsibling}(X, x, y) &\stackrel{\text{def}}{=} \text{followingsibling}(X, y, x) \\ \text{ancestororself}(X, x, y) &\stackrel{\text{def}}{=} \text{descendantorself}(X, y, x) \\ \text{preceding}(X, x, y) &\stackrel{\text{def}}{=} \text{following}(X, y, x) \end{aligned}$$

### 3.3.2 Logical Composition of Steps

This section describes how path composition operators are translated into logical connectives. The translation is formally specified as a “derivor” shown on Figure 3.2 and written  $\mathcal{W}_e \llbracket e \rrbracket_x^y$  where:

- the parameter  $e$  (surrounded by special “syntax” braces  $\llbracket \rrbracket$ ) is the source language parameter that is rewritten;
- the additional parameters  $x$  and  $y$  are respectively the context and the result node of the query.

$$\begin{aligned}
 \mathcal{W}_e[\cdot] : Expression &\rightarrow Node \rightarrow Node \rightarrow \mathcal{L}_{\text{ws2s}} \\
 \mathcal{W}_e[/p]_x &\stackrel{\text{def}}{=} \exists z. \text{isroot}(z) \wedge \mathcal{W}_p[p]_z \\
 \mathcal{W}_e[p]_x &\stackrel{\text{def}}{=} \mathcal{W}_p[p]_x \\
 \mathcal{W}_e[e_1 \mid e_2]_x &\stackrel{\text{def}}{=} \mathcal{W}_e[e_1]_x \vee \mathcal{W}_e[e_2]_x \\
 \mathcal{W}_e[e_1 \cap e_2]_x &\stackrel{\text{def}}{=} \mathcal{W}_e[e_1]_x \wedge \mathcal{W}_e[e_2]_x \\
 \\
 \mathcal{W}_p : Path &\rightarrow Node \rightarrow Node \rightarrow \mathcal{L}_{\text{ws2s}} \\
 \mathcal{W}_p[p_1/p_2]_x &\stackrel{\text{def}}{=} \exists z. \mathcal{W}_p[p_1]_x^z \wedge \mathcal{W}_p[p_2]_z \\
 \mathcal{W}_p[p[q]]_x &\stackrel{\text{def}}{=} \mathcal{W}_p[p]_x \wedge \mathcal{W}_q[q]_y \\
 \mathcal{W}_p[a::\sigma]_x &\stackrel{\text{def}}{=} a(x, y) \wedge y \in X_\sigma \\
 \mathcal{W}_p[a::*]_x &\stackrel{\text{def}}{=} a(x, y) \\
 \\
 \mathcal{W}_q : Qualifier &\rightarrow Node \rightarrow \mathcal{L}_{\text{ws2s}} \\
 \mathcal{W}_q[q_1 \text{ and } q_2]_x &\stackrel{\text{def}}{=} \mathcal{W}_q[q_1]_x \wedge \mathcal{W}_q[q_2]_x \\
 \mathcal{W}_q[q_1 \text{ or } q_2]_x &\stackrel{\text{def}}{=} \mathcal{W}_q[q_1]_x \vee \mathcal{W}_q[q_2]_x \\
 \mathcal{W}_q[\text{not } q]_x &\stackrel{\text{def}}{=} \neg \mathcal{W}_q[q]_x \\
 \mathcal{W}_q[p]_x &\stackrel{\text{def}}{=} \exists y. \mathcal{W}_p[p]_x^y
 \end{aligned}$$

Figure 3.2: Translating XPath into WS2S.

The compilation of an XPath expression to WS2S relies on  $\mathcal{W}_p$  in charge of translating paths into formulas, and the dual derivor  $\mathcal{W}_q$  for translating qualifiers into formulas. The basic principle is that  $\mathcal{W}_p[p]_x^y$  holds for all pairs  $x, y$  of nodes such that  $y$  is accessed from  $x$  through the path  $p$ . Similarly,  $\mathcal{W}_q[q]_x$  holds for all nodes  $x$  such that the qualifier  $q$  is satisfied from the context node  $x$ .

The interpretation of path composition  $\mathcal{W}_p[p_1/p_2]_x^y$  consists in checking the existence of an intermediate node that connects the two paths, and therefore requires a new fresh variable to be inserted. The same holds for  $\mathcal{W}_e[/p]_x^y$  that restarts from the root to interpret  $p$ , whatever the current context node  $x$  is.

Paths can occur inside qualifiers therefore  $\mathcal{W}_e$ ,  $\mathcal{W}_p$  and  $\mathcal{W}_q$  are mutually recursive. Since the interpretations of paths and qualifiers are respectively dyadic and monadic formulas, the translation of a path inside a qualifier  $\mathcal{W}_q[p]_x$  requires the insertion of a new fresh variable whose only purpose consists in testing the existence of the path.

Eventually, the translation of steps relies on the logical definition of axes:  $a(x, y)$  denotes the WS2S predicate defining the XPath axis  $a$ , as described in Section 3.3.1. For instance, Figure 3.3 presents the WS2S translation of the

```

# Translated XPath expression:
# child::book/descendant::citation[parent::section]
ws2s;
# Data Model
var2 $ where ~empty($)
& (all1 x : all1 y : ((y=x.1 | y=x.0)
  & (y in $)) => x in $)
& all1 r : (r in $ & ~(ex1 z : z in $
  & (r=z.1 | r=z.0)))
  => r.1 notin $;

# Characteristic sets
var2 Xbook, Xcitation, Xsection;

# Partition
((all1 x: x in Xbook => x notin Xcitation
  & x notin Xsection)&
(all1 x: x in Xcitation => x notin Xbook
  & x notin Xsection)&
(all1 x: x in Xsection => x notin Xbook
  & x notin Xcitation));

# Query (parameters are context and result nodes)
pred xpath1 (var1 x, var1 y)=
  ex1 x1 : child(x,x1) & x1 in Xbook
  & descendant(x1,y) & y in Xcitation
  & ex1 x2 : parent(y,x2) & x2 in Xsection;

```

Figure 3.3: WS2S Translation of a Sample XPath in MONA Syntax.

XPath expression:

child::book/descendant::citation[parent::section]

### 3.3.3 Formulating XPath Containment

The XPath containment problem can now be expressed in terms of a logical formula. Given two XPath expressions  $e_1$  and  $e_2$ , the WS2S formula corresponding to checking their containment is built in two steps. First, each XPath expression is translated into a WS2S logical relation that connects two nodes in the tree, as presented in Section 3.3.2. Then the data model is unified. Each translation yields a set of characteristic sets. The union of them is built, so that characteristic sets that correspond to symbols used in both expressions are identified.

From a logical point of view,  $e_1 \subseteq e_2$  means that each pair of nodes  $(x, y)$  such that  $x$  and  $y$  are connected by the logical relation corresponding to  $e_1$  is

similarly connected by the logical relation obtained from  $e_2$ :

$$\forall x. \forall y. \mathcal{W}_e \llbracket e_1 \rrbracket_x^y \Rightarrow \mathcal{W}_e \llbracket e_2 \rrbracket_x^y \quad (3.1)$$

The containment relation holds between expressions  $e_1$  and  $e_2$  if and only if the WS2S formula (3.1) is satisfied for all trees. With respect to the notations of Section 3.2, the containment between expressions  $e_1$  and  $e_2$  is thus formulated as:

$$\forall X. \text{XMLTree}(X, X_1, \dots, X_n) \Rightarrow (\forall x \in X. \forall y \in X. \mathcal{W}_e \llbracket e_1 \rrbracket_x^y \Rightarrow \mathcal{W}_e \llbracket e_2 \rrbracket_x^y)$$

where the  $X_i$  are members of the union of all characteristic sets detected for each expression. Consider for instance the two XPath expressions:

$$\begin{aligned} e_1 &\stackrel{\text{def}}{=} \text{child::book/descendant::citation[parent::section]} \\ e_2 &\stackrel{\text{def}}{=} \text{descendant::citation[ancestor::book and ancestor::section]} \end{aligned}$$

Figure 3.4 presents the generated WS2S formula for checking containment between  $e_1$  and  $e_2$ , in MONA syntax. The formula is determined valid (which means  $e_1 \subseteq e_2$ ) in less than 0.2 seconds, the time spent to build the corresponding automaton and analyze it. The formula for the reciprocal containment check between  $e_2$  and  $e_1$  is satisfiable, which means  $e_2 \not\subseteq e_1$ . The total running time of the decision procedure is less than 0.9 seconds, including the generation of the counter-example, shown below:

```
<book>
  <section>
    <other>
      <citation/>
    </other>
  </section>
</book>
```

### 3.3.4 Soundness and Completeness

Soundness and completeness of the decision procedure for XPath Containment are ensured by construction. Indeed, consider the initial definition of the containment problem: provided a XML tree, checking containment between two XPath  $e_1$  and  $e_2$  consists in determining if the following proposition holds:

$$\forall x, \mathcal{S}_e \llbracket e_1 \rrbracket x \subseteq \mathcal{S}_e \llbracket e_2 \rrbracket x \quad (3.2)$$

By definition, (3.2) is logically equivalent to:

$$\forall x, \forall y, y \in \mathcal{S}_e \llbracket e_1 \rrbracket x \Rightarrow y \in \mathcal{S}_e \llbracket e_2 \rrbracket x \quad (3.3)$$

Then the last step remaining to prove is the equivalence between (3.3) and (3.1). To this end, the compilation of XPath expressions into WS2S formulas must preserve XPath denotational semantics, which means:

**Theorem 3.3.1** *The logical translation of XPath expressions is equivalent to XPath denotational semantics:*

$$\mathcal{W}_p \llbracket e \rrbracket_x^y \equiv y \in \mathcal{S}_p \llbracket e \rrbracket x \quad (3.4)$$

```
ws2s;
# Checking XPath Containment between
#'child::book/descendant::citation[parent::section]'
# and 'descendant::citation[ancestor::book
#                               and ancestor::section]'

# Data Model
var2 $ where ~empty($)
  & (all1 x : all1 y : ((y=x.1 | y=x.0)
    & (y in $)) => x in $)
  & all1 r : (r in $ & ~(ex1 z : z in $
    & (r=z.1 | r=z.0)))
    => r.1 notin $;

# Characteristic sets
var2 Xbook, Xcitation, Xsection;

# Queries (parameters are context and result nodes)
pred xpath1 (var1 x, var1 y)=
  ex1 x1 : child(x,x1) & x1 in Xbook
  & descendant(x1,y) & y in Xcitation
  & ex1 x2 : parent(y,x2) & x2 in Xsection;
pred xpath2 (var1 x, var1 y)=
  descendant(x,y) & y in Xcitation
  & ex1 x1 : (ancestor(y,x1) & x1 in Xbook)
  & ex1 x2 : (ancestor(y,x2) & x2 in Xsection);

# Problem formulation
((all1 x: x in Xbook => x notin Xcitation
  & x notin Xsection)&
  (all1 x: x in Xcitation => x notin Xbook
  & x notin Xsection)&
  (all1 x: x in Xsection => x notin Xbook
  & x notin Xcitation))
=>
(all1 x: all1 y: (xpath1(x,y)=> xpath2(x,y)));
```

Figure 3.4: Sample WS2S Formula for XPath Containment in MONA Syntax.

**Proof (Sketch)** The proof uses an induction over the structure of paths. Since the definition of paths and qualifiers is cross-recursive, a mutual induction scheme is used. The scheme relies on the dual property for qualifiers that also needs to be proved:

$$\forall p, \forall x, (\mathcal{S}_q \llbracket q \rrbracket x \equiv \mathcal{W}_q \llbracket q \rrbracket x) \quad (3.5)$$

Specifically (3.4) is proved by taking (3.5) as assumption, and reciprocally (3.5) is proved under (3.4) as assumption. Both equivalences (3.4) and (3.5) are proved inductively for each compositional layer. The idea basically consists in associating corresponding logical connectives to each set-theoretic composition operator used in the denotational semantics. XPath qualifier constructs trivially correspond to logical WS2S connectives. Path constructs involves set-theoretic union and intersection operations which are respectively mapped to logical disjunction and conjunction. Two path constructs:  $p_1/p_2$  and  $p[q]$  require specific attention in the sense their denotational semantics introduce particular compositions over sets of nodes. They are recalled below:

$$\begin{aligned} \mathcal{S}_p \llbracket p_1/p_2 \rrbracket x &\stackrel{\text{def}}{=} \{x_2 \mid x_1 \in \mathcal{S}_p \llbracket p_1 \rrbracket x \wedge x_2 \in \mathcal{S}_p \llbracket p_2 \rrbracket x_1\} \\ \mathcal{S}_p \llbracket p[q] \rrbracket x &\stackrel{\text{def}}{=} \{x_1 \mid x_1 \in \mathcal{S}_p \llbracket p \rrbracket x \wedge \mathcal{S}_q \llbracket q \rrbracket x_1\} \end{aligned}$$

Auxiliary lemmas are introduced in order to clarify how these constructs are mapped to WS2S. The XPath construct  $p_1/p_2$  is generalized as a function  $product()$ , whereas the XPath construct  $p[q]$  is generalized as  $filter()$ :

$$\begin{aligned} product() : \text{Set}(Node) &\rightarrow (Node \rightarrow \text{Set}(Node)) \rightarrow \text{Set}(Node) \\ filter() : \text{Set}(Node) &\rightarrow (Node \rightarrow Boolean) \rightarrow \text{Set}(Node) \end{aligned}$$

$product()$  is characterized by the lemmas (3.6) and (3.7), in which  $y$  and  $z$  are nodes, and  $S$  is a set of nodes. These lemmas abstract over XPath navigational functionalities performed by axes by letting  $f$  denoting a function that returns a set of nodes provided a current node:

$$\forall y, \forall z, \forall S, \forall f : Node \rightarrow \text{Set}(Node), z \in S \Rightarrow y \in (fz) \Rightarrow y \in product(S, f) \quad (3.6)$$

$$\forall y, \forall S, \forall f : Node \rightarrow \text{Set}(Node), y \in product(S, f) \Rightarrow \exists z, z \in S \wedge y \in (fz). \quad (3.7)$$

The function  $filter()$  is in turn characterized by the following lemma:

$$\forall y, \forall g : Node \rightarrow Boolean, y \in filter(S, g) \Rightarrow y \in S \quad (3.8)$$

The auxiliary lemmas (3.6), (3.7), and (3.8) are also proved by induction. Developing the proof in constructive logic involves the (trivial) decidability of set-theoretic inclusion and of the denotational semantics of qualifiers. The full formal proof is detailed in [Genevès and Vion-Dury, 2004]. It has been mechanically checked by the machine using the Coq formal proof management system [Huet et al., 2004].

### 3.4 Complexity Analysis and Optimization

The translation of an XPath query to its logical representation is linear in the size of the input query. Indeed, each expression is decomposed then translated inductively in one pass without any duplication, as shown by the formal definition of  $\mathcal{W}_e$  in Section 3.3.2.

The second step is the decision procedure, which, compared to the translation, represents the major part of the cost. The truth status of a WS2S formula is decided throughout the logic-automaton connection as described in Sections 2.5.4 and 2.5.5 of previous Chapter 2. This translation from logical formulas to tree automata, while effective, is unfortunately non-elementary. This bound may sound discouraging. Fortunately, the worst-case scenario which corresponds to complex formulas, is not likely to occur for small instances of the containment in practice. Furthermore, recent works on MSO solvers - especially those using BDDs techniques [Bryant, 1986] such as MONA [Klarlund and Møller, 2001] - suggest that in particular practical cases the explosiveness of this technique can be effectively controlled.

In practice, the implementation relies on MONA [Klarlund and Møller, 2001] that implements the WS2S decision procedure along with various optimizations. Additionally, a significant optimization that takes advantage of XPath peculiarities for combating automaton size explosion is described in the following subsection.

### 3.4.1 Optimization Based on Guided Tree Automata

A major source of complexity arises from the translation of composed paths. Each translation of the form  $\mathcal{W}_p[[p_1/p_2]]_x^y$  introduces an existentially quantified first-order variable which ranges over all possible tree positions (c.f. Figure 3.5).

The idea in this section is to take advantage of XPath navigational peculiarities for attempting to reduce the scope associated to such variables. XPath navigates the tree step by step: each step selects a set of nodes which is in turn used to select a new one by the next step. The interpretation of a variable inserted during the translation of  $p_1/p_2$  corresponds to the intermediate node which is a result of  $p_1$  and the context node of  $p_2$ . The truth status of the formula is determined by the existence of such an intermediate node at a particular position in the tree. If one can determine regions in the tree in which such a node may appear from those where it cannot appear, valuable positional knowledge is gained that can be used to reduce the variable scope. It is interesting to try to identify the region in the tree (or even some larger approximation) in which the node must be located in order for the formula to be satisfied. XPath sequential structure of steps makes it possible to exploit such positional knowledge. Indeed, consider for instance the expression:

$$e_3 \stackrel{\text{def}}{=} /child::book/descendant::*[child::citation]$$

$e_3$  navigates from the document root through its “book” children elements and then selects all descendant nodes provided they have at least one child named “citation”. Several conditions must be satisfied by a tree  $t_1$  in order to yield a result for  $e_3$ :

- $t_1$  must have at least one “book” element as a child of the root;
- $t_1$  must have at least one element that must be a descendant of the “book” element;
- for this node to be selected it must have at least one child named “citation”.

```

e1(x,y) = ex1 x1 : isroot(x1) & x1 in $
  & ex1 x2 : child(x1,x2) & x2 in Xbook
  & descendant(x2,y) & y in $
  & ex1 x3 : child(y,x3) & x3 in Xcitation;

```

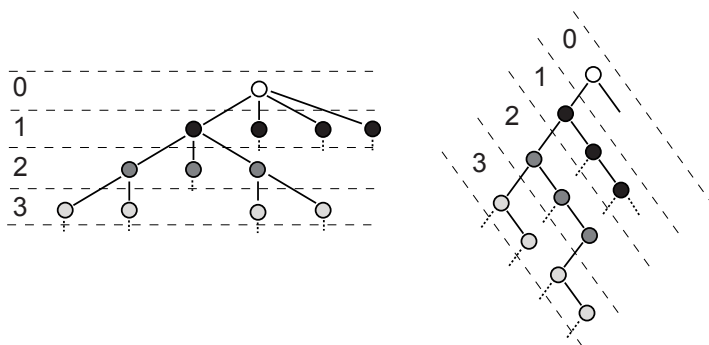
Figure 3.5: WS2S Translation of  $e_3$  in MONA Syntax.

Figure 3.6: Depth Levels in the Unranked and Binary Cases.

This is made explicit by the logical translation  $\mathcal{W}_e[[e_3]]_x^y$  in MONA syntax shown on Figure 3.5. In this translation,  $x1$ ,  $x2$  and  $x3$  denote the respective positions of the root node, a “book” child, and a “citation” child of the selected position  $y$ . These variables actually only range over a particular set of positions in the tree. By definition, the root can only appear at depth level 0, the “book” element can only occur at level 1 and its descendants occur at any depth level  $l$  greater or equals to 2. Eventually, the “citation” element should occur at level  $l + 1$ . This is because each step introduces its particular positional constraint which can be propagated to the next steps.

The idea of taking advantage of positional knowledge is even more general. Theoretically, normal bottom-up FTA are sufficient for deciding validity of a WS2S formula (as presented in Section 2.5.4 of Chapter 2). However composition of such automata is particularly sensitive to state space explosion, as presented in Section 2.5.6. Guided tree automata (GTA) [Biehl et al., 1997] have been introduced in order to combat such state space explosion by following the divide and conquer approach. A GTA is just an ordinary FTA equipped with an additional deterministic top-down tree automaton called the guide. The latter is introduced to take advantage of positional knowledge, and used for partitioning the FTA state space into independent subspaces. Top-down deterministic automata are strictly less powerful than ordinary (bottom-up or non-deterministic top-down) FTA [Comon et al., 1997]. However, this is not a problem since the guide is only intended to provide additional auxiliary information used for optimization purposes. As a consequence, the more precise is the guide, the more efficient is the decision procedure, but an approximation is sufficient. The guide basically splits the state space of the FTA in independent

subsets. Therefore the transition relation of the bottom-up automaton is split into a family of transition functions, one for each state space name. A state space name corresponds to a particular depth level or a set of depth levels. GTA can be composed in the same way than ordinary FTA as explained in Section 2.5.4 of Chapter 2. A GTA can be seen as an ordinary tree automaton, where the state space has been factorized according to the guide. A GTA with only one state space is just an ordinary tree automaton. A detailed description of GTA can be found in [Biehl et al., 1997]. GTA-based optimization may lead to exponential improvements of the decision procedure [Elgaard et al., 2000].

A tree partitioning based on the depth levels is now introduced. It is depicted by Figure 3.6 for a  $n$ -ary sample tree and its binary counterpart. Based on this partitioning, a positional constraint (a restricted set of depth levels) is associated to each node variable. Indeed, a node referred by an XPath can occur at several depth levels since some axes involve transitive closure (c.f. Section 2.2.2 of Chapter 2). Moreover, the set of depth levels can even be infinite since XPath offers recursion in unbounded trees.

The computation of sets of depth levels is calculated by the function shown on Figure 3.7, and written  $L_e[[e]]_N$  where  $e$  is the XPath expression to be analyzed and  $N$  is the set of positional constraints corresponding to the context node from which  $e$  is applied. Again, the algorithm proceeds inductively on the structure of XPath expressions. XPath steps are base cases for which the set of levels is effectively calculated from the previous one. Transitive closure axes such as “descendant” turn the set of depth levels into an infinite one, even if the previous was finite. Path composition basically propagates the level calculations by combining with the base cases. Note that an important precision can be gained with absolute XPath expressions. In this case, the initial set of depth levels is the singleton  $\{0\}$  as opposed to relative XPath expressions for which the context node is not known and the initial set of depth levels is subsequently  $\mathbb{N}$ .

The optimized compilation of XPath expressions to WS2S formulas is given on Figure 3.8.  $\mathcal{W}'_e$ ,  $\mathcal{W}'_p$  and  $\mathcal{W}'_q$  are respective optimized versions of  $\mathcal{W}_e$ ,  $\mathcal{W}_p$  and  $\mathcal{W}_q$ , which convey a set of depth levels as an additional parameter passed to  $L_e$  and  $L_p$ . These functions compute the restrictions on variable scope that are inserted by  $\mathcal{W}'_p$  and  $\mathcal{W}'_q$ . “ $\exists z [D]$ ” denotes the fact that the existentially quantified first-order variable  $z$  is restricted to appear at a depth level among the set of depth levels  $D$ . In practice,  $L_e$  and  $L_p$  can be merged into  $\mathcal{W}'_e$  and can be implemented in a single pass over the XPath expression. Thus the translation and the depth level computation remain linear in the size of the query.

MONA provides an implementation of GTA. The application of the previous algorithm to  $e_3$  leads to the logical formulation shown on Figure 3.9 in MONA syntax.

The guide obtained in this translation means that the root is labeled with “10”; its left and right successor nodes are labeled with “11” and “epsilon” respectively. The “epsilon” is a dummy state space reflecting the fact that the underlying shape is a tree and not a hedge. No variable is associated with this state space. The “lothers” state space represents any tree node occurring at a depth level greater than 3. Such a state space is associated with variables whose scope is of unbounded depth. The size of the guide depends on the maximum depth level found among the computed restrictions. Formally, a

$$\begin{aligned}
 L_e &: \mathcal{L}_{\text{XPath}} \rightarrow \text{Set}(\text{Int}) \rightarrow \text{Set}(\text{Int}) \\
 L_e \llbracket /p \rrbracket_N &\stackrel{\text{def}}{=} L_p \llbracket p \rrbracket_{\{0\}} \\
 L_e \llbracket p \rrbracket_N &\stackrel{\text{def}}{=} L_p \llbracket p \rrbracket_{\mathbb{N}} \\
 L_e \llbracket e_1 \mid e_2 \rrbracket_N &\stackrel{\text{def}}{=} L_e \llbracket e_1 \rrbracket_N \cup L_e \llbracket e_2 \rrbracket_N \\
 L_e \llbracket e_1 \cap e_2 \rrbracket_N &\stackrel{\text{def}}{=} L_e \llbracket e_1 \rrbracket_N \cap L_e \llbracket e_2 \rrbracket_N \\
 \\ 
 L_p &: \text{Path} \rightarrow \text{Set}(\text{Int}) \rightarrow \text{Set}(\text{Int}) \\
 L_p \llbracket p_1/p_2 \rrbracket_N &\stackrel{\text{def}}{=} L_p \llbracket p_2 \rrbracket_{L_p \llbracket p_1 \rrbracket_N} \\
 L_p \llbracket p[q] \rrbracket_N &\stackrel{\text{def}}{=} L_p \llbracket p \rrbracket_N \\
 L_p \llbracket \text{self}::n \rrbracket_N &\stackrel{\text{def}}{=} N \\
 L_p \llbracket \text{child}::n \rrbracket_N &\stackrel{\text{def}}{=} \{n+1 \mid n \in N\} \\
 L_p \llbracket \text{parent}::n \rrbracket_N &\stackrel{\text{def}}{=} \{n-1 \mid n \in N\} \\
 L_p \llbracket \text{descendant}::n \rrbracket_N &\stackrel{\text{def}}{=} \{n' \mid n \in N \wedge n' > n\} \\
 L_p \llbracket \text{descendant-or-self}::n \rrbracket_N &\stackrel{\text{def}}{=} \{n' \mid n \in N \wedge n' \geq n\} \\
 L_p \llbracket \text{ancestor}::n \rrbracket_N &\stackrel{\text{def}}{=} \{n' \mid n \in N \wedge n' >= 0 \wedge n' < n\} \\
 L_p \llbracket \text{ancestor-or-self}::n \rrbracket_N &\stackrel{\text{def}}{=} \{n' \mid n \in N \wedge n' >= 0 \wedge n' \leq n\} \\
 L_p \llbracket \text{following}::n \rrbracket_N &\stackrel{\text{def}}{=} \mathbb{N} - \{0\} \\
 L_p \llbracket \text{preceding}::n \rrbracket_N &\stackrel{\text{def}}{=} \mathbb{N} - \{0\} \\
 L_p \llbracket \text{following-sibling}::n \rrbracket_N &\stackrel{\text{def}}{=} N \\
 L_p \llbracket \text{preceding-sibling}::n \rrbracket_N &\stackrel{\text{def}}{=} N
 \end{aligned}$$

Figure 3.7: Computation of the Depth Levels of Nodes Selected by a Path.

guide for a maximum depth level  $n$  is a top-down deterministic tree automaton with  $\{q_0, \dots, q_{n+1}\} \cup \{q_\epsilon\}$  as set of states,  $q_0$  as the single initial state, and the following set of transitions:

$$\begin{aligned}
 &\{q_0 \rightarrow (q_1, q_\epsilon)\} \\
 \cup &\{q_i \rightarrow (q_{i+1}, q_i) \mid i \in [1..n]\} \\
 \cup &\{q_{n+1} \rightarrow (q_{n+1}, q_{n+1})\} \\
 \cup &\{q_\epsilon \rightarrow (q_\epsilon, q_\epsilon)\}
 \end{aligned}$$

where  $q_i$  ( $i \in [0..n]$ ) denotes the state space name corresponding to the depth level  $i$ , and  $q_{n+1}$  represents all depth levels greater or equal to  $n+1$ . For formulating the XPath containment, the guide is computed from the two XPath expressions. Specifically, the deepest (and thus the most precise) guide is chosen as the guide for both expressions.

Eventually, each variable is restricted with a list of state spaces that represents the regions in the tree where its valuation must be searched. For instance,

$$\begin{aligned}
 \mathcal{W}'_e &: \mathcal{L}_{\text{XPath}} \rightarrow \text{Node} \rightarrow \text{Node} \rightarrow \text{Set}(\text{Int}) \rightarrow \mathcal{L}_{\text{ws2s}} \\
 \mathcal{W}'_e[\!/p](x, y, N) &\stackrel{\text{def}}{=} \exists z [\{0\}] . \text{isroot}(z) \wedge \mathcal{W}'_p[p](z, y, \{0\}) \\
 \mathcal{W}'_e[p](x, y, N) &\stackrel{\text{def}}{=} \mathcal{W}'_p[p](x, y, \mathbb{N}) \\
 \mathcal{W}'_e[e_1 \mid e_2](x, y, N) &\stackrel{\text{def}}{=} \mathcal{W}'_e[e_1](x, y, N) \vee \mathcal{W}'_e[e_2](x, y, N) \\
 \mathcal{W}'_e[e_1 \cap e_2](x, y, N) &\stackrel{\text{def}}{=} \mathcal{W}'_e[e_1](x, y, N) \wedge \mathcal{W}'_e[e_2](x, y, N) \\
 \\ 
 \mathcal{W}'_p &: \text{Path} \rightarrow \text{Node} \rightarrow \text{Node} \rightarrow \text{Set}(\text{Int}) \rightarrow \mathcal{L}_{\text{ws2s}} \\
 \mathcal{W}'_p[p_1/p_2](x, y, N) &\stackrel{\text{def}}{=} \exists z [L_p[p_1]_N] . \mathcal{W}'_p[p_1](x, z, N) \wedge \mathcal{W}'_p[p_2](z, y, N) \\
 \mathcal{W}'_p[p[q]](x, y, N) &\stackrel{\text{def}}{=} \mathcal{W}'_p[p](x, y, N) \wedge \mathcal{W}'_q[q](y, N) \\
 \mathcal{W}'_p[a::\sigma](x, y, N) &\stackrel{\text{def}}{=} a(x, y) \wedge y \in X_\sigma \\
 \mathcal{W}'_p[a::*](x, y, N) &\stackrel{\text{def}}{=} a(x, y) \\
 \\ 
 \mathcal{W}'_q &: \text{Qualifier} \rightarrow \text{Node} \rightarrow \text{Set}(\text{Int}) \rightarrow \mathcal{L}_{\text{ws2s}} \\
 \mathcal{W}'_q[q_1 \text{ and } q_2](x, N) &\stackrel{\text{def}}{=} \mathcal{W}'_q[q_1](x, N) \wedge \mathcal{W}'_q[q_2](x, N) \\
 \mathcal{W}'_q[q_1 \text{ or } q_2](x, N) &\stackrel{\text{def}}{=} \mathcal{W}'_q[q_1](x, N) \vee \mathcal{W}'_q[q_2](x, N) \\
 \mathcal{W}'_q[\text{not } q](x, N) &\stackrel{\text{def}}{=} \neg \mathcal{W}'_q[q](x, N) \\
 \mathcal{W}'_q[p](x, N) &\stackrel{\text{def}}{=} \exists y [L_p[p]_N] . \mathcal{W}'_p[p](x, y, N)
 \end{aligned}$$

Figure 3.8: Translating XPath into WS2S with Restricted Variable Scopes.

```

guide l0 -> (l1, epsilon),
      l1 -> (l2, l1),
      l2 -> (l3, l2),
      l3 -> (lothers, l3),
      lothers -> (lothers, lothers),
      epsilon -> (epsilon, epsilon);

e1(x,y)= ex1 [l0] x1 : (isroot(x) & x=x1 & x in $)
& ex1 [l1] x2 : child(x1,x2) & x2 in Xbook
& descendant(x2,y) & y in $
& ex1 [l3, lothers] x3 : child(y,x3)
& x3 in Xcitation;
    
```

 Figure 3.9: Optimized WS2S Translation of  $e_3$  in MONA Syntax.

“`ex1 [11] x2`” means the scope of the variable `x2` is limited to tree nodes occurring at depth level 1.

This optimization is useful for both kinds of XPath expressions: absolute and relative. More precise restrictions can be computed for absolute XPath expressions (for which the initial set of depth levels is the singleton  $\{0\}$ ).

### 3.5 Implementation and Experiments

The approach has been implemented. A compiler (written in Java) takes XPath expressions and translates them into WS2S formulas. A Java interface controls the C++ implementation of the MONA WS2S solver, and in addition provides precise runtime statistics on the decision procedure.

The evolution of the intermediate automata (in terms of states, number of BDD nodes involved, the minimizations, products, projections...) are reported in realtime during a run of the decision procedure. For example, Figure 3.10 shows detailed statistics on the intermediate automata built during the comparison of the following two XPath expressions  $e_4$  and  $e_5$ :

$$e_4 \stackrel{\text{def}}{=} a/b[\text{descendant}::c]/\text{following-sibling}::d/e$$

$$e_5 \stackrel{\text{def}}{=} a/d[\text{preceding-sibling}::b]/e$$

The horizontal axes of charts of Figure 3.10 correspond to the number of automata operations. In that case, 380 operations were needed to complete the XPath containment test. Once the decision procedure terminates, the result of the comparison is displayed in the console:

```
"a/b[descendant::c]/following-sibling::d/e" is contained in
"a/d[preceding-sibling::b]/e" [Total Time: 00:00:00.18]
```

Extensive tests have been carried out with the implementation. Tests have been reported in [Genevès and Layaida, 2006]. They are not detailed here, since it is difficult to come up with a clear conclusion based on the observed practical behavior of this decision procedure on a few instances. Instead, only the major lessons learned from the practical experiments are summarized:

- The GTA-based optimization has been observed to be particularly useful as guides cause a small overhead compared to the significant performance gains they provide on many instances. Some containment instances cannot be solved without this optimization.
- For small expressions (that are most likely to occur in practice in XSLT transformations, as suggested by [Møller et al., 2005]), it has been observed over many instances that the implementation can run in acceptable time and space bounds. Since this approach is sound and complete over a large XPath fragment, it provides an interesting alternative to the less complex but incomplete decision procedure over a very restricted XPath fragment previously studied in the literature [Miklau and Suciu, 2004].
- For larger XPath expressions however, intermediate tree automata constructed can be so large that blow-ups are observed, even using GTA. Practical experiments notably suggest that the WS2S decision procedure

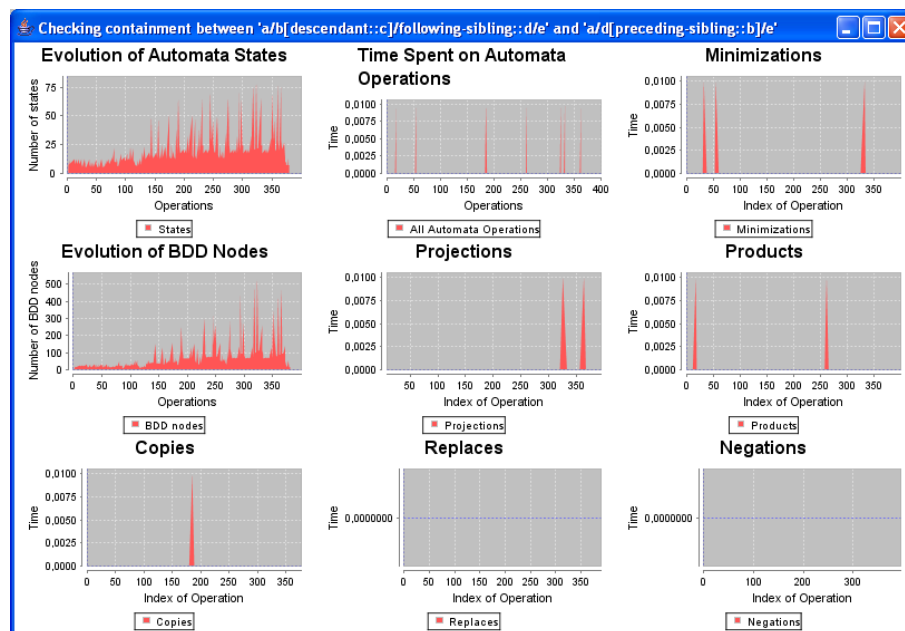


Figure 3.10: Statistics on Intermediate Automata for a Containment Check.

implemented in MONA is particularly sensitive to the alphabet size, which clearly makes the approach inappropriate for XPath expressions that use a large number of tag names.

- The explosiveness of the approach is very difficult to control in practice. It is possible to find relatively small expressions for which blow-ups cannot be controlled, even by the GTA-based optimization. Subsequently, there exist relatively small XPath containment instances for which containment cannot be decided in acceptable time and space bounds.
- As a result, no clear conclusion can be drawn from the experiments, concerning the maximum size and complexity of XPath expressions for which this procedure could offer practical guarantees. Such a characterization is made very difficult by the huge number of parameters that must be taken into account, due to all the optimizations implemented in MONA [Klarlund et al., 2001]. It is thus very difficult to estimate up to which XPath expression size and complexity this decision procedure can be used in practice. Observed results on tested instances suggest that this approach may be efficient for XPath expressions using less than 10 tag names, and indicate that it cannot be reasonably used with larger alphabets.

### 3.6 Outcome

An approach based on MSO has been proposed for the XPath containment problem: query containment is formulated in terms of a WS2S formula, which is then decided using tree automata. An optimization method based on guided

tree automata is proposed in an attempt to take advantage of XPath peculiarities in order to improve time and space requirements of the complex decision procedure.

An advantage of the approach is that it provides a sound and complete decision procedure for a large XPath fragment. Another advantage of this technique is to allow generation of tree examples and counter-examples of the truth status of the formula.

The major drawback of this approach, however, is that the decision procedure is based on the full construction and complementation of the intermediate automata. This makes the explosiveness of the approach very hard to control in practice and unfortunately restricts its use to only small XPath expressions.

Surprisingly enough, the full construction and determinization of intermediate FTA often seems unnecessary. Indeed huge intermediate automata are almost always reduced by following projection operations. This can be observed on most practical scenarios owing to the detailed statistics reported by the implementation (see for instance the peaks in the evolution of intermediate automata states on Figure 3.10). The determinization of huge intermediate automata is the source of uncontrollable blow-ups in practice. On many instances, it has been observed that the memory representation of intermediate automata may require several hundreds of megabytes (or even several gigabytes which is not affordable on most current machines), even if this appears to be unnecessary since the final resulting automaton is only of several kilobytes in size.

One direction of future work is to search for tree automata guides that produce a finer-grained partition of the automaton state space, in order to enhance the scalability of the decision procedure. Another perspective is to search for approaches that do not construct unnecessary parts of intermediate automata, or even do not construct automata at all. This is the motivation that underlies investigations presented in the next chapter.



# XML and the Modal $\mu$ -Calculus

---

## 4.1 Introduction

Investigations presented in this chapter are motivated by a search for automata theoretic approaches that avoid explicit construction of tree automata. In this direction, this chapter attempts to build efficient decision procedures for XML problems by using the alternation-free modal  $\mu$ -calculus. This logic is as expressive as WS2S, less succinct, but has a lower complexity (exponential time).

This chapter shows how XPath can be linearly translated into the  $\mu$ -calculus. In addition, regular tree types (including DTDs) are also linearly embedded in the  $\mu$ -calculus. XPath decision problems (containment, emptiness, equivalence, overlap, coverage) in the presence or absence of XML types are expressed as formulas in this logic. A state of the art decision procedure for  $\mu$ -calculus satisfiability is used to solve the generated formula and to construct relevant example and/or counter-example XML trees. The system has been fully implemented.

**Chapter Outline** The chapter is organized as follows: in Section 4.2 the  $\mu$ -calculus is introduced; Section 4.3 explains how general graph models of this logic can be restricted so that they represent XML trees. The translation of XPath queries into this logic is described in Section 4.4. Section 4.5 embeds regular XML types into the logic. Based on these translations, Section 4.6 explains how to formulate and solve the considered decision problems. A complexity analysis is presented in Section 4.7, along with implementation principles of the system. Finally, the outcome of this approach is discussed in Section 4.8.

## 4.2 The $\mu$ -Calculus

The *propositional  $\mu$ -calculus* is a propositional modal logic extended with least and greatest fixpoint operators [Kozen, 1983]. A *signature*  $\Xi$  for the  $\mu$ -calculus consists of a set *Prop* of atomic propositions, a set *Var* of propositional variables, and a set *FProg* of atomic programs. In the XML context, atomic propositions represent the symbols of the alphabet  $\Sigma$  used to label XML trees. Atomic programs allow navigation in trees.

The  $\mu$ -calculus with converse<sup>1</sup> [Vardi, 1998] augments the propositional  $\mu$ -calculus by associating with each atomic program  $a$  its converse  $\bar{a}$  (such that  $\bar{\bar{a}} = a$ ). A *program*  $\alpha$  is either an atomic program or its converse. *Prog* denotes the set  $FProg \cup \{\bar{a} \mid a \in FProg\}$ . This is the only difference with the propositional  $\mu$ -calculus that lacks converse programs. Equipping the logic with converse programs is useful for supporting query languages that allow both forward and backward navigation in trees (see Section 4.4.2). Converse programs generally provide a mean to reason about the past, which also proved to be useful in the context of program verification [Vardi, 1998]. The interaction of converse programs with other constructs of the logic is known to be quite subtle. In particular, in  $\mu$ -calculus it is known that converse programs interact with recursion in such a way that the finite model property is lost [Vardi, 1998]. The decidability of the  $\mu$ -calculus extended with converse was proved to be in EXPTIME in [Vardi, 1998], by introducing a new class of alternating two-way automata on infinite trees.

The set  $\mathcal{L}_\mu^{\text{full}}$  of formulas of the  $\mu$ -calculus with converse over the signature  $\Xi$  is defined as follows:

$\mathcal{L}_\mu^{\text{full}} \ni \varphi, \psi ::=$	$\top$	formula
		$p$ atomic proposition
		$\neg\varphi$ negation
		$\varphi \wedge \psi$ conjunction
		$[\alpha]\varphi$ universal modality
		$X$ variable
		$\mu X.\varphi$ least fixpoint

where  $p \in Prop$ ,  $X \in Var$  and  $\alpha$  is a program. Note that  $X$  should not occur negatively in  $\mu X.\varphi$ . The following abbreviations are defined:

$$\begin{aligned} \perp &\stackrel{\text{def}}{=} \neg\top \\ \varphi_1 \vee \varphi_2 &\stackrel{\text{def}}{=} \neg(\neg\varphi_1 \wedge \neg\varphi_2) \\ \langle\alpha\rangle\varphi &\stackrel{\text{def}}{=} \neg[\alpha]\neg\varphi \\ \nu X.\varphi &\stackrel{\text{def}}{=} \neg\mu X.\neg\varphi\{\neg X/X\} \end{aligned}$$

$\langle\alpha\rangle\varphi$  is called the existential modality and  $\nu X.\varphi$  the greatest fixpoint. The semantics of the full  $\mu$ -calculus is given with respect to a *Kripke structure*  $K = \langle W, R, L \rangle$  where  $W$  is a set of nodes,  $R : Prog \rightarrow 2^{W \times W}$  assigns to each atomic program a transition relation over  $W$ , and  $L$  is an interpretation function that assigns to each atomic proposition a set of nodes. The formal semantics function  $\llbracket \varphi \rrbracket_V^K$  shown on Figure 4.1 defines the semantics of a  $\mu$ -calculus formula  $\varphi$  in terms of a Kripke structure  $K$  and a valuation  $V$ . A valuation  $V : Var \rightarrow 2^W$  maps each variable to a subset of  $W$ . For a valuation  $V$ , a variable  $X$ , and a set of nodes  $W' \subseteq W$ ,  $V[X/W']$  denotes the valuation that is obtained from  $V$  by assigning  $W'$  to  $X$ .

Note that if  $\varphi$  is a sentence (i.e. all propositional variables occurring in  $\varphi$  are bound), then no valuation is required. For a node  $w \in W$  and a sentence  $\varphi$ ,  $K, w \models \varphi$  iff  $w \in \llbracket \varphi \rrbracket^K$  denotes that  $\varphi$  holds at  $w$  in  $K$ .

<sup>1</sup>The  $\mu$ -calculus with converse is also known as the *full*  $\mu$ -calculus, or alternatively as the *two-way*  $\mu$ -calculus in the literature.

$$\begin{aligned}
 \llbracket \cdot \rrbracket_V^K &: \mathcal{L}_\mu^{\text{full}} \longrightarrow 2^W \\
 \llbracket \top \rrbracket_V^K &\stackrel{\text{def}}{=} W \\
 \llbracket \perp \rrbracket_V^K &\stackrel{\text{def}}{=} \emptyset \\
 \llbracket p \rrbracket_V^K &\stackrel{\text{def}}{=} L(p) \\
 \llbracket \neg\varphi \rrbracket_V^K &\stackrel{\text{def}}{=} W \setminus \llbracket \varphi \rrbracket_V^K \\
 \llbracket \varphi_1 \vee \varphi_2 \rrbracket_V^K &\stackrel{\text{def}}{=} \llbracket \varphi_1 \rrbracket_V^K \cup \llbracket \varphi_2 \rrbracket_V^K \\
 \llbracket \varphi_1 \wedge \varphi_2 \rrbracket_V^K &\stackrel{\text{def}}{=} \llbracket \varphi_1 \rrbracket_V^K \cap \llbracket \varphi_2 \rrbracket_V^K \\
 \llbracket [\alpha]\varphi \rrbracket_V^K &\stackrel{\text{def}}{=} \{w : \forall w'(w, w') \in R(\alpha) \Rightarrow w' \in \llbracket \varphi \rrbracket_V^K\} \\
 \llbracket \langle \alpha \rangle \varphi \rrbracket_V^K &\stackrel{\text{def}}{=} \{w : \exists w'(w, w') \in R(\alpha) \wedge w' \in \llbracket \varphi \rrbracket_V^K\} \\
 \llbracket \mu X.\varphi \rrbracket_V^K &\stackrel{\text{def}}{=} \bigcap \{W' \subseteq W : \llbracket \varphi \rrbracket_{V[X/W']}^K \subseteq W'\} \\
 \llbracket \nu X.\varphi \rrbracket_V^K &\stackrel{\text{def}}{=} \bigcup \{W' \subseteq W : \llbracket \varphi \rrbracket_{V[X/W']}^K \supseteq W'\} \\
 \llbracket X \rrbracket_V^K &\stackrel{\text{def}}{=} V(X)
 \end{aligned}$$

 Figure 4.1: Semantics of the  $\mu$ -Calculus.

The two modalities  $\langle a \rangle \varphi$  (possibility) and  $[a] \varphi$  (necessity) are operators for navigating the structure.

In order to avoid redundancy, only a subset of  $\mathcal{L}_\mu^{\text{full}}$  composed of formulas in negation normal form is of interest. A formula is in *negation normal form* if and only if all negations in the formula appear only before atomic propositions. Every formula is equivalent to a formula in negation normal form [Kozen, 1983], which can be obtained by expanding negations using De Morgan's rules together with standard dualities for modalities and fixpoints (cf. Figure 4.2). For readability purposes, however, translations of XPath expressions given in Section 4.4 are not given in negation normal form.

$$\begin{aligned}
 \neg [\alpha] \varphi &= \langle \alpha \rangle \neg \varphi \\
 \neg \langle \alpha \rangle \varphi &= [\alpha] \neg \varphi \\
 \neg \mu X.\varphi &= \nu X.\neg \varphi\{X/\neg X\} \\
 \neg \nu X.\varphi &= \mu X.\neg \varphi\{X/\neg X\} \\
 \neg(\varphi_1 \wedge \varphi_2) &= \neg \varphi_1 \vee \neg \varphi_2 \\
 \neg(\varphi_1 \vee \varphi_2) &= \neg \varphi_1 \wedge \neg \varphi_2 \\
 \neg \neg \varphi &= \varphi
 \end{aligned}$$

Figure 4.2: Dualities for Negation Normal Form.

For reasoning on XML trees, only a specific subset of  $\mathcal{L}_\mu^{\text{full}}$ , namely the alternation-free modal  $\mu$ -calculus with converse over finite binary trees is of

interest.

A  $\mathcal{L}_\mu^{\text{full}}$  formula  $\varphi$  in negation normal form is *alternation-free* whenever the following condition holds<sup>2</sup>: if  $\mu X.\varphi_1$  (respectively  $\nu X.\varphi_1$ ) is a subformula of  $\varphi$  and  $\nu Y.\varphi_2$  (respectively  $\mu Y.\varphi_2$ ) is a subformula of  $\varphi_1$  then  $X$  does not occur freely in  $\varphi_2$ .

The following section now introduces the additional restrictions of  $\mathcal{L}_\mu^{\text{full}}$  related to finite binary trees.

### 4.3 Kripke Structures and XML Trees

In this section, the satisfiability problem of  $\mathcal{L}_\mu^{\text{full}}$  over Kripke structures is restricted to the satisfiability problem over finite binary trees.

The propositional  $\mu$ -calculus has the *finite tree model property*: a formula that is satisfiable, is also satisfiable on a finite tree [Kozen, 1988]. Unfortunately, the introduction of converse programs causes the loss of the finite model property [Vardi, 1998]. Therefore, the finite model property must be reinforced along with some other properties to ensure finite binary models that encode XML structures.

First, each XML node has at most one  $\Sigma$ -label, i.e.  $p \wedge p'$  never holds for distinct atomic propositions  $p$  and  $p'$ . This can be easily incorporated in a  $\mu$ -calculus satisfiability solver.

Second, for navigating binary trees, only two atomic programs 1 and 2 are used, together their associated relations  $R(1) = \prec_{\text{fc}}$  and  $R(2) = \prec_{\text{ns}}$  whose meaning is to respectively connect a node to its left child and to its right child. For any  $(x, y) \in W \times W$ ,  $x \prec_{\text{fc}} y$  holds iff  $y$  is the left child of  $x$  (i.e. the first child in the unranked tree representation) and  $x \prec_{\text{ns}} y$  holds iff  $y$  is the right child of  $x$  in the binary tree representation (i.e. the next sibling in the unranked tree representation).

For each atomic program  $a \in \{1, 2\}$ ,  $R(\bar{a})$  is defined to be the relational inverse of  $R(a)$ , i.e.,  $R(\bar{a}) = \{(v, u) : (u, v) \in R(a)\}$ . Thus programs  $\alpha \in \{1, 2, \bar{1}, \bar{2}\}$  are considered inside modalities for navigating downward and upward in trees.

Restrictions for a Kripke structure to form a finite binary tree are now defined. A Kripke structure  $T = \langle W, R, L \rangle$  is a finite binary tree if it satisfies the following conditions:

- (1)  $W$  is finite
- (2) the set of nodes  $W$  together with the accessibility relation  $\prec_{\text{fc}} \cup \prec_{\text{ns}}$  define a tree
- (3)  $\prec_{\text{fc}}$  and  $\prec_{\text{ns}}$  are partial functions, i.e. for all  $m \in W$  and  $j \in \{1, 2\}$  there is at most one  $m_j \in W$  such that  $(m, m_j) \in R(j)$ .

A finite binary tree  $T = \langle W, R, L \rangle$  satisfies  $\varphi$  if  $T, r \models \varphi$  where  $r \in W$  is the root of the tree  $T$ .

The previous restrictions are now expressed in  $\mathcal{L}_\mu^{\text{full}}$ . For accessing the root, the  $\mathcal{L}_\mu^{\text{full}}$  formula

$$\varphi_{\text{root}} = [\bar{1}] \perp \wedge [\bar{2}] \perp \wedge \neg \langle 2 \rangle \top$$

<sup>2</sup>For instance,  $\nu X.(\mu Y. \langle 1 \rangle Y \wedge p) \vee \langle 2 \rangle X$  is alternation-free but  $\nu X.(\mu Y. \langle 1 \rangle Y \wedge X) \vee p$  is not since  $X$  bound by  $\nu$  appears freely in the scope of  $\mu Y$ .

is used. Its meaning is to select a node provided it has no parent and no sibling.

The property for ensuring finiteness relies on König's lemma which states that *a finitely branching infinite tree has some infinite path* or, in other words, a finitely branching tree in which every branch is finite is finite. The expression  $\nu X. \langle 1 \rangle X \vee \langle 2 \rangle X$  is only satisfied by structures containing infinite or cyclic paths. To prevent the existence of such paths, the previous formula is negated and, propagating negation using the rules presented on Figure 4.2, yields the following formula:

$$\varphi_{\text{ft}} = \mu X. [1] X \wedge [2] X$$

$\varphi_{\text{ft}}$  states that all descending branches are finite from the current context node ( $\varphi_{\text{ft}}$  is vacuously satisfied at the leaves).  $\varphi_{\text{ft}}$  must hold at the root (i.e.  $\varphi_{\text{root}} \wedge \varphi_{\text{ft}}$  must hold), in order to ensure structure finiteness. This is for condition (1) to be satisfied.

Properties (2) and (3) still need to be enforced. This is done by rewriting existential modalities in such a way that if a successor is supposed to exist, then there exists at least one, and if there are many all verify the same property. This is a way to overcome the difficulty that in  $\mu$ -calculus, one cannot naturally express a property like “a node has exactly  $n$  successors”. Technically,  $\varphi^{\text{FBT}}$  denotes the formula  $\varphi$  where all occurrences of  $\langle \alpha \rangle \psi$  are replaced by  $\langle \alpha \rangle \top \wedge [\alpha] \psi^{\text{FBT}}$ . Furthermore, a node cannot be both a left child and a right child: the formula  $(\neg \langle 1 \rangle \top \vee \neg \langle 2 \rangle \top)$  must be satisfied at each node.

**Theorem 4.3.1** ([Tanabe et al., 2005]) *A  $\mathcal{L}_\mu^{\text{full}}$  formula  $\varphi$  is satisfied by a finite binary tree model if and only if the formula  $\varphi_{\text{root}} \wedge \mu X. (\neg \langle 1 \rangle \top \vee \neg \langle 2 \rangle \top) \wedge [1] X \wedge [2] X \wedge \varphi^{\text{FBT}}$  is satisfied by a Kripke structure.*

The proof of the “if” part iteratively constructs a tree model and proceeds by induction on the structure on  $\varphi$ . The “only if” part is almost immediate. Theorem 4.3.1 gives the adequate framework for formulating decision problems on XML structures in terms of a  $\mu$ -calculus formula.

## 4.4 XPath Embedding

This section explains how an XPath expression can be translated into an equivalent formula in  $\mathcal{L}_\mu^{\text{full}}$ . Navigation as performed by XPath in unranked trees is translated in terms of navigation in the binary tree representation (using the isomorphism presented in Section 2.1.1). The translation adheres to XPath formal semantics in the sense that the translated formula holds for nodes which are selected by the XPath query.

### 4.4.1 Logical Interpretation of Axes

The formal translations of navigational primitives (namely XPath axes) are formally specified on Figure 4.3. The translation function noted “ $A^\top \llbracket a \rrbracket_\chi$ ” takes an XPath axis  $a$  as input, and returns its  $\mathcal{L}_\mu^{\text{full}}$  translation, in terms of a  $\mathcal{L}_\mu^{\text{full}}$  formula  $\chi$  given as a parameter. This parameter represents a context and allows to compose formulas, which is needed for translating path composition.  $A^\top \llbracket a \rrbracket_\chi$  holds for all nodes that can be accessed through the axis  $a$  from some node verifying  $\chi$ .

$$\begin{aligned}
 A^\rightarrow[\cdot] &: Axis \rightarrow \mathcal{L}_\mu^{\text{full}} \rightarrow \mathcal{L}_\mu^{\text{full}} \\
 A^\rightarrow[\text{self}]_\chi &\stackrel{\text{def}}{=} \chi \\
 A^\rightarrow[\text{child}]_\chi &\stackrel{\text{def}}{=} \mu Z. \langle \bar{1} \rangle \chi \vee \langle \bar{2} \rangle Z \\
 A^\rightarrow[\text{following-sibling}]_\chi &\stackrel{\text{def}}{=} \mu Z. \langle \bar{2} \rangle \chi \vee \langle \bar{2} \rangle Z \\
 A^\rightarrow[\text{preceding-sibling}]_\chi &\stackrel{\text{def}}{=} \mu Z. \langle 2 \rangle \chi \vee \langle 2 \rangle Z \\
 A^\rightarrow[\text{parent}]_\chi &\stackrel{\text{def}}{=} \langle 1 \rangle \mu Z. \chi \vee \langle 2 \rangle Z \\
 A^\rightarrow[\text{descendant}]_\chi &\stackrel{\text{def}}{=} \mu Z. \langle \bar{1} \rangle (\chi \vee Z) \vee \langle \bar{2} \rangle Z \\
 A^\rightarrow[\text{descendant-or-self}]_\chi &\stackrel{\text{def}}{=} \mu Z. \chi \vee \mu Y. \langle \bar{1} \rangle (Y \vee Z) \vee \langle \bar{2} \rangle Y \\
 A^\rightarrow[\text{ancestor}]_\chi &\stackrel{\text{def}}{=} \langle 1 \rangle \mu Z. \chi \vee \langle 1 \rangle Z \vee \langle 2 \rangle Z \\
 A^\rightarrow[\text{ancestor-or-self}]_\chi &\stackrel{\text{def}}{=} \mu Z. \chi \vee \langle 1 \rangle \mu Y. Z \vee \langle 2 \rangle Y \\
 A^\rightarrow[\text{following}]_\chi &\stackrel{\text{def}}{=} A^\rightarrow[\text{descendant-or-self}]_{\eta_1(\chi)} \\
 A^\rightarrow[\text{preceding}]_\chi &\stackrel{\text{def}}{=} A^\rightarrow[\text{descendant-or-self}]_{\eta_2(\chi)} \\
 \eta_1(\chi) &\stackrel{\text{def}}{=} A^\rightarrow[\text{following-sibling}]_{A^\rightarrow[\text{ancestor-or-self}]_\chi} \\
 \eta_2(\chi) &\stackrel{\text{def}}{=} A^\rightarrow[\text{preceding-sibling}]_{A^\rightarrow[\text{ancestor-or-self}]_\chi}
 \end{aligned}$$

Figure 4.3: Translation of XPath Axes.

For instance, the translated formula  $A^\rightarrow[\text{child}]_\chi$  is satisfied by children of the context  $\chi$ . These nodes are composed of the first child and the remaining children. From the first child, the context must be reached immediately by going once upward via  $\bar{1}$ . From the remaining children, the context is reached by going upward (any number of times) via  $\bar{2}$  and then finally once via  $\bar{1}$ .

#### 4.4.2 Logical Interpretation of Expressions

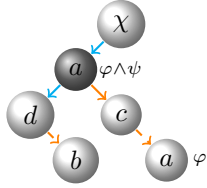
Figure 4.4 gives the translation of XPath expressions into  $\mathcal{L}_\mu^{\text{full}}$ . The translation function “ $E^\rightarrow[e]_\chi$ ” takes an XPath expression  $e$  and a  $\mathcal{L}_\mu^{\text{full}}$  formula  $\chi$  (denoting a particular context) as input, and returns the corresponding  $\mathcal{L}_\mu^{\text{full}}$  translation. The translation of relative XPath expressions use the current context  $\chi$ . The translation of absolute expressions navigates from  $\chi$  to the root which is taken as initial context for the expression.

For example, Figure 4.5 illustrates the translation of the XPath expression “child::a[child::b]”. This expression selects all “a” child nodes of a given context which have at least one “b” child. The translated  $\mathcal{L}_\mu^{\text{full}}$  formula holds for “a” nodes which are selected by the expression. The first part of the translated formula,  $\varphi$ , corresponds to the step “child::a” which selects candidates “a” nodes. The second part,  $\psi$ , navigates downward in the subtrees of these candidate nodes to verify that they have at least one “b” child.

Note that without converse programs it would have been impossible to differentiate selected nodes from nodes whose existence is tested, since properties

$$\begin{aligned}
 E^\rightarrow[\cdot] &: \mathcal{L}_{\text{XPath}} \rightarrow \mathcal{L}_\mu^{\text{full}} \rightarrow \mathcal{L}_\mu^{\text{full}} \\
 E^\rightarrow[\![p]\!]_\chi &\stackrel{\text{def}}{=} P^\rightarrow[\![p]\!](\mu Z. \neg \langle 1 \rangle \top \vee \langle 2 \rangle Z \wedge \mu Y. \chi \vee \langle 1 \rangle Y \vee \langle 2 \rangle Y) \\
 E^\rightarrow[\![p]\!]_\chi &\stackrel{\text{def}}{=} P^\rightarrow[\![p]\!]_{(\chi)} \\
 E^\rightarrow[\![e_1 \mid e_2]\!]_\chi &\stackrel{\text{def}}{=} E^\rightarrow[\![e_1]\!]_\chi \vee E^\rightarrow[\![e_2]\!]_\chi \\
 E^\rightarrow[\![e_1 \cap e_2]\!]_\chi &\stackrel{\text{def}}{=} E^\rightarrow[\![e_1]\!]_\chi \wedge E^\rightarrow[\![e_2]\!]_\chi \\
 \\ 
 P^\rightarrow[\cdot] &: \text{Path} \rightarrow \mathcal{L}_\mu^{\text{full}} \rightarrow \mathcal{L}_\mu^{\text{full}} \\
 P^\rightarrow[\![p_1/p_2]\!]_\chi &\stackrel{\text{def}}{=} P^\rightarrow[\![p_2]\!](P^\rightarrow[\![p_1]\!]_\chi) \\
 P^\rightarrow[\![p[q]\!]_\chi &\stackrel{\text{def}}{=} P^\rightarrow[\![p]\!]_\chi \wedge Q^\leftarrow[\![q]\!]_\top \\
 P^\rightarrow[\![a::\sigma]\!]_\chi &\stackrel{\text{def}}{=} \sigma \wedge A^\rightarrow[\![a]\!]_\chi \\
 P^\rightarrow[\![a::*]\!]_\chi &\stackrel{\text{def}}{=} A^\rightarrow[\![a]\!]_\chi
 \end{aligned}$$

Figure 4.4: Translation of Expressions and Paths.


 Translated Query:  $\text{child}::a [\text{child}::b]$ 

$$\underbrace{a \wedge (\mu Z. \langle 1 \rangle \chi \vee \langle 2 \rangle Z)}_\varphi \wedge \underbrace{\langle 1 \rangle \mu Y. b \vee \langle 2 \rangle Y}_\psi$$

Figure 4.5: XPath Translation Example.

must be stated on both the ancestors and the descendants of the selected node. Equipping the  $\mathcal{L}_\mu^{\text{full}}$  logic with both forward and converse programs is therefore crucial for supporting XPath<sup>3</sup>. Logics without converse programs may only be used for solving XPath emptiness but cannot be used for solving other decision problems such as containment efficiently.

XPath most essential construct  $p_1/p_2$  translates into formula composition in  $\mathcal{L}_\mu^{\text{full}}$ , such that the resulting formula holds for all nodes accessed through  $p_2$  from those nodes accessed from  $\chi$  by  $p_1$ . The translation of the branching construct  $p[q]$  significantly differs. The resulting formula must hold for all nodes that can be accessed through  $p$  and from which  $q$  holds. To preserve semantics, the translation of  $p[q]$  stops the “selecting navigation” to those nodes

<sup>3</sup>One may ask whether it is possible to eliminate upward navigation at the XPath level but it is well known that such XPath rewriting techniques cause exponential blow-ups of expression sizes [Olteanu et al., 2002].

$$\begin{aligned}
Q^\leftarrow[\cdot] &: Qualif \rightarrow \mathcal{L}_\mu^{\text{full}} \rightarrow \mathcal{L}_\mu^{\text{full}} \\
Q^\leftarrow[q_1 \text{ and } q_2]_\chi &\stackrel{\text{def}}{=} Q^\leftarrow[q_1]_\chi \wedge Q^\leftarrow[q_2]_\chi \\
Q^\leftarrow[q_1 \text{ or } q_2]_\chi &\stackrel{\text{def}}{=} Q^\leftarrow[q_1]_\chi \vee Q^\leftarrow[q_2]_\chi \\
Q^\leftarrow[\text{not } q]_\chi &\stackrel{\text{def}}{=} \neg Q^\leftarrow[q]_\chi \\
Q^\leftarrow[p]_\chi &\stackrel{\text{def}}{=} P^\leftarrow[p]_\chi
\end{aligned}$$

$$\begin{aligned}
P^\leftarrow[\cdot] &: Path \rightarrow \mathcal{L}_\mu^{\text{full}} \rightarrow \mathcal{L}_\mu^{\text{full}} \\
P^\leftarrow[p_1/p_2]_\chi &\stackrel{\text{def}}{=} P^\leftarrow[p_1]_{(P^\leftarrow[p_2]_\chi)} \\
P^\leftarrow[p[q]]_\chi &\stackrel{\text{def}}{=} P^\leftarrow[p]_{(\chi \wedge Q^\leftarrow[q]_\top)} \\
P^\leftarrow[a::\sigma]_\chi &\stackrel{\text{def}}{=} A^\leftarrow[a]_{(\chi \wedge \sigma)} \\
P^\leftarrow[a::*]_\chi &\stackrel{\text{def}}{=} A^\leftarrow[a]_\chi
\end{aligned}$$

$$\begin{aligned}
A^\leftarrow[\cdot] &: Axis \rightarrow \mathcal{L}_\mu^{\text{full}} \rightarrow \mathcal{L}_\mu^{\text{full}} \\
A^\leftarrow[a]_\chi &\stackrel{\text{def}}{=} A^\rightarrow[\text{symmetric}(a)]_\chi
\end{aligned}$$

Figure 4.6: Translation of Qualifiers.

reached by  $p$ , then filters them depending on whether  $q$  holds or not. This is expressed by introducing a dual formal translation function for XPath qualifiers, noted  $Q^\leftarrow[q]_\chi$  and defined in Figure 4.6, that performs “filtering” instead of navigation. Specifically,  $P^\leftarrow[\cdot]$  can be seen as the “navigational” translating function: the translated formula holds for target nodes of the given path. On the opposite,  $Q^\leftarrow[\cdot]$  can be seen as the “filtering” translating function: it states the existence of a path *without moving to its end*. The translated formula  $Q^\leftarrow[q]_\chi$  (respectively  $P^\leftarrow[p]_\chi$ ) holds for nodes from which there exists a qualifier  $q$  (respectively a path  $p$ ) leading to a node verifying  $\chi$ .

XPath translation is based on these two translating “modes”, the first one being used for paths and the second one for qualifiers. Whenever the “filtering” mode is entered, it will never be left.

Translations of paths inside qualifiers are also given on Figure 4.6. They use the specific translations for axes inside qualifiers, based on XPath symmetry:  $\text{symmetric}(a)$  denotes the symmetric XPath axis corresponding to the axis  $a$  (for instance  $\text{symmetric}(\text{child}) = \text{parent}$ ).

#### 4.4.3 Correctness and Complexity

The translation of XPath in  $\mathcal{L}_\mu^{\text{full}}$  can be proven correct with respect to XPath denotational semantics. First, a Wadler-like semantics of XPath expressions is defined with respect to Kripke structures that are XML trees. Let  $\mathcal{K}_T$  be the set of Kripke structures that are finite binary trees (as defined in Section 4.3) and  $\mathcal{W}(\mathcal{K}_T) = \{w \in W \mid \langle W, R, L \rangle \in \mathcal{K}_T\}$  the set of nodes of such structures.

Given a finite binary tree  $T = \langle W, R, L \rangle \in \mathcal{K}_T$  and some node  $x \in W$  of  $T$ , the functions  $\mathcal{S}_e[\cdot]_{(T,x)}$ ,  $\mathcal{S}_p[\cdot]_{(T,x)}$ ,  $\mathcal{S}_q[\cdot]_{(T,x)}$  and  $\mathcal{S}_a[\cdot]_{(T,x)}$  respectively define the semantics of XPath expressions, paths, qualifiers, and axes:

$$\begin{aligned}
& \mathcal{S}_e[\cdot]_{(\cdot,\cdot)} : \mathcal{L}_{\text{XPath}} \rightarrow \mathcal{K}_T \rightarrow \mathcal{W}(\mathcal{K}_T) \rightarrow 2^{\mathcal{W}(\mathcal{K}_T)} \\
& \mathcal{S}_e[/p]_{(T,x)} \stackrel{\text{def}}{=} \mathcal{S}_p[p]_{(T,\text{root}(T))} \\
& \mathcal{S}_e[p]_{(T,x)} \stackrel{\text{def}}{=} \mathcal{S}_p[p]_{(T,x)} \\
& \mathcal{S}_e[e_1 \mid e_2]_{(T,x)} \stackrel{\text{def}}{=} \mathcal{S}_e[e_1]_{(T,x)} \cup \mathcal{S}_e[e_2]_{(T,x)} \\
& \mathcal{S}_e[e_1 \cap e_2]_{(T,x)} \stackrel{\text{def}}{=} \mathcal{S}_e[e_1]_{(T,x)} \cap \mathcal{S}_e[e_2]_{(T,x)}
\end{aligned}$$

$$\begin{aligned}
& \mathcal{S}_p[\cdot]_{(\cdot,\cdot)} : \text{Path} \rightarrow \mathcal{K}_T \rightarrow \mathcal{W}(\mathcal{K}_T) \rightarrow 2^{\mathcal{W}(\mathcal{K}_T)} \\
& \mathcal{S}_p[p_1/p_2]_{(T,x)} \stackrel{\text{def}}{=} \{z \in \mathcal{S}_p[p_2]_{(T,y)} \mid y \in \mathcal{S}_p[p_1]_{(T,x)}\} \\
& \mathcal{S}_p[p[q]]_{(T,x)} \stackrel{\text{def}}{=} \{y \in \mathcal{S}_p[p]_{(T,x)} \mid \mathcal{S}_q[q]_{(T,y)}\} \\
& \mathcal{S}_p[a::\sigma]_{(\langle W,R,L \rangle,x)} \stackrel{\text{def}}{=} \{y \in \mathcal{S}_a[a]_{(\langle W,R,L \rangle,x)} \mid y \in L(\sigma)\} \\
& \mathcal{S}_p[a::*]_{(T,x)} \stackrel{\text{def}}{=} \{y \in \mathcal{S}_a[a]_{(T,x)}\}
\end{aligned}$$

$$\begin{aligned}
& \mathcal{S}_q[\cdot]_{(\cdot,\cdot)} : \text{Qualif} \rightarrow \mathcal{K}_T \rightarrow \mathcal{W}(\mathcal{K}_T) \rightarrow \{\text{true}, \text{false}\} \\
& \mathcal{S}_q[q_1 \text{ and } q_2]_{(T,x)} \stackrel{\text{def}}{=} \mathcal{S}_q[q_1]_{(T,x)} \vee \mathcal{S}_q[q_2]_{(T,x)} \\
& \mathcal{S}_q[q_1 \text{ or } q_2]_{(T,x)} \stackrel{\text{def}}{=} \mathcal{S}_q[q_1]_{(T,x)} \wedge \mathcal{S}_q[q_2]_{(T,x)} \\
& \mathcal{S}_q[\text{not } q]_{(T,x)} \stackrel{\text{def}}{=} \neg \mathcal{S}_q[q]_{(T,x)} \\
& \mathcal{S}_q[p]_{(T,x)} \stackrel{\text{def}}{=} \mathcal{S}_p[p]_{(T,x)} \neq \emptyset
\end{aligned}$$

$$\begin{aligned}
 \mathcal{S}_a[\![\cdot]\!]_{(\cdot, \cdot)} &: Axis \rightarrow \mathcal{K}_T \rightarrow \mathcal{W}(\mathcal{K}_T) \rightarrow 2^{\mathcal{W}(\mathcal{K}_T)} \\
 \mathcal{S}_a[\![\text{self}]\!]_{(T,x)} &\stackrel{\text{def}}{=} \{x\} \\
 \mathcal{S}_a[\![\text{child}]\!]_{((W,R,L),x)} &\stackrel{\text{def}}{=} \{y \in W \mid x \prec_{\text{fc}} y\} \cup \{z \in W \mid x \prec_{\text{fc}} y \wedge y \prec_{\text{ns}}^+ z\} \\
 \mathcal{S}_a[\![\text{following-sibling}]\!]_{((W,R,L),x)} &\stackrel{\text{def}}{=} \{z \in W \mid x \prec_{\text{ns}}^+ z\} \\
 \mathcal{S}_a[\![\text{preceding-sibling}]\!]_{((W,R,L),x)} &\stackrel{\text{def}}{=} \{z \in W \mid z \prec_{\text{ns}}^+ x\} \\
 \mathcal{S}_a[\![\text{parent}]\!]_{((W,R,L),x)} &\stackrel{\text{def}}{=} \{p \in W \mid p \prec_{\text{fc}} x\} \cup \{p \in W \mid p \prec_{\text{fc}} y \wedge y \prec_{\text{ns}}^+ x\} \\
 \mathcal{S}_a[\![\text{descendant}]\!]_{(T,x)} &\stackrel{\text{def}}{=} \mathcal{S}_a[\![\text{child}]\!]_{(T,x)} \\
 &\quad \cup \{z \in \mathcal{S}_a[\![\text{descendant}]\!]_{(T,y)} \mid y \in \mathcal{S}_a[\![\text{child}]\!]_{(T,x)}\} \\
 \mathcal{S}_a[\![\text{descendant-or-self}]\!]_{(T,x)} &\stackrel{\text{def}}{=} \mathcal{S}_a[\![\text{descendant}]\!]_{(T,x)} \cup \mathcal{S}_a[\![\text{self}]\!]_{(T,x)} \\
 \mathcal{S}_a[\![\text{ancestor}]\!]_{(T,x)} &\stackrel{\text{def}}{=} \mathcal{S}_a[\![\text{parent}]\!]_{(T,x)} \\
 &\quad \cup \{z \in \mathcal{S}_a[\![\text{ancestor}]\!]_{(T,y)} \mid y \in \mathcal{S}_a[\![\text{parent}]\!]_{(T,x)}\} \\
 \mathcal{S}_a[\![\text{ancestor-or-self}]\!]_{(T,x)} &\stackrel{\text{def}}{=} \mathcal{S}_a[\![\text{ancestor}]\!]_{(T,x)} \cup \mathcal{S}_a[\![\text{self}]\!]_{(T,x)}
 \end{aligned}$$

$$\begin{aligned}
 \mathcal{S}_a[\![\text{following}]\!]_{(T,x)} &\stackrel{\text{def}}{=} \{z \in \mathcal{S}_a[\![\text{descendant-or-self}]\!]_{(T,y)} \mid y \in f(T,x)\} \\
 \mathcal{S}_a[\![\text{preceding}]\!]_{(T,x)} &\stackrel{\text{def}}{=} \{z \in \mathcal{S}_a[\![\text{descendant-or-self}]\!]_{(T,y)} \mid y \in p(T,x)\} \\
 f(T,x) &\stackrel{\text{def}}{=} \{y \in \mathcal{S}_a[\![\text{following-sibling}]\!]_{(T,w)} \mid w \in a(T,x)\} \\
 p(T,x) &\stackrel{\text{def}}{=} \{y \in \mathcal{S}_a[\![\text{preceding-sibling}]\!]_{(T,w)} \mid w \in a(T,x)\} \\
 a(T,x) &\stackrel{\text{def}}{=} \{w \in \mathcal{S}_a[\![\text{ancestor-or-self}]\!]_{(T,x)}\}
 \end{aligned}$$

The auxiliary function  $\text{root}(T)$  returns the root of  $T$ , and the relation symbol  $\prec_{\text{ns}}^+$  used in the semantics of axes denotes the transitive closure of the relation  $\prec_{\text{ns}}$  defined in Section 4.3.

The correctness of the translation of XPath into  $\mathcal{L}_\mu^{\text{full}}$  can now be stated:

**Theorem 4.4.1 (Translation Correctness)** *For any finite binary tree  $T \in \mathcal{K}_T$ , nodes  $x$  and  $y$  of  $T$ , property  $\chi \in \mathcal{L}_\mu^{\text{full}}$ , expression  $e \in \mathcal{L}_{\text{XPath}}$ , and path  $p \in \text{Path}$ , the following equivalences hold:*

$$(\forall \chi \in \mathcal{L}_\mu^{\text{full}} \ T, x \models \chi \Rightarrow T, y \models E^\rightarrow[\![e]\!]_\chi) \iff y \in \mathcal{S}_e[\![e]\!]_{(T,x)} \quad (4.1)$$

$$T, y \models E^\rightarrow[\![e]\!]_\chi \iff y \in \bigcup_{\{x \mid T, x \models \chi\}} \mathcal{S}_e[\![e]\!]_{(T,x)} \quad (4.2)$$

$$(\forall \chi \in \mathcal{L}_\mu^{\text{full}} \ T, x \models \chi \Rightarrow T, y \models P^\rightarrow[\![p]\!]_\chi) \iff y \in \mathcal{S}_p[\![p]\!]_{(T,x)} \quad (4.3)$$

$$(\forall \chi \in \mathcal{L}_\mu^{\text{full}} \ T, y \models \chi \Rightarrow T, x \models P^\leftarrow[\![p]\!]_\chi) \iff y \in \mathcal{S}_p[\![p]\!]_{(T,x)} \quad (4.4)$$

*Proof outline:* Each equivalence is proved by a straightforward structural induction that “peels off” the compositional layers of each set of rules.  $\square$

This result links XPath decision problems to satisfiability in  $\mathcal{L}_\mu^{\text{full}}$ . Note that the size of a translated formula  $E \rightarrow \llbracket e \rrbracket_\chi$  is linear in the length of the XPath expression  $e$  since there is no duplication of subformulas of arbitrary length in the formal translations<sup>4</sup>.

## 4.5 Translation of Regular Tree Languages

The translation of regular tree types into  $\mu$ -calculus is now introduced. It is based on the binary representation of types introduced in Chapter 2. In order to simplify translations, a notation for a  $n$ -ary least fixpoint binder is introduced:

$$\text{let}_\mu (X_i.\varphi_i)_{1 \leq i \leq m} \text{ in } \psi$$

This notation is actually a syntactic sugar for  $\psi$  where all free occurrences of  $X_i$  have been replaced by  $\mu X_i.\varphi_i$  until  $\psi$  becomes closed (that is all  $X_i$  in  $\psi$  are in scope of their corresponding unary  $\mu$ -binder). This provides a shorthand for denoting a  $\mathcal{L}_\mu^{\text{full}}$  formula which would be of exponential size if expressed using only the unary least fixpoint construct. Such a naive expansion contains unnecessary duplicate formulas whereas the satisfiability solver operates only on a single copy of them (see Section 4.7). Therefore, the  $n$ -ary binder is a useful compact notation for representing  $\mathcal{L}_\mu^{\text{full}}$  translations of recursive types, without introducing useless blow-ups between representation of formulas and their satisfiability test.

The translation from binary regular tree types into  $\mathcal{L}_\mu^{\text{full}}$  formulas is given by the following function  $\llbracket \cdot \rrbracket$  :

$$\begin{aligned} \llbracket \cdot \rrbracket &: \mathcal{L}_{\text{bt}} \rightarrow \mathcal{L}_\mu^{\text{full}} \\ \llbracket \emptyset \rrbracket &\stackrel{\text{def}}{=} \perp \\ \llbracket \epsilon \rrbracket &\stackrel{\text{def}}{=} \perp \\ \llbracket T_1 \mid T_2 \rrbracket &\stackrel{\text{def}}{=} \llbracket T_1 \rrbracket \vee \llbracket T_2 \rrbracket \\ \llbracket l(X_1, X_2) \rrbracket &\stackrel{\text{def}}{=} \wedge \text{succ}_1(X_1) \wedge \text{succ}_2(X_2) \\ \llbracket \text{let } \overline{X_i.T_i} \text{ in } T \rrbracket &\stackrel{\text{def}}{=} \text{let}_\mu (X_i.\llbracket T_i \rrbracket)_{1 \leq i \leq m} \text{ in } \llbracket T \rrbracket \end{aligned}$$

where there is an implicit bijective correspondence between  $\mathcal{L}_{\text{bt}}$  variables from  $TVar$  and  $\mathcal{L}_\mu^{\text{full}}$  variables from  $Var$ . Note that the translations of the empty tree type and the empty tree are the same since empty trees should not be explicitly mentioned in satisfiability results. The function  $\text{succ}(\cdot)$  sets the tree frontier accordingly:

$$\begin{aligned} \text{succ}(\cdot) &: Prog \times TVar \rightarrow \mathcal{L}_\mu^{\text{full}} \\ \text{succ}_\alpha(X) &\stackrel{\text{def}}{=} \begin{cases} [\alpha] X & \text{if } \text{nullable}(X) \\ \langle \alpha \rangle X & \text{if } \text{not nullable}(X) \end{cases} \end{aligned}$$

<sup>4</sup>Formulas in which the formal parameter  $\chi$  appears twice (see Figure 4.4 and Figure 4.6) do not cause such duplication since at this stage  $\chi$  carries a constant. Section 4.6 explains how  $\chi$  is initialized with a constant at the expression level.

The predicate  $nullable(\cdot)$  indicates if a type contains the empty tree:

$$\begin{aligned}
nullable(\cdot) &: TVar \cup \mathcal{L}_{bt} \rightarrow \{\text{true}, \text{false}\} \\
nullable(X) &\stackrel{\text{def}}{=} nullable(\theta(X)) \\
nullable(\emptyset) &\stackrel{\text{def}}{=} \text{false} \\
nullable(\epsilon) &\stackrel{\text{def}}{=} \text{true} \\
nullable(l) &\stackrel{\text{def}}{=} \text{false} \\
nullable(T_1 \mid T_2) &\stackrel{\text{def}}{=} nullable(T_1) \vee nullable(T_2) \\
nullable(l(X_1, X_2)) &\stackrel{\text{def}}{=} \text{false} \\
nullable(\text{let } \overline{X_i.T_i} \text{ in } T) &\stackrel{\text{def}}{=} nullable(T)
\end{aligned}$$

## 4.6 Solving XML Decision Problems

Both XPath over unranked trees, and regular unranked tree types have been translated in the unifying  $\mathcal{L}_\mu^{\text{full}}$  logic over binary trees. Owing to these translations, XML decision problems (such as XPath containment, equivalence, emptiness, overlap and coverage) in the presence or absence of XML types are now reduced to satisfiability in  $\mathcal{L}_\mu^{\text{full}}$ .

**Correlating Context Nodes for Path Comparison** In order to correlate two different paths when performing any kind of mutual-relationship checking, a special atomic proposition  $\textcircled{S}$  is introduced. This atomic proposition marks the initial context node(s) from which an XPath expression is applied.  $\textcircled{S}$  is used as initial value of the  $\chi$  parameter of the translating function  $E^{-}[\cdot]_\chi$ . For an XPath expression  $e \in \mathcal{L}_{\text{XPath}}$ ,  $E^{-}[e]_{\textcircled{S}}$  is thus a sentence, that is denoted by  $\varphi_e$  in the remaining. Owing to the introduction of  $\textcircled{S}$ , formulas may refer to the same context multiple times. This allows to compare different XPath expressions applied to the *same* initial context that can be any node in any tree.

**Formulating of XML Problems** Some simplified notations are first introduced:  $\mathcal{T}$  denotes the set of trees: by default,  $\mathcal{T} = \mathcal{T}_\Sigma^n$ , and whenever an optional DTD  $d \in \mathcal{L}_{\text{DTD}}$  is specified  $\mathcal{T} = \llbracket d \rrbracket_\emptyset$ . Additionally,  $\varphi_{\mathcal{T}}$  denotes the  $\mathcal{L}_\mu^{\text{full}}$  embedding of the tree language  $\mathcal{T}$ . In the absence of DTDs  $\varphi_{\mathcal{T}} = \top$ , and  $\varphi_{\mathcal{T}} = \llbracket \mathcal{B}(d) \rrbracket$  in the presence of  $d \in \mathcal{L}_{\text{DTD}}$ .

Several decision problems needed in applications can be expressed in terms of  $\mathcal{L}_\mu^{\text{full}}$  formulas:

- XPath containment
  - Input:  $e_1, e_2 \in \mathcal{L}_{\text{XPath}}$  and optional  $d \in \mathcal{L}_{\text{DTD}}$
  - Problem: Does  $e_2$  always select all nodes selected by  $e_1$ ?
  - Definition:  $\forall t \in \mathcal{T}, \forall x \in t, \mathcal{S}_e[e_1]x \subseteq \mathcal{S}_e[e_2]x$
  - Tested  $\mathcal{L}_\mu^{\text{full}}$  formula:  $\varphi_{e_1} \wedge \neg\varphi_{e_2}$

- XPath equivalence
  - Input:  $e_1, e_2 \in \mathcal{L}_{\text{XPath}}$  and optional  $d \in \mathcal{L}_{\text{dtd}}$
  - Problem: Does  $e_2$  always select exactly the same nodes as  $e_1$ ?
  - Definition:  $\forall t \in \mathcal{T}, \forall x \in t, \mathcal{S}_e[[e_1]]x = \mathcal{S}_e[[e_2]]x$
  - Equivalence can be tested by two successive and separate containment checks
- XPath emptiness
  - Input:  $e \in \mathcal{L}_{\text{XPath}}$  and optional  $d \in \mathcal{L}_{\text{dtd}}$
  - Problem: Will  $e$  ever return a non-empty set of nodes?
  - Definition:  $\forall t \in \mathcal{T}, \forall x \in t, \mathcal{S}_e[[e]]x \neq \emptyset$
  - Tested  $\mathcal{L}_\mu^{\text{full}}$  formula:  $\varphi_e$
- XPath overlap
  - Input:  $e_1, e_2 \in \mathcal{L}_{\text{XPath}}$  and optional  $d \in \mathcal{L}_{\text{dtd}}$
  - Problem: May  $e_1$  and  $e_2$  select common nodes?
  - Definition:  $\forall t \in \mathcal{T}, \forall x \in t, \mathcal{S}_e[[e_1]]x \cap \mathcal{S}_e[[e_2]]x \neq \emptyset$
  - Tested  $\mathcal{L}_\mu^{\text{full}}$  formula:  $\varphi_{e_1} \wedge \varphi_{e_2}$
- XPath coverage
  - Input:  $e_1, e_2, \dots, e_n \in \mathcal{L}_{\text{XPath}}$  and optional  $d \in \mathcal{L}_{\text{dtd}}$
  - Problem: Are nodes selected by  $e_1$  always selected by one of the  $e_2, \dots, e_n$ ?
  - Definition:  $\forall t \in \mathcal{T}, \forall x \in t, \mathcal{S}_e[[e_1]]x \subseteq \bigcup_{2 \leq i \leq n} \mathcal{S}_e[[e_i]]x$
  - Tested  $\mathcal{L}_\mu^{\text{full}}$  formula:  $\varphi_{e_1} \wedge \bigwedge_{2 \leq i \leq n} \neg \varphi_{e_i}$

Note that for the containment problem, the unsatisfiability of  $\varphi_{e_1} \wedge \neg \varphi_{e_2}$  is tested. Indeed, checking that an XPath expression  $e_1$  is contained into another expression  $e_2$  consists in checking that the implication  $\varphi_{e_1} \Rightarrow \varphi_{e_2}$  holds for all trees. In other terms, there exists no tree for which the results of  $e_1$  are not included in those of  $e_2$ , i.e. the negated implication  $\varphi_{e_1} \wedge \neg \varphi_{e_2}$  is unsatisfiable.

Since the finite binary tree model property must be enforced (as seen in Section 4.3.1), decision problems are formulated from the root, and the actually checked formula becomes:

$$\varphi_{\text{root}} \wedge \varphi_{\text{ft}} \wedge (\varphi_{\mathcal{T}} \wedge \mu X. \varphi_{\text{tested}} \vee \langle 1 \rangle X \vee \langle 2 \rangle X)^{\text{FBT}} \quad (4.5)$$

where  $\varphi_{\text{tested}}$  corresponds to a particular XPath decision problem from those given above. Intuitively, the fixpoint is introduced for “plunging” XPath navigation performed by  $\varphi_{\text{tested}}$  at any location in the tree. It is for example necessary for relative XPath expressions that involve upward navigation in the tree.

It is important to note that formula (4.5) is always alternation-free since both embeddings of XPath and tree types produce alternation-free formulas, and the negation of an alternation free sentence remains alternation-free. In practice, negated sentences introduced by XPath embeddings are turned into negation normal form, by applying the rules given on Figure 4.2.

## 4.7 Complexity Analysis and Implementation Principles

The proposed approach has been implemented. A compiler takes XPath expressions as input, and translates them into  $\mathcal{L}_\mu^{\text{full}}$  formulas. Another compiler takes regular tree types as input (DTDs) and outputs their  $\mathcal{L}_\mu^{\text{full}}$  translation. The formula of a particular decision problem is then composed, normalized and solved.

The  $\mu$ -calculus satisfiability solver is specialized for the alternation-free  $\mu$ -calculus with converse. It is closely inspired from the tableau methods described in [Tanabe et al., 2005] and [Pan et al., 2006]. A detailed description of the AFMC solver is beyond the scope of this chapter (see [Tanabe et al., 2005] for more details on an AFMC solver; and Chapter 6 for a detailed description of a logical solver specialized for XML). The focus here is rather given to the AFMC solver aspects which allow to establish precise complexity results for the considered XML decision problems with the  $\mu$ -calculus approach. The algorithm relies on a top-down tableau method which attempts to construct satisfying Kripke structures by a fixpoint computation. Nodes of the tableau are specific subsets of a set called the Lean [Pan et al., 2006]. Given a formula  $\psi \in \mathcal{L}_\mu^{\text{full}}$ , the Lean is the subset of the Fischer-Ladner closure [Fischer and Ladner, 1979] of  $\psi$  composed of atomic and modal subformulas of  $\psi$  [Pan et al., 2006]. The algorithm starts from the set of all possible nodes, and repeatedly removes inconsistent nodes until a fixpoint is reached. At the end of the computation, if  $\psi$  is present in a node of the fixpoint, then  $\psi$  is satisfiable. In this case, the fixpoint contains a satisfying model that can be easily extracted and used as a satisfying example XML tree.

The complexity of the addressed XML decision problems can now be stated:

**Proposition 4.7.1** *XPath containment, equivalence, emptiness, overlap and coverage decision problems, in the presence or absence of regular tree constraints, can be solved in time complexity  $2^{O(n \cdot \log n)}$ , where  $n$  is the Lean size of the corresponding  $\mathcal{L}_\mu^{\text{full}}$  formula.*

This upper-bound is derived from:

1. the linear translations of XPath and regular tree types into the  $\mu$ -calculus;
2. the  $2^{O(n \cdot \log n)}$  time complexity of the solver, which corresponds to the best known complexity for deciding alternation-free  $\mu$ -calculus with converse over Kripke structures [Tanabe et al., 2005]. Note that this complexity is smaller than the best known complexity for the whole  $\mu$ -calculus with converse [Vardi, 1998] which is  $2^{O(n^4 \cdot \log n)}$  [Grädel et al., 2002].

The key observation for the linear translation of regular tree types is that only distinct atomic and modal subformulas of the translated formula are present in the Lean, even for a  $n$ -ary binder  $\varphi = \text{let}_\mu (X_i \cdot \varphi_i)_{1 \leq i \leq m}$  in  $X_k$ . More precisely, the Lean corresponding to the translation of  $\varphi$  contains at most:

- the two eventualities  $\langle a \rangle \top$  for  $a = 1, 2$
- $2 \cdot m$  universalities  $[a] \varphi$  where  $m$  is the number of binary tree type variables in the binder and the constant factor corresponds to the downward programs  $a = 1, 2$

- the atomic propositions representing the alphabet symbols used in  $\varphi$

Deriving complexity from properties of the closure of a formula was first used by Fischer and Ladner for establishing decidability of PDL in single exponential time [Fischer and Ladner, 1979]. Analog observations have also been made for the modal logic K [Pan et al., 2006], and the  $\mu$ -calculus over general Kripke structures [Tanabe et al., 2005]. These results can be seen as an application of this technique to the case where regular tree types are combined with XPath bidirectional queries over finite trees.

Keys for the efficiency of the method on large practical instances are as follows:

1. Nodes of the tableau contain only modal formulas and exactly one atomic proposition (for XML), which greatly reduces the number of enumerated nodes for large alphabets.
2. Negation in the  $\mu$ -calculus is rather straightforward compared to automata techniques. Indeed, handling  $\mathcal{L}_\mu^{\text{full}}$  formulas in negation normal form simply reduces to checking membership of atomic propositions in tableau nodes. This contrasts with tree automata techniques which require for every negation the full construction and complementation of automata with an exponential blow-up. As pointed out in [Baader and Tobies, 2001] and [Pan et al., 2006], tableau methods for logics with the tree model property can be viewed as implementations of the automata-theoretic approach which avoids an explicit automata construction.
3. The implementation relies on representing sets of nodes and operating on them symbolically using Binary Decision Diagrams (BDDs) [Bryant, 1986]. BDDs provide a canonical representation of boolean functions. Their effectiveness is well known in the domain of formal verification of systems [Edmund M. Clarke et al., 1999]. BDD variables encode truth status of Lean formulas. The cost of BDD operations is very sensitive to variable ordering. Finding the optimal variable ordering is known to be NP-complete [Hojati et al., 1996]. However, several heuristics are known to perform well in practice [Edmund M. Clarke et al., 1999]. Choosing a good initial variable order does significantly improve performance. Preserving locality of the initial problem happens to be essential. It can be easily observed that the variable order determined by the breadth-first traversal of the initial formula (thus keeping sister subformulas in close proximity while ordering Lean formulas) yields better results in practice.

There are still areas for improvements though. In particular, a large amount of time is spent in the  $\mu$ -loop detection performed by the solver for avoiding cycles and infinite paths in the case of finite recursion [Tanabe et al., 2005]. From this perspective, transforming the  $\mu$ -calculus formula at the syntactic level (as presented in Section 4.3) and then relying on loop detection to enforce the finite model property is overkill. The approach may be improved by considering XML finite tree structures as models of the logic, and building an appropriate satisfiability solver for such structures.

## 4.8 Outcome

An approach for solving XPath decision problems by reduction to satisfiability of alternation-free modal  $\mu$ -calculus with converse over general Kripke structures has been proposed. XPath queries and regular tree types are linearly translated into the AFMC. XML decision problems are expressed as formulas in this logic, then decided using a solver for AFMC satisfiability. With respect to MSO, this yields much more efficient (exponential time) decision procedures for XML decision problems. Nevertheless, this approach may still be greatly improved, since models of the logic are too general for the XML setting, and one has to pay extra costs for restricting them appropriately. One direction of future work consists in designing a more appropriate calculus where models are finite trees instead of general Kripke structures. This is what is achieved in the remaining of this dissertation.

# A Fixpoint Modal Logic with Converse for XML



# A Fixpoint Modal Logic with Converse for XML

---

## 5.1 Introduction

This chapter and the following introduce the final results of this thesis, based on the lessons learned from the investigations reported in previous chapters.

The decidability of a new logic with converse for finite and ordered trees is proved. The logic is sufficiently expressive to support XPath bidirectional navigation in finite trees along with regular tree languages. The logic is derived from the  $\mu$ -calculus and inherits some of its desirable properties, while improving the best known complexity for finite trees. These discoveries are naturally applied to the static analysis of XML specifications, for which they yield sound, complete and efficient decision procedures. The proof method is based on two auxiliary results. First, XML regular tree types and XPath expressions have a linear translation to *cycle-free* formulas. Second, the least and greatest fixpoints are equivalent for finite trees, hence the logic is closed under negation.

**Chapter Outline** This chapter presents focused trees in Section 5.2 as a convenient data model for XML. The logic is then introduced in Section 5.3, and translations of XML concepts into the logic are presented in Section 5.4.

## 5.2 Focused Trees

In this chapter, a less conventional approach is used to represent XML trees, called *focused trees*. Focused trees are directly inspired by Huet’s Zipper data structure [Huet, 1997], and are closely related to pointed trees introduced in [Podelski, 1992, Nivat and Podelski, 1993], which were extended to pointed hedges and applied to the XML setting in [Murata, 2001]. Focused trees not only describe a tree but also its context: its previous siblings and its parent, recursively. Exploring such a structure has the advantage to preserve all information, which is quite useful when considering languages such as XPath that allow forward and backward axes of navigation.

Formally, an alphabet  $\Sigma$  of labels, ranged over by  $\sigma$  is assumed.

$t$	$::=$	$\sigma[tl]$	tree
$tl$	$::=$	$\epsilon$	list of trees
		$\epsilon$	empty list
		$  \quad t :: tl$	cons cell
$c$	$::=$		context
		$(tl, Top, tl)$	root of the tree
		$  \quad (tl, c[\sigma], tl)$	context node
$f$	$::=$	$(t, c)$	focused tree

In order to deal with XPath containment, it is needed to represent in a focused tree the place where the evaluation was started using a *context mark*. To do so, we consider focused trees where a single tree or a single context node is marked, as in  $\sigma^\circ[tl]$  or  $(tl, c[\sigma^\circ], tl)$ . When the presence of the mark is unknown, it is written as  $\sigma^\circ[tl]$ .

$\mathcal{F}$  denotes the set of finite focused trees with a single mark. The *name* of a focused tree is defined as  $\text{nm}(\sigma^\circ[tl], c) = \sigma$ . Navigation in focused trees is now described, in binary style. Four directions can be followed: for a focused tree  $f$ ,  $f \langle 1 \rangle$  changes the focus to the children of the current tree,  $f \langle 2 \rangle$  changes the focus to the next sibling of the current tree,  $f \langle \bar{1} \rangle$  changes the focus to the parent of the tree *if the current tree is a leftmost sibling*, and  $f \langle \bar{2} \rangle$  changes the focus to the previous sibling.

Formally:

$$\begin{aligned}
 (\sigma^\circ[t :: tl], c) \langle 1 \rangle &\stackrel{\text{def}}{=} (t, (\epsilon, c[\sigma^\circ], tl)) \\
 (t, (tl_l, c[\sigma^\circ], t' :: tl_r)) \langle 2 \rangle &\stackrel{\text{def}}{=} (t', (t :: tl_l, c[\sigma^\circ], tl_r)) \\
 (t, (\epsilon, c[\sigma^\circ], tl)) \langle \bar{1} \rangle &\stackrel{\text{def}}{=} (\sigma^\circ[t :: tl], c) \\
 (t', (t :: tl_l, c[\sigma^\circ], tl_r)) \langle \bar{2} \rangle &\stackrel{\text{def}}{=} (t, (tl_l, c[\sigma^\circ], t' :: tl_r))
 \end{aligned}$$

When the focused tree does not have the required shape, these operations are not defined.

### 5.3 Formulas of the Logic

The logic to which XPath expressions and XML regular tree types are going to be translated is introduced. It is a sub-logic of the alternation free modal  $\mu$ -calculus with converse. Next, a restriction on the considered formulas is introduced, and an interpretation of formulas as sets of finite focused trees is given. Then, it is shown that the logic has a single fixpoint for these models and that it is closed under negation.

In the following definitions,  $a \in \{1, 2, \bar{1}, \bar{2}\}$  are *programs* and atomic propositions  $\sigma$  correspond to labels from  $\Sigma$ . It is also assumed that  $\bar{\bar{a}} = a$ .

Formulas, defined in Fig. 5.1 include the truth predicate, atomic propositions (denoting the name of the tree in focus), start propositions (denoting the presence of the start mark), disjunction and conjunction of formulas, formulas under an existential (denoting the existence a subtree satisfying the subformula), and least and greatest nary fixpoints. We chose to include a nary version of the latter because regular types are often defined as a set of mutually

$\mathcal{L}_\mu \ni \varphi, \psi ::=$	formula
$\top$	true
$\sigma \mid \neg\sigma$	atomic prop (negated)
$\textcircled{\sigma} \mid \neg\textcircled{\sigma}$	context (negated)
$X$	variable
$\varphi \vee \psi$	disjunction
$\varphi \wedge \psi$	conjunction
$\langle a \rangle \varphi \mid \neg \langle a \rangle \top$	existential (negated)
$\mu \overline{X}_i. \varphi_i \text{ in } \psi$	least $n$ -ary fixpoint
$\nu \overline{X}_i. \varphi_i \text{ in } \psi$	greatest $n$ -ary fixpoint

Figure 5.1: Logic formulas

$$\begin{array}{ll}
 \llbracket \top \rrbracket_V \stackrel{\text{def}}{=} \mathcal{F} & \llbracket \sigma \rrbracket_V \stackrel{\text{def}}{=} \{f \mid \mathbf{nm}(f) = \sigma\} \\
 \llbracket X \rrbracket_V \stackrel{\text{def}}{=} V(X) & \llbracket \neg\sigma \rrbracket_V \stackrel{\text{def}}{=} \{f \mid \mathbf{nm}(f) \neq \sigma\} \\
 \llbracket \varphi \vee \psi \rrbracket_V \stackrel{\text{def}}{=} \llbracket \varphi \rrbracket_V \cup \llbracket \psi \rrbracket_V & \llbracket \textcircled{\sigma} \rrbracket_V \stackrel{\text{def}}{=} \{f \mid f = (\sigma^{\textcircled{\sigma}}[tl], c)\} \\
 \llbracket \varphi \wedge \psi \rrbracket_V \stackrel{\text{def}}{=} \llbracket \varphi \rrbracket_V \cap \llbracket \psi \rrbracket_V & \llbracket \neg\textcircled{\sigma} \rrbracket_V \stackrel{\text{def}}{=} \{f \mid f = (\sigma[tl], c)\}
 \end{array}$$

$$\begin{array}{l}
 \llbracket \langle a \rangle \varphi \rrbracket_V \stackrel{\text{def}}{=} \{f \langle \bar{a} \rangle \mid f \in \llbracket \varphi \rrbracket_V \wedge f \langle \bar{a} \rangle \text{ defined}\} \\
 \llbracket \neg \langle a \rangle \top \rrbracket_V \stackrel{\text{def}}{=} \{f \mid f \langle a \rangle \text{ undefined}\} \\
 \llbracket \mu \overline{X}_i. \varphi_i \text{ in } \psi \rrbracket_V \stackrel{\text{def}}{=} \text{let } T_i = \left( \bigcap \left\{ \overline{T}_i \subseteq \overline{\mathcal{F}} \mid \llbracket \overline{\varphi}_i \rrbracket_{V[\overline{T}_i/\overline{X}_i]} \subseteq \overline{T}_i \right\} \right)_i \\
 \quad \text{in } \llbracket \psi \rrbracket_{V[\overline{T}_i/\overline{X}_i]} \\
 \llbracket \nu \overline{X}_i. \varphi_i \text{ in } \psi \rrbracket_V \stackrel{\text{def}}{=} \text{let } T_i = \left( \bigcup \left\{ \overline{T}_i \subseteq \overline{\mathcal{F}} \mid \overline{T}_i \subseteq \llbracket \overline{\varphi}_i \rrbracket_{V[\overline{T}_i/\overline{X}_i]} \right\} \right)_i \\
 \quad \text{in } \llbracket \psi \rrbracket_{V[\overline{T}_i/\overline{X}_i]}
 \end{array}$$

Figure 5.2: Interpretation of formulas

recursive definitions, making their translation in our logic more succinct. In the following we write “ $\mu X. \varphi$ ” for “ $\mu X. \varphi \text{ in } \varphi$ ”.

An interpretation of formulas as sets of finite focused trees with a single start mark is now given on Figure 5.2. The interpretation of the nary fixpoints first compute the smallest or largest interpretation for each  $\varphi_i$  then returns the interpretation of  $\psi$  using these bindings.

The set of valid formulas is now restricted to *cycle-free formulas*, i.e. formulas that have a bound on the number of *modality cycles* independently of the number of unfolding of their fixpoints. A modality cycle is a subformula of the form  $\langle a \rangle \varphi$  where  $\varphi$  contains a *top-level* existential of the form  $\langle \bar{a} \rangle \psi$ . “Top-level” means under an arbitrary number of conjunctions or disjunctions, but not under any other construct. For instance, the formula “ $\mu X. \langle 1 \rangle (\varphi \vee \langle \bar{1} \rangle X) \text{ in } X$ ” is not cycle free: for any integer  $n$ , there is an unfolding of the formula with  $n$  modality cycles. On the other hand, the formula “ $\mu X. \langle 1 \rangle (X \vee Y), Y. \langle \bar{1} \rangle (Y \vee$

$$\begin{array}{c}
 \frac{\varphi = \top, \sigma, \neg\sigma, \textcircled{\sigma}, \text{ or } \neg\textcircled{\sigma}}{\Delta \parallel \Gamma \vdash_I^R \varphi} \qquad \frac{\Delta \parallel \Gamma \vdash_I^R \varphi \quad \Delta \parallel \Gamma \vdash_I^R \psi}{\Delta \parallel \Gamma \vdash_I^R \varphi \vee \psi} \\
 \\
 \frac{\Delta \parallel \Gamma \vdash_I^R \varphi \quad \Delta \parallel \Gamma \vdash_I^R \psi}{\Delta \parallel \Gamma \vdash_I^R \varphi \wedge \psi} \qquad \frac{}{\Delta \parallel \Gamma \vdash_I^R \neg \langle a \rangle \top} \qquad \frac{\Delta \parallel (\Gamma \triangleleft \langle a \rangle) \vdash_I^R \varphi}{\Delta \parallel \Gamma \vdash_I^R \langle a \rangle \varphi} \\
 \\
 \frac{\forall X_j \in \overline{X}_i. \left( (\Delta + \overline{X}_i : \varphi_i) \parallel (\Gamma + \overline{X}_i : \cdot) \vdash_{I \setminus \overline{X}_i}^{R \setminus \overline{X}_i} \varphi_j \right)}{\Delta \parallel \Gamma \vdash_I^R \mu \overline{X}_i. \varphi_i \text{ in } \psi} \qquad \frac{\Delta \parallel \Gamma \vdash_{I \cup \overline{X}_i}^{R \setminus \overline{X}_i} \psi}{\Delta \parallel \Gamma \vdash_I^R \mu \overline{X}_i. \varphi_i \text{ in } \psi} \\
 \\
 \frac{\forall X_j \in \overline{X}_i. \left( (\Delta + \overline{X}_i : \varphi_i) \parallel (\Gamma + \overline{X}_i : \cdot) \vdash_{I \setminus \overline{X}_i}^{R \setminus \overline{X}_i} \varphi_j \right)}{\Delta \parallel \Gamma \vdash_I^R \nu \overline{X}_i. \varphi_i \text{ in } \psi} \qquad \frac{\Delta \parallel \Gamma \vdash_{I \cup \overline{X}_i}^{R \setminus \overline{X}_i} \psi}{\Delta \parallel \Gamma \vdash_I^R \nu \overline{X}_i. \varphi_i \text{ in } \psi} \\
 \\
 \frac{\text{NOREC} \quad X \in R \quad \Gamma(X) = \langle a \rangle}{\Delta \parallel \Gamma \vdash_I^R X} \qquad \frac{\text{REC} \quad X \notin R \quad \Delta \parallel \Gamma \vdash_I^{R \cup \{X\}} \Delta(X)}{\Delta \parallel \Gamma \vdash_I^R X} \qquad \frac{\text{IGN} \quad X \in I}{\Delta \parallel \Gamma \vdash_I^R X}
 \end{array}$$

Figure 5.3: Cycle-free formulas

$\top$ ) in  $X$ ” is cycle free: there is at most one modality cycle.

Cycle-free formulas have a very interesting property, which can now be described. To test whether a tree satisfies a formula, one may define a straightforward inductive relation between trees and formulas that only holds when the root of the tree satisfies the formula, unfolding fixpoints if necessary. Given a tree, if a formula  $\varphi$  is cycle free, then every node of the tree will be tested a finite number of time against any given subformula of  $\varphi$ . The intuition behind this property, which holds a central role in the proof of lemma 5.3.2, is the following. If a tree node is tested an infinite number of times against a subformula, then there must be a cycle in the navigation in the tree, corresponding to some modalities occurring in the subformula, between one occurrence of the test and the next one. As trees are considered, the cycle implies there is a modality cycle in the formula (as cycles of the form  $\langle 1 \rangle \langle 2 \rangle \langle \overline{1} \rangle \langle \overline{2} \rangle$  cannot occur). Hence the number of modality cycles in any expansion of  $\varphi$  is unbounded, thus the formula is not cycle free.

Figure 5.3 gives an inductive relation that decides whether a formula is cycle free.

In the judgement  $\Delta \parallel \Gamma \vdash_I^R \varphi$  of Fig. 5.3,  $\Delta$  is an environment binding some recursion variables to their formulas,  $\Gamma$  binds variables to modalities,  $R$  is a set of variables that have already been expanded (see below), and  $I$  is a set of variables already checked.

The environment  $\Gamma$  used to derive the judgement consists of bindings from variables (from enclosing fixpoint operators) to modalities. A modality may be  $\cdot$ , no information is known about the variable,  $\langle a \rangle$ , the last modality taken  $\langle a \rangle$  was consistent, or  $\perp$ , a cycle has been detected. A formula is not cycle free if an occurrence of a variable under a fixpoint operator is either not under a modality (in this case  $\Gamma(X) = \cdot$ ), or is under a cycle ( $\Gamma(X) = \perp$ ). Cycle

detection uses an auxiliary operator to detect modality cycles:

$$\Gamma \triangleleft \langle a \rangle \stackrel{\text{def}}{=} \{X : (\Gamma(X) \triangleleft \langle a \rangle)\}$$

where

$\cdot \triangleleft \cdot$	$\langle 1 \rangle$	$\langle 2 \rangle$	$\langle \bar{1} \rangle$	$\langle \bar{2} \rangle$
$-$	$\langle 1 \rangle$	$\langle 2 \rangle$	$\langle \bar{1} \rangle$	$\langle \bar{2} \rangle$
$\langle 1 \rangle$	$\langle 1 \rangle$	$\langle 2 \rangle$	$\perp$	$\langle \bar{2} \rangle$
$\langle 2 \rangle$	$\langle 1 \rangle$	$\langle 2 \rangle$	$\langle \bar{1} \rangle$	$\perp$
$\langle \bar{1} \rangle$	$\perp$	$\langle 2 \rangle$	$\langle \bar{1} \rangle$	$\langle \bar{2} \rangle$
$\langle \bar{2} \rangle$	$\langle 1 \rangle$	$\perp$	$\langle \bar{1} \rangle$	$\langle \bar{2} \rangle$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

To check that mutually recursive formulas are cycle-free, one proceeds the following way. When a mutually recursive formula is encountered, for instance  $\mu \overline{X_i} . \varphi_i$  in  $\psi$ , every recursive binding is checked. Because of mutual recursion, formulas cannot be checked independently and a variable must be expanded the first time it is encountered (rule REC). However there is no need to expand it a second time (rule NOREC). When checking  $\psi$ , as the formulas bound to the enclosing recursion have been checked to be cycle free, there is no need to further check these variables (rule IGN). To account for shadowing of variables, newly bound recursion variables are removed from  $I$  and  $R$  when checking a recursion. One may easily prove that if  $\Delta \parallel \Gamma \vdash_I^R \varphi$  holds, then  $I \cap R = \emptyset$ .

This relation decides whether a formula is cycle free because, if it is not, there must be a recursive binding of  $X_i$  to  $\varphi_i$  such that  $\varphi_i \{ \varphi_i / X_i \} \{ \varphi_j / X_j \}$  exhibits a modality cycle above  $X_i$ , where the  $X_j$  are recursion variables being defined (either in the recursion defining  $X_i$  or in an enclosing recursion definition).

With these definitions, a first result can now be shown: in the finite focused-tree interpretation, the least and greatest fixpoints coincide for cycle-free formulas. To this end, a stronger result is proved, which states that a given focused tree is in the interpretation of a formula if it is in a finite unfolding of the formula. In the base case, the formula  $\sigma \wedge \neg \sigma$  is used as “false”.

**Definition 5.3.1 (Finite unfolding)** A finite unfolding of a formula  $\varphi$  belongs to the set  $\text{unf}(\varphi)$  inductively defined as

$$\begin{aligned} \text{unf}(\varphi) &\stackrel{\text{def}}{=} \{\varphi\} \quad \text{for } \varphi = \top, \sigma, \neg \sigma, \textcircled{\sigma}, \neg \textcircled{\sigma}, X, \neg \langle a \rangle \top \\ \text{unf}(\varphi \vee \psi) &\stackrel{\text{def}}{=} \{\varphi' \vee \psi' \mid \varphi' \in \text{unf}(\varphi), \psi' \in \text{unf}(\psi)\} \\ \text{unf}(\varphi \wedge \psi) &\stackrel{\text{def}}{=} \{\varphi' \wedge \psi' \mid \varphi' \in \text{unf}(\varphi), \psi' \in \text{unf}(\psi)\} \\ \text{unf}(\langle a \rangle \varphi) &\stackrel{\text{def}}{=} \{\langle a \rangle \varphi' \mid \varphi' \in \text{unf}(\varphi)\} \\ \text{unf}(\mu \overline{X_i} . \varphi_i \text{ in } \psi) &\stackrel{\text{def}}{=} \text{unf}(\psi \{ \mu \overline{X_i} . \varphi_i \text{ in } X_i / X_i \}) \\ \text{unf}(\nu \overline{X_i} . \varphi_i \text{ in } \psi) &\stackrel{\text{def}}{=} \text{unf}(\psi \{ \nu \overline{X_i} . \varphi_i \text{ in } X_i / X_i \}) \\ \text{unf}(\mu \overline{X_i} . \varphi_i \text{ in } \psi) &\stackrel{\text{def}}{=} \sigma \wedge \neg \sigma \\ \text{unf}(\nu \overline{X_i} . \varphi_i \text{ in } \psi) &\stackrel{\text{def}}{=} \sigma \wedge \neg \sigma \end{aligned}$$

**Lemma 5.3.2** Let  $\varphi$  a cycle-free formula. If  $f \in \llbracket \varphi \rrbracket_V$  then  $f \in \llbracket \text{unf}(\varphi) \rrbracket_V$ .

The reason why this lemma holds is the following. Given a tree satisfying  $\varphi$ , we deduce from the hypothesis that  $\varphi$  is cycle free the fact that every node of the tree will be tested a finite number of times against every subformula of  $\varphi$ . As the tree and the number of subformulas are finite, the satisfaction derivation is finite hence only a finite number of unfolding is necessary to prove that the tree satisfies the formula, which is what the lemma states. As least and greatest fixpoints coincide when only a finite number of unfolding is required, this is sufficient to show that they collapse. Note that this would not hold if infinite trees were allowed: the formula  $\mu X. \langle 1 \rangle X$  is cycle free, but its interpretation is empty, whereas the interpretation of  $\nu X. \langle 1 \rangle X$  includes every tree with an infinite branch of  $\langle 1 \rangle$  children.

We now illustrate why formulas need to be cycle free for the fixpoints to collapse. Consider the formula  $\mu X. \langle 1 \rangle \langle \bar{1} \rangle X$ . Its interpretation is empty. The interpretation of  $\nu X. \langle 1 \rangle \langle \bar{1} \rangle X$  however contains every focused tree that has one  $\langle 1 \rangle$  child.

*Proof outline:*

The result is a consequence of the fact that a sub-formula is never confronted twice to the same node of the focused tree as there is no cycle in the formula. It is thus possible to annotate occurrences of  $\nu$  and  $\mu$  with the direction the formula is exploring for each variable, as in Fig. 5.3, and prove the result by induction on the size of focused tree in this direction.

More precisely, each variable in every  $\mu$  and  $\nu$  of the initial formula is given a unique identifier.

The induction principle relies on the *longest path* of a focused tree. Given a tree and a direction (which may be  $\_$ ), we define the longest path as the longest cycle-free path that starts in the initial direction.

We then prove the property that a tree  $f$  belongs to the finite unfolding of  $\varphi$  by induction on the lexical order of:

1. the number of fixpoints not yet annotated;
2. the max of the lengths of the longest path for a given unique identifier according to the direction for this identifier;
3. the size of the formula.

The interesting case is an annotated formula recursion  $\varphi = \overline{\mu X_i. \varphi_i}$  in  $\psi$ . This formula may only have been produced by an expansion. As the formula is cycle-free, at least one modality has been encountered since the expansion for each identifier associated with the  $X_i$ , and these modalities are compatible with the previous directions (if they existed). The longest path for each identifier is thus shorter hence we have by induction that  $f$  is in a finite expansion of the expansion of  $\varphi$ .  $\square$

In the rest of the dissertation, only least fixpoints are considered. An important consequence of Lemma 5.3.2 is that the logic restricted in this way is closed under negation using De Morgan's dualities, extended to eventualities and fixpoints as follows:

$$\neg \langle a \rangle \varphi \stackrel{\text{def}}{=} \neg \langle a \rangle \top \vee \langle a \rangle \neg \varphi$$

$$\neg \overline{\mu X_i. \varphi_i} \text{ in } \psi \stackrel{\text{def}}{=} \overline{\mu X_i. \neg \varphi_i \{ \overline{X_i / \_} \}} \text{ in } \neg \psi \{ \overline{X_i / \_} \}$$

## 5.4 Translations of XML Concepts

The interpretation of XPath expressions as sets of focused trees is given:

$$\begin{aligned}
 \mathcal{S}_e[\cdot] &: \mathcal{L}_{\text{XPath}} \rightarrow 2^{\mathcal{F}} \rightarrow 2^{\mathcal{F}} \\
 \mathcal{S}_e[/p]_F &\stackrel{\text{def}}{=} \mathcal{S}_p[p]_{\text{root}(F)} \\
 \mathcal{S}_e[p]_F &\stackrel{\text{def}}{=} \mathcal{S}_p[p]_{\{(\sigma^{\otimes}[tl], c) \in F\}} \\
 \mathcal{S}_e[e_1 \mid e_2]_F &\stackrel{\text{def}}{=} \mathcal{S}_e[e_1]_F \cup \mathcal{S}_e[e_2]_F \\
 \mathcal{S}_e[e_1 \cap e_2]_F &\stackrel{\text{def}}{=} \mathcal{S}_e[e_1]_F \cap \mathcal{S}_e[e_2]_F
 \end{aligned}$$

$$\begin{aligned}
 \mathcal{S}_p[\cdot] &: \text{Path} \rightarrow 2^{\mathcal{F}} \rightarrow 2^{\mathcal{F}} \\
 \mathcal{S}_p[p_1/p_2]_F &\stackrel{\text{def}}{=} \{f' \mid f' \in \mathcal{S}_p[p_2]_{(\mathcal{S}_p[p_1]_F)}\} \\
 \mathcal{S}_p[p[q]]_F &\stackrel{\text{def}}{=} \{f \mid f \in \mathcal{S}_p[p]_F \wedge \mathcal{S}_q[q]_f\} \\
 \mathcal{S}_p[a::\sigma]_F &\stackrel{\text{def}}{=} \{f \mid f \in \mathcal{S}_a[a]_F \wedge \text{nm}(f) = \sigma\} \\
 \mathcal{S}_p[a::*]_F &\stackrel{\text{def}}{=} \{f \mid f \in \mathcal{S}_a[a]_F\}
 \end{aligned}$$

$$\begin{aligned}
 \mathcal{S}_q[\cdot] &: \text{Qualif} \rightarrow \mathcal{F} \rightarrow \{\text{true}, \text{false}\} \\
 \mathcal{S}_q[q_1 \text{ and } q_2]_f &\stackrel{\text{def}}{=} \mathcal{S}_q[q_1]_f \wedge \mathcal{S}_q[q_2]_f \\
 \mathcal{S}_q[q_1 \text{ or } q_2]_f &\stackrel{\text{def}}{=} \mathcal{S}_q[q_1]_f \vee \mathcal{S}_q[q_2]_f \\
 \mathcal{S}_q[\text{not } q]_f &\stackrel{\text{def}}{=} \neg \mathcal{S}_q[q]_f \\
 \mathcal{S}_q[p]_f &\stackrel{\text{def}}{=} \mathcal{S}_p[p]_{\{f\}} \neq \emptyset
 \end{aligned}$$

$$\begin{aligned}
 \mathcal{S}_a[\cdot] : Axis &\rightarrow 2^{\mathcal{F}} \rightarrow 2^{\mathcal{F}} \\
 \mathcal{S}_a[\text{self}]_F &\stackrel{\text{def}}{=} F \\
 \mathcal{S}_a[\text{child}]_F &\stackrel{\text{def}}{=} \text{fchild}(F) \cup \mathcal{S}_a[\text{following-sibling}]_{\text{fchild}(F)} \\
 \mathcal{S}_a[\text{following-sibling}]_F &\stackrel{\text{def}}{=} \text{nsibling}(F) \cup \mathcal{S}_a[\text{following-sibling}]_{\text{nsibling}(F)} \\
 \mathcal{S}_a[\text{preceding-sibling}]_F &\stackrel{\text{def}}{=} \text{psibling}(F) \cup \mathcal{S}_a[\text{preceding-sibling}]_{\text{psibling}(F)} \\
 \mathcal{S}_a[\text{parent}]_F &\stackrel{\text{def}}{=} \text{parent}(F) \\
 \mathcal{S}_a[\text{descendant}]_F &\stackrel{\text{def}}{=} \mathcal{S}_a[\text{child}]_F \cup \mathcal{S}_a[\text{descendant}]_{(\mathcal{S}_a[\text{child}]_F)} \\
 \mathcal{S}_a[\text{descendant-or-self}]_F &\stackrel{\text{def}}{=} F \cup \mathcal{S}_a[\text{descendant}]_F \\
 \mathcal{S}_a[\text{ancestor}]_F &\stackrel{\text{def}}{=} \mathcal{S}_a[\text{parent}]_F \cup \mathcal{S}_a[\text{ancestor}]_{(\mathcal{S}_a[\text{parent}]_F)} \\
 \mathcal{S}_a[\text{ancestor-or-self}]_F &\stackrel{\text{def}}{=} F \cup \mathcal{S}_a[\text{ancestor}]_F \\
 \mathcal{S}_a[\text{following}]_F &\stackrel{\text{def}}{=} \mathcal{S}_a[\text{descendant-or-self}]_{(\mathcal{S}_a[\text{following-sibling}]_{(\mathcal{S}_a[\text{ancestor-or-self}]_F)})} \\
 \mathcal{S}_a[\text{preceding}]_F &\stackrel{\text{def}}{=} \mathcal{S}_a[\text{descendant-or-self}]_{(\mathcal{S}_a[\text{preceding-sibling}]_{(\mathcal{S}_a[\text{ancestor-or-self}]_F)})} \\
 \\
 \text{fchild}(F) &\stackrel{\text{def}}{=} \{f \langle 1 \rangle \mid f \in F \wedge f \langle 1 \rangle \text{ defined}\} \\
 \text{nsibling}(F) &\stackrel{\text{def}}{=} \{f \langle 2 \rangle \mid f \in F \wedge f \langle 2 \rangle \text{ defined}\} \\
 \text{psibling}(F) &\stackrel{\text{def}}{=} \{f \langle \bar{2} \rangle \mid f \in F \wedge f \langle \bar{2} \rangle \text{ defined}\} \\
 \text{parent}(F) &\stackrel{\text{def}}{=} \{(\sigma^\circ[\text{rev\_a}(tl_l, t :: tl_r)], c) \\
 &\quad \mid (t, (tl_l, c[\sigma^\circ], tl_r)) \in F\} \\
 \text{rev\_a}(\epsilon, tl_r) &\stackrel{\text{def}}{=} tl_r \\
 \text{rev\_a}(t :: tl_l, tl_r) &\stackrel{\text{def}}{=} \text{rev\_a}(tl_l, t :: tl_r) \\
 \text{root}(F) &\stackrel{\text{def}}{=} \{(\sigma^\circ[tl], (tl, \text{Top}, tl)) \in F\} \cup \text{root}(\text{parent}(F))
 \end{aligned}$$

### 5.4.1 XPath Embedding

An XPath expression can be translated into an equivalent formula in  $\mathcal{L}_\mu$  which performs navigation in focused trees in binary style, as presented in the Section 4.4 of previous Chapter 4. A stronger result can be proved:

**Proposition 5.4.1 (Translation Correctness)** *The following hold for an XPath expression  $e$  and a  $\mathcal{L}_\mu$  formula  $\varphi$ , with  $\psi = E^\rightarrow[e]_\varphi$ :*

1.  $\llbracket \psi \rrbracket_\emptyset = \mathcal{S}_e[\llbracket e \rrbracket_\varphi]_\emptyset$
2.  $\psi$  is cycle-free
3. the size of  $\psi$  is linear in the size of  $e$  and  $\varphi$

*Proof outline:* The proof uses a structural induction that “peels off” the compositional layers of each set of rules over focused trees. The cycle-free

Translation of `following-sibling::a/preceding-sibling::b`  
 into  $\mathcal{L}_\mu$ :  $b \wedge [\mu Y. \langle 2 \rangle ( a \wedge (\mu Z. \langle \bar{2} \rangle \textcircled{\text{S}} \vee \langle \bar{2} \rangle Z ) ) \vee \langle 2 \rangle Y]$

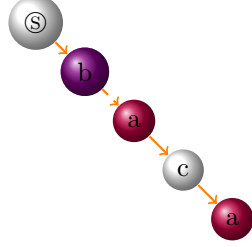


Figure 5.4: Example of Back and Forth XPath Navigation Translation.

part follows from the fact that translated fixpoint formulas are closed and there is no nesting of modalities with converse programs between a fixpoint variable and its binder. Each XPath navigation step is cycle-free, and their composition yields a proper nesting of fixpoint formulas which is also cycle-free. Figure 5.4 illustrates this on an typical example. Finally, formal translations do not duplicate any subformula of arbitrary length.  $\square$

#### 5.4.2 Embedding Regular Tree Languages

The straightforward isomorphism between unranked and binary regular tree types (presented in Section 2.1.3 of Chapter 2) is used. The translation from binary regular tree types into  $\mathcal{L}_\mu$  is given by the function  $\llbracket \cdot \rrbracket$  as follows:

$$\begin{aligned} \llbracket \cdot \rrbracket &: \mathcal{L}_{\text{bt}} \rightarrow \mathcal{L}_\mu \\ \llbracket \emptyset \rrbracket &\stackrel{\text{def}}{=} \sigma \wedge \neg \sigma \\ \llbracket \epsilon \rrbracket &\stackrel{\text{def}}{=} \sigma \wedge \neg \sigma \\ \llbracket T_1 \mid T_2 \rrbracket &\stackrel{\text{def}}{=} \llbracket T_1 \rrbracket \vee \llbracket T_2 \rrbracket \\ \llbracket \sigma(X_1, X_2) \rrbracket &\stackrel{\text{def}}{=} \sigma \wedge \text{succ}_1(X_1) \wedge \text{succ}_2(X_2) \\ \llbracket \text{let } \overline{X_i.T_i} \text{ in } T \rrbracket &\stackrel{\text{def}}{=} \mu \overline{X_i. \llbracket T_i \rrbracket} \text{ in } \llbracket T \rrbracket \end{aligned}$$

where the formula  $\sigma \wedge \neg \sigma$  is used as “false”, and the function  $\text{succ}_\alpha(\cdot)$  takes care of setting the type frontier:

$$\text{succ}_\alpha(X) = \begin{cases} \neg \langle \alpha \rangle \top \vee \langle \alpha \rangle X & \text{if } \text{nullable}(X) \\ \langle \alpha \rangle X & \text{if not } \text{nullable}(X) \end{cases}$$

according to the predicate  $\text{nullable}(\cdot)$  (defined in Section 4.5 of previous chapter) which indicates whether a type contains the empty tree.

Note that the translation of a regular tree type uses only downward modalities since it describes the allowed subtrees at a given context. No additional restriction is imposed on the context from which the type definition starts. In particular, navigation is allowed in the upward direction so that type constraints for which only partial knowledge in a given direction is known can be

supported. However, when the position of the root is known, conditions similar to those of absolute paths are added. This is particularly useful when a regular type is used by an XPath expression that starts its navigation at the root ( $/p$ ) since the path will not go above the root of the type (by adding the restriction  $\mu Z. \neg \langle \bar{1} \rangle \top \vee \langle \bar{2} \rangle Z$ ).

On the other hand, if the type is compared with another type (typically to check inclusion of the result of an XPath expression in this type), then there is no restriction as to where the root of the type is (the translation does not impose the chosen node to be at the root). This is particularly useful since an XPath expression usually returns a set of nodes deep in the tree which may be compared to this partially defined type.

# Satisfiability-Testing Algorithm

---

## 6.1 Introduction

This chapter presents the algorithm for deciding the logic introduced in previous chapter. It is shown sound and complete, and the time complexity boundary is proved. The combination of all these ingredients leads to the main result: a satisfiability algorithm for a logic for finite trees whose time complexity is a simple exponential of the size of a formula. With these proofs, a practically effective system for solving the satisfiability of a formula is described. The system has been experimented with some decision problems such as XPath containment, emptiness, overlap, and coverage, with or without type constraints.

**Chapter Outline** Some preliminary notions are defined in Section 6.2. The satisfiability algorithm is then introduced in Section 6.3 and proven correct in Section 6.4, with details of the implementation discussed in Section 6.5. Applications for type checking are described in Section 6.6 along with some experimental results, before the approach outcome is discussed in 6.7.

## 6.2 Preliminary Definitions

The *unwinding* of a formula  $\varphi = (\mu\overline{X_i}.\varphi_i \text{ in } \psi)$ , noted  $\text{exp}(\varphi)$ , is defined as  $\text{exp}(\varphi) \stackrel{\text{def}}{=} \psi\{\overline{\mu X_i.\varphi_i \text{ in } X_i/X_i}\}$  which denotes the formula  $\psi$  in which every occurrence of a  $X_i$  is replaced by  $(\mu\overline{X_i}.\varphi_i \text{ in } X_i)$ .

The *Fisher-Ladner closure*  $\text{cl}(\psi)$  of a formula  $\psi$  is defined as the set of all subformulas of  $\psi$  where fixpoint formulas are additionally unwound once. Specifically, the relation  $\rightarrow_e \subseteq \mathcal{L}_\mu \times \mathcal{L}_\mu$  is defined as the least relation that satisfies the following:

- $\varphi_1 \wedge \varphi_2 \rightarrow_e \varphi_1, \varphi_1 \wedge \varphi_2 \rightarrow_e \varphi_2$
- $\varphi_1 \vee \varphi_2 \rightarrow_e \varphi_1, \varphi_1 \vee \varphi_2 \rightarrow_e \varphi_2$
- $\langle a \rangle \varphi' \rightarrow_e \varphi'$
- $\mu\overline{X_i}.\varphi_i \text{ in } \psi \rightarrow_e \text{exp}(\mu\overline{X_i}.\varphi_i \text{ in } \psi)$

The closure  $\text{cl}(\psi)$  is the smallest set  $S$  that contains  $\psi$  and closed under the relation  $\rightarrow_e$ , i.e. if  $\varphi_1 \in S$  and  $\varphi_1 \rightarrow_e \varphi_2$  then  $\varphi_2 \in S$ .

$\Sigma(\psi)$  denotes the set of atomic propositions used in  $\psi$  along with an other name,  $\sigma_x$ , representing atomic propositions not occurring in  $\psi$ .

The extended closure is defined as  $\text{cl}^*(\psi) = \text{cl}(\psi) \cup \{\neg\varphi \mid \varphi \in \text{cl}(\psi)\}$ . Every formula  $\varphi \in \text{cl}^*(\psi)$  can be seen as a boolean combination of formulas of a set called the Lean of  $\psi$ , inspired from [Pan et al., 2006]. This set is noted  $\text{Lean}(\psi)$  and defined as follows:

$$\text{Lean}(\psi) = \{\langle a \rangle \top \mid a \in \{1, 2, \bar{1}, \bar{2}\}\} \cup \Sigma(\psi) \cup \{\textcircled{S}\} \cup \{\langle a \rangle \varphi \mid \langle a \rangle \varphi \in \text{cl}(\psi)\}$$

A  $\psi$ -type (or simply a “type”) (Hintikka set in the temporal logic literature) is a set  $t \subseteq \text{Lean}(\psi)$  such that:

- $\forall \langle a \rangle \varphi \in \text{Lean}(\psi), \langle a \rangle \varphi \in t \Rightarrow \langle a \rangle \top \in t$  (modal consistency);
- $\langle \bar{1} \rangle \top \notin t \vee \langle \bar{2} \rangle \top \notin t$  (a tree node cannot be both a first child and a second child);
- exactly one atomic proposition  $\sigma \in t$  (XML labeling); the function  $\sigma(t)$  is used to return the atomic proposition of a type  $t$ ;
- $\textcircled{S}$  may belong to  $t$ .

$\text{Typ}(\psi)$  denotes the set of  $\psi$ -types. For a  $\psi$ -type  $t$ , the *complement* of  $t$  is the set  $\text{Lean}(\psi) \setminus t$ .

A type determines a truth assignment of every formula in  $\text{cl}^*(\psi)$  with the relation  $\dot{\in}$  defined in Figure 6.1.

Note that such derivations are finite because the number of naked  $\mu\bar{X}_i.\varphi_i$  in  $\psi$  (that do not occur under modalities) strictly decreases after each expansion.

The notation  $\varphi \dot{\in} t$  is often used if there are some  $T, F$  such that  $\varphi \dot{\in} t \Rightarrow (T, F)$ . A formula  $\varphi$  is true at a type  $t$  iff  $\varphi \dot{\in} t$ .

The truth status of a formula is now related to the truth assignment of its  $\psi$ -types.

**Proposition 6.2.1** *If  $\varphi \dot{\in} t \Rightarrow (T, F)$ , then  $T \subseteq t$ ,  $F \subseteq \text{Lean}(\psi) \setminus t$ , and  $\bigwedge_{\psi \in T} \psi \wedge \bigwedge_{\psi \in F} \neg\psi \Rightarrow \varphi$ . If  $\varphi \not\dot{\in} t \Rightarrow (T, F)$ , then  $T \subseteq t$ ,  $F \subseteq \text{Lean}(\psi) \setminus t$ , and  $\bigwedge_{\psi \in T} \psi \wedge \bigwedge_{\psi \in F} \neg\psi \Rightarrow \neg\varphi$ .*

*Proof outline:* Immediate by induction on the derivations.  $\square$

A compatibility relation is now defined between types. This relation establishes which formulas must hold in a type in order for it to be a witness for a modal formula.

**Definition 6.2.2 (Compatibility relation)** : *Two types  $t, t'$  are compatible under  $a \in \{1, 2\}$ , written  $\Delta_a(t, t')$ , iff*

$$\begin{aligned} \forall \langle a \rangle \varphi \in \text{Lean}(\psi), \langle a \rangle \varphi \in t &\Leftrightarrow \varphi \dot{\in} t' \\ \forall \langle \bar{a} \rangle \varphi \in \text{Lean}(\psi), \langle \bar{a} \rangle \varphi \in t' &\Leftrightarrow \varphi \dot{\in} t \end{aligned}$$

$$\begin{array}{c}
 \frac{}{\top \dot{\in} t \Rightarrow (\emptyset, \emptyset)} \qquad \frac{\varphi \in \text{Lean}(\psi) \quad \varphi \in t}{\varphi \dot{\in} t \Rightarrow (\{\varphi\}, \emptyset)} \\
 \\
 \frac{\varphi_1 \dot{\in} t \Rightarrow (T_1, F_1) \quad \varphi_2 \dot{\in} t \Rightarrow (T_2, F_2)}{\varphi_1 \wedge \varphi_2 \dot{\in} t \Rightarrow (T_1 \cup T_2, F_1 \cup F_2)} \qquad \frac{\varphi_1 \dot{\in} t \Rightarrow (T_1, F_1)}{\varphi_1 \vee \varphi_2 \dot{\in} t \Rightarrow (T_1, F_1)} \\
 \\
 \frac{\varphi_2 \dot{\in} t \Rightarrow (T_2, F_2)}{\varphi_1 \vee \varphi_2 \dot{\in} t \Rightarrow (T_2, F_2)} \qquad \frac{\varphi \dot{\notin} t \Rightarrow (T, F)}{\neg \varphi \dot{\in} t \Rightarrow (T, F)} \\
 \\
 \frac{\text{exp}(\overline{\mu X_i. \varphi_i} \text{ in } \psi) \dot{\in} t \Rightarrow (T, F)}{\overline{\mu X_i. \varphi_i} \text{ in } \psi \dot{\in} t \Rightarrow (T, F)} \qquad \frac{\varphi \in \text{Lean}(\psi) \quad \varphi \dot{\notin} t}{\varphi \dot{\notin} t \Rightarrow (\emptyset, \{\varphi\})} \\
 \\
 \frac{\varphi_1 \dot{\notin} t \Rightarrow (T_1, F_1) \quad \varphi_2 \dot{\notin} t \Rightarrow (T_2, F_2)}{\varphi_1 \vee \varphi_2 \dot{\notin} t \Rightarrow (T_1 \cup T_2, F_1 \cup F_2)} \qquad \frac{\varphi_1 \dot{\notin} t \Rightarrow (T_1, F_1)}{\varphi_1 \wedge \varphi_2 \dot{\notin} t \Rightarrow (T_1, F_1)} \\
 \\
 \frac{\varphi_2 \dot{\notin} t \Rightarrow (T_2, F_2)}{\varphi_1 \wedge \varphi_2 \dot{\notin} t \Rightarrow (T_2, F_2)} \qquad \frac{\varphi \dot{\in} t \Rightarrow (T, F)}{\neg \varphi \dot{\notin} t \Rightarrow (T, F)} \\
 \\
 \frac{\text{exp}(\overline{\mu X_i. \varphi_i} \text{ in } \psi) \dot{\notin} t \Rightarrow (T, F)}{\overline{\mu X_i. \varphi_i} \text{ in } \psi \dot{\notin} t \Rightarrow (T, F)}
 \end{array}$$

Figure 6.1: Truth Assignment of a Formula

### 6.3 The Algorithm

The algorithm works on sets of triples of the form  $(t, w_1, w_2)$  where  $t$  is a type, and  $w_1$  and  $w_2$  are sets of types which represent all possible witnesses for  $t$  according to relations  $\Delta_1$  and  $\Delta_2$ .

The algorithm proceeds in a bottom-up approach, repeatedly adding new triples until a satisfying model is found (i.e. a triple whose first component is a type implying the formula), or until no more triple can be added. Each iteration of the algorithm builds types representing deeper trees (in the 1 and 2 direction) with pending backward modalities that will be fulfilled at later iterations. Types with no backward modalities are satisfiable, and if such a type implies the formula being tested, then it is satisfiable. The main iteration is as follows:

```

X ← ∅
repeat
  X' ← X
  X ← Upd(X')
  if FinalCheck(ψ, X) then
    return “ψ is satisfiable”
until X = X'
return “ψ is unsatisfiable”
    
```

where  $X \subseteq \text{Typ}(\psi) \times 2^{\text{Typ}(\psi)} \times 2^{\text{Typ}(\psi)}$  and the operations  $\text{Upd}(\cdot)$  and  $\text{FinalCheck}(\cdot)$  are defined on Figure 6.2.

$$\begin{aligned}
 \text{Upd}(X) \stackrel{\text{def}}{=} & X \cup \{(t, \mathbf{w}_1(t, X^\circ), \mathbf{w}_2(t, X^\circ)) \mid \begin{array}{l} \textcircled{\text{S}} \notin t \subseteq \text{Typ}(\psi) \\ \wedge \langle 1 \rangle \top \in t \Rightarrow \mathbf{w}_1(t, X^\circ) \neq \emptyset \\ \wedge \langle 2 \rangle \top \in t \Rightarrow \mathbf{w}_2(t, X^\circ) \neq \emptyset \end{array} \\
 & \cup \{(t, \mathbf{w}_1(t, X^\circ), \mathbf{w}_2(t, X^\circ))^{\textcircled{\text{S}}} \mid \begin{array}{l} \textcircled{\text{S}} \in t \subseteq \text{Typ}(\psi) \\ \wedge \langle 1 \rangle \top \in t \Rightarrow \mathbf{w}_1(t, X^\circ) \neq \emptyset \\ \wedge \langle 2 \rangle \top \in t \Rightarrow \mathbf{w}_2(t, X^\circ) \neq \emptyset \end{array} \\
 & \cup \{(t, \mathbf{w}_1(t, X^{\textcircled{\text{S}}}), \mathbf{w}_2(t, X^{\textcircled{\text{S}}}))^{\textcircled{\text{S}}} \mid \begin{array}{l} \textcircled{\text{S}} \notin t \subseteq \text{Typ}(\psi) \\ \wedge \langle 1 \rangle \top \in t \Rightarrow \mathbf{w}_1(t, X^{\textcircled{\text{S}}}) \neq \emptyset \\ \wedge \langle 2 \rangle \top \in t \Rightarrow \mathbf{w}_2(t, X^{\textcircled{\text{S}}}) \neq \emptyset \end{array} \\
 & \cup \{(t, \mathbf{w}_1(t, X^\circ), \mathbf{w}_2(t, X^{\textcircled{\text{S}}}))^{\textcircled{\text{S}}} \mid \begin{array}{l} \textcircled{\text{S}} \notin t \subseteq \text{Typ}(\psi) \\ \wedge \langle 1 \rangle \top \in t \Rightarrow \mathbf{w}_1(t, X^\circ) \neq \emptyset \\ \wedge \langle 2 \rangle \top \in t \Rightarrow \mathbf{w}_2(t, X^{\textcircled{\text{S}}}) \neq \emptyset \end{array} \\
 \\
 \mathbf{w}_a(t, X) \stackrel{\text{def}}{=} & \{\mathbf{type}(x) \mid x \in X \wedge \langle \bar{a} \rangle \top \in \mathbf{type}(x) \wedge \Delta_a(t, \mathbf{type}(x))\} \\
 \text{FinalCheck}(\psi, X) \stackrel{\text{def}}{=} & \exists x \in X^{\textcircled{\text{S}}}, \text{dsat}(x, \psi) \wedge \forall a \in \{\bar{1}, \bar{2}\}, \langle a \rangle \top \notin \mathbf{type}(x) \\
 \text{dsat}((t, w_1, w_2), \psi) \stackrel{\text{def}}{=} & \psi \in t \vee \exists x', \text{dsat}(x', \psi) \wedge (x' \in w_1 \vee x' \in w_2)
 \end{aligned}$$

$$X^{\textcircled{\text{S}}} \stackrel{\text{def}}{=} \{x \in X \mid x = (-, -, -)^{\textcircled{\text{S}}}\}$$

$$X^\circ \stackrel{\text{def}}{=} \{x \in X \mid x = (-, -, -)\}$$

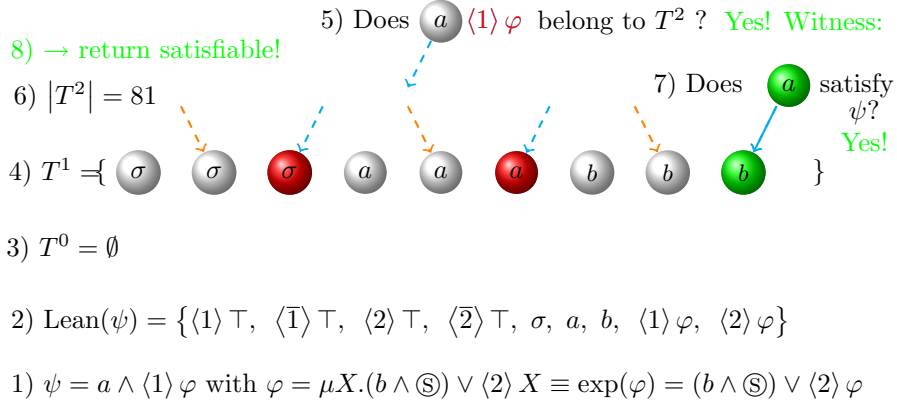
$$\mathbf{type}((t, w_1, w_2)) \stackrel{\text{def}}{=} t$$

Figure 6.2: Operations used by the Algorithm.

$X^i$  and  $T^i$  respectively denote the set of triples and the set of types after  $i$  iterations:  $T^i = \{\mathbf{type}(x) \mid x \in X^i\}$ . Note that  $T^{i+1}$  is the set of types for which at least one witness belongs to  $T^i$ .

### 6.3.1 Example Run of the Algorithm

Figure 6.3 illustrates a run of the algorithm for checking the non-emptiness of the simple XPath expression  $e = \mathbf{self}::\mathbf{b}/\mathbf{parent}::\mathbf{a}$ . This expression is first compiled into the logic as explained in section 5.4.1. The resulting formula  $\psi = E^{\rightarrow} \llbracket e \rrbracket_{\top}$  is shown on Figure 6.3 (step 1). As a second step,  $\text{Lean}(\psi)$  is computed. Then the fixpoint computation starts: the set of types  $T^1$  contains all possible leaves (step 3). For each type in  $T^2 \setminus T^1$ , a witness must be found in  $T^1$ . The algorithm notably finds a witness for a particular  $\psi$ -type  $t$  such that  $a \wedge \langle 1 \rangle \varphi \in t$  (step 5).  $T^2$  finally contains 81  $\psi$ -types (step 6).  $t$  happens to satisfy the initial formula  $\psi$  (step 7), therefore the algorithm stops just after computing  $T^2$  (step 8) because the structure built by connecting  $t$  and its witness (as drawn on Figure 6.3) is a finite tree which contains a node on which  $\psi$  is satisfied. Thus  $\mathbf{self}::\mathbf{b}/\mathbf{parent}::\mathbf{a}$  is satisfiable.


 Figure 6.3: Run of the Algorithm for Checking Emptiness of `self::b/parent::a`

## 6.4 Correctness and Complexity

In this section the correctness of the satisfiability testing algorithm, is proved, and it is shown that its time complexity is  $2^{O(|\text{Lean}(\psi)|)}$ .

**Theorem 6.4.1 (Correctness)** *The algorithm decides satisfiability of  $\mathcal{L}_\mu$  formulas over finite focused trees.*

**Termination** For  $\psi \in \mathcal{L}_\mu$ , since  $\text{cl}(\psi)$  is a finite set,  $\text{Lean}(\psi)$  and  $2^{\text{Lean}(\psi)}$  are also finite. Furthermore,  $\text{Upd}(\cdot)$  is monotonic and each  $X^i$  is included in the finite set  $\text{Typ}(\psi) \times 2^{\text{Typ}(\psi)} \times 2^{\text{Typ}(\psi)}$ , therefore the algorithm terminates. To finish the proof, it thus suffices to prove soundness and completeness.

**Preliminary Definitions for Soundness** First, a notion of partial satisfiability is introduced for a formula. In this partial satisfiability notion, backward modalities are only checked up to a given level. A formula  $\varphi$  is partially satisfied iff  $\llbracket \varphi \rrbracket_V^0 \neq \emptyset$  as defined in Figure 6.4.

For a type  $t$ ,  $\varphi_c(t)$  denotes the most constrained formula, where atoms are taken from  $\text{Lean}(\psi)$ . In the following,  $\circ$  stands for  $\textcircled{S}$  if  $\textcircled{S} \in t$ , and for  $\neg\textcircled{S}$  otherwise.

$$\varphi_c(t) = \sigma(t) \wedge \bigwedge_{\sigma \in \Sigma, \sigma \notin t} \neg \sigma \wedge \circ \wedge \bigwedge_{\langle a \rangle \varphi \in t} \langle a \rangle \varphi \wedge \bigwedge_{\langle a \rangle \varphi \notin t} \neg \langle a \rangle \varphi$$

A notion of *paths* is now introduced. Paths written  $\rho$  are concatenations of modalities: the empty path is written  $\epsilon$ , and path concatenation is written  $\rho a$ .

Every path may be given a *depth*:

$$\begin{aligned} \text{depth}(\epsilon) &\stackrel{\text{def}}{=} 0 \\ \text{depth}(\rho a) &\stackrel{\text{def}}{=} \text{depth}(\rho) + 1 \quad \text{if } a \in \{1, 2\} \\ \text{depth}(\rho a) &\stackrel{\text{def}}{=} \text{depth}(\rho) - 1 \quad \text{if } a \in \{\bar{1}, \bar{2}\} \end{aligned}$$

$$\begin{array}{ll}
 \llbracket \top \rrbracket_V^n \stackrel{\text{def}}{=} \mathcal{F} & \llbracket X \rrbracket_V^n \stackrel{\text{def}}{=} V(X) \\
 \llbracket \varphi \vee \psi \rrbracket_V^n \stackrel{\text{def}}{=} \llbracket \varphi \rrbracket_V^n \cup \llbracket \psi \rrbracket_V^n & \llbracket p \rrbracket_V^n \stackrel{\text{def}}{=} \{f \mid \mathbf{nm}(f) = p\} \\
 \llbracket \varphi \wedge \psi \rrbracket_V^n \stackrel{\text{def}}{=} \llbracket \varphi \rrbracket_V^n \cap \llbracket \psi \rrbracket_V^n & \llbracket \neg p \rrbracket_V^n \stackrel{\text{def}}{=} \{f \mid \mathbf{nm}(f) \neq p\} \\
 \llbracket \langle \bar{1} \rangle \varphi \rrbracket_V^0 \stackrel{\text{def}}{=} \mathcal{F} & \llbracket \textcircled{S} \rrbracket_V^n \stackrel{\text{def}}{=} \{f \mid f = (\sigma^{\textcircled{S}}[tl], c)\} \\
 \llbracket \langle \bar{2} \rangle \varphi \rrbracket_V^0 \stackrel{\text{def}}{=} \mathcal{F} & \llbracket \neg \textcircled{S} \rrbracket_V^n \stackrel{\text{def}}{=} \{f \mid f = (\sigma[tl], c)\} \\
 \\
 \llbracket \langle \bar{1} \rangle \varphi \rrbracket_V^{n>0} \stackrel{\text{def}}{=} \{f \langle 1 \rangle \mid f \in \llbracket \varphi \rrbracket_V^{n-1} \wedge f \langle 1 \rangle \text{ defined}\} \\
 \llbracket \langle \bar{2} \rangle \varphi \rrbracket_V^{n>0} \stackrel{\text{def}}{=} \{f \langle 2 \rangle \mid f \in \llbracket \varphi \rrbracket_V^{n-1} \wedge f \langle 2 \rangle \text{ defined}\} \\
 \llbracket \langle 1 \rangle \varphi \rrbracket_V^n \stackrel{\text{def}}{=} \{f \langle \bar{1} \rangle \mid f \in \llbracket \varphi \rrbracket_V^{n+1} \wedge f \langle \bar{1} \rangle \text{ defined}\} \\
 \llbracket \langle 2 \rangle \varphi \rrbracket_V^n \stackrel{\text{def}}{=} \{f \langle \bar{2} \rangle \mid f \in \llbracket \varphi \rrbracket_V^{n+1} \wedge f \langle \bar{2} \rangle \text{ defined}\} \\
 \llbracket \neg \langle a \rangle \top \rrbracket_V^n \stackrel{\text{def}}{=} \{f \mid f \langle a \rangle \text{ undefined}\} \\
 \llbracket \mu \overline{X_i} . \varphi_i \text{ in } \psi \rrbracket_V^n \stackrel{\text{def}}{=} \text{let } T_i = \left( \bigcap \left\{ \overline{T_i} \subseteq \overline{\mathcal{F}} \mid \llbracket \varphi_i \rrbracket_{V[\overline{T_i}/\overline{X_i}]}^n \subseteq \overline{T_i} \right\} \right)_i \\
 \text{in } \llbracket \psi \rrbracket_{V[\overline{T_i}/\overline{X_i}]}^n
 \end{array}$$

Figure 6.4: Partial Satisfiability

A forward path is a path that only mentions forward modalities.

A tree of types  $\mathcal{T}$  is defined as a tree whose nodes are types,  $\mathcal{T}(\bullet) = t$ , with at most two children,  $\mathcal{T} \langle 1 \rangle$  and  $\mathcal{T} \langle 2 \rangle$ . The navigation in tree of types is trivially extended to forward paths. A tree of types is *consistent* iff for every forward path  $\rho$  and for every child  $a$  of  $\mathcal{T} \langle \rho \rangle$ , the following holds:  $\mathcal{T} \langle \rho \rangle (\bullet) = t$ ,  $\mathcal{T} \langle \rho a \rangle (\bullet) = t'$  implies  $\langle a \rangle \top \in t$ ,  $\langle \bar{a} \rangle \top \in t'$ , and  $\Delta_a(t, t')$ .

Given a consistent tree of types  $\mathcal{T}$ , a dependency graph is now defined. In this graph, nodes are pairs of a forward path  $\rho$  and a formula in  $t = \mathcal{T} \langle \rho \rangle (\bullet)$  or the negation of a formula in the complement  $t$ . The directed edges of the graph are modalities consistent with the tree. For every  $(\rho, \varphi)$  in the nodes the following edges are built:

- $\varphi \in \Sigma(\psi) \cup \neg \Sigma(\psi) \cup \{\textcircled{S}, \neg \textcircled{S}, \langle a \rangle \top, \neg \langle a \rangle \top\}$ : no edge
- $\rho = \epsilon, \varphi = \langle \bar{a} \rangle \varphi'$  with  $a \in \{1, 2\}$ : no edge
- $\rho = \rho' a, \varphi = \langle a' \rangle \varphi'$ : let  $t = \mathcal{T} \langle \rho \rangle (\bullet)$ . Let first consider the case where  $a' \in \{1, 2\}$  and let  $t' = \mathcal{T} \langle \rho a' \rangle (\bullet)$ . As  $\mathcal{T}$  is consistent,  $\varphi' \in t'$  hence there are  $T, F$  such that  $\varphi' \in t' \implies (T, F)$  with  $T$  a subset of  $t'$ , and  $F$  a subset of the complement of  $t'$ . For every  $\varphi_T \in T$  an edge  $a'$  is added to  $(\rho a', \varphi_T)$ , and for every  $\varphi_F \in F$  an edge  $a'$  is added to  $(\rho a', \neg \varphi_F)$ . Consider now the case where  $a' \in \{\bar{1}, \bar{2}\}$  and first show that  $a' = \bar{a}$ . As  $\mathcal{T}$  is consistent,  $\langle \bar{a} \rangle \top$  in  $t$ . Moreover, as  $t$  is a tree type, it must contain  $\langle a' \rangle \top$ . As  $a'$  is a backward modality, it must be equal to  $\bar{a}$  as at most one may be present. Hence  $\rho' a a' = \rho'$  holds. Let  $t' = \mathcal{T} \langle \rho' \rangle (\bullet)$ . By

consistency,  $\varphi' \in t'$ , hence  $\varphi' \in t' \implies (T, F)$  and edges are added as in the previous case: to  $(\rho', \varphi_T)$  and to  $(\rho', \neg\varphi_F)$ .

- $\rho = \rho'a, \varphi = \neg\langle a' \rangle \varphi'$ : let  $t = \mathcal{T} \langle \rho \rangle (\bullet)$ . If  $\langle a' \rangle \top$  is not in  $t$  then no edge is added. Otherwise, one proceeds as in the previous case. For downward modalities, let  $t' = \mathcal{T} \langle \rho a' \rangle (\bullet)$  and compute  $\varphi' \notin t' \implies (T, F)$  which is known to hold by consistency. Edges are then added to  $(\rho a', \varphi_T)$  and to  $(\rho a', \neg\varphi_F)$  as before. For upward modalities, as  $\langle a' \rangle \top$  holds in  $t$ , one must have  $a' = \bar{a}$  and let  $t' = \mathcal{T} \langle \rho' \rangle (\bullet)$ .  $\varphi' \notin t' \implies (T, F)$  is computed and edges are added to  $(\rho', \varphi_T)$  and to  $(\rho', \neg\varphi_F)$  as before.

**Lemma 6.4.2** *The dependency graph of a consistent tree of types of a cycle-free formula is cycle free.*

*Proof outline:* The proof proceeds by induction on the depth of the cycle, relying on the fact that the dependency graph is consistent with the tree structure (i.e. if a 1 edge reaches a node, no 2 edge may leave this node). The induction case is trivial: if there is a cycle of depth  $n$ , there must be a cycle of depth  $n - 1$ , a contradiction.

The base case is for a cycle of depth 1. One case is described, where the cycle is  $(\rho, \langle 1 \rangle \varphi) \xrightarrow{1} (\rho 1, \langle \bar{1} \rangle \psi) \xrightarrow{\bar{1}} (\rho, \langle 1 \rangle \varphi)$ . As  $\varphi$  must be a subformula of  $\psi$  and  $\psi$  a subformula of  $\varphi$ , they are both recursive formula. An analysis of the shape of  $\varphi$ , based on the derivations  $\varphi \in t \implies (T, F)$  and  $\psi \in t' \implies (T', F')$  with  $\langle 1 \rangle \psi \in T$  and  $\langle \bar{1} \rangle \varphi \in T'$  then shows that  $\varphi$  is not a cycle-free formula, a contradiction.  $\square$

**Lemma 6.4.3 (Soundness)** *Let  $T$  be the result set of the algorithm. For any type  $t \in T$  and any  $\varphi$  such that  $\varphi \in t$ , then  $\llbracket \varphi \rrbracket_{\emptyset}^0 \neq \emptyset$ .*

*Proof outline:*

The proof proceeds by induction on the number of steps of the algorithm. For every  $t$  in  $T^n$  and every witness tree  $\mathcal{T}$  rooted at  $t$  built from  $X^n$ , one can show that  $\mathcal{T}$  is a consistent tree type and one can build a focused tree  $f$  that is rooted (i.e. of the shape  $(\sigma^\circ[t], (\epsilon, Top, t'))$ ). The tree  $f$  is in the partial interpretation of  $\varphi_c(t)$ :  $f \langle \rho \rangle \in \llbracket \varphi_c(\mathcal{T} \langle \rho \rangle (\bullet)) \rrbracket_{\emptyset}^{depth(\rho)}$  for any path  $\rho$  whose depth is 0 or more, and  $f$  contains the context marker only if  $\textcircled{S}$  occurs in  $\mathcal{T}$ . Then one shows that for all  $\varphi \in t$ ,  $f \in \llbracket \varphi \rrbracket_{\emptyset}^0$  holds.

The base case is trivial by the shape of  $t$ : it may only contain backward modalities (trivially satisfied at level 0), one atomic proposition, and one context proposition. Moreover there is only one tree of witnesses to consider, the tree whose only node is  $t$ . If the atomic proposition is  $\sigma$ , then the focused tree returned is either  $(\sigma^{\textcircled{S}}[\epsilon], (\epsilon, Top, \epsilon))$  or  $(\sigma[\epsilon], (\epsilon, Top, \epsilon))$  depending on the context proposition.

In the inductive case, every witness types for both downward modalities,  $t_1$  and  $t_2$  are considered. For each of them, every tree type  $\mathcal{T}_1$  and  $\mathcal{T}_2$  are considered and a tree type rooted at  $t$  is built which is consistent by definition of the algorithm. By induction,  $f_1$  and  $f_2$  such that  $f_1 \langle \rho \rangle \in \llbracket \varphi_c(\mathcal{T} \langle 1\rho \rangle (\bullet)) \rrbracket_{\emptyset}^{depth(\rho)}$  and  $f_2 \langle \rho \rangle \in \llbracket \varphi_c(\mathcal{T} \langle 2\rho \rangle (\bullet)) \rrbracket_{\emptyset}^{depth(\rho)}$  for any path  $\rho$  whose depth is 0 or more. If either  $\mathcal{T}_1$  or  $\mathcal{T}_2$  contains  $\textcircled{S}$ , then  $f_1$  or  $f_2$  contains the context marker by

induction. Moreover, by definition of the algorithm, it is the case for only one of them and  $\textcircled{S}$  is not in  $t$ .

Let  $f_1$  be  $(\sigma_1^\circ[tl_1], (\epsilon, Top, tr_1))$  and  $f_2$  be  $(\sigma_2^\circ[tl_2], (\epsilon, Top, tr_2))$ . Let  $f = (\sigma(t)^\circ[\sigma_1^\circ[tl_1] :: tr_1], (\epsilon, Top, \sigma_2^\circ[tl_2] :: tr_2))$  where  $\sigma(t)^\circ$  is  $\sigma(t)^{\textcircled{S}}$  if  $\textcircled{S} \in t$ , and  $\sigma(t)$  otherwise. Note that  $f$  contains exactly one context marker iff  $\textcircled{S} \in \mathcal{T}$ .

Next, one shows that  $f_1 \langle \rho \rangle \in \llbracket \varphi_c(\mathcal{T} \langle 1\rho \rangle (\bullet)) \rrbracket_\emptyset^{\text{depth}(\rho)}$  implies  $f \langle 1\rho \rangle \in \llbracket \varphi_c(\mathcal{T} \langle 1\rho \rangle (\bullet)) \rrbracket_\emptyset^{\text{depth}(\rho)}$ , and the same for the other modality, by induction on the depth of the path, remarking that every backward modality at level 0 is trivially satisfied.

Then one proceeds to show that  $f$  satisfies  $\varphi_c(t)$  at level 0. To do so, a further induction on the dependency tree is needed. Let  $\rho$  be a path of the dependency tree and  $\psi$  be a formula at that path in the dependency tree, one shows that  $f \langle \rho \rangle \in \llbracket \psi \rrbracket_V^{\text{depth}(\rho)}$ . To do so, one relies on  $f \langle \rho \rangle \in \llbracket \psi \rrbracket_V^{\text{depth}(\rho)-1}$  if  $\text{depth}(\rho) \neq 0$ . In the base case at depth 0, the result is by construction as the formula is either a backward modality or an atomic formula. In the base case at another depth, the case is immediate by induction as the formula has to be an atomic formula whose interpretation does not depend on the depth. In the induction case, one concludes by the inductive hypothesis and by definition of partial satisfiability.

The proof is concluded by noticing that the final selected type has no backward modality, hence  $\llbracket \varphi_c(t) \rrbracket_\emptyset^0 = \llbracket \varphi_c(t) \rrbracket_\emptyset$ .  $\square$

**Lemma 6.4.4 (Completeness)** *For a cycle-free closed formula  $\varphi \in \mathcal{L}_\mu$ , if  $\llbracket \varphi \rrbracket_\emptyset \neq \emptyset$  then the algorithm terminates with a set of triples  $X$  such that  $\text{FinalCheck}(\varphi, X)$ .*

*Proof outline:* Let  $f \in \llbracket \varphi \rrbracket_\emptyset$  be a smallest focused tree validating the formula such that the names occurring in  $f$  are either also occurring in  $\varphi$  or are a single other name  $\sigma_x$ . By Lemma 5.3.2, there is a finite unfolding of  $\varphi$  such that  $f$  belongs to its interpretation. Hence there is a finite satisfiability derivation, defined in Figure 6.5, of  $f \Vdash_\epsilon \varphi$ .

In the satisfiability derivation, paths are assumed to be normalized ( $1\bar{1} = \epsilon$ ). Hence every path is a concatenation of a (possibly empty) backward path  $\rho_b$  followed by a forward path  $\rho_f$ .

This derivation has the following property, immediate by induction: let  $f$  the initial focused tree, then  $f' \Vdash_\rho \varphi$  implies  $f' = f \langle \rho \rangle$ . Hence if  $f_1 \Vdash_\rho \varphi_1$  and  $f_2 \Vdash_\rho \varphi_2$ , then  $f_1 = f_2$ .

Next, one uses the satisfiability derivation to construct a run of the algorithm that concludes that  $\varphi$  is satisfiable. One first associates each path to a type, which one then saturates (adding formulas that are true even though the satisfiability relation does not mention them at that path). One next shows that every formula at a path in the satisfiability relation is implied by the type at that path, and that types are consistent according to the  $\Delta_a(t, t')$  relation. One then concludes that the types are created by a run of the algorithm by induction on the paths.

More precisely, let first describe how  $t_\rho$  is built. Let  $\Phi_\rho$  the set of formulas at path  $\rho$ . One first adds every formula of  $\Phi_\rho$  that is in  $\text{Lean}(\varphi)$ , then one completes this set to yield a correct type: if  $\langle a \rangle \psi \in \Phi_\rho$  then one adds  $\langle a \rangle \top$ ;

$$\begin{array}{c}
 \frac{}{f \Vdash_{\rho} \top} \qquad \frac{\text{nm}(f) = \sigma}{f \Vdash_{\rho} \sigma} \qquad \frac{\text{nm}(f) \neq \sigma}{f \Vdash_{\rho} \neg \sigma} \qquad \frac{}{(\sigma^{\textcircled{S}}[tl], c) \Vdash_{\rho} \textcircled{S}} \\
 \\
 \frac{}{(\sigma[tl], c) \Vdash_{\rho} \neg \textcircled{S}} \qquad \frac{f \Vdash_{\rho} \varphi}{f \Vdash_{\rho} \varphi \vee \psi} \qquad \frac{f \Vdash_{\rho} \psi}{f \Vdash_{\rho} \varphi \vee \psi} \qquad \frac{f \Vdash_{\rho} \varphi \quad f \Vdash_{\rho} \psi}{f \Vdash_{\rho} \varphi \wedge \psi} \\
 \\
 \frac{f \langle 1 \rangle \Vdash_{\rho 1} \varphi}{f \Vdash_{\rho} \langle 1 \rangle \varphi} \qquad \frac{f \langle 2 \rangle \Vdash_{\rho 2} \varphi}{f \Vdash_{\rho} \langle 2 \rangle \varphi} \qquad \frac{f \langle \bar{1} \rangle \Vdash_{\rho \bar{1}} \varphi}{f \Vdash_{\rho} \langle \bar{1} \rangle \varphi} \qquad \frac{f \langle \bar{2} \rangle \Vdash_{\rho \bar{2}} \varphi}{f \Vdash_{\rho} \langle \bar{2} \rangle \varphi} \\
 \\
 \frac{f \langle a \rangle \text{ undefined}}{f \Vdash_{\rho} \neg \langle a \rangle \top} \qquad \frac{f \Vdash_{\rho} \exp(\mu \bar{X}_i . \varphi_i \text{ in } \psi)}{f \Vdash_{\rho} \mu \bar{X}_i . \varphi_i \text{ in } \psi}
 \end{array}$$

Figure 6.5: Satisfiability Relation

for every modality  $a$  for which  $f \langle a \rangle$  is defined one adds  $\langle a \rangle \top$ ; if there is no atomic proposition in  $\Phi_{\rho}$  then one adds  $\text{nm}(f \langle \rho \rangle)$ ; finally if  $f \langle \rho \rangle$  has the context marker one adds  $\textcircled{S}$ .

One next saturates the types. For every path  $t_{\rho}$  if  $t_{\rho a}$  exists, if  $\langle a \rangle \psi \in \text{Lean}(\varphi)$ , and if  $\psi \in t_{\rho a}$  then one adds  $\langle a \rangle \psi$  to  $t_{\rho}$ . This procedure is repeated until it does not change any type. Termination is a consequence of the finite size of the lean and of the number of paths. The resulting types are satisfiable as they are before saturation (since a focused tree satisfies them) and each formula added during saturation is first checked to be implied by the type.

One next shows (\*): for any given path  $\rho$ , if  $\varphi_{\rho} \in \Phi_{\rho}$  then  $\varphi_{\rho} \in t_{\rho}$ , by induction on the satisfiability derivation. Base cases with no negation are immediate by definition of  $t_{\rho}$  as these are formulas of the lean. For base cases with negation, one relies on the fact that  $f \langle \rho \rangle$  satisfies the formula, hence one cannot for instance have  $\sigma$  and  $\neg \sigma$  in  $\Phi_{\rho}$ . If  $\neg \langle a \rangle \top \in \Phi_{\rho}$  then one cannot also have  $\langle a \rangle \psi \in \Phi_{\rho}$  as  $\rho a$  is not a valid path, hence  $\langle a \rangle \top$  is not in  $t_{\rho}$  thus  $\neg \langle a \rangle \top \in t_{\rho}$ . The inductive cases of this induction (disjunction, conjunction, recursion) are immediate as they correspond to the definition of  $\cdot \in \cdot$ .

One next shows that for every type  $t_{\rho}$  and  $t_{\rho a}$  where  $a$  is a forward modality,  $\langle \bar{a} \rangle \top \in t_{\rho a}$  and  $\Delta_a(t_{\rho}, t_{\rho a})$  hold. (Note that, by path normalization, the types considered may be  $t_{\bar{1}\bar{2}}$  and  $t_{\bar{1}}$  for modality 2.) The first condition is immediate by construction of  $t_{\rho a}$  as  $f \langle \rho a \rangle$  is defined. For the second condition, let  $\langle a \rangle \psi \in t_{\rho}$ . If  $\langle a \rangle \psi \in \Phi_{\rho}$ , then it occurs in the satisfiability derivation with an hypothesis  $f_{\rho a} \Vdash_{\rho a} \psi$ . In this case  $\psi \in t_{\rho a}$  holds by (\*). If  $\langle a \rangle \psi \notin \Phi_{\rho}$  then it was added during saturation and the result is immediate by construction. Conversely, if  $\psi \in t_{\rho a}$  then by saturation  $\langle a \rangle \psi \in t_{\rho}$ . The case  $\langle \bar{a} \rangle \psi \in t_{\rho a}$  is now considered. The proof goes exactly as before, distinguishing the case where the formula is in  $\Phi_{\rho a}$  and the case where it was added by saturation.

One now shows that there is a run of the algorithm that produces these types. The proof proceeds by induction on the paths in the downward direction: if  $t_{\rho a}$  has been proven for a partial run for  $a \in \{1, 2\}$ , then  $t_{\rho}$  is proven for the next step of the algorithm. Moreover, one shows that  $(t_{\rho}, \{t_{\rho 1}\}, \{t_{\rho 2}\})$  is marked iff a forward subtree of  $f \langle \rho \rangle$  contains the context mark. The base case

is for paths with no descendants, hence no witness is required. The algorithm then adds  $(t_\rho, \emptyset, \emptyset)$  to its set of types, with a mark iff  $\textcircled{S} \in t_\rho$ , iff  $f \langle \rho \rangle$  is marked.

The inductive case is now considered. By induction, a partial run of the algorithm returns  $t_{\rho_1}$  and/or  $t_{\rho_2}$ . One first shows that  $t_\rho$  is returned in the next step of the algorithm, taking these two types as witnesses. One first remarks that if either witness is marked then the other is not and the mark is not at  $f \langle \rho \rangle$ , since there is only one context mark in  $f$ , and if the mark is at  $f \langle \rho \rangle$ , then neither witness is marked. For each child  $a \in \{1, 2\}$ ,  $\Delta_a(t_\rho, t_{\rho_a})$  and  $\langle \bar{a} \rangle \top \in t_{\rho_a}$ , hence the triple  $(t_\rho, W_1, W_2)$  with  $t_{\rho_1} \in W_1$  and  $t_{\rho_2} \in W_2$  is added by the algorithm.

One may now conclude. At the end of the induction, the last path considered,  $\rho_0$ , has no predecessor, hence it is the longest backward only path. Since  $f \langle \rho_0 \rangle$  is the root of the tree,  $\langle \bar{1} \rangle \top \notin t_{\rho_0}$  and  $\langle \bar{2} \rangle \top \notin t_{\rho_0}$ . Moreover, as the context mark is somewhere in  $f$ , it is in a forward subtree of  $f \langle \rho_0 \rangle$ , hence the final type is marked. Finally,  $t_\epsilon$  is in the witness tree of the final type, and since  $f \Vdash_\epsilon \varphi$ ,  $\varphi \in t_\epsilon$ .  $\square$

**Lemma 6.4.5 (Complexity)** *For a formula  $\psi \in \mathcal{L}_\mu$  the satisfiability problem  $\llbracket \psi \rrbracket_\emptyset \neq \emptyset$  is decidable in time  $2^{O(n)}$  where  $n = |\text{Lean}(\psi)|$ .*

*Proof outline:*  $|\text{Typ}(\psi)|$  is bounded by  $|2^{\text{Lean}(\psi)}|$  which is  $2^{O(n)}$ . During each iteration, the algorithm adds at least one new type (otherwise it terminates), thus it performs at most  $2^{O(n)}$  iterations. What is done at each iteration is now detailed. For each type that may be added (there are  $2^{O(n)}$  of them), there are two traversals of the set of types at the previous step to collect witnesses. Hence there are  $2 * 2^{O(n)} * 2^{O(n)} = 2^{O(n)}$  witness tests at each iteration. Each witness test involves a membership test and a  $\Delta_a$  test. In the implementation these are precomputed: for every formula  $\langle a \rangle \varphi$  in the lean, the subsets  $(T, F)$  of the lean that must be true and false respectively for  $\varphi$  to be true are precomputed, so testing  $\varphi \in t$  are simple inclusion and disjunction tests. The `FinalCheck` condition test at most  $2^{O(n)}$   $\psi$ -types and each test takes at most  $2^{O(n)}$  (testing the formulas containing  $\textcircled{S}$  against  $\psi$ ). Therefore, the worst case global time complexity of the algorithm does not exceed  $2^{O(n)}$ .  $\square$

## 6.5 Implementation Techniques

This section describes the main techniques used in the complete implementation [Genevès, 2006] of the  $\mathcal{L}_\mu$  decision procedure.

### 6.5.1 Implicit Representation of Sets of $\psi$ -Types

The implementation relies on a symbolic representation and manipulation of sets of types using Binary Decision Diagrams (BDDs) [Bryant, 1986]. BDDs provide a canonical representation of boolean functions. Experience has shown that this representation is very compact for very large boolean functions. Their effectiveness is notably well known in the area of formal verification of systems [Edmund M. Clarke et al., 1999].

First, one may observe that the implementation can avoid keeping track of every possible witnesses of each  $\psi$ -type. In fact, for a formula  $\varphi$ , one can test  $\llbracket \varphi \rrbracket_\emptyset \neq \emptyset$  by testing the satisfiability of the (linear-size) “plunging” formula

$\psi = \mu X. \varphi \vee \langle 1 \rangle X \vee \langle 2 \rangle X$  at the root of focused trees. That is, checking  $\llbracket \psi \rrbracket_0^0 \neq \emptyset$  while ensuring there is no unfulfilled upward eventuality at top level 0. One advantage of proceeding this way is that the implementation only need to deal with a current set of  $\psi$ -types at each step.

A bit-vector representation of  $\psi$ -types is now introduced. Types are complete in the sense that either a subformula or its negation must belong to a type. It is thus possible for a formula  $\varphi \in \text{Lean}(\psi)$  to be represented using a single BDD variable. For  $\text{Lean}(\psi) = \{\varphi_1, \dots, \varphi_m\}$ , a subset  $t \subseteq \text{Lean}(\psi)$  is represented by a vector  $\vec{t} = \langle t_1, \dots, t_m \rangle \in \{0, 1\}^m$  such that  $\varphi_i \in t$  iff  $t_i = 1$ . A BDD with  $m$  variables is then used to represent a set of such bit vectors.

For a program  $a \in \{1, 2\}$ , some auxiliary predicates on a vector  $\vec{t}$  are defined:

- $\text{isparent}_a(\vec{t})$  is read “ $\vec{t}$  is a parent for program  $a$ ” and is true iff the bit for  $\langle a \rangle \top$  is true in  $\vec{t}$
- $\text{ischild}_a(\vec{t})$  is read “ $\vec{t}$  is a child for program  $a$ ” and is true iff the bit for  $\langle \bar{a} \rangle \top$  is true in  $\vec{t}$

For a set  $T \subseteq 2^{\text{Lean}(\psi)}$ , its corresponding characteristic function is denoted  $\chi_T$ . Encoding  $\chi_{\text{Typ}(\psi)}$  is straightforward with the previous definitions.

The equivalent of  $\dot{\epsilon}$  is defined on the bit vector representation:

$$\text{status}_\varphi(\vec{t}) \stackrel{\text{def}}{=} \begin{cases} t_i & \text{if } \varphi \in \text{Lean}(\psi) \\ \text{status}_{\varphi'}(\vec{t}) \wedge \text{status}_{\varphi''}(\vec{t}) & \text{if } \varphi = \varphi' \wedge \varphi'' \\ \text{status}_{\varphi'}(\vec{t}) \vee \text{status}_{\varphi''}(\vec{t}) & \text{if } \varphi = \varphi' \vee \varphi'' \\ \neg \text{status}_{\varphi'}(\vec{t}) & \text{if } \varphi = \neg \varphi' \\ \text{status}_{\text{exp}(\varphi)}(\vec{t}) & \text{if } \varphi = \mu \bar{X}_i. \varphi_i \text{ in } \psi \end{cases}$$

$a \rightarrow b$  and  $a \leftrightarrow b$  respectively denote the implication and equivalence of two boolean formulas  $a$  and  $b$  over vector bits. The BDD of the relation  $\Delta_a$  for  $a \in \{1, 2\}$  can now be constructed. This BDD relates all pairs  $(\vec{x}, \vec{y})$  that are consistent w.r.t the program  $a$ , i.e., such that  $\vec{y}$  supports all of  $\vec{x}$ 's  $\langle a \rangle \varphi$  formulas, and vice-versa  $\vec{x}$  supports all of  $\vec{y}$ 's  $\langle \bar{a} \rangle \varphi$  formulas:

$$\Delta_a(\vec{x}, \vec{y}) \stackrel{\text{def}}{=} \bigwedge_{1 \leq i \leq m} \begin{cases} x_i \leftrightarrow \text{status}_\varphi(\vec{y}) & \text{if } \varphi_i = \langle a \rangle \varphi \\ y_i \leftrightarrow \text{status}_\varphi(\vec{x}) & \text{if } \varphi_i = \langle \bar{a} \rangle \varphi \\ \top & \text{otherwise} \end{cases}$$

For  $a \in \{1, 2\}$ , the set of witnessed vectors is defined:

$$\chi_{\text{wit}_a(T)}(\vec{x}) \stackrel{\text{def}}{=} \text{isparent}_a(\vec{x}) \rightarrow \exists \vec{y} [ h(\vec{y}) \wedge \Delta_a(\vec{x}, \vec{y}) ]$$

where  $h(\vec{y}) = \chi_T(\vec{y}) \wedge \text{ischild}_a(\vec{y})$ .

Then, the BDD of the fixpoint computation is initially set to the false constant, and the main function  $\text{Upd}(\cdot)$  is implemented as:

$$\chi_{\text{Upd}(T)}(\vec{x}) \stackrel{\text{def}}{=} \chi_T(\vec{x}) \vee \left( \chi_{\text{Typ}(\psi)}(\vec{x}) \wedge \bigwedge_{a \in \{1, 2\}} \chi_{\text{wit}_a(T)}(\vec{x}) \right)$$

Finally, the solver can be implemented as iterations over the sets  $\chi_{\text{Upd}(T)}$  until a fixpoint is reached. The final satisfiability condition consists in checking

whether  $\psi$  is present in a  $\psi$ -type of this fixpoint with no unfulfilled upward eventuality:

$$\exists \vec{t} \left[ \chi_T(\vec{t}) \wedge \bigwedge_{a \in \{1,2\}} \neg \text{ischild}_a(\vec{t}) \wedge \text{status}_\psi(\vec{t}) \right]$$

### 6.5.2 Satisfying Model Reconstruction

The implementation keeps a copy of each intermediate set of types computed by the algorithm, so that whenever a formula is satisfiable, a minimal satisfying model can be extracted. The top-down (re)construction of a satisfying model starts from a root (a  $\psi$ -type for which the final satisfiability condition holds), and repeatedly attempts to find successors. In order to minimize model size, only required left and right branches are built. Furthermore, for minimizing the maximal depth of the model, left and right successors of a node are successively searched in the intermediate sets of types, in the order they were computed by the algorithm. For readability purposes, the extracted satisfying model can be enriched by annotating the context mark  $\textcircled{S}$  from which XPath evaluation started and a target node selected by the XPath expression. The annotated model is then provided to the user in XML unranked tree syntax.

### 6.5.3 Conjunctive Partitioning and Early Quantification

The BDD-based implementation involves computations of *relational products* of the form:

$$\exists \vec{y} [ h(\vec{y}) \wedge \Delta_a(\vec{x}, \vec{y}) ] \quad (6.1)$$

It is well-known that such a computation may be quite time and space consuming, because the BDD corresponding to the relation  $\Delta_a$  may be quite large.

One famous optimization technique consists in using *conjunctive partitioning* [Edmund M. Clarke et al., 1999] and *early quantification* [Pan et al., 2006]. The idea is to compute the relational product without ever building the full BDD of the relation  $\Delta_a$ . This is possible by taking advantage of the form of  $\Delta_a$  along with properties of existential quantification. By definition,  $\Delta_a$  is a conjunction of  $n$  equivalences relating  $\vec{x}$  and  $\vec{y}$  where  $n$  is the number of  $\langle b \rangle \varphi$  formulas in  $\text{Lean}(\psi)$  where  $\varphi \neq \top$  and  $b \in \{a, \bar{a}\}$ :

$$\Delta_a(\vec{x}, \vec{y}) = \bigwedge_{i=1}^n R_i(\vec{x}, \vec{y})$$

If a variable  $y_k$  does not occur in the clauses  $R_{i+1}, \dots, R_n$  then the relational product (6.1) can be rewritten as:

$$\exists_{y_1, \dots, y_{k-1}, y_{k+1}, \dots, y_m} \left[ \exists y_k \left[ h(\vec{y}) \wedge \bigwedge_{1 \leq j \leq i} R_j(\vec{x}, \vec{y}) \right] \wedge \bigwedge_{i+1 \leq l \leq n} R_l(\vec{x}, \vec{y}) \right]$$

This allows to apply existential quantification on intermediate BDDs and thus to compose smaller BDDs. Of course, there are many ways to compose the  $R_i(\vec{x}, \vec{y})$ . Let  $\rho$  be a permutation of  $\{0, \dots, n-1\}$  which determines the order in which the partitions  $R_i(\vec{x}, \vec{y})$  are combined. For each  $i$ , let  $D_i$  be the

set of variables  $y_k$  with  $k \in \{1, \dots, m\}$  that  $R_i(\vec{x}, \vec{y})$  depends on.  $E_i$  is defined as the set of variables contained in  $D_{\rho(i)}$  that are not contained in  $D_{\rho(j)}$  for any  $j$  larger than  $i$ :

$$E_i = D_{\rho(i)} \setminus \bigcup_{j=i+1}^{n-1} D_{\rho(j)}$$

The  $E_i$  are pairwise disjoint and their union contains all the variables. The relational product (6.1) can be computed by starting from:

$$h_1(\vec{x}, \vec{y}) = \bigoplus_{y_k \in E_0} [ h(\vec{y}) \wedge R_{\rho(0)}(\vec{x}, \vec{y}) ]$$

and successively computing  $h_{p+1}$  defined as follows:

$$h_{p+1}(\vec{x}, \vec{y}) = \begin{cases} \bigoplus_{y_k \in E_p} [ h_p(\vec{x}, \vec{y}) \wedge R_{\rho(p)}(\vec{x}, \vec{y}) ] & \text{if } E_p \neq \emptyset \\ h_p(\vec{x}, \vec{y}) \wedge R_{\rho(p)}(\vec{x}, \vec{y}) & \text{if } E_p = \emptyset \end{cases}$$

until reaching  $h_n$  which is the result of the relational product. The ordering  $\rho$  determines how early in the computation variables can be quantified out. This directly impact the sizes of BDDs constructed and therefore the global efficiency of the decision procedure. It is thus important to choose  $\rho$  carefully. The overall goal is to minimize the size of the largest BDD created during the elimination process. A heuristic taken from [Edmund M. Clarke et al., 1999] is used. It seems to provide a good approximation as in practice it yields the best observed performance. It defines the cost of eliminating a variable  $y_k$  as the sum of the sizes of all the  $D_i$  containing  $y_k$ :

$$\sum_{1 \leq i \leq n, y_k \in D_i} |D_i|$$

The ordering  $\rho$  on the relations  $R_i$  is then defined in such a way that variables can be eliminated in the order given by a greedy algorithm which repeatedly eliminates the variable of minimum cost.

#### 6.5.4 BDD Variable Ordering

The cost of BDD operations is very sensitive to variable ordering. Finding the optimal variable ordering is known to be NP-complete [Hojati et al., 1996]. However, several heuristics are known to perform relatively well in practice [Edmund M. Clarke et al., 1999]. Choosing a good initial order of  $\text{Lean}(\psi)$  formulas does significantly improve performance. To this end, preserving locality of the initial problem happens to be essential. Experience has shown that the variable order determined by the breadth-first traversal of the formula  $\psi$  to solve, which keeps sister subformulas in close proximity, yields better results in practice.

## 6.6 Typing Applications and Experimental Results

For XPath expressions  $e_1, \dots, e_n \in \mathcal{L}_{\text{XPath}}$ , the decision problems presented in Section 4.6 can be generalized in the presence of several XML type expressions  $T_1, \dots, T_n$  and formulated as follows:

- XPath containment:  $E^{\rightarrow} \llbracket e_1 \rrbracket_{(\otimes \wedge [T_1])} \wedge \neg E^{\rightarrow} \llbracket e_2 \rrbracket_{(\otimes \wedge [T_2])}$  (if the formula is unsatisfiable then all nodes selected by  $e_1$  under type constraint  $T_1$  are selected by  $e_2$  under type constraint  $T_2$ )
- XPath emptiness:  $E^{\rightarrow} \llbracket e_1 \rrbracket_{(\otimes \wedge [T_1])}$
- XPath overlap:  $E^{\rightarrow} \llbracket e_1 \rrbracket_{(\otimes \wedge [T_1])} \wedge E^{\rightarrow} \llbracket e_2 \rrbracket_{(\otimes \wedge [T_2])}$
- XPath coverage:  $E^{\rightarrow} \llbracket e_1 \rrbracket_{(\otimes \wedge [T_1])} \wedge \bigwedge_{2 \leq i \leq n} \neg E^{\rightarrow} \llbracket e_i \rrbracket_{(\otimes \wedge [T_i])}$

The advantage of generalizing all the previous problem formulations with distinct types  $T_1$  and  $T_2$  is particularly useful for applications where types evolve. For instance, it is common that a file format of some company (described by an XML schema for instance) evolves over time. In this case, transformations that operated on the old document type must be updated to operate on the new type. Analysing XPath queries of a transformation (written in XSLT for instance) under different type constraints (the old one and the new one) can be used for helping the programmer to identify and understand the consequences of the evolution of the document type.

The system can also be used to check basic subtyping:  $\llbracket T_1 \rrbracket \wedge \neg \llbracket T_2 \rrbracket$ . However, since XPath (and therefore reverse navigation) is not used in that case, algorithms specialized for this restricted case such as the ones proposed in [Hosoya and Pierce, 2003] or in [Tozawa and Hagiya, 2003] may perform better on practical instances.

Additionally, two decision problems are of special interest for XML static type checking:

- Static type checking of an annotated XPath query:  $E^{\rightarrow} \llbracket e_1 \rrbracket_{(\otimes \wedge [T_1])} \wedge \neg \llbracket T_2 \rrbracket$  (if the formula is unsatisfiable then all nodes selected by  $e_1$  under type constraint  $T_1$  are included in the type  $T_2$ .)
- XPath equivalence under type constraints, checked by  $E^{\rightarrow} \llbracket e_1 \rrbracket_{(\otimes \wedge [T_1])} \wedge \neg E^{\rightarrow} \llbracket e_2 \rrbracket_{(\otimes \wedge [T_2])}$  and  $\neg E^{\rightarrow} \llbracket e_1 \rrbracket_{(\otimes \wedge [T_1])} \wedge E^{\rightarrow} \llbracket e_2 \rrbracket_{(\otimes \wedge [T_2])}$  (This test can be used to check that the nodes selected after a modification of a type  $T_1$  by  $T_2$  and an XPath expression  $e_1$  by  $e_2$  are the same, typically when an input type changes and the corresponding XPath query has to change as well.)

### 6.6.1 Experimental Results

Extensive tests of the implementation [Genevès, 2006] have been carried out<sup>1</sup>. This section gathers a few of them. All times reported correspond to the actual running time (in milliseconds) of the  $\mathcal{L}_\mu$  satisfiability solver without the extra (negligible) time spent for parsing XPath and translating into  $\mathcal{L}_\mu$ .

First, an XPath benchmark [Franceschet, 2005] is used. Its goal is to cover XPath features by gathering a significant variety of XPath expressions met in real-world applications. In this first test series, types are not yet considered, and the focus is only given to the XPath containment problem, since its logical formulation (presented in Section 4.6) is the most complex (as it requires the

<sup>1</sup>Experiments have been conducted with a Java implementation running on a Pentium 4, 3 Ghz, with 512Mb of RAM with Windows XP.

```

q1 /site/regions/*/item
q2 /site/auctions/auction/annotation/description/parlist/listitem/text/keyword
q3 //keyword
q4 /descendant-or-self::listitem/descendant-or-self::keyword
q5 /site/regions/*/item[parent::namerica or parent::samerica]
q6 //keyword/ancestor::listitem
q7 //keyword/ancestor-or-self::mail
q8 /site/regions/namerica/item | /site/regions/samerica/item
q9 /site/people/person[address and (phone or homepage)]
    
```

Figure 6.6: Queries Taken from the XPathmark Benchmark.

logic to be closed under negation). This first test series consists in finding the relation holding for each pair of queries from the benchmark. This means checking the containment of each query of the benchmark against all the others.  $q_i \subseteq q_j$  denotes that the query  $q_i$  is contained in the query  $q_j$ . Comparisons of two queries  $q_i$  and  $q_j$  may yield to three different results:

1.  $q_i \subseteq q_j$  and  $q_j \subseteq q_i$ , the queries are semantically equivalent, which is denoted by  $q_i \equiv q_j$
2.  $q_i \subseteq q_j$  but  $q_j \not\subseteq q_i$ , denoted by  $q_i \subset q_j$  or alternatively by  $q_j \supset q_i$
3.  $q_i \not\subseteq q_j$  and  $q_j \not\subseteq q_i$ , queries are not related, denoted by  $q_i \not\sim q_j$

Queries are presented on Figure 6.6 (where “/” is used as a shorthand for “/descendant-or-self::\*”). Corresponding results together with running times of the decision procedure are summarized on Table 6.1. Obtained results show that all tests are solved in several milliseconds. These first results suggest that several XPath expressions used in real-world scenarios can be efficiently handled in practice.

As a second test series, several expressions found in research papers on the containment of XPath expressions are compared. Figure 6.7 presents the collected expressions. Figure 6.7 also shows the obtained results. The first containment instance of Figure 6.7 was first formulated in [Miklau and Suciu, 2004] as an example for which the proposed tree pattern homomorphism technique is incomplete. The third example was not solvable in acceptable time and space bounds using the technique based on WS2S presented in Chapter 3. For this instance, the  $\mathcal{L}_\mu$  technique is orders of magnitude faster, and yields acceptable memory footprints. These results suggest that the system is reasonably able to handle containment instances which are difficult or impossible to solve using other techniques.

Figure 6.8 presents the results of a third test series including examples with intersection, and axes such as “following” and “preceding”, which are not illustrated in the previous series.

In the fourth test series, several XPath expressions (shown on Figure 6.9) are used in the presence of two real-world XML types: the DTDs of the SMIL [Hoschka, 1998] and XHTML [Pemberton, 2000] W3C recommendations. Table 6.2 gives the size of each DTD by presenting the number of symbols used (alphabet size) and the number of grammar production rules (type variables) in the unranked and binary representations. Several decision problems and

Relation	Time (ms)		Relation	Time (ms)	
	$\subseteq$	$\supseteq$		$\subseteq$	$\supseteq$
$q_1 \not\sim q_2$	17	21	$q_3 \not\sim q_7$	13	11
$q_1 \not\sim q_3$	13	20	$q_3 \not\sim q_8$	16	4
$q_1 \not\sim q_4$	12	16	$q_3 \not\sim q_9$	13	16
$q_1 \supset q_5$	14	9	$q_4 \not\sim q_5$	22	14
$q_1 \not\sim q_6$	21	17	$q_4 \not\sim q_6$	5	12
$q_1 \not\sim q_7$	13	11	$q_4 \not\sim q_7$	22	11
$q_1 \supset q_8$	8	13	$q_4 \not\sim q_8$	13	17
$q_1 \not\sim q_9$	14	17	$q_4 \not\sim q_9$	15	17
$q_2 \subset q_3$	32	35	$q_5 \not\sim q_6$	10	10
$q_2 \subset q_4$	33	38	$q_5 \not\sim q_7$	13	8
$q_2 \not\sim q_5$	24	22	$q_5 \equiv q_8$	9	14
$q_2 \not\sim q_6$	21	38	$q_5 \not\sim q_9$	17	21
$q_2 \not\sim q_7$	30	31	$q_6 \not\sim q_7$	21	22
$q_2 \not\sim q_8$	22	23	$q_6 \not\sim q_8$	17	17
$q_2 \not\sim q_9$	35	37	$q_6 \not\sim q_9$	13	19
$q_3 \supset q_4$	14	23	$q_7 \not\sim q_8$	22	19
$q_3 \not\sim q_5$	7	9	$q_7 \not\sim q_9$	14	17
$q_3 \not\sim q_6$	5	8	$q_8 \not\sim q_9$	9	11

Table 6.1: Results for Comparisons of Benchmark Queries.

$e_1$  /a[./b[c/\*//d]/b[c//d]/b[c/d]]  
 $e_2$  /a[./b[c/\*//d]/b[c/d]]

$e_3$  a[b]/\*//d/\*//g  
 $e_4$  a[b]/(b | c)/d/(e|f)/g  
 $e_5$  (a[b]/b/d/e/g) | (a/b/d/f/g)

$e_6$  a/b/s//c/b/s/c//d  
 $e_7$  a//b/\*//c//\*/d

$e_8$  a[b/e][b/f][c]  
 $e_9$  a[b/e][b/f]

$e_{10}$  /descendant::editor[parent::journal]  
 $e_{11}$  /descendant-or-self::journal/editor

Relation	Time (ms)	
	$\subseteq$	$\supseteq$
$e_1 \subset e_2$	323	248
$e_3 \supset e_4$	18	25
$e_3 \supset e_5$	23	17
$e_4 \supset e_5$	24	25
$e_6 \subset e_7$	37	30
$e_8 \subset e_9$	8	9
$e_{10} \equiv e_{11}$	17	14

Figure 6.7: Results for Instances Found in Research Papers.

$e_{12}$	<code>a/b//c/following-sibling::d/e</code>	Relation	Time (ms)	
$e_{13}$	<code>a//d[preceding-sibling::c]/e</code>		$\subseteq$	$\supseteq$
$e_{14}$	<code>//a//b//c/following-sibling::d/e</code>	$e_{12} \subset e_{13}$	23	17
$e_{15}$	<code>//b[ancestor::a]//*[preceding-sibling::c]/e</code>	$e_{14} \subset e_{15}$	12	23
$e_{16}$	<code>/b[preceding::a]//following::c</code>	$e_{16} \subset e_{17}$	18	22
$e_{17}$	<code>/a/b//following::c</code>	$e_{18} \subset e_{19}$	17	15
$e_{18}$	<code>a/b[/c]/following::d/e</code>	$e_{20} \equiv e_{12}$	23	24
$e_{19}$	<code>a//d[preceding::c]/e</code>	$e_{21} \not\subset e_{22}$	15	19
$e_{20}$	<code>a/b//d[preceding-sibling::c]/e</code>	$e_{23} \subset e_{21}$	22	19
$e_{21}$	<code>a/c/following::d/e</code>	$e_{24} \not\subset e_{18}$	16	11
$e_{22}$	<code>a/d[preceding::c]/e</code>			
$e_{23}$	<code>a/b[/c]/following::d/e <math>\cap</math> a/d[preceding::c]/e</code>			
$e_{24}$	<code>a/c/following::d/e <math>\cap</math> a/d[preceding::c]/e</code>			

Figure 6.8: Results for Instances with Horizontal Navigation.

```

p5  switch/layout
p6  smil/head//layout
p7  smil/head//layout[ancestor::switch]
p8  *//switch[ancestor::head]/descendant::seq//audio[preceding-sibling::video]

p9  descendant::a[ancestor::a]
p10 /descendant::*
p11 html/(head | body)
p12 html/head/descendant::*
p13 html/body/descendant::*
p14 //img
p15 //img[not *]
    
```

Figure 6.9: Queries Used in the Presence of DTDs.

DTD	Labels	Tree Type Variables
SMIL 1.0 [Hoschka, 1998]	19	29 unranked, 11 binary
XHTML 1.0 [Pemberton, 2000]	77	104 unranked, 325 binary

Table 6.2: Types Used in Experiments.

their results are presented on Table 6.3. For example, the emptiness test for  $p_9$  shows that the official XHTML DTD does not syntactically prohibit the nesting of anchors. Obtained results suggest that deciding XPath problems remains practically feasible, especially for static analysis purposes where such operations are performed at compile-time.

An additional benefit of the technique is that it automatically outputs a satisfying XML document, which can easily be enriched with XPath context and target information. For instance, the solver trace for the emptiness test for  $p_8$  is given below:

```

Checking emptiness of
*//switch[ancestor::head]/descendant::seq//audio[preceding-sibling::video]
in the presence of 'smil.dtd'.
Parsing XPath [249 ms].
Compilation of XPath to Tree Logic Formulas [15 ms].
Input DTD read from 'sampleDTDs/smil.dtd'.
    
```

## 6. SATISFIABILITY-TESTING ALGORITHM

---

Question	Instance	DTD	Answer	Time (ms)
Non-Emptiness	$p_5$	SMIL	yes	56
Overlap	$p_5 \cap p_6 \neq \emptyset$	SMIL	no	75
Containment	$p_6 \subseteq p_7$	SMIL	no	81
Non-Emptiness	$p_8$	SMIL	yes	94
Non-Emptiness	$p_9$	XHTML	yes	2530
Coverage	$p_{10} \subseteq p_{11} \cup p_{12} \cup p_{13}$	XHTML	yes	2723
Containment	$p_{14} \subseteq p_{15}$	XHTML	yes	2937

Table 6.3: Results in the Presence of DTDs.

Start symbol is \$smil

Converted DTD into BTT [140 ms].

CFT: 29 type variables and 19 terminals.

BTT: 11 type variables and 17 terminals.

Translated BTT into Tree Logic [16 ms].

Computing Relevant Closure

Computed Relevant Closure [46 ms].

Computed Lean [0 ms].

The Lean has size 53. It contains 35 eventualities and 18 symbols.

Fixpoint Computation Initialized [31 ms].

Computing Fixpoint.....[94 ms].

Formula is satisfiable [171 ms].

A satisfying finite binary tree model was found [94 ms]:

smil(head(switch(seq(video(#, audio), layout), meta), #), #)

In XML syntax:

```
<smil context="true">
  <head>
    <switch>
      <seq>
        <video/>
        <audio target="true"/>
      </seq>
      <layout/>
    </switch>
    <meta/>
  </head>
</smil>
```

```
*/switch[ancestor::head]/descendant::seq//audio[preceding-sibling::video]
is satisfiable in presence of 'smil.dtd'
```

These experiments shed a first light on the cost of solving XML decision problems in practice, and suggest that the presented system is already able to handle realistic scenarios.

## 6.7 Outcome

The essence of the obtained results lives in a sub-logic of the alternation free modal  $\mu$ -calculus with converse, with some syntactic restrictions on formulas, and where models are finite trees. As detailed in Chapter 5, the syntactic restrictions allow to prove that formulas of the logic are cycle-free. The cycle-free property is used to prove that the least and greatest fixpoint operators collapse in a single fixpoint operator. This provides closure under negation, which is the key property for solving the containment (a logical implication). Deep connections between this logic and XML decision problems can then be revealed: XPath expressions and regular tree type formulas conform to the  $\mathcal{L}_\mu$  syntactic restrictions. Furthermore, XPath expressions and regular tree languages can surprisingly be efficiently embedded since they are linear in the size of the corresponding formulas in the logic.

A sound and complete algorithm for testing the satisfiability of the logic is described in this chapter. Its upper bound time complexity is  $2^{O(n)}$  w.r.t. the length  $n$  of the given formula. The combination of all these ingredients yields the main result: sound and complete decision procedures, with the same upper bound complexity, for XML decision problems involving regular tree types and XPath queries. The implementation appears efficient in practice. A benefit of the approach is that the system can be effectively used in static analyzers for programming languages manipulating both XPath expressions and XML type annotations (input and output).



# Conclusion

---

## 7.1 Summary of the Main Contributions

The main contribution of this thesis is a new logic for finite trees, derived from the  $\mu$ -calculus. The logic is expressive enough to capture regular tree types along with multi-directional navigation in finite trees. It is decidable in single exponential time (specifically in  $2^{O(n)}$  steps where  $n$  is the size of the input formula defined as its number of atomic propositions and eventualities). This improves the best known computational complexity for finite trees. As such, this logic offers a new compromise between expressivity and complexity, specifically interesting in the context of XML.

Another contribution of this thesis is to show how to linearly compile queries and regular tree types (including DTDs and XML Schemas) in the logic. The logic takes almost full XPath into account and supports the largest fragment that has been treated for static analysis. Another advantage is that the logic is a sublogic of an existing one (the  $\mu$ -calculus) thus facilitating known optimization techniques and warranting extensibility.

This solves the major decision problems needed in the static analysis of XML specifications. These problems involve containment, emptiness, equivalence, overlap, and coverage of XPath queries (in the presence or absence of regular tree types), static type-checking of an annotated XPath query, and XPath equivalence under type constraints.

Furthermore, implementation techniques that yield concrete design and effective algorithm implementation in practice are presented. The fully implemented system is already able to handle realistic scenarios.

## 7.2 Perspectives

There are a number of interesting and promising directions for further research that builds on the results and ideas developed in this dissertation.

### 7.2.1 Further Optimizations of the Logical Solver

The worst-case complexity upper bound for deciding  $\mathcal{L}_\mu$  cannot be less than exponential time (since it can be used to decide FTA containment, or alterna-

tively since it contains the CTL subsystem). Nevertheless, several techniques can be further developed for continuing to improve the performance of the implementation. One perspective is to use dynamic reordering of BDD variables whenever it can speed up the decision procedure. Another interesting direction of further research is to attempt to statically reduce Lean contents by exploiting peculiarities of particular problem instances such as locality.

### 7.2.2 Pushing the XPath Decidability Envelope Further

One perspective of this thesis consists in extending the considered XPath fragment in order to support restricted data value comparisons (in the manner of [Bojanczyk et al., 2006]). Another direction for extending the fragment consists in integrating related work on counting [Dal-Zilio et al., 2004, Seidl et al., 2004] to the logic.

### 7.2.3 Enhancing the Translation of Regular Tree Types

Another perspective consists in considering a modification of the translation of types such that it imposes the context of a type to also follow the regular tree language definition (stating for instance that the parent of a given node may only be some specific other nodes). This would allow a yet more precise and interesting reporting on type-checking instances.

### 7.2.4 Efficiently Supporting Attributes and Data Values

Most theoretical work on XML and XPath models XML documents by finite labeled ordered trees, where the labels are taken from a finite alphabet. Attributes and data values are usually ignored. This thesis makes the same abstractions, and thus still offers perspectives for supporting more XML features. There is a reason for each previous widespread abstractions.

The difficulty for supporting XML attributes arises from the fact that they are unordered [Bray et al., 2004] which forces to consider mixed ordered and unordered contents in the underlying data model. There are several directions that can be followed for supporting constraints over mixed content while avoiding blow-ups caused by a naive modeling of unordered data on top of an ordered data model. Shuffle automata introduced in the 1970's provide a way to deal with an interleave operator [Jedrzejowicz and Szepietowski, 2001]. The work found in [Dal-Zilio and Lugiez, 2006] introduces the Sheaves logic and a related new class of automata (sheaves automata) suited for ordered trees. The logic combines regularity and counting constraints, and provides an interleaving operator. The work found in [Murata and Hosoya, 2003] proposes an automata rewriting technique for handling attribute-element constraints, which has been implemented in a validator for RELAX NG. The approach presented in this dissertation can easily be extended for supporting unordered XML attributes in an alternative manner, by observing that the algorithm proposed in Chapter 6 is based on  $\psi$ -types. Since a  $\psi$ -type is simply a set of formulas, attributes could naturally be modeled by a new class of atomic propositions, with the same complexity.

The usual reason for ignoring data values comes from the fact that they quickly lead to languages whose static analysis is undecidable [Alon et al., 2003,

[Neven and Schwentick, 2003, Benedikt et al., 2005]. Nevertheless, there exists examples of decidable static reasoning tasks involving attribute values [Arenas et al., 2005, Buneman et al., 2003, Bojanczyk et al., 2006]. A perspective of this thesis is to extend the algorithm proposed in Chapter 6 to deal with attribute values. This could help at identifying more precisely the upper-bound complexity of decision problems involving XPath with limited data value comparison, which has been observed to be between NEXPTIME and 3-NEXPTIME in the recent work found in [Bojanczyk et al., 2006].

### 7.2.5 Query Optimization

Another perspective of this thesis is to take advantage of the static analysis of XPath expressions for optimization purposes. This allows for instance to automatically detect contradictions and eliminate redundancies from XML queries at compile time, as preliminary investigated in [Genevès and Vion-Dury, 2004]. One perspective is to extend this work with some trace-based semantics for XPath (in the manner of [Hartel, 2005]) in order to capture optimality of a query w.r.t a given evaluation context. Then, the optimal query could be calculated by using the automatic comparison of queries described in this thesis.

### 7.2.6 Query Evaluation via Model-Checking

The linear translation of XPath into the  $\mu$ -calculus opens perspectives for query evaluation. A direction of future work consists in revisiting XPath evaluation (reduced to model-checking) based on existing techniques [Mateescu, 2002, Mateescu and Sighireanu, 2003].

### 7.2.7 Application to the Static Analysis of Transformations

Last but not least, a perspective of this thesis is to apply the presented XPath static analysis techniques to the type-checking of XML transformation languages. Results presented in this dissertation open the way to the construction of debuggers, compilers, and type-checkers for XSLT and XQuery. For example, the recent work found in [Møller et al., 2005] could benefit from using the exact algorithm of Chapter 6 instead of their conservative approximation. The practical experiments reported in Chapter 6 strengthen the hope for an effective analysis of this kind in the near future.



# Bibliography

---

- [Abiteboul and Vianu, 1999] Abiteboul, S. and Vianu, V. (1999). Regular path queries with constraints. *Journal of Computer and System Sciences*, 58(3):428–452. 6, 131
- [Afanasiev et al., 2005] Afanasiev, L., Blackburn, P., Dimitriou, I., Gaiffe, B., Goris, E., Marx, M., and de Rijke, M. (2005). PDL for ordered trees. *Journal of Applied Non-Classical Logics*, 15(2):115–135. 32
- [Aiken and Murphy, 1991] Aiken, A. and Murphy, B. R. (1991). Implementing regular tree expressions. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 427–447. Springer-Verlag. 34
- [Alon et al., 2003] Alon, N., Milo, T., Neven, F., Suciu, D., and Vianu, V. (2003). XML with data values: typechecking revisited. *J. Comput. Syst. Sci.*, 66(4):688–727. 109
- [Arenas et al., 2005] Arenas, M., Fan, W., and Libkin, L. (2005). Consistency of XML specifications. In Bertossi, L. E., Hunter, A., and Schaub, T., editors, *Inconsistency Tolerance*, volume 3300 of *Lecture Notes in Computer Science*, pages 15–41. Springer. 109
- [Arnold and Niwinski, 1992] Arnold, A. and Niwinski, D. (1992). Fixed point characterization of weak monadic logic definable sets of trees. In *Tree Automata and Languages*, pages 159–188. North-Holland. 32
- [Audebaud and Rose, 2000] Audebaud, P. and Rose, K. H. (2000). Stylesheet validation. Technical Report 2000-37, Laboratoire de l’informatique du parallélisme, Ecole Normale Supérieure de Lyon. 33
- [Baader and Tobies, 2001] Baader, F. and Tobies, S. (2001). The inverse method implements the automata approach for modal satisfiability. In *IJ-CAR '01: Proceedings of the First International Joint Conference on Automated Reasoning*, pages 92–106. Springer-Verlag. 73
- [Barceló and Libkin, 2005] Barceló, P. and Libkin, L. (2005). Temporal logics over unranked trees. In *LICS '05: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, pages 31–40. 23, 32

- [Benedikt et al., 2005] Benedikt, M., Fan, W., and Geerts, F. (2005). XPath satisfiability in the presence of DTDs. In *PODS '05: Proceedings of the twenty-fourth ACM Symposium on Principles of Database Systems*, pages 25–36, Baltimore, Maryland. ACM Press. [6](#), [20](#), [21](#), [109](#)
- [Benedikt and Segoufin, 2005] Benedikt, M. and Segoufin, L. (2005). Regular tree languages definable in FO. In *STACS '05: Proceedings of the 22nd Annual Symposium on Theoretical Aspects of Computer Science*, volume 3404 of *LNCS*, pages 327–339, Stuttgart, Germany. Springer-Verlag. [25](#), [32](#)
- [Benzaken et al., 2003] Benzaken, V., Castagna, G., and Frisch, A. (2003). CDuce: An XML-centric general-purpose language. In *ICFP '03: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 51–63, Uppsala, Sweden. ACM Press. [34](#)
- [Berglund et al., 2006] Berglund, A., Boag, S., Chamberlin, D., Fernández, M. F., Kay, M., Robie, J., and Siméon, J. (2006). XML path language (XPath) 2.0, W3C candidate recommendation. <http://www.w3.org/TR/xpath20/>. [3](#), [127](#)
- [Biehl et al., 1997] Biehl, M., Klarlund, N., and Rauhe, T. (1997). Algorithms for guided tree automata. In *WIA '96: Revised Papers from the First International Workshop on Implementing Automata*, volume 1260 of *LNCS*, pages 6–25. Springer-Verlag. [51](#), [52](#)
- [Boag et al., 2006] Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J., and Siméon, J. (2006). XQuery 1.0: An XML query language, W3C candidate recommendation. <http://www.w3.org/TR/xquery/>. [1](#), [4](#), [36](#), [125](#), [128](#)
- [Bojanczyk et al., 2006] Bojanczyk, M., David, C., Muscholl, A., Schwentick, T., and Segoufin, L. (2006). Two-variable logic on data trees and XML reasoning. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 10–19, Chicago, IL, USA. ACM Press. [21](#), [23](#), [24](#), [25](#), [108](#), [109](#)
- [Boneva and Talbot, 2005] Boneva, I. and Talbot, J.-M. (2005). Expressiveness of a spatial logic for trees. In *LICS '05: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*, pages 280–289. IEEE Computer Society. [36](#)
- [Bradfield and Stirling, 2001] Bradfield, J. and Stirling, C. (2001). *Handbook of Process Algebra*, chapter 4 – Modal logics and mu-calculi: an introduction, pages 293–330. Elsevier. [32](#)
- [Bray et al., 2004] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. (2004). Extensible markup language (XML) 1.0 (third edition), W3C recommendation. <http://www.w3.org/TR/2004/REC-xml-20040204/>. [1](#), [6](#), [12](#), [14](#), [18](#), [108](#), [125](#), [131](#)
- [Bryant, 1986] Bryant, R. E. (1986). Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691. [31](#), [35](#), [50](#), [73](#), [96](#)

- 
- [Buneman et al., 2003] Buneman, P., Davidson, S. B., Fan, W., Hara, C. S., and Tan, W. C. (2003). Reasoning about keys for XML. *Inf. Syst.*, 28(8):1037–1063. [109](#)
- [Calcagno et al., 2003] Calcagno, C., Cardelli, L., and Gordon, A. D. (2003). Deciding validity in a spatial logic for trees. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 62–73, New Orleans, Louisiana, USA. ACM Press. [36](#)
- [Cardelli et al., 2002] Cardelli, L., Gardner, P., and Ghelli, G. (2002). A spatial logic for querying graphs. In *ICALP '02: Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, pages 597–610. Springer-Verlag. [36](#)
- [Cardelli and Ghelli, 2004] Cardelli, L. and Ghelli, G. (2004). TQL: A query language for semistructured data based on the ambient logic. *Mathematical Structures in Computer Science*, 14:285–327. [36](#)
- [Cardelli and Gordon, 2000] Cardelli, L. and Gordon, A. D. (2000). Mobile ambients. *Theoretical Computer Science*, 240:177–213. [36](#)
- [Cardelli and Gordon, 2006] Cardelli, L. and Gordon, A. D. (2006). Ambient logic. *Mathematical Structures in Computer Science. To appear.* [36](#)
- [Charatonik et al., 2003] Charatonik, W., Dal-Zilio, S., Gordon, A. D., Mukhopadhyay, S., and Talbot, J.-M. (2003). Model checking mobile ambients. *Theoretical Computer Science*, 308(1-3):277–331. [36](#)
- [Clark, 1999] Clark, J. (1999). XSL transformations (XSLT) version 1.0, W3C recommendation. <http://www.w3.org/TR/1999/REC-xslt-19991116>. [1](#), [4](#), [125](#), [128](#)
- [Clark and DeRose, 1999] Clark, J. and DeRose, S. (1999). XML path language (XPath) version 1.0, W3C recommendation. <http://www.w3.org/TR/1999/REC-xpath-19991116>. [1](#), [3](#), [4](#), [21](#), [125](#), [127](#), [128](#)
- [Clark and Murata, 2001] Clark, J. and Murata, M. (2001). RELAX NG specification, OASIS committee specification. <http://relaxng.org/spec-20011203.html>. [12](#), [18](#)
- [Clarke and Emerson, 1981] Clarke, E. M. and Emerson, E. A. (1981). Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, volume 131 of *LNCS*, pages 52–71. Springer-Verlag. [32](#)
- [Colazzo et al., 2004] Colazzo, D., Ghelli, G., Manghi, P., and Sartiani, C. (2004). Types for path correctness of XML queries. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 126–137, Snow Bird, UT, USA. ACM Press. [34](#)
- [Colazzo et al., 2006] Colazzo, D., Ghelli, G., Manghi, P., and Sartiani, C. (2006). Static analysis for path correctness of XML queries. *Journal of Functional Programming. To appear.* [36](#)

- [Comon et al., 1997] Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. (1997). Tree automata techniques and applications. Available on: <http://www.grappa.univ-lille3.fr/tata>. release October, 1st 2002. **15, 18, 19, 20, 30, 31, 51**
- [Conforti et al., 2002] Conforti, G., Ghelli, G., Albano, A., Colazzo, D., Manghi, P., and Sartiani, C. (2002). The query language tql. In *WebDB'02: Proceedings of the 5th International Workshop on Web and Databases*, pages 13–18. **37**
- [Dal-Zilio and Lugiez, 2003] Dal-Zilio, S. and Lugiez, D. (2003). XML schema, tree logic and sheaves automata. In Nieuwenhuis, R., editor, *RTA'03: Proceedings of the 14th International Conference on Rewriting Techniques and Applications*, volume 2706 of *Lecture Notes in Computer Science*, pages 246–263, Valencia, Spain. Springer. **36**
- [Dal-Zilio and Lugiez, 2006] Dal-Zilio, S. and Lugiez, D. (2006). Xml schema, tree logic and sheaves automata. *Appl. Algebra Eng., Commun. Comput.*, 17(5):337–377. **37, 108**
- [Dal-Zilio et al., 2004] Dal-Zilio, S., Lugiez, D., and Meyssonier, C. (2004). A logic you can count on. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 135–146, Venice, Italy. ACM Press. **21, 36, 108**
- [Dawar et al., 2004] Dawar, A., Gardner, P., and Ghelli, G. (2004). Expressiveness and complexity of graph logic. Technical report, Imperial College. **36**
- [DeRose et al., 2002] DeRose, S., Jr., R. D., Grosso, P., Maler, E., Marsh, J., and Walsh, N. (2002). XML pointer language (XPointer), W3C working draft. <http://www.w3.org/TR/xptr/>. **4, 128**
- [DeRose et al., 2001] DeRose, S., Maler, E., and Orchard, D. (2001). XML linking language (XLink) version 1.0, W3C recommendation. <http://www.w3.org/TR/xlink/>. **4, 128**
- [Deutsch and Tannen, 2001] Deutsch, A. and Tannen, V. (2001). Containment of regular path expressions under integrity constraints. In *KRDB '01: Proceedings of the 8th International Workshop on Knowledge Representation meets Databases*, volume 45 of *CEUR Workshop Proceedings*, pages 1–11, Rome, Italy. CEUR. **20**
- [Doner, 1970] Doner, J. (1970). Tree acceptors and some of their applications. *Journal of Computer and System Sciences*, 4:406–451. **25, 27, 29**
- [Dong and Bailey, 2004] Dong, C. and Bailey, J. (2004). Static analysis of XSLT programs. In Schewe, K.-D. and Williams, H. E., editors, *ADC'04: Proceedings of the Fifteenth Australasian Database Conference*, volume 27 of *CRPIT*, pages 151–160, Dunedin, New Zealand. Australian Computer Society. **35**
- [Ebbinghaus and Flum, 2005] Ebbinghaus, H. and Flum, J. (2005). *Finite Model Theory*. Springer Monographs in Mathematics. Springer. **23**

- 
- [Edmund M. Clarke et al., 1999] Edmund M. Clarke, J., Grumberg, O., and Peled, D. A. (1999). *Model checking*. MIT Press. **73, 96, 98, 99**
- [Elgaard et al., 2000] Elgaard, J., Møller, A., and Schwartzbach, M. I. (2000). Compile-time debugging of C programs working on trees. In *ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems*, volume 1782 of *LNCS*, pages 119–134. Springer-Verlag. **52**
- [Emerson and Jutla, 1991] Emerson, E. A. and Jutla, C. S. (1991). Tree automata,  $\mu$ -calculus and determinacy. In *Proceedings of the 32nd annual Symposium on Foundations of Computer Science*, pages 368–377, San Juan, Puerto Rico. IEEE Computer Society Press. **32**
- [Fallside and Walmsley, 2004] Fallside, D. C. and Walmsley, P. (2004). XML Schema part 0: Primer second edition, W3C recommendation. <http://www.w3.org/TR/xmlschema-0/>. **1, 4, 6, 12, 18, 125, 128, 131**
- [Fan et al., 2004] Fan, W., Chan, C.-Y., and Garofalakis, M. (2004). Secure XML querying with security views. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 587–598, Paris, France. ACM Press. **5, 6, 128, 131**
- [Fischer and Ladner, 1979] Fischer, M. J. and Ladner, R. E. (1979). Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211. **32, 72, 73**
- [Franceschet, 2005] Franceschet, M. (2005). XPathMark - an XPath benchmark for XMark generated data. In *XSYM '05: Proceedings of The Third International Symposium on Database and XML Technologies*, volume 3671 of *LNCS*, pages 129–143, Trondheim, Norway. Springer-Verlag. **100**
- [Frisch, 2004] Frisch, A. (2004). *Théorie, conception et réalisation d'un langage adapté à XML*. PhD thesis, Université Paris 7 – Denis Diderot. **34**
- [Gapeyev and Pierce, 2003] Gapeyev, V. and Pierce, B. C. (2003). Regular object types. In *ECOOP'03: Proceedings of the European Conference on Object-Oriented Programming*, Darmstadt, Germany. A preliminary version was presented at FOOL'03. **34**
- [Gapeyev and Pierce, 2004] Gapeyev, V. and Pierce, B. C. (2004). Paths into patterns. Technical Report MS-CIS-04-25, University of Pennsylvania. **35**
- [Genevès, 2006] Genevès, P. (2006). A satisfiability solver for XML and XPath decision problems. <http://wam.inrialpes.fr/xml/>. **96, 100**
- [Genevès and Layaïda, 2006] Genevès, P. and Layaïda, N. (2006). A decision procedure for XPath containment. Research Report 5867, INRIA. **55**
- [Genevès and Vion-Dury, 2004] Genevès, P. and Vion-Dury, J.-Y. (2004). Logic-based XPath optimization. In *DocEng'04: Proceedings of the 2004 ACM Symposium on Document Engineering*, pages 211–219, Milwaukee, Wisconsin, USA. ACM Press. **6, 109, 131**

- [Genevès and Vion-Dury, 2004] Genevès, P. and Vion-Dury, J.-Y. (2004). XPath formal semantics and beyond: A Coq-based approach. In *TPHOLs '04: Emerging Trends Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics*, pages 181–198, Park City, Utah, United States. University Of Utah. [24](#), [49](#)
- [Gottlob et al., 2005] Gottlob, G., Koch, C., and Pichler, R. (2005). Efficient algorithms for processing XPath queries. *ACM Transactions on Database Systems*, 30(2):444–491. [24](#)
- [Grädel and Otto, 1999] Grädel, E. and Otto, M. (1999). On logics with two variables. *Theor. Comput. Sci.*, 224(1-2):73–113. [24](#)
- [Grädel et al., 2002] Grädel, E., Thomas, W., and Wilke, T., editors (2002). *Automata logics, and infinite games: a guide to current research*. Springer-Verlag. [32](#), [72](#), [123](#)
- [Grzegorzczak, 1953] Grzegorzczak, A. (1953). Some classes of recursive functions. *Rozprawy Matematyczne*, 4:1–45. [31](#)
- [Harren et al., 2005] Harren, M., Raghavachari, M., Shmueli, O., Burke, M. G., Bordawekar, R., Pechtchanski, I., and Sarkar, V. (2005). XJ: facilitating XML processing in Java. In Ellis, A. and Hagino, T., editors, *WWW'05: Proceedings of the 14th international conference on World Wide Web*, pages 278–287, Chiba, Japan. ACM. [35](#)
- [Hartel, 2005] Hartel, P. H. (2005). A trace semantics for positive core XPath. In *TIME'05: Proceedings of the 12th International Symposium on Temporal Representation and Reasoning*, pages 103–112, Burlington, Vermont, USA. IEEE Computer Society. [109](#)
- [Henglein and Mairson, 1991] Henglein, F. and Mairson, H. G. (1991). The complexity of type inference for higher-order lambda calculi. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 119–130, Orlando, Florida, United States. ACM Press. [41](#)
- [Hojati et al., 1996] Hojati, R., Krishnan, S. C., and Brayton, R. K. (1996). Early quantification and partitioned transition relations. In *ICCD '96: Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, pages 12–19. [73](#), [99](#)
- [Hopcroft et al., 2000] Hopcroft, J. E., Motwani, R., Rotwani, and Ullman, J. D. (2000). *Introduction to Automata Theory, Languages and Computability*. Addison-Wesley Longman Publishing Co., Inc. [14](#)
- [Hoschka, 1998] Hoschka, P. (1998). Synchronized multimedia integration language (SMIL) 1.0 specification, W3C recommendation. <http://www.w3.org/TR/REC-smil/>. [101](#), [103](#)
- [Hosoya et al., 2005a] Hosoya, H., Frisch, A., and Castagna, G. (2005a). Parametric polymorphism for XML. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 50–62, Long Beach, California, USA. ACM Press. [34](#)

- 
- [Hosoya and Pierce, 2001] Hosoya, H. and Pierce, B. (2001). Regular expression pattern matching for XML. In *POPL'01: Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 67–80, London, United Kingdom. ACM Press. [22](#)
- [Hosoya and Pierce, 2003] Hosoya, H. and Pierce, B. C. (2003). XDuce: A statically typed XML processing language. *ACM Trans. Inter. Tech.*, 3(2):117–148. [1](#), [5](#), [11](#), [34](#), [100](#), [125](#), [129](#)
- [Hosoya et al., 2005b] Hosoya, H., Vouillon, J., and Pierce, B. C. (2005b). Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27(1):46–90. [14](#), [15](#)
- [Huet et al., 2004] Huet, G., Kahn, G., and Paulin-Mohring, C. (2004). *The Coq Proof Assistant - A tutorial - Version 8.0*. INRIA. [49](#)
- [Huet, 1997] Huet, G. P. (1997). The zipper. *Journal of Functional Programming*, 7(5):549–554. [77](#)
- [Jedrzejowicz and Szepietowski, 2001] Jedrzejowicz, J. and Szepietowski, A. (2001). Shuffle languages are in p. *Theoretical Computer Science*, 250(1-2):31–53. [108](#)
- [Klarlund and Møller, 2001] Klarlund, N. and Møller, A. (2001). *MONA Version 1.4 User Manual*. BRICS Notes Series NS-01-1. [31](#), [42](#), [50](#)
- [Klarlund et al., 2001] Klarlund, N., Møller, A., and Schwartzbach, M. I. (2001). MONA implementation secrets. In *CIAA '00: Revised Papers from the 5th International Conference on Implementation and Application of Automata*, volume 2088 of *LNCS*, pages 182–194. Springer-Verlag. [31](#), [56](#)
- [Kozen, 1983] Kozen, D. (1983). Results on the propositional  $\mu$ -calculus. *Theoretical Computer Science*, 27:333–354. [32](#), [59](#), [61](#)
- [Kozen, 1988] Kozen, D. (1988). A finite model theorem for the propositional  $\mu$ -calculus. *Studia Logica*, 47(3):233–241. [62](#)
- [Kupferman and Vardi, 1999] Kupferman, O. and Vardi, M. (1999). The weakness of self-complementation. In *Proc. 16th Symp. on Theoretical Aspects of Computer Science*, volume 1563 of *LNCS*, pages 455–466. [32](#)
- [Levin and Pierce, 2005] Levin, M. Y. and Pierce, B. C. (2005). Type-based optimization for regular patterns. In *DBPL '05: Proceedings of the 10th International Symposium on Database Programming Languages*, volume 3774 of *LNCS*. Springer-Verlag. [6](#), [131](#)
- [Martens and Neven, 2003] Martens, W. and Neven, F. (2003). Typechecking top-down uniform unranked tree transducers. In Calvanese, D., Lenzerini, M., and Motwani, R., editors, *ICDT'03: In Proceedings of the 9th International Conference on Database Theory*, volume 2572 of *Lecture Notes in Computer Science*, pages 64–78, Siena, Italy. Springer. [34](#)

- [Marx, 2004a] Marx, M. (2004a). Conditional XPath, the first order complete XPath dialect. In *PODS '04: Proceedings of the twenty-third ACM Symposium on Principles of Database Systems*, pages 13–22, Paris, France. ACM Press. [24](#)
- [Marx, 2004b] Marx, M. (2004b). XPath with conditional axis relations. In *Proceedings of the 9th International Conference on Extending Database Technology*, volume 2992 of *LNCS*, pages 477–494. Springer-Verlag. [32](#)
- [Mateescu, 2002] Mateescu, R. (2002). Local model-checking of modal  $\mu$ -calculus on acyclic labeled transition systems. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 281–295. Springer-Verlag. [109](#)
- [Mateescu and Sighireanu, 2003] Mateescu, R. and Sighireanu, M. (2003). Efficient on-the-fly model-checking for regular alternation-free  $\mu$ -calculus. *Sci. Comput. Program.*, 46(3):255–281. [109](#)
- [Meyer, 1975] Meyer, A. (1975). Weak monadic second-order theory of successor is not elementary-recursive. In Parikh, R., editor, *Proceedings of Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 132–154. Springer-Verlag. [31](#), [123](#)
- [Miklau and Suciu, 2004] Miklau, G. and Suciu, D. (2004). Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51(1):2–45. [20](#), [32](#), [55](#), [101](#)
- [Milo et al., 2003] Milo, T., Suciu, D., and Vianu, V. (2003). Typechecking for xml transformers. *J. Comput. Syst. Sci.*, 66(1):66–97. [1](#), [33](#), [34](#), [125](#)
- [Møller et al., 2005] Møller, A., Olesen, M. O., and Schwartzbach, M. I. (2005). Static validation of XSL Transformations. Technical Report RS-05-32, BRICS. [6](#), [35](#), [55](#), [109](#), [131](#)
- [Møller and Schwartzbach, 2005] Møller, A. and Schwartzbach, M. I. (2005). The design space of type checkers for XML transformation languages. In *Proc. Tenth International Conference on Database Theory, ICDT '05*, volume 3363 of *LNCS*, pages 17–36. Springer-Verlag. [33](#), [35](#)
- [Mortimer, 1975] Mortimer, M. (1975). On languages with two variables. *Z. Math. Logik Grundlagen Math.*, 21:135–140. [24](#)
- [Murata, 1996] Murata, M. (1996). Transformation of documents and schemas by patterns and contextual conditions. In *PODP '96: Proceedings of the Third International Workshop on Principles of Document Processing*, pages 153–169. Springer-Verlag. [1](#), [125](#)
- [Murata, 1998] Murata, M. (1998). Data model for document transformation and assembly. In Munson, E. V., Nicholas, C. K., and Wood, D., editors, *PODDP'98: In Proceedings of the 4th International Workshop on Principles of Digital Document Processing*, volume 1481 of *Lecture Notes in Computer Science*, pages 140–152, Saint Malo, France. Springer. [18](#)

- 
- [Murata, 2001] Murata, M. (2001). Extended path expressions for XML. In *PODS '01: Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 126–137, Santa Barbara, California, United States. ACM Press. 77
- [Murata and Hosoya, 2003] Murata, M. and Hosoya, H. (2003). Validation algorithm for attribute-element constraints of RELAX NG. In *Proceedings of the Extreme Markup Languages 2003 Conference*, Montréal, Quebec, Canada. 108
- [Murata et al., 2005] Murata, M., Lee, D., Mani, M., and Kawaguchi, K. (2005). Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology*, 5(4):660–704. 5, 12, 18, 129
- [Neven, 2002a] Neven, F. (2002a). Automata, logic, and xml. In *CSL '02: Proceedings of the 16th International Workshop and 11th Annual Conference of the EACSL on Computer Science Logic*, pages 2–26. Springer-Verlag. 18, 23
- [Neven, 2002b] Neven, F. (2002b). Automata theory for XML researchers. *SIGMOD Record*, 31(3):39–46. 11
- [Neven and Schwentick, 2003] Neven, F. and Schwentick, T. (2003). XPath containment in the presence of disjunction, DTDs, and variables. In *ICDT '03: Proceedings of the 9th International Conference on Database Theory*, volume 2572 of *LNCS*, pages 315–329. Springer-Verlag. 20, 109
- [Nivat and Podelski, 1993] Nivat, M. and Podelski, A. (1993). Another variation on the common subexpression problem. *Discrete Math.*, 114(1-3):379–401. 77
- [Olteanu et al., 2002] Olteanu, D., Meuss, H., Furche, T., and Bry, F. (2002). XPath: Looking forward. In *EDBT '02: Proceedings of the Workshop on XML-Based Data Management*, volume 2490 of *LNCS*, pages 109–127. Springer-Verlag. 65
- [Pan et al., 2006] Pan, G., Sattler, U., and Vardi, M. Y. (2006). BDD-based decision procedures for the modal logic K. *Journal of Applied Non-classical Logics*, 16(1-2):169–208. 72, 73, 88, 98
- [Pemberton, 2000] Pemberton, S. (2000). XHTML 1.0 the extensible hypertext markup language (second edition), W3C recommendation. <http://www.w3.org/TR/xhtml1/>. 101, 103
- [Podelski, 1992] Podelski, A. (1992). A monoid approach to tree automata. In *Tree Automata and Languages*, pages 41–56. North-Holland. 77
- [Schwentick, 2004] Schwentick, T. (2004). XPath query containment. *SIGMOD Record*, 33(1):101–109. 20, 21
- [Seidl, 1990] Seidl, H. (1990). Deciding equivalence of finite tree automata. *SIAM J. Comput.*, 19(3):424–437. 20

- [Seidl et al., 2004] Seidl, H., Schwentick, T., Muscholl, A., and Habermehl, P. (2004). Counting in trees for free. In *ICALP'04 : In Proceedings of the 31st International Colloquium on Automata, Languages and Programming*, volume 3142 of *Lecture Notes in Computer Science*, pages 1136–1149. Springer. [108](#)
- [Siméon and Wadler, 2003] Siméon, J. and Wadler, P. (2003). The essence of XML. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–13, New Orleans, Louisiana, USA. ACM Press. [34](#), [35](#)
- [Stockmeyer, 1974] Stockmeyer, L. (1974). The complexity of decision problems in automata theory and logic. Technical Report MAC-TR-133, Project MAC, M.I.T. [31](#)
- [Stockmeyer and Meyer, 1973] Stockmeyer, L. and Meyer, A. (1973). Word problems requiring exponential time. In *STOC '73: Proceedings of the 5th ACM symposium on Theory of computing*, pages 1–9, Austin, Texas, United States. ACM Press. [31](#)
- [Su et al., 2002] Su, Z., Aiken, A., Niehren, J., Priesnitz, T., and Treinen, R. (2002). The first-order theory of subtyping constraints. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 203–216, Portland, Oregon. ACM Press. [34](#)
- [Suciu, 2002] Suciu, D. (2002). The XML typechecking problem. *SIGMOD Rec.*, 31(1):89–96. [33](#), [34](#)
- [Sur et al., 2004] Sur, G., Hammer, J., and Siméon, J. (2004). Updatex - an XQuery-based language for processing updates in XML. In *PLAN-X 2004: Proceedings of the International Workshop on Programming Language Technologies for XML, Venice, Italy*, volume NS-03-4 of *BRICS Notes Series*, pages 40–53, Venice, Italy. BRICS. [5](#), [128](#)
- [Tanabe et al., 2005] Tanabe, Y., Takahashi, K., Yamamoto, M., Tozawa, A., and Hagiya, M. (2005). A decision procedure for the alternation-free two-way modal  $\mu$ -calculus. In *TABLEAUX '05: Proceedings of the 14th International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, volume 3702 of *LNCS*, pages 277–291, Koblenz, Germany. Springer-Verlag. [31](#), [33](#), [63](#), [72](#), [73](#), [123](#)
- [Thatcher and Wright, 1968] Thatcher, J. W. and Wright, J. B. (1968). Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1):57–81. [25](#), [27](#), [29](#)
- [Tozawa, 2001] Tozawa, A. (2001). Towards static type checking for XSLT. In *DocEng '01: Proceedings of the 2001 ACM Symposium on Document Engineering*, pages 18–27, Atlanta, Georgia, USA. ACM Press. [1](#), [33](#), [34](#), [125](#)
- [Tozawa, 2004] Tozawa, A. (2004). On binary tree logic for XML and its satisfiability test. In *PPL '04: Informal Proceedings of the Sixth JSSST Workshop on Programming and Programming Languages*. [33](#)

- 
- [Tozawa and Hagiya, 2003] Tozawa, A. and Hagiya, M. (2003). XML Schema containment checking based on semi-implicit techniques. In Ibarra, O. H. and Dang, Z., editors, *CIAA '03: Proceedings of the 8th International Conference on Implementation and Application of Automata*, volume 2759 of *Lecture Notes in Computer Science*, pages 213–225, Santa Barbara, California, USA. Springer. 35, 100
- [Vardi, 1998] Vardi, M. Y. (1998). Reasoning about the past with two-way automata. In *ICALP '98: Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, pages 628–641. Springer-Verlag. 20, 32, 60, 62, 72
- [Wadler, 2000] Wadler, P. (2000). Two semantics for XPath. Internal Technical Note of the W3C XSL Working Group, <http://homepages.inf.ed.ac.uk/wadler/papers/xpath-semantics/xpath-semantics.pdf>. 21
- [Wood, 2003] Wood, P. T. (2003). Containment for XPath fragments under DTD constraints. In *ICDT '03: Proceedings of the 9th International Conference on Database Theory*, volume 2572 of *LNCS*, pages 300–314. Springer-Verlag. 20



# Computational Complexity for Logical Satisfiability Dealt With in this Dissertation

---

Undecidable	↑
Decidable	<ul style="list-style-type: none"> <li>• <math>2^{2^{2^{O(n)}}}</math> WS2S [<a href="#">Meyer, 1975</a>] used in Chapter 3.</li> </ul>
Elementary	
EXPSPACE	
EXPTIME	<ul style="list-style-type: none"> <li>• <math>2^{O(n^4 \cdot \log(n))}</math> Full <math>\mu</math>-calculus [<a href="#">Grädel et al., 2002</a>].</li> <li>• <math>2^{O(n \cdot \log(n))}</math> AFMC [<a href="#">Tanabe et al., 2005</a>] used in Chapter 4.</li> <li>• <math>2^{O(n)}</math> <math>\mathcal{L}_\mu</math> logic proposed in Chapters 5 and 6.</li> </ul>
PSPACE	
NP	
P (PTIME)	



# Résumé étendu

---

## Motivation et objectifs

Ce travail a été initialement motivé par le besoin d'analyseurs statiques efficaces pour les langages de manipulation de données et de documents XML. Ces langages de programmation utilisent des schémas [Fallside and Walmsley, 2004] et des requêtes XPath [Clark and DeRose, 1999] comme constructions de première classe. Des exemples actuels de ces langages incluent la recommandation du W3C XSLT [Clark, 1999] pour la transformation de documents XML, et la future recommandation XQuery [Boag et al., 2006] pour l'interrogation de bases de données XML. Equiper ces langages de systèmes décidables et efficaces pour la vérification statique de types a été l'un des défis de recherche majeurs de la dernière décennie, qui a entre autres rassemblé les communautés travaillant sur les langages de programmation, les bases de données, les documents structurés, et l'informatique théorique. Ce travail poursuit l'effort de recherche initié dans les travaux décrits dans [Murata, 1996, Tozawa, 2001, Milo et al., 2003, Hosoya and Pierce, 2003].

Ce travail a abouti à la conception d'une logique d'arbre finis adaptée à XML, et sa procédure de décision, présentées dans cette thèse. Le solveur logique a été implanté au cœur d'un système pour l'analyse statique générale et le typage des spécifications XML. Le système peut être utilisé comme un composant d'analyseurs statiques pour les langages de programmation utilisant à la fois des expressions XPath et des types XML.

Cette thèse présente les investigations théoriques qui ont conduit aux fondations de cette nouvelle logique d'arbres finis, avec les bases algorithmiques et les principes d'implantation sur lesquels le solveur logique repose. Ces découvertes sont appliquées à la résolution des problèmes de typage XML, qui sont traduits dans la logique. Les problèmes résolus incluent le typage statique du langage XPath en présence de types réguliers d'arbres.

## Documents XML et schémas

*Extensible Markup Language* (XML) [Bray et al., 2004] est un format de fichier texte pour la représentation de structures arborescentes sous une forme standard.

La structure complète d'un document XML, si on s'abstrait des détails d'importance moindre, est un arbre d'arité variable, dans lequel les nœuds (aussi appelés *éléments* dans le jargon XML) sont étiquetés, les feuilles de l'arbre sont des nœuds textes, et l'ordre entre les nœuds enfants est important. XML peut être vu comme une syntaxe concrète pour la description de telles structures en utilisant des balises. Un exemple de document XML suit:

```
<plante>
  <categorie>Vasculaire</categorie>
  <tissu>
    <nom>Phloeme</nom>
    <def>Le phloeme est un tissu vivant servant au transport
      dans toutes les parties de la plante.</def>
    <note>Dans les arbres, c'est une partie de l'écorce.</note>
  </tissu>
</plante>
```

Un élément est décrit par une paire composée d'une balise ouvrante `< ... >` et d'une balise fermante `</... >`, entre lesquelles le contenu de l'élément est inséré. Dans l'exemple précédent "plante", "categorie", "tissu", "nom", "def", et "note" sont des étiquettes (*noms d'élément* dans le jargon XML).

La spécification XML ne définit pas a priori l'ensemble des étiquettes permises dans un document XML, et ne définit pas non plus de sémantique pour les étiquettes. Seules des conditions de bonne formation sont définies pour s'assurer que les éléments sont bien imbriqués, ce qui permet de considérer les documents XML comme les arbres. Par exemple, la Figure 1 donne une représentation plus visuelle du précédent document XML bien formé.

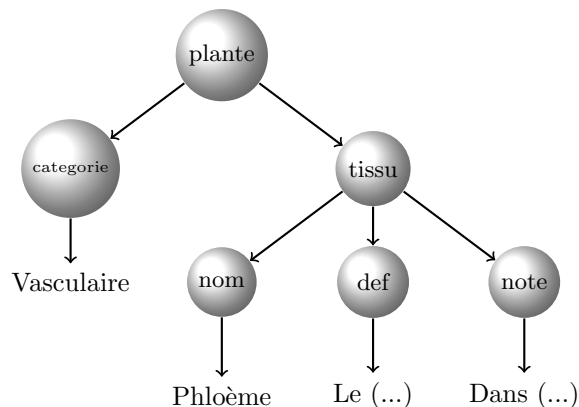


Figure 1: Exemple: arbre d'un document bien-formé.

L'ensemble des étiquettes qui apparaissent dans un document XML est déterminé par des *schémas* qui peuvent être librement définis par les utilisateurs. Un *schéma* (aussi appelé un *type XML*) est une description des contraintes sur la structure des documents, comme les étiquettes permises et leur possible structure d'imbrication. Un schéma définit ainsi une classe de documents XML. Deux niveaux de correction peuvent donc être distingués pour les documents XML:

- 
- le caractère *bien-formé* qui s'applique aux documents qui vérifient la condition nécessaire et suffisante (définie par la spécification XML) pour être interprétés comme des arbres;
  - la *validité* qui s'applique aux documents qui vérifient les contraintes additionnelles décrites par un schéma donné.

La validité d'un document implique son caractère bien-formé puisque un schéma décrit des contraintes sur l'arbre et non sur la représentation textuelle du document XML.

Chaque application peut définir son propre format de données en définissant des schémas, à un plus haut niveau d'abstraction (structures arborescentes). De ce fait, XML est souvent appelé un métalangage ou un "format pour les formats de données".

Séparer les deux niveaux de correction permet aux applications de partager des outils logiciels génériques pour manipuler des documents bien formés (analyseurs syntaxiques, éditeurs, requêtes, outils d'interrogation et de transformation...). Ces outils implantent tous les mêmes conventions définies par la spécification XML (comme la façon d'inclure des commentaires, des fragments externes, des caractères spéciaux...). XML rend ainsi possible un premier niveau de traitement pour un document XML dès lors qu'il est bien-formé, sans faire l'hypothèse additionnelle beaucoup plus forte qu'il est valide par rapport à un certain schéma. Cette généralité est l'une des forces de XML. En conséquence, l'adoption de XML s'est faite à une vitesse et une ampleur inégalée. De nombreux schémas ont été définis et sont actuellement largement utilisés en pratique, par exemple: XHTML (la version XML de HTML), SVG (pour le graphisme vectoriel), SMIL (pour la synchronisation des documents multimédias), MatML (pour les formules mathématiques), SOAP (pour l'appel de procédure à distance), XBRL et FIX (pour les informations financières et les transactions de valeurs), SMD (pour la musique), X3D (pour la modélisation 3D), et CML (pour les structures chimiques).

## XPath

XPath [Clark and DeRose, 1999, Berglund et al., 2006] a été introduit par le W3C comme le langage de requêtes standard pour sélectionner et récupérer de l'information dans les documents XML. Il permet de naviguer dans les arbres XML et de retourner un ensemble de nœuds vérifiant certaines conditions. En tant que tel, XPath forme l'essence de l'accès aux données XML.

Dans leur forme la plus simple, les expressions XPath ressemblent à des "chemins de navigation dans les répertoires". Par exemple, l'expression XPath

/livre/chapitre/section

navigue à partir de la racine d'un document (désignée par le "/" en tête) à travers les nœuds "livre" au premier niveau, vers leurs nœuds enfants "chapitre", jusqu'à leurs nœuds enfants nommés "section". Le résultat de l'évaluation de l'expression complète est l'ensemble de tous les nœuds "section" qui peuvent être atteints de cette manière. De plus, à chaque étape de la navigation, les nœuds sélectionnés peuvent être filtrés avec des qualifieurs. Un qualifieur est

une expression booléenne entre crochets qui peut tester l'existence ou l'absence de chemins. Si on formule par exemple la requête suivante :

```
/livre/chapitre/section[citation]
```

alors le résultat est constitué de *tous* les éléments “section” qui ont au moins un élément fils nommé “citation”. La situation devient plus intéressante lorsque les capacités de navigation de XPath selon d'autres “axes” que l'axe “child” sont utilisées. En effet, l'expression XPath précédente est un raccourci pour :

```
/child::livre/child::chapitre/child::section[child::citation]
```

qui fait apparaître de manière explicite que chaque étape de navigation utilise l'axe “child” contenant tous les nœuds enfants des nœuds sélectionnés lors de l'étape précédente. Si on formule la requête suivante :

```
/child::livre/descendant::*[child::citation]
```

alors la dernière étape sélectionne les nœuds de n'importe quel nom qui sont parmi les descendants de l'élément “livre” et qui ont un sous-élément nommé “citation”. Il est aussi possible d'utiliser des axes comme “preceding-sibling” pour naviguer vers les précédents nœuds fils du même parent, ou “ancestor” pour naviguer récursivement vers le haut (cf. Figure 2). *L'ordre du document* est défini comme l'ordre dans lequel les nœuds sont visités par un parcours en profondeur d'abord de l'arbre. Les axes qui effectuent de la navigation dans l'ordre inverse de l'ordre du document sont appelés “axes inverses”.

Les exemples précédents illustrent tous des expressions XPath absolues puisqu'elles commencent toutes avec un “/” qui se réfère à la racine. La sémantique d'une expression *relative* (sans le “/” en tête) est définie par rapport à un *nœud de contexte* dans l'arbre. Le *nœud de contexte* désigne simplement le nœud de l'arbre depuis lequel la navigation débute. A partir d'un nœud de contexte quelconque dans un arbre, tous les autres nœuds peuvent être facilement atteints: les axes XPath forment une partition de l'arbre. La Figure 2 illustre cela sur un exemple. Plus de détails informels sur le langage XPath complet peuvent être trouvés dans la spécification du W3C [Clark and DeRose, 1999].

XPath est de plus en plus populaire du fait de son expressivité et de sa syntaxe compacte. Ces deux avantages ont conféré à XPath un rôle central dans d'autres spécifications clés et applications XML. Il est utilisé dans XQuery [Boag et al., 2006] comme le langage cœur pour formuler des requêtes; dans XSLT [Clark, 1999] pour la sélection des nœuds dans les transformations; dans XML Schema [Fallside and Walmsley, 2004] pour définir les clés; dans XLink [DeRose et al., 2001] et XPointer [DeRose et al., 2002] pour référencer des parties de données XML. XPath est aussi utilisé dans de nombreuses applications comme les langages de mise à jour [Sur et al., 2004] et de contrôle d'accès [Fan et al., 2004].

## Vérification statique de type

Les applications XML utilisent la plupart du temps les schémas pour effectuer de la validation (aussi appelée *vérification dynamique de type*). La validation consiste en l'utilisation d'un validateur de schéma qui analyse un document

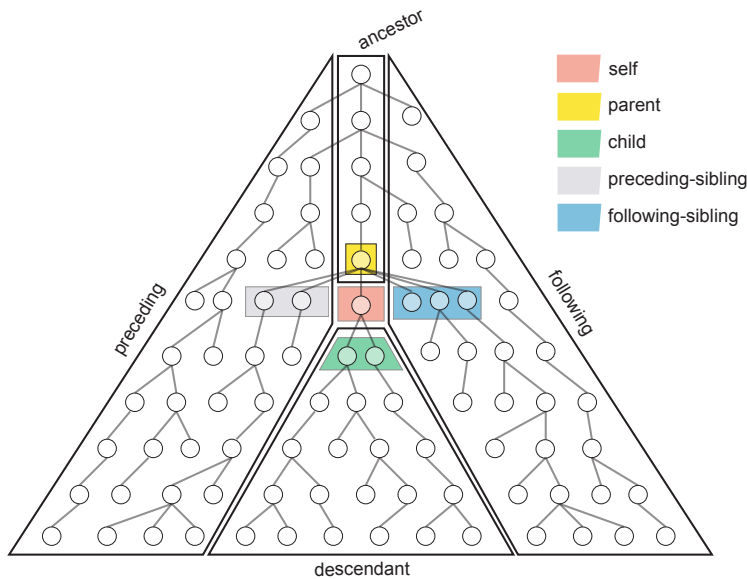


Figure 2: Partition des axes depuis un nœud de contexte.

XML particulier par rapport à un certain schéma dans le but de s'assurer que le document est bien conforme aux attentes de l'application.

En pratique cependant, les documents XML sont souvent générés dynamiquement par un certain programme. Typiquement, les programmes qui manipulent du XML accèdent tout d'abord aux données (se conformant possiblement à un certain schéma) avec des expressions XPath, et construisent et retournent ensuite un document XML résultat qui se conforme aux exigences d'un autre schéma.

Une approche ambitieuse est la *vérification statique de type* pour ces programmes, qui consiste à s'assurer au moment de la compilation, que le code traitant les données XML ne peut pas produire de document non valide. Un vérificateur statique de type analyse un programme, possiblement en présence des schémas qui décrivent ses entrées et sorties (si ceux-ci s'avèrent disponibles). La difficulté du problème est fonction du langage dans lequel le programme et les schémas sont exprimés.

Les langages de schémas ont fait l'objet de nombreuses études et sont maintenant bien compris comme des sous-ensembles des langages réguliers d'arbres [Murata et al., 2005]. Cependant, bien que de nombreuses tentatives aient été faites pour mieux comprendre les techniques de typage statique, en particulier à travers la conception de langages de programmation spécifiques au domaine [Hosoya and Pierce, 2003], aucune approche est effectivement capable de supporter XPath, qui demeure néanmoins l'essence de la navigation et de l'accès aux données XML.

## Défis de recherche

Les limitations des approches existantes sont justifiées par la difficulté de l'analyse statique de XPath. Il est connu que l'analyse statique du langage

XPath complet est indécidable. L'importance et l'ampleur des applications motivent néanmoins des questions de recherche: quel est le plus gros fragment de XPath dont l'analyse statique est décidable ? Quels fragments peuvent être efficacement décidés en pratique ? Comment déterminer si une expression XPath est satisfaisable sur l'un des arbres XML définis par un schéma donné ? Comment savoir si deux requêtes vont toujours donner le même résultat lorsqu'elles sont évaluées sur un document valide par rapport à un certain schéma ? Est ce que le résultat d'une expression XPath sur un document valide se conforme toujours aux exigences d'un autre schéma ? Existe-t-il un algorithme capable de répondre à ces questions d'une manière efficace de telle sorte qu'il soit utilisable en pratique ?

Une source de difficulté pour un tel algorithme est qu'il doit vérifier des propriétés sur une quantification possiblement infinie sur un ensemble d'arbres. Une variété d'autres facteurs contribuent de plus à sa complexité comme les opérateurs permis dans les requêtes XPath et leur composition (cf. Chapitre 2.2). Une conséquence de ces difficultés est que de telles questions de recherche sont toujours ouvertes.

## Aperçu de cette thèse

Cette thèse part de l'idée que deux problèmes doivent être résolus pour pouvoir répondre à des problèmes de décision dans le monde XML. Tout d'abord, identifier une logique appropriée avec une expressivité suffisante pour supporter à la fois les langages d'arbres réguliers et la navigation et la sémantique de sélection de nœuds à la XPath. Deuxièmement, résoudre efficacement le problème de la satisfaisabilité de cette logique qui permet de déterminer si une formule donnée de la logique admet un document XML qui la satisfait.

## Principales contributions

La contribution principale de cette thèse est une nouvelle logique pour les arbres finis, dérivée du  $\mu$ -calcul. La logique est suffisamment expressive pour capturer les langages réguliers d'arbres et la navigation multi-directionnelle dans les arbres finis. Elle est décidable en temps simplement exponentiel (plus précisément en  $2^{O(n)}$  étapes où  $n$  est la taille de la formule dont le statut de vérité est déterminé définie comme le nombre de propositions atomiques et d'éventualités qu'elle comporte). Cela améliore la meilleure complexité computationnelle connue pour les arbres finis. En tant que telle, cette logique offre un nouveau compromis entre expressivité et complexité, spécifiquement intéressant dans le contexte de XML).

Une autre contribution de cette thèse est de montrer comment traduire linéairement les requêtes et les types réguliers d'arbres (incluant les DTDs et les XML Schemas) dans la logique. La logique prend en compte XPath dans sa quasi globalité, et supporte le plus gros fragment qui a été traité pour l'analyse statique. Un autre avantage est que la logique est une sous-logique d'une existante (le  $\mu$ -calcul) ce qui facilite l'application de techniques d'optimisation connues et l'extensibilité.

Cela résout les problèmes de décision majeurs rencontrés dans l'analyse statique des langages manipulant des structures XML. Ces problèmes englobent l'inclusion, la satisfaisabilité, l'équivalence, le recouvrement, la couverture des

---

requêtes XPath (en présence ou absence de types réguliers d'arbres), le typage statique d'une requête XPath annotée, et l'équivalence des requêtes sous contraintes de type.

De plus, des techniques d'implantation sont présentées, qui conduisent à la réalisation concrète et à l'implantation d'algorithmes efficaces en pratique. Le système entièrement implanté est déjà capable de traiter des scénarios réalistes.

## Applications

La principale application de ce travail est une nouvelle classe d'analyseurs statiques de programmes manipulant des données et des documents XML. Cette classe d'analyseurs utilise directement les résultats décrits dans cette thèse, qui résolvent les problèmes de décision auxquels ils sont confrontés. La résolution de chaque problème particulier offre des applications importantes.

Par exemple, le problème le plus fondamental pour un langage de requête est la satisfaisabilité: une expression retourne-t-elle toujours un résultat vide ? La satisfaisabilité de XPath est importante pour l'optimisation des langages hôtes de XPath: par exemple, si on peut savoir au moment de la compilation qu'une requête est insatisfaisable, alors tous les calculs qui en dépendent peuvent être évités.

Un autre problème fondamental est le problème de l'équivalence: deux requêtes retournent-elles toujours les mêmes résultats ? Savoir déterminer l'équivalence entre deux requêtes est utile pour la reformulation et l'optimisation de la requête elle-même [Genevès and Vion-Dury, 2004], qui vise à s'assurer de propriétés opérationnelles tout en préservant la sémantique de la requête [Abiteboul and Vianu, 1999, Levin and Pierce, 2005].

Le problème le plus critique pour le typage statique des transformations XML est l'inclusion de requêtes XPath: est ce que, pour tout arbre, le résultat d'une requête particulière est inclus dans le résultat d'une autre ? Ce problème se pose pour l'analyse du flot de contrôle de XSLT [Møller et al., 2005]. Savoir déterminer l'inclusion est aussi utile pour vérifier les contraintes d'intégrités [Fallside and Walmsley, 2004], et pour vérifier la politique et les droits d'accès dans les applications de sécurité XML [Fan et al., 2004].

D'autres problèmes de décision utiles dans les applications incluent par exemple la couverture mutuelle des requêtes (deux expressions peuvent-elles sélectionner des nœuds communs ?) et la couverture (les nœuds sélectionnés par une requête sont-ils toujours contenus dans l'union des résultats sélectionnés par d'autres requêtes ?). Ces problèmes sont par exemple utiles pour détecter statiquement les erreurs de programmation.

Cette thèse résout ces problèmes de décision, en présence ou en l'absence de contraintes de types XML comme les DTDs [Bray et al., 2004] ou les XML Schemas [Fallside and Walmsley, 2004]. Cela permet de s'assurer de propriétés locales ou globales importantes (comme le bon typage ou des optimisations) au moment de la compilation, pour un traitement plus sûr et plus efficace des données XML. Les résultats présentés dans cette thèse ouvrent notamment des perspectives prometteuses concernant l'analyse statique des transformations XML.

## Organisation de la thèse

Cette thèse est divisée en trois parties. La première partie est dédiée à l'état de l'art et présente les techniques de pointe existantes et les travaux de recherche reliés. A cette fin, le chapitre 2 introduit quelques fondations théoriques et formalismes utilisés dans la suite de cette thèse, tout en introduisant progressivement les travaux reliés au fur et à mesure que leurs concepts sous-jacents ont été présentés.

Dans une seconde partie, les chapitres 3 et 4 conduisent des investigations préliminaires avec des logiques connues dans le cadre de XML. Plus précisément, le chapitre 3 étudie dans quelle mesure la logique monadique du second ordre peut être utilisée en pratique, en dépit de sa grande complexité, pour résoudre des problèmes d'analyse statique comme l'inclusion des requêtes XPath. Une procédure de décision correcte pour l'inclusion est proposée. Le chapitre 4 introduit le  $\mu$ -calcul sans alternance comme un puissant remplacement pour la logique monadique du second ordre, et étudie son usage pour raisonner sur les arbres XML. Les problèmes de décision mettant en jeu les requêtes XPath et les types réguliers sont réduits à la satisfaisabilité de cette logique sur des structures de Kripke générales.

Grâce aux leçons tirées des investigations précédemment menées, la troisième partie de cette thèse présente la contribution finale. Le chapitre 5 propose une logique d'arbres finis spécifiquement conçue pour XML. Le chapitre 6 propose un algorithme pour tester la satisfaisabilité de la logique, ainsi que les techniques pour son implantation. Des expérimentations sont menées avec une implantation complète du système, qui s'avère efficace sur plusieurs scénarios réalistes. Enfin, le chapitre 7 conclut cette thèse et donne de nouvelles perspectives.