University of Kentucky
UKnowledge

Theses and Dissertations--Computer Science

Computer Science

2012

MULTIHIERARCHICAL DOCUMENTS AND FINE-GRAINED ACCESS CONTROL

Neil Moore
University of Kentucky, neil@uky.edu

Right click to open a feedback form in a new tab to let us know how this document benefits you.

Recommended Citation

Moore, Neil, "MULTIHIERARCHICAL DOCUMENTS AND FINE-GRAINED ACCESS CONTROL" (2012). *Theses and Dissertations--Computer Science*. 6.
https://uknowledge.uky.edu/cs_etds/6

This Doctoral Dissertation is brought to you for free and open access by the Computer Science at UKnowledge. It has been accepted for inclusion in Theses and Dissertations--Computer Science by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained and attached hereto needed written permission statements(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine).

I hereby grant to The University of Kentucky and its agents the non-exclusive license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless a preapproved embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's dissertation including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Neil Moore, Student

Dr. Jerzy Wl. Jaromczyk, Major Professor

Dr. Raphael Finkel, Director of Graduate Studies

MULTIHIERARCHICAL DOCUMENTS AND FINE-GRAINED ACCESS CONTROL

DISSERTATION

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the College of Engineering at the University of Kentucky

By
Neil Moore
Lexington, Kentucky

Director: Dr. Jerzy Wl. Jaromczyk
Lexington, Kentucky 2012

Copyright © Neil Moore 2012

ABSTRACT OF DISSERTATION

MULTIHIERARCHICAL DOCUMENTS AND FINE-GRAINED ACCESS CONTROL

This work presents new models and algorithms for creating, modifying, and controlling access to complex text. The digitization of texts opens new opportunities for preservation, access, and analysis, but at the same time raises questions regarding how to represent and collaboratively edit such texts. Two issues of particular interest are modelling the relationships of markup (annotations) in complex texts, and controlling the creation and modification of those texts. This work addresses and connects these issues, with emphasis on data modelling, algorithms, and computational complexity; and contributes new results in these areas of research.

Although hierarchical models of text and markup are common, complex texts often exhibit layers of overlapping structure that are best described by multihierarchical markup. We develop a new model of multihierarchical markup, the globally ordered GODDAG, that combines features of both graph- and range-based models of markup, allowing documents to be unambiguously serialized. We describe extensions to the XPath query language to support globally ordered GODDAGs, provide semantics for a set of update operations on this structure, and provide algorithms for converting between two different representations of the globally ordered GODDAG.

Managing the collaborative editing of documents can require restricting the types of changes different editors may make, while not altogether restricting their access to the document. Fine-grained access control allows precisely these kinds of restrictions on the operations that a user is or is not permitted to perform on a document. We describe a rule-based model of fine-grained access control for updates of hierarchical documents, and in this context analyze the document generation problem: determining whether a document could have been created without violating a particular access control policy. We show that this problem is undecidable in the general case and provide computational complexity bounds for a number of restricted variants of the problem.

Finally, we extend our fine-grained access control model from hierarchical to multihierarchical documents. We provide semantics for fine-grained access control policies that control splice-in, splice-out, and rename operations on globally ordered GODDAGs, and show that the multihierarchical version of the document generation problem remains undecidable.

KEYWORDS: Text encoding, markup, multiple hierarchies, access control, globally ordered GODDAG

Author's signature: Neil Moore

Date: May 2, 2012

MULTIHIERARCHICAL DOCUMENTS AND FINE-GRAINED ACCESS CONTROL

By
Neil Moore

Director of Dissertation: Jerzy Wl. Jaromczyk

Director of Graduate Studies: Raphael Finkel

Date: May 2, 2012

Dedicated to my wife, Amanda Mefford, whose patience, understanding, and constant encouragement made this work possible.

ACKNOWLEDGMENTS

The current work, though the product of individual research, would not have been possible without the support of a number of individuals. Foremost thanks go to my advisor, Dr Jerzy Jaromczyk, for his guidance, technical advice, and support along what has been a long road through my graduate studies. I also thank Drs Kenneth Calvert, Raphael Finkel, and William Rayens for serving on my dissertation committee and providing constructive advice that has helped improve the final version of this work.

Also important are a long list of educators who have helped shape me into the researcher I have become. Though a full list would be impractical, I would like to single out a few particular individuals for acknowledgment. Drs Victor Marek and Mirosław Truszczyński provided me, many years ago, with the opportunity to perform undergraduate research, introducing me to the world of academics. Furthermore, I am especially appreciative of my high school mathematics teacher, Greg Moore (no relation), who nearly twenty years ago encouraged my developing interests in computers and mathematics, going so far as to provide me with a computer when my family could not afford one. Without people such as these, I would not be in a position to write and defend this dissertation today.

Finally, I would like to thank my family. My wife Amanda Mefford has been tremendously supportive throughout my graduate school career, encouraging me to press on even when I worried that I could not. Her sacrifices and patience have made an often difficult process much less so. My mother, father, stepfather, and grandparents have also been extremely important to me, encouraging me to pursue my dreams and instilling me with the values that make me who I am today.

TABLE OF CONTENTS

Acknowledgments iii

Table of Contents iv

List of Tables v

List of Figures vi

Chapter 1 Introduction 1
- 1.1 Our contributions 1
- 1.2 Motivations for our research 4
- 1.3 Structure of dissertation 7

Chapter 2 XML and hierarchical document models 8
- 2.1 Hierarchical documents 8
- 2.2 Selecting nodes in hierarchical documents: XPath and its subsets . . 14
- 2.3 Updating hierarchical documents 22

Chapter 3 Multihierarchical text: encodings and models 30
- 3.1 Encoding multiple hierarchies 31
- 3.2 Models of multihierarchical documents 39
- 3.3 Resolving inconsistencies in the GODDAG: global order 49

Chapter 4 Fine-grained access control 72
- 4.1 Definitions 72
- 4.2 Sub-problems with NP algorithms 75
- 4.3 Lower bounds for more complicated subproblems 82
- 4.4 General policies 91
- 4.5 Fine-grained access control for multihierarchical markup 108

Chapter 5 Conclusions and future research directions 115
- 5.1 Thesis contributions 115
- 5.2 Future work 115

Bibliography 118

Vita 125

LIST OF TABLES

4.1 Subproblems of PGen and lower bounds on their complexities 76
4.2 Subproblems of PGen and upper bounds on their complexities 76

LIST OF FIGURES

2.1 A hierarchical document. . . . 19
2.2 Evaluating an XPath expression on Figure 2.1. . . . 19

3.1 Self-overlapping and nested strike-through 33
3.2 Spurious overlap 33
3.3 Complete overlap 34
3.4 Multiple encoding of multiple hierarchies 35
3.5 TEI fragmentation encoding of multiple hierarchies 36
3.6 TEI fragmentation encoding of multiple hierarchies, with chaining 37
3.7 Milestone encoding of multiple hierarchies 38
3.8 CONCUR encoding of multiple hierarchies 39
3.9 Inconsistent sibling axes in an ordered DAG 43
3.10 Intersecting axes in an ordered DAG 43
3.11 Self-following in an ordered DAG 44
3.12 Containment relationships in a restricted GODDAG 48
3.13 A non-example of a restricted GODDAG 48
3.14 Figure 3.13 corrected to form a restricted GODDAG 49
3.15 Evaluating a multihierarchical XPath expression 52
3.16 Spurious and non-spurious overlap 53
3.17 A multihierarchical document 57
3.18 Inserting a node into a GODDAG 57
3.19 Splicing a node into a GODDAG 58
3.20 Deleting a node from a GODDAG 59
3.21 Non-triviality of narrow deletion 60
3.22 Splicing a node out of a GODDAG 61

4.1 Algorithm for deciding $\textsc{PGen}^i_+$ (positive monotone policies) 77
4.2 Nondeterministic algorithm for deciding $\textsc{PGen}^i$ (monotone policies) 80
4.3 Steps of the vertex cover simulation 83
4.4 Semantic tree for a QBF on two variables. 87
4.5 Tree for a QBF and its subformulae 90
4.6 A configuration tree 92
4.7 Simulating a Turing machine transition 101

Chapter 1 Introduction

This work presents new models and algorithms for creating, modifying, and controlling access to complex text. These models aim to support the digitization of complex texts that do not fit the commonly-used hierarchical model, and to allow controlled collaborative editing of such texts. We consider and connect two issues: modelling the relationships of markup in complex texts, and controlling the creation and modification of such markup.

Text documents are often encoded in the Extensible Markup Language (XML) [9], which, with its simple and elegant structure, offers a natural choice for many kinds of documents [22]. However, documents with multiple layers of structure or analysis, such as electronic editions of manuscripts [34, 45], often expose limitations of XML's elegant hierarchical document model and as a result require multihierarchical models of text and markup [58]. A number of such models have been introduced over the years [16, 57, 61, 64]; we identify some problems with these models, and introduce our own model that addresses these problems [55].

Complex text documents are often constructed by multiple users with different areas and levels of expertise and authority. This situation suggests that the editing process be governed by fine-grained access control rules restricting the types and locations of edits that may be made by certain users, groups, or roles. We examine a problem in the analysis of fine-grained access control policies for updating XML documents, obtaining a number of complexity results.

Finally, most models of fine-grained access control for document editing assume that documents are hierarchical, with edits consisting of tree update operations [49, 27]. In order to better support collaborative editing of complex documents that are best represented multihierarchically, we develop a model of access control for updates of multihierarchical documents.

1.1 Our contributions

In this dissertation we introduce and study models of multihierarchical text documents, as well as fine-grained access control of both hierarchical and multihierarchical documents to support managed collaborative editing. Our contributions advance the state of the art in multihierarchical markup and fine-grained access control. They include the following, described in more detail in Sections 1.1.1–1.1.3:

1. We develop a model of multihierarchical documents, the globally ordered GODDAG [55], in which markup and text have both a containment structure (in terms of directed acyclic graphs) and a global order with certain properties inspired by XML. We also introduce extensions to the XPath language to support these multihierarchical documents, as well as update operations to allow editing of such structures.

2. We prove that the globally ordered GODDAG characterizes multihierarchical documents with unambiguous serializations free of spurious overlaps, through correspondence with a structure, the range GODDAG, that models the markup relations in such documents. We furthermore provide algorithms for converting between the two structures. This characterization provides an important connection between graph-based models of documents and serialized depictions of markup ranges.

3. We establish complexity bounds for the document generation problem in certain classes of rule-based fine-grained access control systems for updates of trees, where rules are based on expressions in XPath or subsets thereof [54, 56]. This problem asks whether a given document could have been generated while following the rules of a given access control policy. We show that the most general version of the problem is in fact undecidable, indicating that such fine-grained access control systems are too powerful for many kinds of static analysis.

4. We develop a model of fine-grained access control on the globally ordered GODDAG, providing the first steps towards a fine-grained access control system for collaborative editing of multihierarchical documents. We also show that the multihierarchical version of the document generation problem, like the tree version, is undecidable.

1.1.1 The globally ordered GODDAG

We present a model of multihierarchical markup, which we call the globally ordered GODDAG or GOG, that augments the GODDAG (generalized ordered-descendant DAG) structure of Sperberg-McQueen and Huitfeldt [61] with an order relation and certain natural properties on that order and the graph structure. We also provide an alternative characterization of GOG documents in terms of serialization properties. We prove that documents in the model have (1) a unique representation as a directed acyclic graph, and (2) a unique textual serialization that avoids spurious overlap.

We also provide algorithms for converting between these representations. Finally, we provide semantics for an extension of XPath to globally ordered GODDAGs, and show how the model provides a convenient way to characterize the update operations that may be performed on a document while maintaining serializability. This work restates and extends results we previously published in 2010 [55].

1.1.2 The document generation problem

Fine-grained access control is the problem of specifying the set of operations that may be performed on a complex structure. For tree-structured databases and documents, particularly XML, a rule-based approach is most common. In this model, access control policies consist of rules that select the allowed or disallowed targets of queries or updates based on their hierarchical relationships to other nodes.

We consider the problem of deciding whether a fine-grained access control policy for tree updates allows a particular document to be constructed. This problem, which we call PGen, is motivated by a number of database administration and collaborative editing scenarios. For example, an administrator might wish to audit a document by verifying that it was (or at least could have been) created under the existing policy; such auditing is particularly important when logs of previous operations are missing or unavailable. Another application is to evaluate a proposed policy change, by verifying that existing documents could be reconstructed under the policy. Finally, an algorithm for PGen would allow administrators to verify that particular undesirable document states are disallowed: that is, that the policy does not generate particular trees.

We show that, for a typical form of rule-based fine-grained access control policies based on a simple fragment of XPath, the problem PGen is undecidable. We also prove lower bounds on the complexity of various restrictions of this problem, including those that limit both the subset of XPath and the set of update operations allowed. For two restrictions in particular, we demonstrate deterministic and nondeterministic (respectively) polynomial-time algorithms for PGen. This work extends results we previously published in 2009 [54] and 2011 [56].

1.1.3 Fine-grained access control on multihierarchical structures

Although fine-grained access control has been well-studied for hierarchical documents [27, 28], it has not previously been applied to multihierarchical structures. In order to facilitate managed collaborative editing of multihierarchical documents, we define

a model for fine-grained access control on globally ordered GODDAGs, based on the update operations and XPath extensions described elsewhere in this work. We show that, as with hierarchical documents, the document generation problem $\textsc{PGen}_{\mathrm{MH}}$ is undecidable.

1.2 Motivations for our research

Digital editions provide an important means of preserving documents while facilitating access by researchers, students, and the general public. Although such editions can be based on page images [1], the use of marked-up text [22], possibly in conjunction with images [46], is a recognized means of making these texts available to computational processing and analysis as well [65]. Our interest in multihierarchical markup and fine-grained access control arose through in involvement with the ARCHway [45] project, developing tools for constructing electronic editions such as Electronic Boethius [34].

The encoding of text and markup is often based on a model of text as an "ordered hierarchy of content objects" [22]. This model, exemplified by languages such as the Standard Generalized Markup Language (SGML) [39] and the Extensible Markup Language [9], represents a document as an ordered tree with plain text (that is to say, character strings) forming leaves and markup appearing as both internal nodes and leaves. This model aims to recapture the hierarchical nature of many of the concepts represented by markup, such as books that are divided into chapters, themselves divided into sections.

1.2.1 Multihierarchical markup

Although XML, with its simple and elegant hierarchical structure, is a popular choice for text encoding, this very simplicity leads to problems with complex documents; in particular, a single hierarchy may not be sufficient to capture the relationships among multiple layers of descriptions [58]. For example, in preparing an electronic edition of a literary manuscript, an editor may wish to describe both the division of the document into pages and lines, *and* the division into paragraphs and sentences. However, as most printed texts readily demonstrate, sentences and lines do not necessarily nest within one another to form hierarchical structures; a sentence may overlap with a line without either being included within the other. Even two annotations that encode the same type of feature, such as stricken-out text [58] or components of a portmanteau word, may overlap with one another. The question, then, is how to represent such

multihierarchical structures in a way that offers the same advantages of portability, flexibility, etc. as XML does for hierarchical documents. Such a solution would ideally have two components: a mathematical model of multihierarchical documents, corresponding to the tree-based document object model (DOM) of XML [4]; and an encoding of such documents in a format for data interchange, corresponding to the tag-based textual serialization of XML.

Many representations for multiple hierarchies have been proposed, ranging from straightforward embeddings in XML [21, 64] to entirely new data structures [16, 43, 57, 61]. One model in particular, Sperberg-McQueen and Huitfeldt's generalized ordered-descendant directed acyclic graph (GODDAG) [61], defines a straightforward but powerful generalization of the DOM tree model to multihierarchical structures. As a result, the GODDAG is used as the basis for a number of models of and systems for multihierarchical documents [37, 38, 52]. However, the GODDAG lacks many of the desirable properties of XML trees, such as a consistent order among nodes in the document; as a result, tag-based serializations along the lines of XML cannot unambiguously represent the full range of GODDAG documents without significant added complexity. Furthermore, restricted versions of GODDAG designed to avoid these problems [52, 61] can make it difficult to concisely specify the set of updates that may be applied to a document while maintaining the integrity of the structure. Our model attempts to address these problems.

1.2.2 Fine-grained access control

The construction of digital documents is often conducted by groups of individuals, sometimes separated by geographical and organizational boundaries. Although projects such as Wikipedia [2] show that it is sometimes possible to organize collaborative editing as a (more-or-less) "free for all" process, often some kind of control is desired. In many instances, the many contributors to a document play different roles in the construction process. For example, some contributors to an electronic edition may concentrate on the physical nature of the document (for example, fire and water damage); while others might instead focus on lyrical properties of the poetry contained therein. A project manager may wish to make this division of labor explicit, by ensuring that editors work only on their assigned tasks. Access control is the problem of determining which users can access or modify which resources, and of enforcing those determinations, and has many applications to collaborative editing:

- Protecting sensitive data, such as personally identifying information in medical

data. Although this data may be necessary for the attending physician, it should not be available to outside researchers studying the efficacy of treatment methods.

- Enforcing editorial roles and delegations. In an image-based electronic edition project, a student assistant editor might be allowed to add certain kinds of markup, but not to change the underlying text. In a business environment, the marketing team might be permitted to modify product brochures but (we hope) not technical manuals.

- Gradually expanding privileges for initially-untrusted users. For example, in Wikipedia [2], all visitors are permitted and encouraged to add and edit content. However, newly-registered users cannot edit certain "semi-protected" pages, and unregistered users are even more restricted. [3]

Because encoded text documents have an internal structure that is often associated with a division of labor, it is natural to ask whether it is possible to control updates to certain parts of this structure, rather than to entire documents. This topic of fine-grained access control (FGAC) for hierarchical structures has been heavily studied, particularly in the context of XML documents. For FGAC of XML documents, a rule-based approach is prevalent in the literature [5, 14, 28, 33]: in this model, an **access control policy** comprises a collection of **rules**. Each rule permits or denies a user, role, program, etc. (the **subject**) to use a particular kind of operation on certain parts of the document (the **object**). Objects are nodes or subtrees of the document, usually specified with a language of **path expressions**, usually XPath [12] or some fragment thereof [23, 53], but sometimes with tree automata and term rewriting systems [40].

Although most of the early research on FGAC for XML focused on query operations [5, 11, 14], there has also been a great deal of work in applying FGAC to update operations [10, 15, 27, 49]. Three problems have been particularly well-studied for hierarchical documents: specifying and formalizing the semantics of FGAC policies [6, 27, 28, 33]; safely and efficiently enforcing these policies [5, 11, 15, 47, 50]; and analyzing policies for consistency [7, 8, 27]. Another important problem is to determine a posteriori whether a document could have been constructed under a given access control policy [54, 56]; applications include verifying that a document was (or could have been) created pursuant to a policy, and checking whether a policy can result in documents that violate schemas or other structural rules [40]. This problem thus has

important implications for document security, verifiability, and provenance, even in the absence of multihierarchical markup.

Finally, multihierarchical text documents, including annotated electronic editions of literary works [34, 45], may be created through a collaborative but curated process, a perfect situation for fine-grained access control. This method of creation suggests the need for models of update operations, path expressions, and finally fine-grained access control policies for multihierarchical structures. To our knowledge, this problem has not been previously addressed in the literature.

1.3 Structure of dissertation

In Chapter 2 we discuss hierarchical models of text and markup, and formulate a model that characterizes documents in a simplified subset of XML. We also discuss the XPath language for querying node sets according to their hierarchical and order relationships. Finally, we discuss various types of update operations on hierarchical documents.

In Chapter 3 we discuss models of multihierarchical documents, beginning with three important existing classes of models (concurrent trees, overlapping ranges, and ordered DAGs) and concluding with our structure, the globally ordered GODDAG (GOG). We show that the GOG connects the ordered DAG model of multihierarchical documents with serializable and range-based models of markup, and provide algorithms for converting between a range-based and a graph-based representation. Finally, we provide extensions of XPath and the update operations discussed in Chapter 2 to support globally ordered GODDAGs and some other models of multihierarchical documents.

Chapter 4 concerns fine-grained access control for document updates. We briefly discuss existing models of FGAC, and present in detail a model that simplifies many of those found in the literature. We describe the document generation problem on access control policies, and provide new complexity results, including in some cases undecidability, for this problem under various subsets of our access control model. Furthermore, we extend our access control model to updates of globally ordered GODDAGs.

Finally, Chapter 5 summarizes the key results and contributions of this work, and briefly discusses possible directions for future research in fine-grained access control and multihierarchical documents.

Copyright © Neil Moore, 2012.

2.1 Hierarchical documents

We begin with a simple hierarchical model of text as an "ordered hierarchy of content objects" [22]. In this model, a document is an arborescence[1] with its vertices divided into **text nodes** (with outdegree zero) and **element nodes** or **elements** (with any outdegree, including zero). At each node ν, there is a total order (the **child order at** ν) on the successors (**children**) of ν. Each text node is associated with a string (over, for example, the Unicode alphabet [13], as in XML [9]) called the node's **text**. Each element node has a **name** from some language L of labels, as well as an **attribute** map from a language of keys (which, for simplicity, we assume to be the same as L) to a language of values (assumed to be Σ^*; in particular, attributes may have empty values, and an attribute with an empty value is distinguished from a missing attribute).

We begin with some terminology regarding binary relations.

Definition 2.1.1 *A **strict partial order** is a transitive irreflexive binary relation. A **strict total order** is a strict partial order that is additionally **trichotomous**: For any x, y in the domain, either $x = y$, $x < y$, or $x > y$.*

Definition 2.1.2 *The **transitive closure** R^+ of a binary relation R is the minimal transitive binary relation that contains R. If the domain of R is finite with cardinality k, then the transitive closure of R is:*

$$R^+ = R \cup R^2 \cup \cdots \cup R^k$$

.

*A **transitive reduction** of a binary relation R is a minimal binary relation R^- such that $(R^-)^+ = R^+$. If R is acyclic, then it has a unique transitive reduction:*

$$R^- = \left\{(x, z) \in R \middle| \neg\exists y : \ x \, R^+ \, y \wedge y \, R \, z\right\}.$$

We may now define our model of hierarchical documents.

[1] That is to say, a tree with all edges directed away from the root. Alternatively, a directed graph with a **root** node ρ such that, for every vertex v, there is exactly one path from ρ to v.

Definition 2.1.3 *A **hierarchical document** is a tuple*

$$(\Sigma, L, T, E, A, \text{text}, \text{label}, \text{order}, \text{attribute})\,,$$

where

- Σ *is a **text alphabet**;*
- L *is a set of element (and attribute) **labels**;*
- T *is a set of **terminal** or **text** nodes;*
- E *is a set of **nonterminal** or **element** nodes, with* $T \cap E = \emptyset$*;*
- $A \subset E \times (T \cup E)$ *is a set of arcs from elements to nodes such that, if* $T \cup E$ *is non-empty,* $(T \cup E, A)$ *is an arborescence;*
- text *is a function from* T *to* Σ^+*;*
- label *is a function from* E *to* L*;*
- order *is a function mapping each node of* $T \cup E$ *to a strict total order over that node's children in* A*; and*
- attribute *is a function mapping elements in* E *to partial functions from attribute keys in* L *to strings in* Σ^**. That is,* $\text{attribute} : E \to L \to (\Sigma^* \cup \bot)$*.*

To simplify later definitions, we define notations for a number of relations and functions on the nodes of a document. In these notations, the document is not specified and is assumed to be clear from context.

Definition 2.1.4 *Given a document* $D = (\Sigma, L, T, E, A, \text{text}, \text{label}, \text{order}, \text{attribute})$*, we write:*

- $<_\nu$ *for* $\text{order}(\nu)$*.*
- $\nu \to \xi$ *(or* $\xi \leftarrow \nu$*) if there is an arc* $(\nu, \xi) \in A$*. We say also that* ν ***directly contains*** ξ*.*
- $\nu \to^* \xi$ *(or* $\xi \leftarrow^* \nu$*) if there is a (possibly empty) path from* ν *to* ξ *in* A*. We say also that* ν ***dominates*** ξ*.*
- $\nu \to^+ \xi$ *(or* $\xi \leftarrow^+ \nu$*) if there is a nonempty path from* ν *to* ξ *in* A*; that is, if* $(\nu, \xi) \in A^+$*. We say also that* ν ***contains*** ξ*.*

- $\mathrm{root}(D)$ *for the root of the arborescence* $(T \cup E, A)$*, or undefined if* $T \cup E$ *is empty.*

- $\mathrm{children}(\nu)$ *for either the sequence of nodes* ξ *such that* $\nu \to \xi$*, ordered by* $<_\nu$*; or the set of these nodes.*

- $\mathrm{parent}(\nu)$ *for the (unique) node* π *such that* $\pi \to \nu$ *(with* $\mathrm{parent}(\mathrm{root}(D))$ *undefined).*

- $\mathrm{descendants}(\nu)$ *for the set of nodes* ξ *such that* $\nu \to^+ \xi$ *(the empty set if* $\nu \in T$*).*

- $\mathrm{ancestors}(\nu)$ *for the set of nodes* ξ *such that* $\nu \leftarrow^+ \xi$ *(the empty set if* $\nu = \mathrm{root}(D)$*).*

We write $\mathsf{Doc}_{\Sigma,L}$ for the set of all documents with node names from L and text from Σ^+.

The child orders of document nodes can be combined and extended to a (strict) partial order, the **precedes** relation, over all nodes of the document. Two nodes are ordered according to the order of their respective subtrees in their lowest common ancestor; nodes are incomparable with their ancestors and descendants.

Definition 2.1.5 *We say that node* ν ***precedes*** *node* ξ *(equivalently,* ξ ***follows*** ν*) in document* D *if there is some node* $a_\xi \to^* \xi$ *(*ξ *or an ancestor of* ξ*) and some node* $a_\nu \to^* \nu$ *such that* a_ξ *and* a_ν *have the same parent* p*, and* $a_\nu <_p a_\xi$*. If* ν *precedes* ξ*, we write* $\nu <_D \xi$ *and* $\xi >_D \nu$*.*

Theorem 2.1.1 $<_D$ *is a strict partial order over the set* $T \cup E$ *of nodes in* D*.*

PROOF:

- *Irreflexivity*: Because the nodes of D form an arborescence, for any nodes p and d there is at most one child c of p such that $c \to^* d$. Hence no node has distinct ancestors a_ν and a_ξ with the same parent, and Definition 2.1.5 does not apply when ν and ξ are the same node.

- *Transitivity*: Let $\mu, \nu, \xi \in T \cup E$ with $\mu <_D \nu$ and $\nu <_D \xi$. Let p_1 be the common ancestor of μ and ν from Definition 2.1.5, with a_μ and a_ν being the corresponding children of p_1; and let p_2 be the common ancestor of ν and ξ, with children a'_ν and a'_ξ. Since both p_1 and p_2 are ancestors of ν, either they are the same node, or one is an ancestor of the other.

If $p_1 = p_2 = p$, then $a_\mu <_p a_\nu = a'_\nu <_p a'_\xi$; by transitivity, $a_\mu <_p a'_\xi$, so $\mu <_D \xi$. Otherwise, one is an ancestor of the other; assume without loss of generality that p_1 is an ancestor of p_2. Then the child a_ξ of p_1 containing ξ and the child a_ν of p_1 containing ν both contain or equal p_2, and are hence the same node. Because $a_\mu <_{p_1} a_\nu$, we have that $a_\mu <_{p_1} a_\xi$, and $\mu <_D \xi$

□

The precedes relation may be combined with the ancestor-of relation to form a total order, known as the **document order** in XML. In fact, the relationships may be combined in two distinct ways, each giving a total order.

Definition 2.1.6 *Given a document D, the **document order** on D is the binary relation $<_+$ on nodes, where $\mu <_+$ if $\mu <_D \nu$ or $\mu \rightarrow^+ \nu$. The **inverted document order** is the relation $<_-$, where $\mu <_- \nu$ if $\mu <_D \nu$ or $\nu \rightarrow^+ \mu$.*

The document order is the preorder depth-first traversal of the document, beginning from the root and processing children of each node ν in the order given by $<_\nu$. The inverted document order is the corresponding postorder traversal. As we shall see, these are the orders of text nodes along with start or end tags, respectively, in the XML serialization of the document.

To begin, we note that each node in a document partitions that document into five sets: the node itself, its ancestors, its descendants, its preceding nodes, and its following nodes.

Lemma 2.1.1 ***Orthogonality**: For any two distinct nodes ν and ξ of a document D, exactly one of the following is true: $\nu <_D \xi$, $\xi <_D \nu$, $\nu \rightarrow^+ \xi$, or $\xi \rightarrow^+ \nu$.*

PROOF: If $\nu \rightarrow^+ \xi$ or $\xi \rightarrow^+ \nu$ (by irreflexivity, only one of these can be true), then for any common ancestor p of ν and ξ, the children $a_\nu \rightarrow^* \nu$ and $a_\xi \rightarrow^* \xi$ of p are the same node. Hence Definition 2.1.5 does not apply, neither $\nu <_D \xi$ nor $\xi <_D \nu$, and exactly one of the cases is true.

Otherwise, neither $\nu \rightarrow^+ \xi$ nor $\xi \rightarrow^+ \nu$. Let p be the lowest common ancestor of ν and ξ; it is distinct from both ν and ξ. Now consider the children a_ν and a_ξ of p that are ancestors of ν and ξ respectively. If $a_\nu = a_\xi$ then it is a lower common ancestor than p, a contradiction. Therefore, because the child order is a strict total order, either $a_\nu <_p a_\xi$ and $\nu <_D \xi$; or $a_\xi <_p a_\nu$ and $\xi <_D \nu$; but not both. Hence exactly one of the cases is true. □

We shall return to this orthogonality property later, when in Section 3.3 we discuss models for multihierarchical documents. For now, we use it to show that the document orders are (strict) total orders.

Theorem 2.1.2 *The document order $<_+$ and inverted document order $<_-$ on a document D are strict total orders over the set $T \cup E$ of nodes of D.*

PROOF:

- *Trichotomy*: This property follows straightforwardly from Lemma 2.1.1 and the definitions of $<_+$ and $<_-$.
- *Transitivity*: We consider only $<_+$; the proof for $<_-$ is analogous. Suppose $\mu <_+ \nu$ and $\nu <_+ \xi$. If the former and latter relationship derive from the same underlying relation ($<_D$ or $\rightarrow^+$), transitivity follows from the transitivity of that relation. We therefore need only consider two cases:
  - $\mu \rightarrow^+ \nu <_D \xi$. If ξ is a descendant of μ, then $\mu <_+ \xi$. Otherwise, the lowest common ancestor of ξ and ν is also a common ancestor of ξ and μ, with a single child $a_\mu = a_\nu$ of this ancestor containing both μ and ν. Since $\nu <_D \xi$, this child precedes the corresponding ancestor of ξ, so $\mu <_D \xi$ and thus $\mu <_+ \xi$.
  - $\mu <_D \nu \rightarrow^+ \xi$. If ξ is a descendant of μ, then $\mu <_+ \xi$. Otherwise, the lowest common ancestor of μ and ν is also a common ancestor of μ and ξ, with a single child $a_\nu = a_\xi$ of this ancestor containing both ν and ξ. Since $\mu <_D \nu$, this child follows the corresponding ancestor of μ, so $\mu <_D \xi$ and thus $\mu <_+ \xi$.

□

2.1.1 XML

The Extensible Markup Language (XML) [9] defines documents as strings (**serializations**) obeying the XML grammar, with the XML and related Document Object Model (DOM) [4] standards defining a tree, similar to our hierarchical model, for each document. Text nodes are serialized as simple text strings, while elements are represented by the sequence of a **start tag** <*label*>, the children of the node in order, and an **end tag** </*label*>. XML requires that start and end tags be nested:

if the start tag of element e_1 occurs between the start and end tags of e_2, the end tag of e_1 also occurs between the tags of e_2, ensuring that the document does in fact form a tree. In a somewhat simplified form:

Definition 2.1.7 *The* ***XML serialization*** serial (D) *of a hierarchical document D is defined recursively on the nodes of the document.*

$$
\begin{aligned}
\mathrm{serial}\,(D) &= \mathrm{serial}\,(\mathrm{root}(D)) \\
\mathrm{serial}\,(t \in T) &= \mathrm{text}(t) \\
\mathrm{serial}\,(e \in E) &= <\mathrm{name}(e)>\mathrm{serial}\,(\chi_1)\,\mathrm{serial}\,(\chi_2)\ldots\mathrm{serial}\,(\chi_n)\,</\mathrm{name}(e)>,
\end{aligned}
$$

where $\chi_1, \chi_2, \ldots \chi_n$ *is the sequence* children(e).

If element names and text nodes are not permitted to contain the characters `<` or `>`, the serialization is unambiguous and reversible. XML, in fact, does not allow these characters to occur except where specifically indicated in the grammar; other methods, such as CDATA sections and entity references, are required to encode these (and a few other) characters in an XML document.

The document tree corresponding to a given serialized XML document is, essentially, the parse tree for that string, with elements containing those nodes that occur between their start and end tags, in order; and with other types of nodes appearing as leaves. Likewise, an XML tree may be serialized recursively, by a depth-first traversal of the nodes; start tags appear before content, and end tags after. XML defines a document order based on the sequence of start tags and leaf nodes; this order is the same as the document order of Definition 2.1.6.

The XML document model is somewhat more complicated than ours; besides elements and text nodes, XML documents may contain other types of nodes including processing instructions, comments, CDATA sections, and entity references; each type of node has its own serialization, none of which contain other nodes. The document as a whole is represented by a document node (the root of the document tree, distinct from the root element). The document also contains an **XML declaration** indicating character encoding and version of XML to which the document conforms; and an optional **document type declaration**, which, among other things, defines a simple grammar (the document type definition or DTD) to which elements and their children must conform. Finally, XML allows an **empty tag** shorthand <t/> for a start tag immediately followed by the corresponding end tag.

In the remainder of this work we shall, for simplicity, refer to XML documents and our hierarchical documents interchangeably. We shall for the most part disregard

the additional features of XML (document nodes, comments, etc.), referring to them only when they would require special treatment. In general, the document node may be treated as a special case of an element; and other node types may be treated as text nodes with labels distinct from those of ordinary text nodes.

2.2 Selecting nodes in hierarchical documents: XPath and its subsets

The nature of a hierarchical document as an arborescence implies that each node has a unique path from the root to that node. This fact, along with the often constrained structure of these paths, suggests a query language that locates and identifies nodes based on the paths from the root to those nodes. XPath [12], part of the W3 Consortium's family of XML technologies, is precisely such a language.

Rather than dealing with the full complexities of XPath, we consider the fragment known as **core XPath** [24, 31]. This subset of XPath dispenses with numeric, string, and Boolean expressions; rather, each expression evaluates to a set of nodes, greatly simplifying the semantics.

Briefly, a path expression in core XPath consists of a sequence of location steps, each containing an **axis** (the name of a relation on the document: **parent**, **descendant**, **following**, and so on), a **node test** (usually an element name or node type predicate such as `text()`), and optional **predicates** (themselves path expressions). A location step selects those nodes that have the axis relationship to nodes selected by the previous location step; that satisfy the node test; and at which each predicate evaluates to a nonempty set of nodes.

2.2.1 Core XPath

We follow Gottlob, Koch, and Pichler [31], with some exceptions noted, in our formulation of the syntax and semantics of core XPath. We use the symbol ε to represent the empty string.

Grammar 2.2.1 *A* ***core XPath expression*** *is given by the expression production of the following grammar, where the name nonterminal produces a language of element names.*

$$
\begin{aligned}
\textit{expression} &\Rightarrow \textit{relative} \;\big|\; \texttt{/}\,\textit{relative-path}\\
\textit{relative-path} &\Rightarrow \textit{location-step} \;\big|\; \textit{relative-path}\,\texttt{/}\,\textit{location-step}\\
\textit{location-step} &\Rightarrow \textit{axis}\;\texttt{::}\;\textit{node-test}\;\textit{predicate-list}\\
\textit{predicate-list} &\Rightarrow \varepsilon \;\big|\; \texttt{[}\,\textit{predicate}\,\texttt{]}\\
\textit{predicate} &\Rightarrow \texttt{(}\,\textit{predicate}\;\texttt{and}\;\textit{predicate}\,\texttt{)} \;\big|\; \texttt{(}\,\textit{predicate}\;\texttt{or}\;\textit{predicate}\,\texttt{)}\\
&\;\big|\; \texttt{not}\;\texttt{(}\,\textit{predicate}\,\texttt{)} \;\big|\; \textit{expression}\\
\textit{axis} &\Rightarrow \texttt{self} \;\big|\; \texttt{parent} \;\big|\; \texttt{child}\\
&\;\big|\; \texttt{ancestor} \;\big|\; \texttt{ancestor-or-self}\\
&\;\big|\; \texttt{descendant} \;\big|\; \texttt{descendant-or-self}\\
&\;\big|\; \texttt{following} \;\big|\; \texttt{following-sibling}\\
&\;\big|\; \texttt{preceding} \;\big|\; \texttt{preceding-sibling}\\
\textit{node-test} &\Rightarrow \textit{name} \;\big|\; \texttt{*} \;\big|\; \texttt{node()} \;\big|\; \texttt{text()}
\end{aligned}
$$

Remark: Gottlob, Koch, and Pichler use slightly different terminology, writing "locationpath" for relative path expressions (what we call "relative-path"), and folding our predicate-list production into the location-step production. Furthermore, they do not require Boolean expressions to be parenthesized, instead implicitly relying on relative precedences of `and` and `or` ; and their node tests are simpler, permitting only element names and `*` .

We also follow Gottlob *et al.*[31], with some modifications, in our definition of the semantics of a path expression. The semantics $\mathcal{S}_{\mathrm{C}}[[p]]_D$ of a path expression p on a document D is a function from sets of nodes (the **context**) to sets of nodes. The semantics of other productions in Grammar 2.2.1 are either functions from sets of nodes to sets of nodes, or (in the case of predicate-list and node-test), simply sets of nodes.

In the following definitions, we assume a document

$$D = (\Sigma, L, T, E, A, \text{text}, \text{label}, \text{order}, \text{attribute})\,,$$

and write $N = T \cup E$ for the set of nodes of D.

An axis represents a relation between nodes. We identify axis terminals `parent` , `child`, etc. with relations on the document D. The semantics of an axis is a function from sets of context nodes to sets of nodes that bear the indicated relation to a context node.

Definition 2.2.1 *An **axis** is one of the following relationships on a set N of nodes:*

- *self* $= \{(\nu, \nu) | \, \nu \in N\}$
- *parent* $= \{(\mu, \nu) | \, \mu \to \nu\}$
- *child* $= \{(\mu, \nu) | \, \nu \to \mu\}$
- *ancestor-or-self* $= \{(\mu, \nu) | \, \mu \to^* \nu\}$
- *descendant-or-self* $= \{(\mu, \nu) | \, \nu \to^* \mu\}$
- *ancestor* $= \{(\mu, \nu) | \, \mu \to^+ \nu\}$
- *descendant* $= \{(\mu, \nu) | \, \nu \to^+ \mu\}$
- *following* $= \{(\mu, \nu) | \, \mu >_D \nu\}$
- *preceding* $= \{(\mu, \nu) | \, \mu <_D \nu\}$
- *following-sibling* $= \{(\mu, \nu) \in$ *following*$| \, \mu$ *and* ν *have the same parent*$\}$
- *preceding-sibling* $= \{(\mu, \nu) \in$ *preceding*$| \, \mu$ *and* ν *have the same parent*$\}$

The semantics of an axis χ on document D, $\mathcal{S}_{\mathrm{C}}[[\chi]]_D : \mathcal{P}(N) \to \mathcal{P}(N)$, is given by:

$$\mathcal{S}_{\mathrm{C}}[[\chi]]_D \, (M) = \{\nu \in N | \, \exists \mu \in M : (\nu, \mu) \in \chi\}$$

A node test matches nodes of a particular type or (in the case of elements) name. In particular, $*$ matches all elements, `text()` matches all text nodes, `node()` matches all nodes, and an element name matches all elements with that name. The semantics of a node test is the set of matching nodes.

Definition 2.2.2 *The semantics of a node test τ on document D, $\mathcal{S}_{\mathrm{C}}[[\tau]]_D : \mathcal{P}(N)$, is given by:*

$$\begin{aligned} \mathcal{S}_{\mathrm{C}}[[\, * \,]]_D &= E \\ \mathcal{S}_{\mathrm{C}}[[\, \texttt{text()} \,]]_D &= T \\ \mathcal{S}_{\mathrm{C}}[[\, \texttt{node()} \,]]_D &= T \cup E \\ \mathcal{S}_{\mathrm{C}}[[\ell\,]]_D &= \{e \in E | \, \mathrm{name}(e) = \ell\} \, , \end{aligned}$$

where ℓ is a string produced by the nonterminal name.

We say that a node test τ matches the label $\ell \in (L \cup \Sigma^+)$ if one of the following is true:

- $\tau =$ `node()`;
- $\tau = \ell \in L$;
- $\tau =$ `*` *and* $\ell \in L$; *or*
- $\tau =$ `text()` *and* $\ell \in \Sigma^+$.

A predicate is a Boolean expression formed from path expressions, where a path expression is considered to be true at a given node if, when evaluated in the context of that node, it produces a nonempty set of nodes. A predicate list is simply an optional predicate, where the lack of a predicate is taken to be true of all nodes. Gottlob *et al.* use a bottom-up semantics for path expressions in predicates, contrary to the top-down semantics used elsewhere. We instead use the ordinary top-down semantics for all path expressions, reducing the number of definitions required at the expense of slightly complicating Definition 2.2.3.

Definition 2.2.3 *The semantics of a predicate Q on document D, $\mathcal{S}^{\mathrm{Q}}_{\mathrm{C}}[[Q]]_D \in \mathcal{P}(N)$, is given by the following, where $\mathcal{S}_{\mathrm{C}}[[P]]_D\,(C)$, defined later (Definition 2.2.6), is the result of evaluating the path expression P in context C.*

$$\begin{aligned}
\mathcal{S}^{\mathrm{Q}}_{\mathrm{C}}[[P]]_D &= \{\nu \in N \mid \mathcal{S}_{\mathrm{C}}[[P]]_D\,(\{\nu\}) \neq \emptyset\} \\
\mathcal{S}^{\mathrm{Q}}_{\mathrm{C}}[[\ \texttt{(}\,Q_1\ \texttt{and}\ Q_2\,\texttt{)}\ ]]_D &= \mathcal{S}^{\mathrm{Q}}_{\mathrm{C}}[[Q_1]]_D \cap \mathcal{S}^{\mathrm{Q}}_{\mathrm{C}}[[Q_2]]_D \\
\mathcal{S}^{\mathrm{Q}}_{\mathrm{C}}[[\ \texttt{(}\,Q_1\ \texttt{or}\ Q_2\,\texttt{)}\ ]]_D &= \mathcal{S}^{\mathrm{Q}}_{\mathrm{C}}[[Q_1]]_D \cup \mathcal{S}^{\mathrm{Q}}_{\mathrm{C}}[[Q_2]]_D \\
\mathcal{S}^{\mathrm{Q}}_{\mathrm{C}}[[not(Q')]]_D &= N \setminus \mathcal{S}^{\mathrm{Q}}_{\mathrm{C}}[[Q']]_D\,,
\end{aligned}$$

The semantics of a predicate list P on document D, $\mathcal{S}_{\mathrm{C}}[[P]]_D : \mathcal{P}(N)$, is given by the following:

$$\begin{aligned}
\mathcal{S}_{\mathrm{C}}[[\ \texttt{[}\,Q\,\texttt{]}\ ]]_D &= \mathcal{S}^{\mathrm{Q}}_{\mathrm{C}}[[Q]]_D \\
\mathcal{S}_{\mathrm{C}}[[\varepsilon]]_D &= N
\end{aligned}$$

A location step comprises an axis, a node test, and a (possibly empty) predicate list. A location step is evaluated on a set of **context** nodes $M \subseteq T \cup E$, typically the result of the previous location step. It evaluates to the set of nodes that bear the axis relation to any context node, at which the node test and predicate list both evaluate to true.

Definition 2.2.4 *The semantics of a location step S of the form χ::τP, $\mathcal{S}_{\mathrm{C}}[[S]]_D : \mathcal{P}(N) \to \mathcal{P}(N)$, is given by:*

$$\mathcal{S}_{\mathrm{C}}[[\chi{::}\tau P]]_D\,(C) = \mathcal{S}_{\mathrm{C}}[[\chi]]_D\,(C) \cap \mathcal{S}_{\mathrm{C}}[[\tau]]_D \cap \mathcal{S}_{\mathrm{C}}[[P]]_D$$

A relative path expression consists of a sequence of one or more location steps. Evaluating a relative path expression on a set of context nodes results in a set of nodes; the location steps are evaluated, from left to right, with each of the second and subsequent steps using the previous step's result as its context.

Definition 2.2.5 *The semantics of a relative path expression R, $\mathcal{S}^{\mathrm{R}}_{\mathrm{C}}[[R]]_D : \mathcal{P}(N) \to \mathcal{P}(N)$, is given by the following, where S is a location step.*

$$\begin{aligned}\mathcal{S}^{\mathrm{R}}_{\mathrm{C}}[[S]]_D\,(N) &= \mathcal{S}_{\mathrm{C}}[[S]]_D\,(N)\\ \mathcal{S}^{\mathrm{R}}_{\mathrm{C}}[[R' \,/\, S]]_D\,(N) &= \mathcal{S}_{\mathrm{C}}[[S]]_D\,(\mathcal{S}^{\mathrm{R}}_{\mathrm{C}}[[R']]_D\,(N))\end{aligned}$$

Finally,

Definition 2.2.6 *The semantics of a core XPath expression E, $\mathcal{S}_{\mathrm{C}}[[E]]_D : \mathcal{P}(N) \to \mathcal{P}(N)$, is given by the following, where R is a relative path expression, and $\rho(D)$ is either the empty set (if N is empty) or the singleton containing the root of D (if it is not empty).*

$$\begin{aligned}\mathcal{S}_{\mathrm{C}}[[R]]_D\,(N) &= \mathcal{S}^{\mathrm{R}}_{\mathrm{C}}[[R]]_D\,(N)\\ \mathcal{S}_{\mathrm{C}}[[\,/\,R]]_D\,(N) &= \mathcal{S}^{\mathrm{R}}_{\mathrm{C}}[[R]]_D\,(\rho(D))\end{aligned}$$

The semantics of a core XPath expression E on a document D is a set of nodes in that document:

$$\mathcal{S}_{\mathrm{C}}[[E]]\,(D) = \begin{cases}\emptyset, & N = \emptyset\\ \mathcal{S}_{\mathrm{C}}[[E]]_D\,(\rho(D)), & \textit{otherwise}\end{cases}$$

If $\nu \in \mathcal{S}_{\mathrm{C}}[[E]]\,(D)$, we say that E ***matches*** *ν in D.*

Remark: Gottlob, Koch, and Pichler showed that queries in Core XPath (that is to say, computing $\mathcal{S}_{\mathrm{C}}[[P]]_D\,(M)$ for some expression P and node set M) can be evaluated in time $\mathcal{O}\left(|P|\,|N|\right)$ [31].

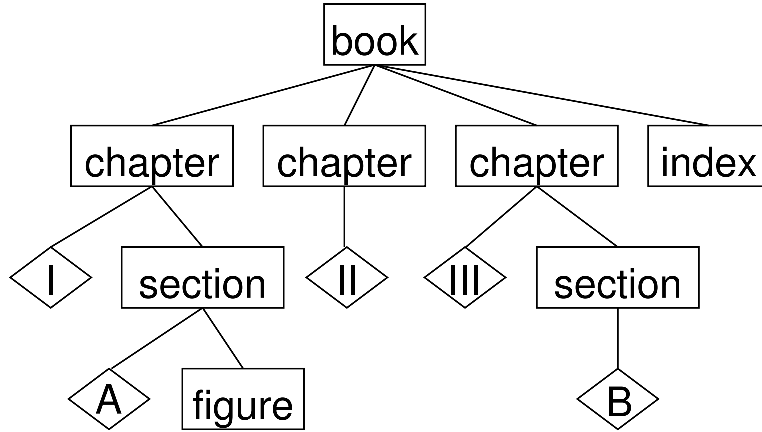

Figure 2.1: A hierarchical document.

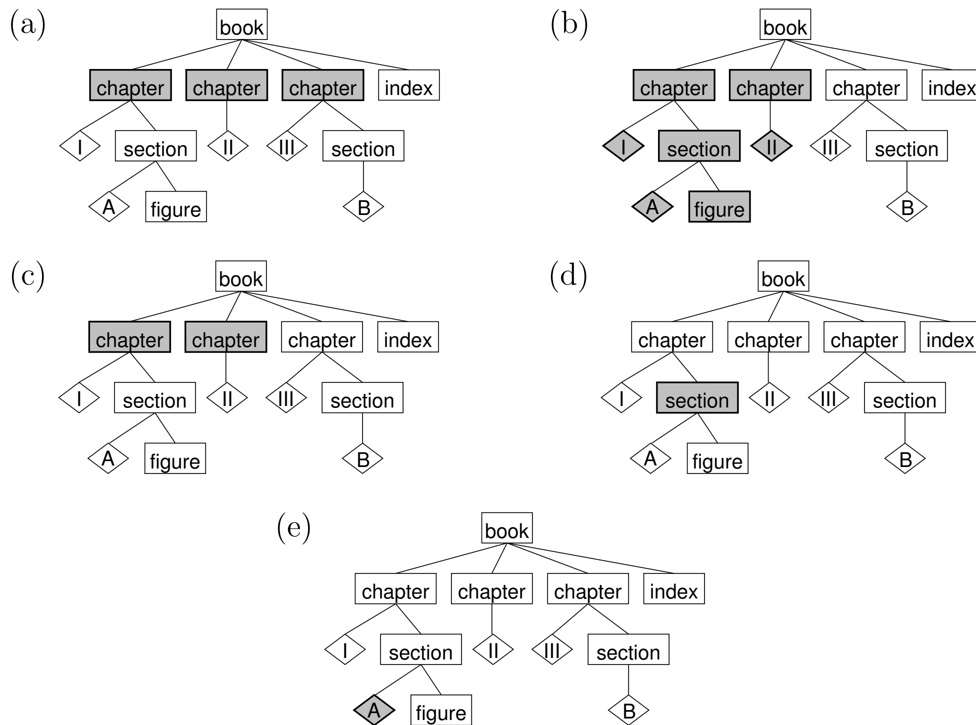

Figure 2.2: Evaluating an XPath expression on Figure 2.1.

2.2.2 XPath example

Consider the hierarchical document in Figure 2.1. In this diagram, and in subsequent diagrams of hierarchical and multihierarchical documents, we assume unless otherwise specified that edges are directed from top to bottom, and that children of a node are ordered from left to right. Rectangles represent elements, and diamonds text nodes.

In Figure 2.2 we trace the evaluation of the XPath expression

/descendant::chapter[following::chapter]/child::section/ `text()` .

The first location step descendant::chapter, evaluated in the context of the root, evaluates to the set of chapter nodes (a). The predicate [following::chapter] evaluates to the set of nodes that are followed by a chapter node in document order (b). The complete location step descendant::chapter[following::chapter] thus evaluates to the intersection of these two sets (c). The next location step, child::section, is evaluated in the context of these nodes, yielding a single section child (d). Finally, the location step `text()` is evaluated in the context of this node, yielding the single text node A (e) as the result of the full expression.

2.2.3 The language $XP^{\{[],*,//\}}$

For the purposes of access control rules (Chapter 4), we use an even simpler language than Core XPath. The $XP^{\{[],*,//\}}$ fragment of XPath consists of path expressions that always descend the tree [23, 53], with location steps containing conjunctive lists of predicates rather than arbitrary Boolean formulae.

Grammar 2.2.2 *The language* $XP^{\{[],*,//\}}$ *is given by the expression production of the following grammar, where name produces a language of element names.*

$$
\begin{array}{rcl}
\textit{expression} & \Rightarrow & \textit{relative-path} \;\mid\; \texttt{/}\,\textit{relative-path} \;\mid\; \texttt{//}\,\textit{relative-path} \\
\textit{relative-path} & \Rightarrow & \textit{location-step} \\
 & \mid & \textit{relative-path}\,\texttt{/}\,\textit{location-step} \\
 & \mid & \textit{relative-path}\,\texttt{//}\,\textit{location-step} \\
\textit{location-step} & \Rightarrow & \textit{node-test}\;\textit{predicate-list} \\
\textit{predicate-list} & \Rightarrow & \varepsilon \;\mid\; \texttt{[}\,\textit{relative-path}\,\texttt{]}\;\textit{predicate-list} \\
\textit{node-test} & \Rightarrow & \textit{name} \;\mid\; \texttt{*} \;\mid\; \texttt{node()} \;\mid\; \texttt{text()}
\end{array}
$$

We note that the $XP^{\{[],*,//\}}$ syntax is somewhat simplified and therefore not a subset of Core XPath. In particular:

- Axes are omitted; the `/` separator implies the child axis on the next location step, while `//` implies the descendant axis. It is not possible to traverse up or across the tree, only down.

- Conjunctive lists of predicates are represented sequentially, rather than as expressions using `and`. It is not possible to disjoin or negate predicates.

Nonetheless, it is possible to express the semantics of $\mathrm{XP}^{\{[],*,//\}}$ in terms of the semantics of a subset of Core XPath. We define a mapping from $\mathrm{XP}^{\{[],*,//\}}$ to the subset of Core XPath with only the `child` and `descendant` axes and only the `and` operation on predicates.

Definition 2.2.7 *The **core equivalent** of a path expression $P \in \mathrm{XP}^{\{[],*,//\}}$ is a path expression $\mathcal{E}_{\mathrm{core}}(P)$ in Core XPath given by the following rules:*

- *Node tests are identical: $\mathcal{E}_{\mathrm{core}}(\tau) = \tau$.*
- *Predicate lists are converted using an auxiliary function $\mathcal{E}_{\mathrm{core}p}$ that handles the non-empty cases.*
  - $\mathcal{E}_{\mathrm{core}}(\varepsilon) = \varepsilon$;
  - $\mathcal{E}_{\mathrm{core}}([R]P) = [\mathcal{E}_{\mathrm{core}p}([R]P)]$;
  - $\mathcal{E}_{\mathrm{core}p}([R]) = \mathcal{E}_{\mathrm{core}}(R)$
  - $\mathcal{E}_{\mathrm{core}p}([R]P) = (\mathcal{E}_{\mathrm{core}}(R)$ *and* $\mathcal{E}_{\mathrm{core}p}(P))$
- *Location steps: $\mathcal{E}_{\mathrm{core}}(\tau P) =$ `child`$::\tau\, \mathcal{E}_{\mathrm{core}}(P)$*
- *Relative paths:*
  - $\mathcal{E}_{\mathrm{core}}(S) = \mathcal{E}_{\mathrm{core}}(S)$;
  - $\mathcal{E}_{\mathrm{core}}(P/S) = \mathcal{E}_{\mathrm{core}}(P)/\,\mathcal{E}_{\mathrm{core}}(S)$;
  - $\mathcal{E}_{\mathrm{core}}(P//S) = \mathcal{E}_{\mathrm{core}}(P)/$ `descendant-or-self` :: `node()` $/\,\mathcal{E}_{\mathrm{core}}(S)$;
- *Path expressions:*
  - $\mathcal{E}_{\mathrm{core}}(R) = \mathcal{E}_{\mathrm{core}}(R)$
  - $\mathcal{E}_{\mathrm{core}}(/R) = /\,\mathcal{E}_{\mathrm{core}}(R)$
  - $\mathcal{E}_{\mathrm{core}}(//R) = /$ `descendant-or-self` :: `node()` $/\,\mathcal{E}_{\mathrm{core}}(R)$

The semantics of a path expression $P \in \mathrm{XP}^{\{[],*,//\}}$ is the function $\mathcal{S}[[P]]_D : \mathcal{P}(N) \rightarrow \mathcal{P}(N)$ given by $\mathcal{S}[[P]]_D\,(N) = \mathcal{S}_{\mathrm{C}}[[\mathcal{E}_{\mathrm{core}}(P)]]_D\,(N)$.

2.3 Updating hierarchical documents

In order to define update operations on hierarchical documents, we begin with a number of notations for representing modifications to functions or total order. We shall make heavy use of these notations when defining the semantics of the various update operations.

Definition 2.3.1 *Let f be a function from domain A to codomain B, and let x and y be arbitrary objects. The* ***extension of*** *f* ***with*** *$f(x) = y$, written $\mathrm{f}^{\mathrm{x} \to \mathrm{y}}$, is the function from $A \cup \{x\}$ to $B \cup \{y\}$ such that:*

$$f^{x \to y}(z) = \begin{cases} y, & z = x \\ f(z), & \textit{otherwise} \end{cases}$$

We write $\mathrm{f}^{\mathrm{x_0 \to y_0, \ldots, x_n \to y_n}}$ as a shorthand for $(\mathrm{f}^{\mathrm{x_0 \to y_0}})^{\mathrm{\ldots x_n \to y_n}}$.

We also define the extension of a total order, obtained by adding the members of another total order (disjoint from the first) either at the end of, or immediately preceding some item in, the first order.

Definition 2.3.2 *If R is a total order on the set A, B is a set disjoint from A, and S is a total order over B, the* ***extension of*** *R* ***by*** *S is the total order $R^{S<\infty}$ on $A \cup B$ such that $m\,(R^{S<\infty})\,n$ if and only if one of the following is true:*

- *$m, n \in A$ and $m\,R\,n$;*
- *$m, n \in B$ and $m\,S\,n$;*
- *$m \in A$, $n \in B$.*

If R is a total order on the set A, B is a set disjoint from A, S is a total order over B, and $a \in A$, the ***extension of*** *R* ***by*** *S* ***before*** *a, is the total order $R^{S<a}$ on $A \cup B$ such that $m\,(R^{S<a})\,n$ if and only if one of the following is true:*

- *$m, n \in A$ and $m\,R\,n$;*
- *$m, n \in B$ and $m\,S\,n$;*
- *$m \in B$, $n \in A$, and $a\,R\,n$;*
- *$m \in A \setminus \{a\}$, $n \in B$, and $m\,R\,a$.*

If $x \notin A$ and $p \in A \cup \{\infty\}$, we write $R^{x<p}$ as a shorthand for $R^{(x,x)p}$, where (x, x) is the (unique) total order over $\{x\}$.

Similarly, a relation (including functions and total orders) may be restricted to some subset of its domain. We define restriction slightly differently for functions and for binary relations on a single set; in the former case only the domain is restricted, while in the latter both components are restricted.

Definition 2.3.3 *If f is a function from A to B, and $C \subseteq A$, the **restriction** of f to C, $f|_C$, is $f \cap (C \times B)$.*

*If R is a binary relation on A, and $C \subseteq A$, the **restriction** of R to C, $R|_C$, is the relation $R \cap C^2$.*

Finally, a few operations may affect either elements or text nodes. We define a notation to concisely represent the changes to the sets of nodes and the text and label functions .

Definition 2.3.4 *Let $P = (\nu, \ell)$ be a pair of either an element and an element name, or a text node and a string; we do not require that ν be a node of the document. Then the extensions by P of the sets T and E of elements, and of the* text *and* label *functions, by P is:*

$$T^{(\nu,\ell)} = \begin{cases} T, & \ell \in L \\ T \cup \{\nu\}, & \ell \in \Sigma^+ \end{cases}$$

$$E^{(\nu,\ell)} = \begin{cases} E \cup \{\nu\}, & \ell \in L \\ E, & \ell \in \Sigma^+ \end{cases}$$

$$\text{text}^{(\nu,\ell)} = \begin{cases} \text{text}, & \ell \in L \\ \text{text}^{\nu \to \ell}, & \ell \in \Sigma^+; \end{cases}$$

$$\text{label}^{(\nu,\ell)} = \begin{cases} \text{label}^{\nu \to \ell}, & \ell \in L \\ \text{label}, & \ell \in \Sigma^+; \end{cases}$$

2.3.1 Insert

The **insert** operation adds nodes to a document in a specified location. We consider three forms of the insert operation, in order of increasing complexity. First, a **simple insert** adds a new leaf node to a document. This operation corresponds to the insert-before, insert-after, and append methods of DOM [4] and XUpdate [48].

Definition 2.3.5 *A **simple insert** on the document*

$$D = (\Sigma, L, T, E, A, \text{text}, \text{label}, \text{order}, \text{attribute})$$

is a tuple of the form $I = (insert, \pi, \nu, \ell)$, where:

- $\pi \in E \cup \{\top\}$ *is the node which will be the parent of the new node; if $\pi = \top$, the new node will be the root of the document.*
- ν *is either ∞ or a child of E (if $E = \top$, then $\nu = \infty$); this node will immediately follow the new node in the child order of π, or ∞ if the new node will be the last child of π.*
- $\ell \in L \cup \Sigma^+$ *is the name or text of the new node.*

*The **result** of I on D, written $I(D)$, is:*

- *If $\pi = \top$ and $E \cup T$ is nonempty, an error ($\bot$).*
- *If $\pi = \top$ and $E \cup T$ is empty, the document*

$$\left(\Sigma, L, T^P, E^P, A, \text{text}^P, \text{label}^P, \text{order}^{\chi \to \emptyset}, \text{attribute}^{\chi \to \emptyset}\right)$$

  , where χ is a new node and $P = (\chi, \ell)$.

- *If $\pi \in E$, the document*

$$\left(\Sigma, L, T^P, E^P, A \cup (\pi, \chi), \text{text}^P, \text{label}^P, \text{order}^{\chi \to \emptyset, \pi \to <_\pi^{\chi < \nu}} \text{attribute}^{\chi \to \emptyset}\right),$$

  where χ is a new node not in $T \cup E$, and $P = (\chi, \ell)$.

While a simple insert adds a single new node to a tree, a **complex insert** adds an entire subtree:

Definition 2.3.6 *A **complex insert** on the document*

$$D = (\Sigma, L, T, E, A, \text{text}, \text{label}, \text{order}, \text{attribute})$$

is a tuple of the form $(insert, \pi, \nu, D')$, where π and ν are as in a simple insert, and

$$D' = (\Sigma, L, T', E', A', \text{text}', \text{label}', \text{order}', \text{attribute}')$$

is a non-empty document over the same languages as D with the sets of nodes $T \cup E$ and $T' \cup E'$ being disjoint.

*The **result** of a complex insert* $(insert, \pi, \nu, D')$ *on document* D *is:*

- *If* $\pi = \top$ *and* $E \cup T$ *is nonempty, an error* $(\bot)$*.*
- *If* $\pi = \top$ *and* $E \cup T$ *is empty, the document* D'*.*
- *If* $\pi \in E$*, the document*

$$\big(\Sigma, L, (T \cup T'), (E \cup E'), A \cup \{(\pi, \mathrm{root}(D'))\}, \mathrm{text} \cup \mathrm{text}', \mathrm{label} \cup \mathrm{label}', \mathrm{order}^{\pi \to <_{\pi}^{\mathrm{root(D')}<\nu}} \cup \mathrm{order}', \mathrm{attribute} \cup \mathrm{attribute}'\big)$$

Finally, a **compound insert** adds a sequence of subtrees (a **document fragment**) to the location in question.

Definition 2.3.7 *A **compound insert** on the document*

$$D = (\Sigma, L, T, E, \mathrm{text}, \mathrm{label}, \mathrm{order})$$

is a tuple of the form $(insert, \pi, \nu, \mathcal{X})$*, where* π *and* ν *are as in a simple insert, and* $\mathcal{X} = \langle X_0, X_1, \ldots, X_n \rangle$ *is a sequence of non-empty documents over the same languages as D, with the node set of each* X_i *disjoint from that of* D *as well as that of each* X_j *where* $i \neq j$*.*

*The **result** of a compound insert* $I = (insert, \pi, \nu, \mathcal{X})$ *on document* D *is the composition* $I(D) = I_n(\cdots(I_1(I_0(D)))\cdots)$ *of complex inserts, where* $I_k = (insert, \pi, \nu, X_k)$ *for* $0 \leq k \leq n$*. In particular, if* $\mathcal{X}$ *is empty,* $I(D) = D$*.*

Note that if $\pi = \top$ but $\mathcal{X}$ contains more than one item, then $I_1(I_0(D))$ (and hence $I(D)$) is undefined, as I_1 attempts to insert a root node into a document that either is undefined, or already has a root node.

2.3.2 Delete

The delete operation removes a node of the document; since all paths from the root to a descendant of the deleted node must pass through the deleted node, it is necessary to remove the entire subtree rooted at that node.

Definition 2.3.8 *A **deletion** on the document* D *is a tuple of the form* $(delete, \nu)$*, where* ν *is a node of* D*. The **result** of a deletion* $X = (delete, \nu)$ *on*

$$D = (\Sigma, L, T, E, A, \mathrm{text}, \mathrm{label}, \mathrm{order})$$

is the document

$$X(D) = (\Sigma, L, T \cap N', E \cap N', A|_{N'}, \text{text}|_{N'}, \text{label}|_{N'}, \text{order'})\,,$$

where N' is the set of nodes that are neither ν nor descendants of ν, and order' *is the function:*

$$\text{order}' = \begin{cases} (\text{order}|_{\text{N'}})^{\pi \rightarrow <_\pi|_{\text{children}(\pi) \cap \text{N'}}}, & \nu \in \text{children}(\pi) \\ \emptyset, & \nu \textit{ is the root node} \end{cases}$$

One could also define simple deletions, which remove only a single node. The node in question must be a leaf node.

Definition 2.3.9 *A* ***simple deletion*** *on the document D is a deletion $(delete, \nu)$, where ν has no children in D. The result $X(D)$ of a simple deletion X on document D is its result when treated an ordinary deletion.*

2.3.3 Move

A **move** operation removes a document node ν as a child of its parent node and re-adds it in a different location of the tree. It is, conceptually speaking, the composition of an insert and a delete, though it should be considered an atomic operation. It is not possible to move a node to the root of the document, nor to move a node beneath itself or one of its own descendants.

Definition 2.3.10 *A* ***move*** *on the document D is a tuple of the form $(move, \mu, \pi, \nu)$, where μ is a node of D, π is node of D that is neither μ nor one of its descendants, and ν is either ∞ or a child of π other than μ.*

The ***result*** *of a move on D is the result of deleting ν then inserting it in the location identified by π and ν:*

$$(insert, \pi, \nu, D_\mu)\,((delete, \mu)\,(D)),$$

where D_μ is the document obtained from D by retaining only the nodes reachable from μ:

$$D_\mu = \left(\Sigma, L, T \cap N_\mu, E \cap N_\mu, A|_{N_\mu}, \text{text}|_{N_\mu}, \text{label}|_{N_\mu}, \text{order}|_{N_\mu} \text{attributes}|_{N_\mu}\right),$$

where $N_\mu = \{\mu\} \cup \text{descendants}(\mu)$.

2.3.4 Rename

The **rename** operation changes the name of an element or the text of a text node, but leaves all node relationships unchanged. A rename operation could be viewed as a sequence of an insertion (adding a node with the new label as a sibling of the target node), a number of moves (moving each child of the target node to the new node), and a deletion (removing the old, now empty, node). However, these operations entail changes to the document tree that may result in invalid or incorrect intermediate documents. Even worse, such a model would not allow renaming the root node, as it is not possible to add a sibling. In order to avoid this problem, we consider rename as a separate, atomic, operation.

Definition 2.3.11 *A* ***rename*** *on the document D is a tuple of the form $R = (rename, \nu, \ell)$, where $\nu \in T \cup E$ is a node of D; and ℓ is a name from L if $\nu \in E$, or a text string from Σ^+ if $\nu \in T$. The* ***result*** *of a rename R on the document*

$$D = (\Sigma, L, T, E, A, \text{text}, \text{label}, \text{order}, \text{attribute})$$

is the document

$$R(D) = \left(\Sigma, L, T, E, A, \text{text}^{(\nu,\ell)}, \text{label}^{(\nu,\ell)}, \text{order}, \text{attribute}\right)$$

.

2.3.5 Splice-out

While the delete operation allows removing entire subtrees of the document, sometimes a more delicate approach is required. We define a **splice-out** operation that replaces a target node with its children. It is not always possible to perform this operation on the root; if the root has more than one child, such an operation would produce a document without a unique root node; instead, we make the result of the operation undefined.

Definition 2.3.12 *A* ***splice-out*** *on the document D is a tuple of the form $X = (splice\text{-}out, \nu)$, where ν is a node of D. The* ***result*** *$X(D)$ of a splice-out X on the document*

$$D = (\Sigma, L, T, E, A, \text{text}, \text{label}, \text{order})$$

is $\perp$ (an error) if ν is the root of D and contains more than one child. Otherwise,

$$X(D) = \big(\Sigma, L, T \cap N', E \cap N', A|_{N'} \cup (\{\text{parent}(\nu)\} \times \text{children}(\nu)), \\ \text{text}|_{N'}, \text{label}|_{N'}, \text{order}', \text{attribute}|_{N'}\big),$$

where $N' = (T \cup E) \setminus \{\nu\}$ *is the set of nodes other than* ν*, and*

$$\text{order}' = \begin{cases} \left(\text{order}|_{\text{N'}}\right)^{\pi \to \left(<_{\pi}^{(<\nu)<\nu} \big|_{\text{children}(\nu) \cup \text{children}(\pi) \setminus \{\nu\}} \right)}, & \nu \leftarrow \pi \\ \text{order}|_{N'}, & \nu = \text{root}(D) \end{cases}$$

Splice-outs can usually be simulated by a sequence of moves to relocate the target node's children, followed by a deletion of the now-empty target node. However, such an implementation would not allow splicing out the root node, even when it contains one child or no children.

2.3.6 Splice-in

The inverse of the splice-out operation is **splice-in**. While the insert operation allows building a document from the top down by adding new leaf nodes and/or subtrees, splice-in builds a document from the bottom up, by adding new nodes that contain existing nodes. Specifically, a sequence of consecutive child nodes (or the root node) is replaced by a single node, with the replaced nodes becoming children of the new node. This operation corresponds to the "tagging" method of markup, where markup elements are added to an existing text [19].

Definition 2.3.13 *A* ***splice-in*** *on the document*

$$D = (\Sigma, L, T, E, A, \text{text}, \text{label}, \text{order}, \text{attribute})$$

is a tuple of the form $S = (\textit{splice-in}, \alpha, \beta, \ell)$*, where:*

- $\alpha = \beta$*, or* α *and* β *share a parent* π *with* $\alpha <_{\pi} \beta$*; these nodes will be children of the new node; and*
- $\ell \in L$ *is an element name.*

The result of a splice-in S *on the document* D *is the document*

$$\begin{aligned} S(D) = \big(&\Sigma, L, T, E \cup \{\chi\}, (A \cup (\pi, \chi) \cup (\{\chi\} \times R)) \setminus (\{\pi\} \times R), \\ &\text{text}, \text{label}^{\nu \to \ell}, \text{order}^{\nu \to <_{\pi}|_{\text{R}}, \pi \to \left(<_{\pi}^{\chi < \alpha} \big|_{\{\nu\} \cup \text{children } \pi \setminus \text{R}} \right)}, \\ &\text{attribute}^{\chi \to \emptyset} \big), \end{aligned}$$

where R *is the set of children of* π *between* α *and* β*, inclusive, in* $<_{\pi}$*; and* χ *is a new element node.*

Note that there is no overlap with insert operations: (simple) inserts add only leaf nodes, while splice-in operations always add nodes with at least one child. In particular, this fact means that splice-in operations are not sufficient to construct a tree, as they provide no means of adding a first node to the tree, nor of increasing the maximum out-degree of nodes in an existing tree. We shall see later, in Definition 3.3.5, that a variant of the splice-in operation on multihierarchical documents can allow inserting nodes anywhere in the document, including as leaves.

Copyright © Neil Moore, 2012.

Chapter 3 Multihierarchical text: encodings and models

In 1990, DeRose *et al.* introduced a model of text as an "Ordered Hierarchy of Content Objects" (OHCO), a model that is now reflected in XML. Even as early as this paper, however, it was clear that the OHCO model was insufficient for some purposes, as "many documents have more than one useful structure" [22]. The problem of modelling and encoding documents with such **multiple hierarchies** has, in the two decades since the DeRose paper, seen many often incompatible solutions.

A few years after the OHCO paper, several of its authors published an article [58] arguing that the notion of text as an ordered hierarchy is flawed. The authors suggest a series of extensions of the OHCO thesis that account for some of the difficulties presented by multiple hierarchies:

> **OHCO**: Text is an ordered hierarchy of content objects. [22]
>
> **OHCO-2**: An analytical perspective on a text determines an ordered hierarchy of content objects. [58]
>
> **OHCO-3**: Analytical perspectives can be decomposed into hierarchies: for every distinct pair of objects x and y that overlap in the structure determined by some perspective $P(1)$, there exist diverse perspectives $P(2)$ and $P(3)$ such that $P(2)$ and $P(3)$ are sub-perspectives of $P(1)$ and x is an object in $P(2)$ and not in $P(3)$ and y is an object in $P(3)$ and not in $P(2)$. [58]

Unfortunately, as the authors note, each of these theses itself has flaws. OHCO-2 claims, essentially, that a document can be treated as a collection of hierarchical views, each associated with a particular analytical perspective. However, the notion of enjambment, wherein a syntactic structure spans portions of two or more lines of verse, reflects an inherently non-hierarchical perspective of text. As a simple example, consider the first two lines of T.S. Eliot's famous poem *The Waste Land*:

```
<l>April is the cruellest month, <cl>breeding</l>
<l>Lilacs out of the dead land</cl>, mixing</l>
```

The verse structure, represented by `<l>` tags for verse lines, forms a hierarchy, as does the syntactic structure, here represented by `<cl>` tags delimiting clauses.

However, treating these two hierarchies as entirely separate views of the document limits the analysis of enjambment, which as we see is inherently non-hierarchical.

The OHCO-3 thesis aims to address this issue by declaring the verse and syntactic structures to be sub-perspectives of a unified "literary studies" perspective. However, other situations such as self-overlap (discussed in Section 3.1.1) challenge even this model of text, by providing examples of non-nesting overlap that cannot reasonably be divided into sub-perspectives.

We first discuss methods of encoding documents with multiple hierarchies, including both XML and XML-like languages; followed by data models for such documents. Finally, we describe our model, the **globally ordered GODDAG**, which brings together a number of the existing data models.

3.1 Encoding multiple hierarchies

Although XML documents are modeled as trees, documents are usually stored, transferred, and often even edited, in their serialized form: strings of characters containing tags and text. The hierarchical form of XML documents corresponds to certain constraints on the components of these strings, in particular the nesting of tags. By relaxing these constraints, or by using alternative serializations that lack them, it is possible to produce a markup language that can represent at least some kinds of overlapping features. Alternatively, overlapping features can be encoded in ordinary hierarchical XML documents by a variety of means, although ordinary XML processing tools can typically manipulate the multiple hierarchies only indirectly.

Of course, even if we have a serialized representation for multihierarchical documents, processing and reasoning about them requires a model. In XML, this representation is the tree structure of the Document Object Model (DOM) [4], but non-nested overlap results in documents that cannot be represented as trees. We shall consider a number of such models later, in Section 3.2. The encodings described in the present section may be used to store and exchange documents belonging to many models. They can therefore, with certain caveats we address later, also serve as a medium for converting among those models.

3.1.1 Non-nested tags

Recall from Section 2.1.1 that a hierarchical document may be serialized, as in XML, as a string consisting of text, start tags, and end tags; start tags and end tags appear in pairs representing elements, with (the serialization of) an element's content

appearing between its tags. Naturally, a start tag precedes its corresponding end tag. Furthermore, the hierarchical nature of XML results in tags being **nested**: if s_1, e_1 are the start and end tags of element ν_1, and s_2, e_2 of ν_2, then they appear in one of the following orders:

- $s_1 e_1 s_2 e_2$ (ν_1 precedes ν_2)
- $s_2 e_2 s_1 e_1$ (ν_1 follows ν_2)
- $s_1 s_2 e_2 e_1$ (ν_1 is an ancestor of ν_2)
- $s_2 s_1 e_1 e_2$ (ν_1 is a descendant of ν_2)

In particular, $s_1 s_2 e_1 e_2$ and $s_2 s_1 e_2 e_1$ are disallowed.

In some ways, the simplest approach to the problem of encoding multiple hierarchies is to remove this nesting constraint, while maintaining the requirement that tags occur in pairs with the start tag following the end tag. We call the resulting superset of XML **pseudo-XML** [19]. As with XML, an element's content is the set of nodes that occur between its start and end tags. Unlike XML, it is possible for only part of an element B (for example, its start tag) to occur between the tags of element A; in this case we do not say that B is part of A's content, though they may in fact share content; instead we say that A and B **overlap**.[1] Of course, it is not possible to use standard XML-processing tools with such documents, because conforming XML-processing tools are required to reject documents with improperly nested tags [9]. However, approaches such as just-in-time trees [25] allow reconstructing hierarchies at the time of document processing rather than parsing, allowing XML-based tools to be used for processing the resulting hierarchies.

Since pseudo-XML documents are not processed directly by standard XML tools, it is not necessary to retain the same syntax as XML. For example, the TexMECS language [37] uses tags of the form "<`start`|" and "|`end`>," as well as additional syntax to encode features that represent discontinuous [62] and unordered elements. Another language, the Layered Markup and Annotation Language (LMNL) [66], uses tags of the form "{`start`|" and "{`end`|" instead, and furthermore allows annotations (the equivalent of attributes) to themselves contain hierarchical structures. At least some extensions to the XML syntax are necessary to represent certain kinds of overlap with non-nested tags. For example, if an element overlaps another element with the

[1] Though one could take the term "overlap" to include case where two elements share content, including when one element is nested within another, we do not do so. We instead use the term "intersect" or "intersection" when we wish to include both nesting and proper overlap.

same name—so-called **self-overlap**—straightforward encodings such as pseudo-XML are likely to be ambiguous.

(a) ~~No longer necessary~~
(b) ~~No longer necessary~~

Figure 3.1: Self-overlapping and nested strike-through. In the absence of special support for self-overlap, (a) and (b) would have the same serialization "<strike>no <strike>longer</strike> necessary</strike>".

For example, to represent the text "no longer necessary" with the substrings "no longer" and "longer necessary" separately stricken out (Figure 3.1(a)), we would expect "<strike>no <strike>longer</strike> necessary </strike>". However, this string is identical to the serialization of a text where "no longer necessary" and "longer" are stricken (Figure 3.1(b)). Alternative syntaxes can help solve this problem, by adding an additional identifier to the representation of both start and end tags of self-overlapping elements; a start tag then only corresponds to the end tags with a matching identifier. For example, in TexMECS [37], tags may have a suffix separated from the element name by a tilde; tags only match (forming an element) if the start and end tags have the same suffix. Suffixes in TexMECS need not be unique; rather, tags with the same suffix (including no suffix) are assumed to be nested with respect to one another; so in TexMECS, the first situation could be encoded as "<strike|no <strike~1|longer|strike> necessary |strike~1>," while the second could be simply "<strike|no <strike|longer|strike> necessary |strike>", or for more clarity "<strike|no <strike~1|longer|strike~1> necessary |strike>".

Another problematic situation, present in pseudo-XML as well as both TexMECS and LMNL, is **spurious overlap**: the representation of two elements as overlapping when they could equivalently be represented as non-overlapping. Sperberg-McQueen and Huitfeldt [61] define spurious overlap to occur when two elements overlap, but either the overlapping region of the document or the region of the document exclusive to one of the elements is empty. In the former case (Figure 3.2(a)), the same contain-

(a) <link>edit<suffix></link>ed</suffix>
(b) <link>edit</link><suffix>ed</suffix>

Figure 3.2: Spurious overlap. (a) Elements link and suffix spuriously overlap. (b) A non-overlapping document fragment with the same containment relations.

(a) `<link><verb>edit</link></verb>`
(b) `<link><verb>edit</verb></link>`

Figure 3.3: Complete overlap. (a) Elements link and verb overlap but have exactly the same content. (b) Removing the overlap introduces a containment relationship between link and verb.

ment relations can be represented with two adjacent but non-overlapping elements (Figure 3.2(b)). The latter case (Figure 3.3(a)) is more difficult, however, as the "obvious" solution of nesting one element within the other (Figure 3.3(b)) changes the containment relations among elements, though not between elements and text nodes. In fact, it is not clear that this case is spurious at all: it may be important to represent two or more elements that have the same content, without claiming that any of the elements is in fact nested within another [52]. Therefore, we use the term to refer only to the first situation.

Definition 3.1.1 *Two elements e_1 and e_2 in a tag-based serialization of a multihierarchical document **spuriously overlap** if the start tag for e_2 immediately precedes the end tag for e_1, or if the start tag for e_1 immediately precedes the end tag for e_2.*

We return to the topic of spurious overlap in Section 3.3, where we consider a model of markup with a serialization that avoids such spurious overlap.

3.1.2 Encoding multiple hierarchies in XML

Although enhanced markup languages, such as those discussed in the previous section, can effectively represent many kinds of multihierarchical structures, they have a significant disadvantage in that documents encoded in such languages cannot be processed by the wealth of existing tools based on XML. The Text Encoding Initiative guidelines [65] suggest four methods of representing multihierarchical documents within the XML framework [64].

Multiple encoding

By far the simplest method of encoding multiple hierarchies is to encode each hierarchy separately. The document, then, contains one or more "root" elements, one per hierarchy, along with an actual root element containing all these hierarchy roots. The textual content of each hierarchy is the same, though it may be divided differently into text nodes.

```
<poem>
<lg><l>April is the cruellest month, breeding</l>
  <l>Lilacs out of the dead land, mixing</l>
  <l>Memory and desire, stirring</l>
  <l>Dull roots with spring rain.</l>
</lg>
<seg>April is the cruellest month, <cl>breeding
  Lilacs out of the dead land</cl>, <cl>mixing
  Memory and desire</cl>, <cl>stirring
  Dull roots with spring rain</cl>.
</seg>
...
</poem>
```

Figure 3.4: Multiple encoding of multiple hierarchies

Encoding multiple hierarchies in this way has a number of disadvantages. The duplication of text, once per hierarchy, invites inconsistency when the document is updated. Any change made to the text in one tree (including deleting elements in that tree) must be replicated in each other tree; failure to do so results in hierarchies that describe different documents. Worse, cross-hierarchy queries, such as searches for instances of enjambment or even for words occurring within lines of poetry, are difficult or impossible in this representation.

One possible XPath extension to help support such queries is an axis that navigates from text nodes in one hierarchy to the corresponding text nodes in other hierarchies. However, because each hierarchy divides text into text nodes in its own way, such an axis is not usually be one-to-one, even when restricted to two hierarchies. For example, in Figure 3.4, the text node "breeding Lilacs out of the dead land" from the second (red) hierarchy corresponds to two text nodes from the first (blue) hierarchy: "April is..." and "Lilacs out..."; each of these nodes, in turn, corresponds to two or three text nodes from the second hierarchy. Some progress is possible by implicitly fragmenting text nodes so as to contain only sequences of characters that appear contiguously, with no intervening tags, in *all* hierarchies; then text nodes may be placed in one-to-one correspondence with those from another hierarchy.

Fragmentation

The **fragmentation** of text nodes suggested above can be extended to elements as well. Doing so makes it possible to represent the underlying text only once, with markup elements from all the hierarchies interspersed among one another. Essentially,

one dominant hierarchy is encoded normally, while elements of the other hierarchy are divided as necessary to allow their parts to nest within elements of the first hierarchy.

```
<poem>
<lg><seg><l>April is the cruellest month, <cl
part="I">breeding</cl></l>
  <l><cl part="F">Lilacs out of the dead land</cl>,
        <cl part="I">mixing</cl></l>
  <l><cl part="F">Memory and desire</cl>,
        <cl part="I">stirring</cl></l>
  <l><cl part="F">Dull roots with spring rain</cl>.</l></seg>
</lg> ...
</poem>
```

Figure 3.5: TEI fragmentation encoding of multiple hierarchies, with (I)nitial, (M)edial, and (F)inal fragments.

Users of this encoding must take care when fragmenting elements. With text nodes, fragmentation could be applied arbitrarily without changing the appearance or meaning of the document; two consecutive text nodes are encoded and interpreted no differently from a single text node containing the concatenation of their text. The same can be true for some kinds of elements, particularly those such as "in an italic typeface" with what we might call intensive semantics; that is, those that express that their content has some property. However, many if not most types of elements have extensive semantics, indicating that their content forms some particular object or quantity. For example, in Figure 3.5, the first line of the poem contains a `<cl>` element (ostensibly representing a clause) that contains only the text "breeding"; this text, however, is not a clause, but only part of one. In order to maintain the proper sense of elements such as `<cl>`, the fragmented elements must somehow be distinguished from stand-alone elements; and the fragments must somehow be associated with one another so that they can be (at least conceptually) reassembled into coherent units.

The TEI guidelines provide two ways of labelling fragments to support such **virtual joins**. The simpler, but less widely applicable, method is demonstrated in Figure 3.5: certain TEI elements may contain a `part` attribute that indicates the element represents only the (I)nitial, a (M)edial, or the (F)inal portion of a feature. However, this method is not supported for element types that may self-nest or self-overlap, because in such a situation fragments of a virtual element may appear between the initial and final fragments of another virtual element of the same type (i.e. with the same element name).

```
<poem>
<lg><seg><l>April is the cruellest month, <cl xml:id="cl01"
next="#cl02">breeding</cl></l>
  <l><cl xml:id="cl02" prev="#cl01">Lilacs out of the dead land</cl>,
        <cl xml:id="cl03" next="#cl04">mixing</cl></l>
  <l><cl xml:id="cl04" prev="#cl03">Memory and desire</cl>,
        <cl xml:id="cl05" next="#cl06">stirring</cl></l>
  <l><cl xml:id="cl06" prev="#cl05">Dull roots with spring
rain</cl>.</l></seg>
</lg> ...
</poem>
```

Figure 3.6: TEI fragmentation encoding of multiple hierarchies, with chaining

More general situations may be represented by **chaining**, demonstrated in Figure 3.6. In this method of encoding virtual joins, each fragment has a unique identifier as well as references to the identifier of the previous and next fragments of the virtual element. The virtual element may then be reconstructed by following the chain of `next` pointers from the first element. Other, less direct but just as general, methods of expressing virtual joins are discussed in Section 3.1.2.

Cross-hierarchy queries are much simpler with fragmentation encoding than with multiple encoding. For example, if the verse hierarchy is primary and the syntactic hierarchy secondary, instances of enjambment may be found by searching for clause fragments that occur within a different line from the preceding fragment of the same clause. However, XPath alone is not sufficient for such a query, because it is not in general possible to express both that the two lines are in fact distinct and that the two fragments belong to the same virtual element.

Milestones

Milestone elements or **boundary marking** is a third, and one of the most popular [21], means of representing multiple hierarchies in XML. As with fragmentation, one hierarchy is typically encoded normally. Features in other hierarchies are then represented by empty elements called **milestones** representing the beginning and end of each element. In fact, it is possible to avoid privileging one hierarchy by using milestone encodings for all hierarchies.

Milestone encodings, like fragmentation encodings, can be made to support self-overlap. As with non-XML encodings such as TexMECS [37], milestone encodings can disambiguate self-overlapping markup by adding to each milestone element an

```
<poem>
<lg><anchor type="delimiter" subtype="sentenceStart"/>
<l>April is the cruellest month,
    <anchor type="delimiter" subtype="clStart"/>breeding</l>
<l>Lilacs out of the dead land<anchor type="delimiter" subtype="clEnd"/>,
    <anchor type="delimiter" subtype="clStart"/>mixing</l>
<l>Memory and desire<anchor type="delimiter" subtype="clEnd"/>,
    <anchor type="delimiter" subtype="clStart"/>stirring</l>
<l>Dull roots with spring rain<anchor type="delimiter" subtype="clEnd"/>.
</l> <anchor type="delimiter" subtype="sentenceEnd"/></lg> ...
</poem>
```

Figure 3.7: Milestone encoding of multiple hierarchies

attribute that uniquely identifies its matching milestone. For example, the trojan milestone approach, used in the Canonical LMNL in XML (CLIX) serialization of LMNL [21], uses attributes `sID` and `eID` to identify start and end milestones, respectively. Each milestone element has exactly one of these attributes, and every value of such an attribute occurs exactly twice: once as the value of `sID`, and once as the value of `eID`. A pair of milestones represents a virtual markup element if and only if the `sID` of the first is equal to the `eID` of the second.

The milestone approach is conceptually very similar to pseudo-XML, adapted to produce well formed XML documents by replacing the improperly nested tags with an entire element representing each tag. It is therefore straightforward to convert between pseudo-XML and related languages and milestone encodings; in fact, particularly if milestones are used uniformly across hierarchies, the conversion can be done entirely at the lexical level, without regard for the large-scale structure of the document.

Standoff markup

Standoff markup is the fourth and final method proposed in the TEI guidelines for encoding multiple hierarchies. A single hierarchy of the document is selected as the **link base**. Other hierarchies may be specified elsewhere in the document, or even in other documents; they contain, rather than text, pointers to ranges of elements or of text characters in the first hierarchy, using a notation such as XPointer [32]. In some models, such as XStandoff [63], the link base may be just the text of the document, with each hierarchy containing pointers to ranges of characters in the text.

Standoff markup combines features of fragmentation and of multiple encoding. As with fragmentation, elements from hierarchies other than the link base can be thought of a virtual joins of ranges of elements from a single hierarchy. As with multiple encoding, hierarchies other than the link base are stored separately from it, with elements nested as in an ordinary single-hierarchy document without interference from other markup layers.

Although standoff markup avoids duplicating text, it still requires considerable work to synchronize the representations of the various hierarchies. If a node in the link base is deleted or moved, it is necessary to update every range that has that node as an endpoint. Likewise, if pointers reference ranges of text characters rather than of elements, modifying the text is likely to require updating a large number of pointers (in an implementation using integer character indices, potentially every character range ending after the modified text).

3.2 Models of multihierarchical documents

3.2.1 Concurrent hierarchies

One model of multihierarchical markup, **concurrent hierarchies**, represents documents as a collection of markup hierarchies over the same underlying text. This model reflects the OHCO-2 or OHCO-3 thesis, that analytical perspectives correspond to or may be decomposed into hierarchies.

```
<poem>
<(V)lg>
<(S)seg><(V)l>April is the cruellest month, <(S)cl>breeding</(V)l>
  <(V)l>Lilacs out of the dead land</(S)cl>, <(S)cl>mixing</(S)(V)l>
  <(V)l>Memory and desire</(S)cl>,<(S)cl>stirring</(V)l>
  <(V)l>Dull roots with spring rain</(S)cl>.</(V)l></(S)seg>
</(V)lg> ...
</poem>
```

Figure 3.8: CONCUR encoding of multiple hierarchies: `S` (syntax) and `V` (verse structure)

XML's predecessor SGML has, with its CONCUR feature, supported concurrent hierarchies from the outset [39]. CONCUR allows markup tags to be annotated with the name of the schema (document type definition) or instance of a schema to which they belong (Figure 3.8). When constructing a tree for the document, only one schema instance is used, with tags belonging to other schemata and other instances

of the same schema being ignored. This method allows a single SGML document to represent concurrent hierarchies over the document text, with one schema instance for each hierarchy. However, CONCUR is an optional feature, not supported by all or even most SGML processing systems [58], and the SGML developers have argued against its use for representing "multiple logical views of a document" [30]. CONCUR has never been included in XML, although there exist XML extensions such as MULAX [35] that add a CONCUR feature.

Other models of concurrent hierarchies include distributed XML [16]. In this model, a multihierarchical document is a collection of XML documents, each associated with a schema, such that the documents

- share a single root node; and
- have the same underlying text.

In a document with concurrent hierarchies, relationships among nodes in the same hierarchy may be expressed in the ordinary language of XML. Relationships among nodes from separate hierarchies are more complex, but may be described in terms of the (shared) text content of the nodes [21, 20], allowing for cross-hierarchy queries. For example, a node A may be said to contain a node B from a different hierarchy if every text node descendant of B is also a descendant of A.

Documents with concurrent hierarchies may be represented as graphs [38], in particular using the generalized ordered-descendant directed acyclic graph (GODDAG) structure of Sperberg-McQueen and Huitfeldt [61], described in detail in Section 3.2.3. However, concurrent hierarchies do not exhibit the full generalities of multihierarchical markup. In particular, self-overlap is difficult to represent: two overlapping elements must be assigned to different schema instances, even if they share the same name and should therefore logically belong to the same schema. This problem ultimately reflects the limitations of the OHCO-3 model: some annotation perspectives cannot logically be divided into hierarchies.

3.2.2 Overlapping ranges

One way to model more general forms of multihierarchical markup is to return to the serialization, based on non-nested tags, described in Section 3.1.1. This serialization, where elements contain the objects (elements or text nodes) that fall between a pair of start and end tags, immediately suggests a model of markup in terms of overlapping intervals in some coordinate system over the document's text.

A number of coordinate systems are possible. One possibility, used in the core range algebra formalism [57], is to assign an integer position to each character in the document. Markup elements are then associated with intervals of integers, with a range containing a character if the interval contains that character's position. There are difficulties, however, in using integer ranges of characters to represent some forms of markup. In particular, such a representation of element containment does not distinguish between two elements containing the same text, even though one element might contain the other. Likewise, it is difficult to represent elements with no text content, which occur between consecutive text nodes and thus correspond to empty intervals at different positions in the document. Particularly difficult are subtrees of elements containing no characters; in this case, every element in the subtree corresponds to the same empty interval, and the relationships of the elements are lost.

One solution to these problems is to represent ranges as intervals of real numbers, with characters continuing to corresponding to integers. Then two elements may be represented by intervals with distinct endpoints, even if they contain the same characters, allowing disambiguation of their nesting (and order) relationships. Likewise, elements between two consecutive characters may be represented by subintervals of the open interval between characters; since such a nonempty open interval is homeomorphic to the real line, any relationships that may be expressed among elements with content may also be expressed among such elements.

Alternatively, integer positions may be assigned in sequence to lexical tokens of the document (start tags, end tags, and contiguous sequences of text), with an element's range being the open interval of the integers between the positions of its start and end tags. Since each element's range has endpoints distinct from those of every other element, the relationships among elements, even those containing no text, is always unambiguous.

Finally, for text associated with a digitized image, for example in an image-based electronic edition [19], it may be possible to use pixel or sub-pixel coordinates within the image [18]. One advantage of this method is that is allows for visualization of markup directly on the image whose features that markup describes [41, 42]. However, effort is required to define mappings between image regions and lines of text, in order to create a linear coordinate system for markup; and between pixel coordinates and characters, so that those characters may be put in relationship with the markup ranges [18].

3.2.3 Ordered DAGs

Rather than using XML's serialized form as the basis for a model of multiple hierarchies, we may begin with the tree structure of XML. Generalizing this tree structure to allow directed acyclic graphs yields an initially straightforward model of multihierarchical markup, general enough to represent many types of structures, including distributed markup.

Definition 3.2.1 *An* ***ordered directed acyclic graph (ordered DAG)*** *is a tuple* (V, A, order)*, where* V *is a set of nodes,* $A \subset V^2$ *is an acyclic relation on nodes known as the* ***arc*** *or* ***child*** *relation, and* $\mathrm{order} : V \to \mathcal{P}(V^2)$ *is a function associating with each node of* V *a strict total order on the children of that node.*

Ordered trees, such as those of our hierarchical document model, are special cases of ordered DAGs containing a single root node ρ with exactly one path from ρ to each node. Extending our definition of hierarchical documents (Definition 2.1.3) to ordered DAGs is straightforward; the only change is to loosen the restriction on the set A of arcs in the document's graph, and to account for the potential presence of multiple root nodes.

Definition 3.2.2 *A* ***ordered DAG document*** *is a tuple*

$$D = (\Sigma, L, T, E, A, \mathrm{text}, \mathrm{label}, \mathrm{order}, \mathrm{attribute}),$$

where:

- Σ *is the* ***text alphabet****;*
- L *is a set of element and attribute* ***labels****;*
- T *is a set of* ***terminal*** *or* ***text*** *nodes;*
- E *is a set of* ***nonterminal*** *or* ***element*** *nodes, with* $T \cap E = \emptyset$*;*
- $A \subset E \times (T \cup E)$ *is a set of arcs from elements to nodes such that* $(T \cup E, A)$ *is an acyclic directed graph;*
- text *is a function from* T *to* Σ^+*;*
- label *is a function from* E *to* L*;*
- order *is a function from nodes of* $T \cup E$ *to strict total orders over those nodes' children in* A*; and*

- attribute *is a function mapping elements in* E *to partial functions from attribute keys in* L *to values in* Σ^*.

We write roots(D) *for the set of nodes in* $T \cup E$ *that do not have parents; this set is empty if and only if* $T \cup E$ *is empty.*

The ordered DAG associated with a document D *is* $\text{odag}(D) = (T \cup E, A, \text{order})$.

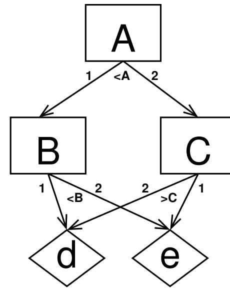

Figure 3.9: Inconsistent sibling axes in an ordered DAG. Numbers on edges represent the child order at the parent node. The node e is both a following-sibling and a preceding-sibling of d.

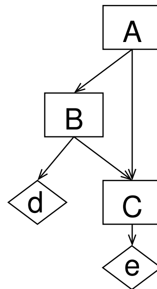

Figure 3.10: Intersecting axes in an ordered DAG. The node C is both a child and a following-sibling of B.

Axes similar to those of standard XPath may be defined for ordered DAGs, but the more general document structure requires some compromise. Most importantly, there is no unique way to order nodes; even if two nodes share a common ancestor, different paths from that ancestor to those nodes may suggest different orders. As a result, the following and preceding axes (and likewise following-sibling and preceding-sibling are no longer disjoint with one another (Figure 3.9), nor with the child and parent axes (Figure 3.10). In fact, under the definition of the follows relation we used for hierarchical documents (Definition 2.1.5), a node would follow itself whenever it has two ancestors that share a common parent (Figure 3.11). For these reasons, we

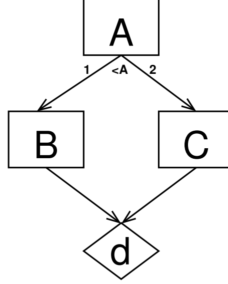

Figure 3.11: Self-following in an ordered DAG. Under the tree-based definition of preceding and following nodes, the node d follows itself, since its ancestor C is a following-sibling of its ancestor B.

temporarily put aside consideration of these axes until we more thoroughly explore the notion of order on multihierarchical documents.

Definition 3.2.3 *An* ***unordered axis*** *on an ordered DAG is one of the following relationships on the set of nodes:*

- *self* $= \{(\nu, \nu) | \; \nu \in N\}$
- *parent* $= \{(\mu, \nu) | \; \mu \to \nu\} = A$
- *child* $= \{(\mu, \nu) | \; \nu \to \mu\} = A^{-1}$
- *ancestor-or-self* $= \{(\mu, \nu) | \; \mu \to^{*} \nu\}$
- *descendant-or-self* $= \{(\mu, \nu) | \; \nu \to^{*} \mu\}$
- *ancestor* $= \{(\mu, \nu) | \; \mu \to^{+} \nu\}$
- *descendant* $= \{(\mu, \nu) | \; \nu \to^{+} \mu\}$

The inconsistencies inherent in extending these originally tree-based relations to a more general document structure lead to a desire for a more restricted form of multihierarchical document, still generalizing XML while preserving more properties of the axis relations. One DAG-based model that aims to address these concerns is the generalized ordered-descendant directed acyclic graph (GODDAG) of Sperberg-McQueen and Huitfeldt [61].

In its most general form, a GODDAG is simply an ordered DAG document with the additional restriction that the child relation be antitransitive (no node dominates another node both directly and indirectly). This restriction eliminates cases such as Figure 3.10 where two nodes have both a parent-child and a sibling relationship;

and furthermore ensures that the child axis can be reconstructed from the descendant relation by taking its transitive reduction. In the original definition of GODDAG, leaf and non-leaf nodes are strictly separated, with leaves always corresponding to text nodes, and otherwise empty elements containing an empty text node. In order to avoid the necessity of empty text nodes, we instead require only that text nodes be leaves (and not that leaves be text nodes).

Definition 3.2.4 *A **GODDAG** is an ordered DAG document*

$$(\Sigma, L, T, E, A, \text{text}, \text{label}, \text{order}, \text{attribute})$$

such that A is antitransitive: if $(a_0, a_1), (a_1, a_2), \ldots, (a_{n-1}, a_n) \in A$, with $(n > 1)$, then $(a_0, a_n) \notin A$.

Since the GODDAG structure still permits inconsistent orders (such as in Figure 3.9), Sperberg-McQueen and Huitfeldt furthermore defined the **restricted GODDAG** [61], which replaces the child order with a total order on the set of leaf nodes, along with two constraints on the sets of leaf nodes contained within elements. We begin with a slightly relaxed version of the restricted GODDAG that has only one of these constraints. This definition is very similar to the definition of ordered DAG documents (Definition 3.2.2), with the order function replaced with a total order $<_T$ on the leaf nodes.

Definition 3.2.5 *A **semi-restricted GODDAG** is a tuple*

$$D = (\Sigma, L, T, E, A, <_T, \text{text}, \text{label}, \text{attribute}),$$

where:

- *Σ is the **text alphabet**;*
- *L is a set of element and attribute **labels**;*
- *T is a set of **terminal** or **text** nodes;*
- *E is a set of **nonterminal** or **element** nodes, with $T \cap E = \emptyset$;*
- *$A \subset E \times (T \cup E)$ is a set of arcs from elements to nodes such that $(T \cup E, A)$ is an acyclic directed graph;*
- *$<_T \subset T^2$ is a strict total order, the **leaf order**, on T;*

- text *is a function from* T *to* Σ^+*;*

- label *is a function from* E *to* L*; and*

- attribute *is a function mapping elements in* E *to partial functions from attribute keys in* L *to values in* Σ^**;*

that furthermore satisfies the ***leaf contiguity*** *constraint: If* $\nu \in E$ *and* $\alpha <_T \beta <_T \gamma \in T$*, and* α *and* γ *are descendants of* ν *in* A*, then* β *is also a descendant of* ν*.*

The set of leaves reachable from a node $\nu \in T \cup E$ *is called the* ***frontier*** *of* ν*, denoted by* $\mathsf{frontier}(\nu)$*.*

The restricted GODDAG further adds a uniqueness constraint. The strict separation of leaf and internal nodes, which we disregarded in our description of the GODDAG, also becomes important for the properties of the restricted GODDAG, so we include it here.

Definition 3.2.6 *A* ***restricted GODDAG*** *is a semi-restricted GODDAG that also satisfied the constraints:*

1. *(Uniqueness) No two elements have the same frontier. For any two nodes* $\mu, \nu \in E$*, at least one of* $\mathsf{frontier}(\mu) \setminus \mathsf{frontier}(\nu)$ *and* $\mathsf{frontier}(\nu) \setminus \mathsf{frontier}(\mu)$ *is nonempty.*

2. *(Leaves) No element is a leaf node.*

The leaf order and the uniqueness criterion allow us to define a more consistent document order.

Definition 3.2.7 *Let* A *and* B *be two nodes of a restricted GODDAG* D*. We say that* A ***precedes*** B *(*$A <_D B$*) if* $\mathsf{frontier}(A) \setminus \mathsf{frontier}(B)$ *contains a leaf that precedes in* $<_T$ *every leaf in* $\mathsf{frontier}(B) \setminus \mathsf{frontier}(A)$*, and* $\mathsf{frontier}(B) \setminus \mathsf{frontier}(A)$ *contains a leaf that follows in* $<_T$ *every leaf in* $\mathsf{frontier}(A) \setminus \mathsf{frontier}(B)$*.*

This definition gives us a variant of the orthogonality property of hierarchical documents (Lemma 2.1.1).

Lemma 3.2.1 *Let* A *and* B *be two distinct nodes of a restricted GODDAG* D*. Then exactly one of the following is true:* $A <_D B$*;* $B <_D A$*;* A *dominates every leaf node that* B *dominates; or* B *dominates every leaf node that* A *dominates.*

PROOF: Because of the uniqueness constraint, either or both of $F_A = \mathsf{frontier}(A) \setminus \mathsf{frontier}(B)$ and $F_B = \mathsf{frontier}(B) \setminus \mathsf{frontier}(A)$ are non-empty.

If F_A is empty, then B dominates every node that A dominates; if F_B is empty, then A dominates every node that B dominates. In neither case can A precede B, as doing so requires that both F_A and F_B be non-empty. Furthermore, by the uniqueness constraint F_A and F_B cannot both be empty.

Now consider the case where F_A and F_B are both nonempty; we must show that either A precedes B or B precedes A. By the definition of precedence, this case is equivalent to stating that the smallest and largest members of $F_A \cup F_B$ do not appear in the same set. But if they did both occur in A, then by contiguity A would dominate every leaf node that B dominates, and F_B would be empty. Hence either the smallest element is in A and the largest in B, in which case A precedes B; or vice versa, with B preceding A. □

Furthermore, the $<_D$ relation allows us to define the missing XPath axes for restricted GODDAGs. Overlapping nodes are those that share descendants without themselves having an ancestor/descendant relationship.

Definition 3.2.8 *An axis on a restricted GODDAG D is one of the following:*

- *One of the unordered axes (Definition 3.2.3); none of these definitions refers to the child order of an ordered DAG, so each may be applied to the restricted GODDAG unchanged.*
- $\textsf{\textit{following}} = \{(\mu, \nu) |\ \mu >_D \nu\}$
- $\textsf{\textit{preceding}} = \{(\mu, \nu) |\ \mu <_D \nu\}$
- $\textsf{\textit{following-sibling}} = \{(\mu, \nu) \in \textsf{\textit{following}} |\ \mu$ *and* ν *have a parent in common*$\}$
- $\textsf{\textit{preceding-sibling}} = \{(\mu, \nu) \in \textsf{\textit{preceding}} |\ \mu$ *and* ν *have a parent in common*$\}$
- $\textsf{\textit{following-overlap}} = \{(\mu, \nu) \in \textsf{\textit{following}} |\ \mu$ *and* ν *have a descendant in common*$\}$
- $\textsf{\textit{preceding-overlap}} = \{(\mu, \nu) \in \textsf{\textit{preceding}} |\ \mu$ *and* ν *have a descendant in common*$\}$
- $\textsf{\textit{following-strict}} = \textsf{\textit{following}} \setminus \textsf{\textit{following-overlap}}$
- $\textsf{\textit{preceding-strict}} = \textsf{\textit{preceding}} \setminus \textsf{\textit{preceding-overlap}}$

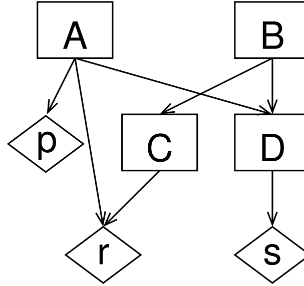

Figure 3.12: Containment relationships in a restricted GODDAG. Although A contains every leaf node that C contains, it does not contain C itself.

Note that the condition in Theorem 3.2.1 is not that A dominates B, but rather that A dominates every leaf node that B dominates. It is entirely possible for the latter to be true without the former being so; for example, consider the nodes A and C in Figure 3.12. Such a situation poses a problem for tag-based serializations: the relationship among the start and end tags of A and C is the same as if A dominated C, so it is impossible to determine from the serialization whether there is in fact an edge from A to C.

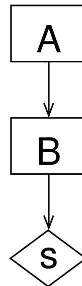

Figure 3.13: A non-example of a restricted GODDAG. A and B contain the same set of text nodes, violating the uniqueness constraint.

Although the uniqueness and leaf constraints establish an order on elements based on the sets of text nodes they contain, this order comes at a cost. Certain situations, such as an element with another element as its only child (Figure 3.13), are impossible to represent under this constraint. The usual solution, demonstrated in Figure 3.14, is to use empty text nodes to disambiguate the relationship [52, 61]. Since the encoding of an empty text node is simply the empty string, these nodes do not change the serialization of the document. Care must be taken to add them in appropriate places when de-serializing a document; one possibility is to add empty text nodes between all adjacent pairs of tags [61].

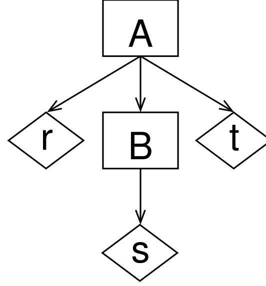

Figure 3.14: Figure 3.13 corrected to form a restricted GODDAG. The nodes r and t are empty text nodes, with $\text{text}(r) = \text{text}(t) = \varepsilon$. Either r or t, but not both, may be omitted.

3.3 Resolving inconsistencies in the GODDAG: global order

The infelicities described in Figures 3.12 and 3.13 motivate us to put forward a different DAG-based model for markup, the globally ordered GODDAG [55]. We maintain the per-node child order of the generalized GODDAG, but also introduce a global document order with which it must be compatible. An additional orthogonality constraint ensures that, as in XML, each node partitions the set of other nodes into four sets: those that follow, precede, dominate, and are dominated by that node.

Definition 3.3.1 *A* ***globally ordered GODDAG (GOG)*** *is a pair* $(G, <_G)$ *where* G *is a GODDAG and* $<_G$ *is a strict partial order on the nodes of* G *such that:*

1. *(Compatibility) The child order* $<_\nu$ *at each node* $\nu \in T \cup E$ *is the restriction of* $<_G$ *to the children of* ν*.*

2. *(Orthogonality) For any nodes* $\mu, \nu \in G$*, exactly one of the following is true:* $\mu = \nu$*,* $\mu \to^+ \nu$*,* $\nu \to^+ \mu$*,* $\mu <_G \nu$*, or* $\nu <_G \mu$*.*

We begin by proving two properties of globally ordered GODDAGs that are important to the connection between GOGs and markup serialization in Section 3.3.4. First, we show that the globally ordered GODDAG satisfies a contiguity constraint analogous to that of the restricted GODDAG, but applying to all descendants, not just leaves.

Theorem 3.3.1 *A globally ordered GODDAG* $(G, <_G)$ *is* ***globally contiguous****: If* $\pi, \alpha, \beta, \gamma \in (T \cup E)$ *such that* $\pi \to^+ \alpha$*,* $\pi \to^+ \gamma$*, and* $\alpha <_G \beta <_G \gamma$*, then* $\pi \to^+ \beta$*.*

Proof: Let $\pi, \alpha, \beta, \gamma \in (T \cup E)$ satisfy the antecedent. Now consider the relation between π and β. If $\pi <_G \beta$ then $\pi <_G \gamma$ by transitivity, contradicting orthogonality between π and γ. Likewise, if $\beta <_G \pi$ then $\alpha <_G \pi$, contradicting orthogonality between π and α. Finally, if $\beta \to^+ \pi$, then by transitivity $\beta \to^+ \alpha$, contradicting orthogonality between α and β. By orthogonality between π and β, then, we have that $\pi \to^+ \beta$. □

Orthogonality results in the global order being compatible with the ancestor relation in another way. Given two nodes related by $<_G$, the descendants of the earlier node alone precede (or are ancestors of) the common descendants of the two nodes, which in turn precede (or are descendants of) the descendants of the later node alone; and likewise for ancestors.

Theorem 3.3.2 *Let μ and ν be two nodes of a globally ordered GODDAG $(G, <_G)$, with $\mu <_G \nu$. Every descendant of μ but not of ν precedes or is an ancestor of every descendant of ν; every descendant of ν but not of μ follows or is an ancestor of every descendant of μ; every ancestor of μ but not of ν precedes or is a descendant of every ancestor of ν; and every ancestor of ν but not of μ follows or is an descendant of every ancestor of μ.*

Proof: We prove only the first statement; the others have analogous proofs. Let $\mu, \nu, \alpha, \beta \in G$ such that $\mu <_G \nu$, $\mu \to^+ \alpha$, $\nu \not\to^+ \alpha$, and $\nu \to^+ \beta$. By transitivity, if $\nu <_G \alpha$ then $\mu <_G \alpha$, and if $\alpha \to^+ \nu$ then $\mu \to^+ \nu$, both contradicting orthogonality. Hence $\alpha <_G \nu$. Again by transitivity, if $\beta <_G \alpha$ then $\beta <_G \nu$, and if $\beta \to^+ \alpha$ then $\nu \to^+ \alpha$, both contradicting orthogonality. Therefore either $\alpha \to^+ \beta$ or $\alpha <_G \beta$. □

The global order constraints are strictly stronger than the constraints on semi-restricted GODDAGs.

Theorem 3.3.3 *Each globally ordered GODDAG is a semi-restricted GODDAG.*

Proof: Let $<_L$ be the restriction of the partial order $<_G$ to the leaf nodes of G. Let x and y be two distinct leaf nodes of G; then neither $x \to^+ y$ nor $y \to^+ x$, so by orthogonality either $x <_L y$ or $y <_L x$. Hence $<_L$ is a strict total order. Furthermore, global contiguity implies the weaker contiguity constraint on $<_L$. Hence G is a semi-restricted GODDAG with leaf order $<_L$. □

Theorem 3.3.4 *There exist restricted (and therefore semi-restricted) GODDAGs that are not globally ordered.*

PROOF: Consider the GODDAG G in Figure 3.12 (page 48). Each of the non-terminal nodes A, B, C, and D dominates a different, contiguous, subrange of the terminal nodes $p <_L r <_L s$; hence G is a restricted GODDAG. Now suppose G were globally ordered by $<_G$. The compatibility constraint requires, among other relations, that $C <_G D$ and $p <_G r$. Then there are three possibilities for the relationships among p, C, and s:

- If $p <_G C <_G s$, then contiguity is violated: A should dominate C, but it does not.
- If $s <_G C$ then $s <_G C <_G D$, violating orthogonality because $D \rightarrow^+ s$.
- If $C <_G p$ then $C <_G p <_G r$, violating orthogonality, because $C \rightarrow^+ r$.

Since no choice of relationships between p and C and between s and C satisfies all of the constraints of Definition 3.3.1, G is not globally ordered. □

3.3.1 XPath on globally ordered GODDAGs

The order relation $<_G$ on GOGs allows us to define the ordered XPath axes. Other than using the global order $<_G$ rather than the extension $<_D$ of the child order, the axes are defined identically to the axes on restricted GODDAGs (Definition 3.2.8).

Definition 3.3.2 *An axis on a globally ordered GODDAG G is one of the following:*

- *One of the unordered axes (Definition 3.2.3).*
- $\mathit{following} = \{(\mu, \nu) |\ \mu >_G \nu\}$
- $\mathit{preceding} = \{(\mu, \nu) |\ \mu <_G \nu\}$
- $\mathit{following\text{-}sibling} = \{(\mu, \nu) \in \mathit{following} |\ \mu \text{ and } \nu \text{ have a parent in common}\}$
- $\mathit{preceding\text{-}sibling} = \{(\mu, \nu) \in \mathit{preceding} |\ \mu \text{ and } \nu \text{ have a parent in common}\}$
- $\mathit{following\text{-}overlap} = \{(\mu, \nu) \in \mathit{following} |\ \mu \text{ and } \nu \text{ have a descendant in common}\}$
- $\mathit{preceding\text{-}overlap} = \{(\mu, \nu) \in \mathit{preceding} |\ \mu \text{ and } \nu \text{ have a descendant in common}\}$

- *following-strict* $=$ *following* $\setminus$ *following-overlap*
- *preceding-strict* $=$ *preceding* $\setminus$ *preceding-overlap*

From this set of definitions, we may define the semantics of an XPath expression on a globally ordered GODDAG. The syntax is as in Grammar 2.2.1, with the addition of new productions for the four new axes (following-overlap, preceding-overlap, following-strict, and preceding-strict).

Definition 3.3.3 *A **multihierarchical core XPath expression** is a string produced by Grammar 2.2.1 with the additional production*

$$\begin{aligned} axis \Rightarrow \quad & \texttt{following-strict} \mid \texttt{following-overlap} \\ \mid \quad & \texttt{preceding-strict} \mid \texttt{preceding-overlap} \\ \mid \quad & \texttt{descendant} \mid \texttt{descendant-or-self} \end{aligned}$$

The semantics $\mathcal{S}_{MH}[[E]]_G$ of a multihierarchical core XPath expression E is the function from sets of nodes from sets of nodes given by $\mathcal{S}_{\mathrm{C}}[[E]]_G$, with the semantics of axes replaced with those from Definitions 3.2.3 and 3.3.2.

The semantics $\mathcal{S}_{MH}[[E]]\,(G)$ of E on the globally ordered GODDAG G is

$$\bigcup_{\rho \in \mathrm{roots}(G)} \mathcal{S}_{MH}[[E]]_G\,(\rho).$$

*If $\nu \in \mathcal{S}_{MH}[[E]]\,(G)$, we say that E **matches** ν in G.*

We make use of these semantics when we define multihierarchical access control policies in Section 4.5.

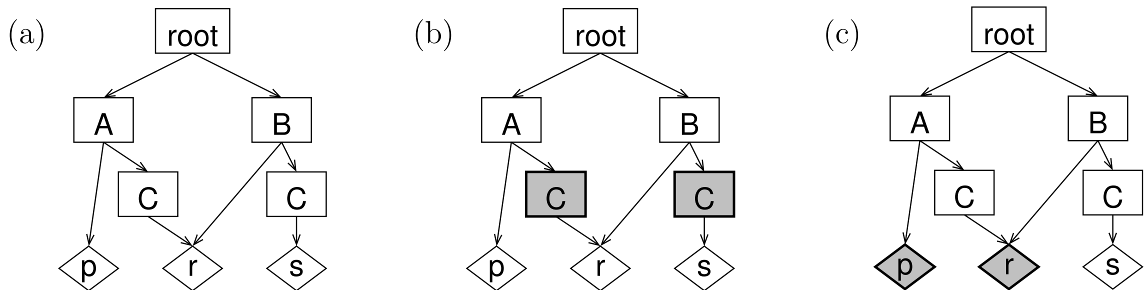

Figure 3.15: Evaluating a multihierarchical XPath expression

As an example, consider the document D in Figure 3.15(a), and the path expression /descendant::C/preceding-sibling:: `node()` . The first location step selects descendants of the root labelled C (Figure 3.15(b)). The second location step is evaluated

in the context of those nodes and selects siblings that precede those nodes (Figure 3.15(c)): p as a sibling of the first C node (with shared parent A), and r as a sibling of the second (with shared parent B).

3.3.2 Connection with tag- and range-based models

The new GOG structure reflects many of the constraints faced when serializing markup in a text-and-tags form. To make the connection to serialization more clear, and to make the constraints easier to verify, we develop an alternative formulation.

The global contiguity property of GOGs (Theorem 3.3.1) suggests that the structure avoids some of the problems encountered when attempting to serialize documents such as the one in Figure 3.12. However, the question remains of how accurately GOG documents may be serialized, and whether the GOG can represent all forms of tag-and-text markup. We first note that, in the absence of empty text nodes, it is impossible to represent spurious overlap (Definition 3.1.1) in the GOG: two nodes overlap only if they share descendants (Figure 3.16), but a non-empty shared descendant as in Figure 3.16(b) would cause the overlap to not be spurious. We consider this limitation to be an advantage rather than a disadvantage, but it does raise the question of whether there are other infelicities when attempting to model serializable markup using globally ordered GODDAGs. To answer this question, we return to the issue of modelling the text and tags serialization.

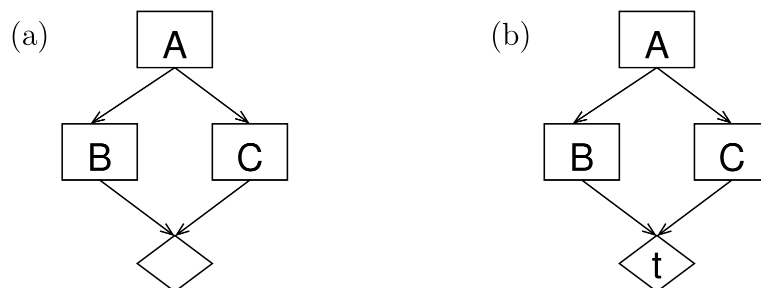

Figure 3.16: Spurious and non-spurious overlap. The nodes B and C overlap in both (a) and (b), but this overlap is spurious only in (a).

The connection between serializable markup (in the TexMECS notation) and GODDAGs was studied by Marcoux [52], who characterized the serializable GODDAGs by deriving starts-before and ends-after relations on start tags and end tags from a GODDAG's child order: If these relations are acyclic, the document is serializable. We take a similar but distinct approach, beginning with these tag order relations and deriving from them the graph structure. In addition, unlike Marcoux,

our construction avoids tag orders with spurious overlap; therefore, we avoid both the ambiguity of empty text nodes and the inability to represent some documents without using empty text nodes.

A tag-based serialization of a document with elements E and text nodes T may be thought of as a total order $<_S$ on the tags $E \times \{\mathsf{start}, \mathsf{end}\}$ and text nodes T of the document, where $(\nu, \mathsf{start}) <_S (\nu, \mathsf{end})$ for every element ν. Spurious overlap occurs when one element's start tag is immediately before another node's end tag. By abstracting away the relative ordering of start tags and end tags, we can avoid this situation by having a single representation for both adjacency and spurious overlap; then the sequence of tags without spurious overlap can be taken as the canonical order.

We remove the ordering between start and end tags by using two total orders on the set of nodes, similar to the relations of Marcoux [52]. The order $<^o$ ("opens before") in a range GODDAG reflects the order of text nodes and start tags in serialized markup; $<^c$ ("closes before") reflects the order of text nodes and end tags.

Definition 3.3.4 *A* ***range GODDAG*** *is a GODDAG along with two total orders $<^o$ and $<^c$ over its nodes, such that:*

1. *(Domination) $\mu \to^+ \nu$ if and only if $\mu <^o \nu$ and $\nu <^c \mu$.*
2. *(Range compatibility) For each node ν, the restriction of $<^o$ to the children of ν and the restriction of $<^c$ to the children of ν are both equal to the child order $<_\nu$.*

As we shall see in Theorem 3.3.7, this alternative formulation allows the GODDAG to be reconstructed from the order relations alone. First, however, we show that the globally ordered GODDAG and the range GODDAG are in fact equivalent structures.

Theorem 3.3.5 *For each globally ordered GODDAG $(G, <_G)$, there is a corresponding range GODDAG $(G, <^o, <^c)$ with the same GODDAG structure G.*

PROOF: Let $(G, <_G)$ be a globally ordered GODDAG. Take $<^o$ to be $<_G \cup \to^+$, and $<^c$ to be $<_G \cup \to^+$. Then $<^o \cap >^c = (<_G \cup \to^+) \cap (>_G \cup \to^+)$. Since $<_G$ is asymmetric, this relation is precisely the $\to^+$ relation. Thus $a \to^+ b$ if and only if $a <^o b$ and $b <^c a$, satisfying the domination constraint.

Let a be a node of G. By antitransitivity, none of a's children can can dominate the another (lest a dominate a node both directly and indirectly). Hence the restrictions

of $<^o$ and $<^c$ to this set of children is identical with the restriction of $<_G$. Since $<_G$ is compatible with $<_a$, so are $<^o$ and $<^c$. Hence the range GODDAG compatibility constraint is satisfied.

Finally, it remains to be shown only that $<^o$ and $<^c$ are strict total orders (that is, trichotomous and transitive). Let a and b be two distinct nodes of G. Then, by orthogonality, exactly one of the expressions $a <_G b$, $b <_G a$, $a \to^+ b$, or $b \to^+ a$ is true. In each of these cases, either $a <^o b$ or $b <^o a$ is true, and either $a <^c b$ or $b <^c a$ is true. On the other hand, neither $a <^o a$ nor $a <^c a$ is ever true. Hence both of these orders are trichotomous.

Now we consider transitivity. Suppose we have $a <^o b <^o c$. Then either $a <_G b$ or $a \to^+ b$; and either $b <_G c$ or $b \to^+ c$. If $a <_G b <_G c$ or $a \to^+ b \to^+ c$ it is clear by transitivity of $<_G$ and $\to^+$ that $a <^o c$. Therefore consider the two remaining cases:

- Suppose $a <_G b \to^+ c$. It cannot be the case that $c <_G a$, as then $c <_G b$ by transitivity of $<_G$, violating orthogonality between b and c (since $b \to^+ c$). Likewise, it cannot be that $c \to^+ a$, for then $b \to^+ a$, also violating orthogonality. Hence, by orthogonality between a and c, either $a <_G c$ or $a \to^+ c$, so $a <^o c$.

- Suppose $a \to^+ b <_G c$. Again, we cannot have that $c <_G a$, for then we would have $b <_G a$ and violate orthogonality; and we cannot have $c \to^+ a$, for then we would have $c \to^+ b$. By orthogonality, then, either $a <_G c$ or $a \to^+ c$, and $a <^o c$.

Since in every possible case we can show that $a <^o c$, the $<^o$ relation is transitive. Therefore it is a strict partial order. A similar case analysis, omitted here, shows that $<^c$ is also transitive and hence a strict partial order. Therefore $(G, <^o, <^c)$ is a range GODDAG. □

Theorem 3.3.6 *For each range GODDAG $(G, <^o, <^c)$ there is a corresponding globally ordered GODDAG $(G, <_G)$ with the same GODDAG (graph and local order) structure G.*

PROOF: Let $(G, <^o, <^c)$ be a range GODDAG, and take $<_G = <^o \cap <^c$. As the intersection of two strict total orders, $<_G$ is itself irreflexive, asymmetric, and transitive, and hence is a strict partial order. For each node a, both $<^o$ and $<^c$ are compatible with $<_a$, so their intersection is also compatible with $<_a$. Hence the global compatibility constraint is satisfied.

Given two distinct nodes $a, b \in G$, there are four possible ways they may be related under $<^o$ and $<^c$. Either a precedes b in both orders, so $a <_G b$; b precedes a in both orders, so $b <_G a$; a precedes b in $<^o$ but follows it in $<^c$, so $a \to^+ b$; or a follows b in $<^o$ but precedes it in $<^c$, so $b \to^+ a$. Since exactly one of these is true when $a \neq b$, and none are true when $a = b$, the orthogonality constraint is satisfied. Global contiguity is further satisfied by Theorem 3.3.1), so $(G, <_G)$ is a globally ordered GODDAG. $\square$

One benefit of the new definition is that the order relations alone completely determine the GODDAG graph structure.

Theorem 3.3.7 *Given a set V of nonterminal nodes and two total orders $(<^o, <^c)$ on V, there is a range GODDAG $(G, <^o, <^c)$ over the nodes V; and any two such range GODDAGs have the same arc relation and child orders.*

PROOF: We begin by noting that the domination relation is uniquely determined by $<^o$ and $<^c$; specifically, it is the intersection of $>^o$ and $<^c$. Along with the antitransitivity constraint, this intersection determines the arcs of G: they are simply the pairs in the transitive reduction $\to$ of $\to^+$.

Now consider a node a with children C_a. For our GODDAG to satisfy the range compatibility constraint, the child order $o(a)$ must equal the restriction of $<^o$ to C_a, and also the restriction of $<^c$ to C_a. The child order is therefore uniquely defined, and exists whenever the restrictions of $<^o$ and $<^c$ to C_a are equal. But they are equal: if $b <^o c <^c b$, then b is an ancestor of c, violating antitransitivity. Hence the child order at each node is uniquely determined. Therefore, there is (up to the labelling function name) a unique GODDAG G such that $(G, <^o, <^c)$ is a range GODDAG. $\square$

Although Theorem 3.3.7 considers only nonterminal nodes, it is straightforward to extend the result to documents with both terminal and nonterminal nodes, giving the following corollary:

Corollary 3.3.1 *Let N and T be disjoint sets of nonterminal and terminal nodes, and $<^o$ and $<^c$ two total orders on $N \cup T$ such that every terminal node is minimal under $>^o \cap <^c$. Then there is a GODDAG G with nonterminal set N and terminal set T such that $(G, <^o, <^c)$ is a range GODDAG; and any two such GODDAGs have the same arc relation and child orders.*

3.3.3 Update operations

We have seen that the range GODDAG model combines many of the useful properties of graph-based and tag-based models of multihierarchical markup. Another advantage, particularly compared to the restricted GODDAG, is that we can define update operations in such a way that the result is uniquely determined, and is itself a range GODDAG.

In the generalized GODDAG, a new node may be inserted as the child of any set of existing nodes, anywhere in the child orders of those nodes, subject only to the constraint that it not be the child of both a node and one of its descendants. However, in the semi-restricted GODDAG the situation is more complicated.

Inserting new nodes

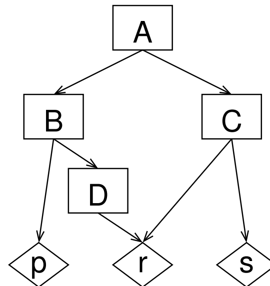

Figure 3.17: A multihierarchical document

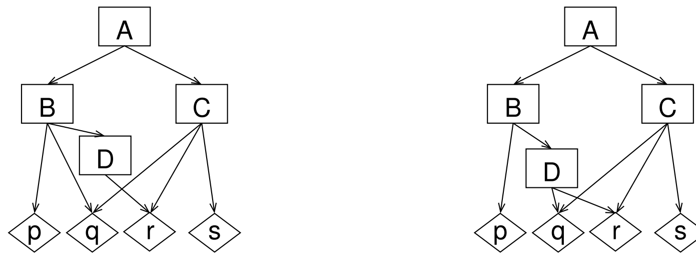

Figure 3.18: Inserting a node into a GODDAG. Both documents result from inserting q into Figure 3.17 as a child of C preceding r; only the document on the right is a range GODDAG.

Consider the semi-restricted GODDAG in Figure 3.17, and suppose we wish to insert a new terminal node q as a child of C preceding r. In order to avoid violating the contiguity constraint at C, q must follow p; but then in order to avoid violating the same constraint at B, q must also be a descendant of B. There are two possibilities

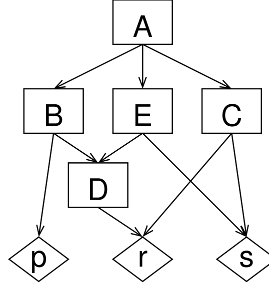

Figure 3.19: Splicing a node into a GODDAG. E has been spliced into Figure 3.17 as a parent of D and s

(Figure 3.18): q can be a child of B preceding D, or q can be a child of D preceding r. The former choice, though a valid restricted GODDAG, results in a document without a contiguous serialization: D follows q as a child of B, and should thus be a child of C as well; this situation is similar to the situation in Figure 3.12. The second choice, making q a child of D, is the only option that results in a valid GODDAG with a clear serialization.

Now consider Figure 3.17 as a range GODDAG; its order relations are

$$<^o : (A, B, p, D, C, r, s)$$
$$<^c : (p, r, D, B, s, C, A)$$

A child of C preceding r must lie between C and r in $<^o$; hence the new $<^o$ is (A, B, p, D, C, q, r, s). Furthermore, in order to preserve the total order on leaf nodes, the new node must follow p and precede r in $<^c$. Hence the new $<^c$ is (p, q, r, D, B, s, C, A), giving us the second document from Figure 3.18.

It may also be desirable to insert a node as a parent of existing nodes. For example, when annotating an electronic manuscript, we may wish to "tag" existing text by making it the descendant of a new element [17]. The element inserted by this "splice-in" operation must still be placed as a descendant of existing nodes, unless it happens to be a root of the document. Figure 3.19, for example, shows the result of inserting a node E that contains both D and s. We define the insert operation to cover this case as well as the more traditional case of inserting a new leaf node.

Definition 3.3.5 *A splice-in operation on a range GODDAG*

$$G = ((\Sigma, L, T, E, A, \text{text}, \text{label}, \text{order}, \text{attribute}), <^o, <^c)$$

is a tuple of the form $S = (\textsf{splice-in}, \alpha, \beta, \ell)$, *where* $\alpha, \beta \in T \cup E \cup \{\top\}$ *are two, not necessarily distinct, nodes of* G, *or* $\top$ *indicating an end of the document; and* $\ell \in L \cup \Sigma^+$ *is the name or text of the new node.*

The result $S(G)$ of a splice-in S on the range GODDAG G is defined as follows. If there exists a terminal node t such that $t <^o \alpha$ and $\beta <^c t$, the result is undefined and $S(G) = \bot$; likewise if $\ell \in \Sigma^+$ and there exists some node ν such that $\alpha <^o \nu$ and $\nu <^c \beta$. Otherwise, $S(G)$ is given by:

$$S(G) = \left(\left(\Sigma, L, T^P, E^P, A', \mathrm{text}^P, \mathrm{label}^P, \mathrm{order}', \mathrm{attribute}^{\xi \to \emptyset}\right), (<^o)^{\xi < \alpha}, (<^c)^{\xi < \beta^+}\right),$$

where χ is a new node not in $T \cup E$; $P = (\chi, \ell)$; x^P is as in Definition 2.3.4; β^+ is the successor of β in $<^c$ (with the successor of $\top$ being the $<^c$-minimal node, and the successor of the $<^c$-maximal node being $\top$); and A' and order′ *are the arc and child order relations obtained from the new $<^o$ and $<^c$ orders by Theorem 3.3.7.*

Remark: We insert the new node by choosing a node α that it will immediately precede in $<^o$ and a node β that it will immediately follow in $<^c$. We use $\alpha = \top$ to indicate that the new node should be maximal under $<^o$, and $\beta = \top$ to indicate that it should be minimal under $<^c$. The two undefined cases correspond to inserting a new node as a descendant of a text node t, and to inserting a text node as an ancestor of an existing node ν, respectively.

As an example, the document on the right of Figure 3.18 results from inserting q such that it immediately precedes r in both $<^o$ and $<^c$. The document in Figure 3.19 results from inserting E such that it immediately precedes D in $<^o$ and such that it immediately follows s in $<^c$.

Deleting nodes

In hierarchical models of documents, the delete operation typically removes an entire subtree: the target node along with all descendants of that node. When extending this operation to multihierarchical documents, there are at least two different notions of "subtree" available to us.

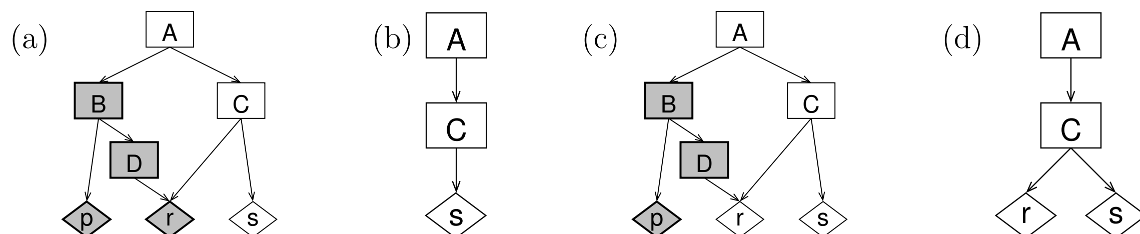

Figure 3.20: Deleting a node from a GODDAG. (a) Broad subtree rooted at B. (b) Document with the broad subtree deleted. (c) Narrow subtree rooted at B. (d) Document with the narrow subtree deleted.

Definition 3.3.6 *Let G be a GODDAG with node set N, and $\nu \in N$. The* ***broad subtree*** *of G rooted at ν is the set of nodes $\chi \in N$ such that $\nu \rightarrow^* \chi$. The* ***narrow subtree*** *of G rooted at ν is the set of nodes $\chi \in N$ such that χ is in the broad subtree of G rooted at ν, and such that furthermore either $\chi = \nu$ or every parent of χ is in the broad subtree rooted at ν.*

When deleting the subtree rooted at a particular node, we have the option of removing all nodes in the broad subtree, or all nodes in the narrow subtree. As an example, consider again the document from Figure 3.17, and suppose we wish to delete the node B. The nodes p and D are in both the narrow and broad subtrees rooted at B. On the other hand, r has a parent, C, that is neither B nor one of its descendants. Hence r is in the broad but not the narrow subtree. Deleting the broad subtree rooted at B results in the document shown in Figure 3.20(b); deleting the narrow subtree results in the document in Figure 3.20(d).

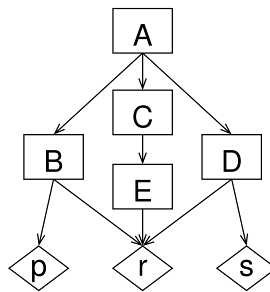

Figure 3.21: Non-triviality of narrow deletion. Deleting E while preserving r requires making r a child of C.

We note here that deletion is not entirely straightforward: deleting a narrow subtree may require adding additional links to the GODDAG in order to produce a range GODDAG. For example, suppose we wish to delete the narrow subtree rooted at E in Figure 3.21. The only node in this narrow subtree is E itself. However, simply removing E would make C a leaf node, which cannot consistently be placed in the $<^o$ and $<^c$ orders. Instead, it is necessary to make r a child of C in the resulting tree.

Instead, we return to the splice-out operation (Section 2.3.6). Recall that, in a hierarchical document, this operation removes only a single node, replacing it with the sequence of its children; hence descendant relations among the remaining nodes are left intact. Unlike in hierarchical documents, it is not always necessary to add arcs from the deleted node's parent(s) to its children, though it often is. Figure 3.22(b) shows the result of splicing out node B from Figure 3.17.

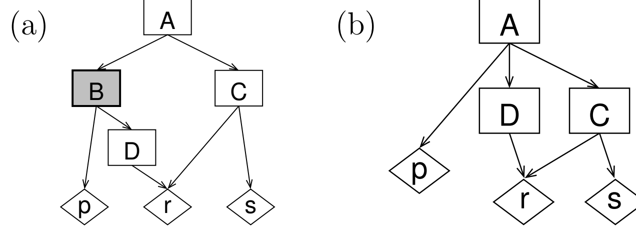

Figure 3.22: Splicing a node out of a GODDAG. (a) Original tree with B highlighted. (b) Tree with B spliced out.

The splice-out operation is more general than the broad and narrow subtree delete operations described above, as these operations may be expressed as a sequence of splice-out operations, one for each node in the broad (respectively, narrow) subtree.

Definition 3.3.7 *A splice-out operation on a range GODDAG*

$$G = ((\Sigma, L, T, E, A, \text{text}, \text{label}, \text{order}, \text{attribute}), <^o, <^c)$$

is a tuple of the form $X = (\textsf{splice-out}, \nu)$, where $\nu \in T \cup E$ is a node of G.

The result $X(G)$ of a splice-out X on the range GODDAG G is:

$$X(G) = \Big(\Big(\Sigma, L, T \cap N', E \cap N', A', \text{text}|_{N'}, \text{label}|_{N'}, \text{order}', \text{attribute}^{\xi \to \emptyset}\Big), <^o|_{N'}, <^c|_{N'}\Big),$$

where $N' = T \cup E \setminus \{\nu\}$ is the set of nodes of G other than ν; and A' and order' *are the arc and child order relations obtained from $<^o|_{N'}$ and $<^c|_{N'}$ by Theorem 3.3.7.*

Renaming nodes

Finally, nodes in a range GODDAG may be renamed. As with the rename operation on hierarchical documents (Definition 2.3.11, this operation does not affect the document's graph structure or order relations, but only its label and text functions.

Definition 3.3.8 *A rename operation on a range GODDAG*

$$G = ((\Sigma, L, T, E, A, \text{text}, \text{label}, \text{order}, \text{attribute}), <^o, <^c)$$

is a tuple of the form $R = (\textsf{rename}, \nu, \ell)$, where $\nu \in T \cup E$ is a node of G; and ℓ is a name from L if $\nu \in E$, or a text string from Σ^+ if $\nu \in T$.

The result $X(G)$ of a splice-in S on the range GODDAG G is:

$$X(G) = \Big(\Big(\Sigma, L, T, E, A, \text{text}^{(\nu,\ell)}, \text{label}^{(\nu,\ell)}, \text{order}, \text{attribute}\Big), <^o, <^c\Big)$$

3.3.4 Converting between range and globally ordered GODDAGS

Dushnik and Miller [26] show that every partially ordered set can be written as the intersection of a set of total orders (each a linear extension of the partial order) called a **realizer** of P. In general there may be many distinct realizers for a given partial order; the **dimension** of the partial order is the cardinality of (one of) the smallest realizers of that set. We shall use these terms to refer also to strict partial orders: a realizer of a strict partial order P is a set of strict total orders, the intersection of which is P; and its dimension is the minimum cardinality among its realizers. It may be verified that $\{T_1, T_2, \ldots\}$ is a realizer of the strict partial order P if and only if $\{T_1 \cup \mathrm{Id}, T_2 \cup \mathrm{Id}, \ldots\}$ is a realizer of the (non-strict) partial order $P \cup \mathrm{Id}$. We shall therefore refer to strict and non-strict partial orders interchangeably.

By construction, the ancestor relation of a range GODDAG is a one- or two-dimensional partial order, with realizer $\{<^o, >^c\}$; conversely, any two-dimensional partial order can be represented as the ancestor relation of two distinct range GODDAGs per realizer $\{A, B\}$, corresponding to the ordered pairs (A, B) and (B, A) of total orders; and any one-dimensional (and therefore total) order P as the ancestor relation of the single range GODDAG with $<^o = >^c = P$. The two range GODDAGs corresponding to a single realizer are reverses of one another: $(<^o, >^c)$ and $(>^c, <^o)$ have the same ancestor relation but the reverse precedes relation (and thus a reversed child order at each node). The one-dimensional orders correspond to chains with no choice of child order.

Algorithm 1 `parents`: find the parents (direct predecessors) of an element in a two-dimensional partial order [51].

Require: X is a set
Require: P is a two-dimensional partial order on X with realizer $\{<_a, <_b\}$.
Require: $\nu \in X$
Ensure: Returns the set of elements $\pi \in X$ such that $\pi P \nu$ and such that there is no μ with $\pi P \mu P \nu$.

  $\mathit{last} \leftarrow \min_{<_b}(X)$ // *last parent under* $<_b$
  for all $\pi <_a \nu$ in descending $<_a$ order **do**
    if $\pi <_b \nu$ **then** // $\pi P \nu$
      if $\nu \geq_b \mathit{last}$ **then** // *does not precede any parent*
        $\mathit{last} \leftarrow \pi$
        $\mathit{parents} \leftarrow \mathit{parents} \cup \{\pi\}$
      end if
    end if
  end for
  return *parents*

Ma and Spinrad [51] define $\mathcal{O}(n^2)$ algorithms for computing the transitive closure and transitive reduction of two-dimensional partial orders given in any form (transitively reduced, transitively closed, or any intermediate form). In particular, it is possible to find the parents of a single node in $\mathcal{O}(n^2)$ time.

Lemma 3.3.1 *Algorithm 1 runs in time $\mathcal{O}(n)$ and returns the direct predecessors of ν in P.*

PROOF: Suppose π is a parent of ν. Then $\pi <_a \nu$, so the loop (lines 2–9) iterates over it; and $\pi <_b \nu$ so the outer conditional (line 3) executes. When the loop reaches the iteration for π, all nodes μ such that $\pi <_a \mu <_a \nu$ have been iterated over. Since π is a parent of ν, none of these nodes had $\pi <_b \mu <_b \nu$. Hence $\pi \geq_b$ *last*, the inner conditional (line 4) executes, and π is added to *parents*. Since *parents* grows monotonically, π is therefore among the returned nodes.

Now suppose conversely that π is in the returned list. Then it was added to *parents*, so the loop iterated over π (hence $\pi <_a \nu$) and both conditions were true (so $\pi <_b \nu$ and $\pi \geq_b$ *first*). Because $\pi \geq_b$ *first*, there was no previously-iterated μ (that is, $\pi <_a \mu <_a \nu$) such that $\pi <_b \mu <_b \nu$. Hence $\pi P \nu$ and there is no μ such that $\pi P \mu P \nu$; that is, π is a direct predecessor of ν. □

This algorithm can help convert between a range GODDAG and its often more convenient representation as a globally ordered GODDAG. We presume that a range GODDAG is represented by two lists of its nodes, one in $<^o$ order and one in $<^c$ order. A GOG, on the other hand, is represented in a more traditional GODDAG form: an ordered list of pointers to the root nodes, with each node containing an ordered list of pointers its child nodes.

Converting from a range GODDAG to a GOG is a straightforward application of the algorithm of Ma and Spinrad. Given the listings L_o and L_c, we find the parents of each node, and add that node to each parent's child list (or to the list of root nodes if there are no parents).

Theorem 3.3.8 *The procedure* `range2goddag` *(Algorithm 2) runs in time $\mathcal{O}(n^2)$. For each node ν, children$[\nu]$ contains the list of children of ν in opening (and thus closing) order; and roots contains the list of roots in opening (and thus closing) order.*

Algorithm 2 `range2goddag`: convert a range GODDAG into a GOG

Require: L_o is a list of n distinct nodes.
Require: L_c is a list of nodes that forms a permutation (perhaps the identity permutation) of L_o.
Ensure: Returns a list R the root nodes of the range GODDAG (() L_o, L_c) as well as a mapping *children* from nodes to lists of their child nodes.

```
// Compute the permutation φ from indexes of L_o to corresponding indexes of L_c.
φ ← an array of n integers
for all i ∈ [1, n] do
  for all j ∈ [1, n] do
    if L_o[i] = L_c[j] then
      φ[i] ← j
    end if
  end for
end for
// Compute the transitive reduction of <^o ∩ >_c (Algorithm 1)[51].
children ← an empty map from nodes to lists of nodes
roots ← an empty list of nodes
for all c from 2 to n do
  first ← n + 1 // earliest-closing parent so far
  for all p from c − 1 down to 1 do
    if φ[p] > φ[c] then // an ancestor
      if φ[p] ≤ first then // not the ancestor of any parent
        first ← φ[p]
        push(children[L_o[p]], L_o[c])
      end if
    end if
  end for
  if first = n + 1 then // no ancestors
    push(roots, L_o[c])
  end if
end for
return (roots, children)
```

PROOF: Suppose ν is the ith node of L_o and that it is a parent of the jth node χ; clearly $i < j$ so $j \geq 2$. Consider the iteration of the inner loop (lines 15–22) when $c = j$ and $p = i$. This loop is a straightforward translation of Algorithm 1; hence by Lemma 3.3.1, it added χ to the child list of ν.

Now suppose conversely that χ was added to the child list of ν; this situation occurs at iteration $c = j$. By Lemma 3.3.1, ν is a parent of χ.

Finally, note that a node ν with index i is added to its parents' child lists in the ith iteration of the outer loop (lines 13–26); hence it follows in each child list those nodes with index less that i and precede those with index greater than i. Since the index of a node corresponds to its position in L_o, each child list is sorted by $<^o$.

Finally, we observe that the algorithm consists of two nested pairs of loops, each iterating over up to n nodes. The body of these nested loops performs a constant amount of work (assigning an array element and/or appending to a list); hence the algorithm runs in time $\mathcal{O}(n^2)$. □

This algorithm also allows us to update a GODDAG to account for a splice-in operation. Recall that a splice-in operation (Definition 3.3.5) specifies the α and β such that the new node immediately precedes α in $<^o$ and immediately follows β in $>^c$; the arc relation A' and child orders order$'$ of the resulting GODDAG are specified in terms of the globally ordered GODDAG corresponding to the new opening and closing orders. Instead, we may apply Algorithm 1 to find the parents of the new node in time $\mathcal{O}(n)$, and a straightforward variant (reversing all comparisons and iterations) to find its children also in time $\mathcal{O}(n)$. For each parent, we iterate over its child list, removing those nodes that also children of the new node (and hence no longer children of the parent) and replacing them with the new node. In this way, we can compute the updated GODDAG in time $\mathcal{O}(n + p * C)$, where p is the number of parents of the new node and C the maximum number of children of all its parents.

Finally, we may proceed in the opposite direction, computing the range GODDAG orders associated with a given GOG. Our algorithm uses only the GODDAG structure (an ordered list of children of each node), not the global order.

In general, a GODDAG need not have a single root node; there may be multiple nodes with no incoming edges. To simplify our algorithm, we assume that the document D has been augmented with a single synthetic root node ρ with no parent, and with the parentless nodes of D as its children. In a globally ordered GODDAG, these parentless nodes are unrelated by the ancestor-descendant relation; by orthogonality, they are related to one another by the global order. There is therefore in a

Algorithm 3 `traverse`: traverse a GODDAG, building lists of its nodes in opening and closing order

---

Require: G is a GODDAG over the set N of nodes.
Require: children() is a function mapping nodes of G to ordered lists of their children (in child or document order).
Require: G is rooted with root ρ.
Ensure: Returns a pair of sequences of nodes generating a range GODDAG isomorphic to G.

```
for all ν ∈ N do
    tovisit[ν] ← in-degree of ν
    uplink[ν] ← empty list of nodes
end for
openorder ← empty list of nodes
closeorder ← empty list of nodes
visit(ρ)
return (openorder, closeorder)
```

Algorithm 4 `visit`: visit a node in a GODDAG, opening the node if all its parents have been opened

---

Require: ν is a node of G
Ensure: If all of this node's parents have been opened, open this element. Then, if this element has been opened and either there are no children or the last child element has been closed, returns true. Otherwise, if this element has been opened, add it to the uplink list of its last child.

```
tovisit[ν] ← tovisit[ν] − 1
if tovisit[ν] > 0 then
    return false
else
    started(ν)
    for all c ∈ children(ν) do
        if c is the last child of ν then
            if visit(c) = false then
                append ν to uplink[c]
                return false
            end if
        else
            visit(c)
        end if
    end for
    ended(ν)
    return true
end if
```

Algorithm 5 `started`: mark that a node has opened

Require: ν is a node of G
  Append ν to *openorder*.

Algorithm 6 `ended`: mark that a node has closed

Require: ν is a node of G
  Append ν to *closeorder*.
  for all $p \in \mathit{uplink}[\nu]$ **do**
    `ended`(p)
  end for

globally ordered GODDAG only a single choice of child order at ρ. In a more general GODDAG, we may choose this order arbitrarily.

Lemma 3.3.2 *A call to* ***`traverse`*** *on a rooted GODDAG results in* ***`started`*** *being called exactly once for each node in the document.*

PROOF: Define the **depth** of a node ν as the length of the longest path from the root to ν. Since the document graph is rooted and acyclic, each node has a finite depth, with nodes other than the root having nonzero depth.

We proceed by induction over the node depth. The root node ρ, with depth 0, is visited directly from `traverse`(); its in-degree is zero, so `started`() is in fact called. Furthermore, because no nodes contain ρ as a child, it is never visited again; hence `started`(ρ) is called exactly once.

Now suppose that all nodes with depth less than d are visited exactly once. Consider a node ν with depth d, and let k be its in-degree. Each of the k parents of ν has depth $< d$. By the inductive hypothesis, `started`() is called on each of these nodes exactly once. As a result, each child of these parents is visited once per parent; in particular, ν is visited exactly k times. On the kth and final visit, $\mathit{tovisit}[\nu]$ is decremented to zero, and `started`(ν) is called. Hence `started`(ν) is called exactly once for each node of depth d; by induction it is called exactly once for each node. □

Lemma 3.3.3 *In a single call to* ***`traverse`***() *on a GOG, if π is an ancestor of ν, then* ***`started`***(π) *is called before* ***`started`***(ν).

PROOF: Suppose π is a parent of ν. Recall that `started`(ν) is not called until ν has been visited once per parent, and in particular from π. Since the algorithm visits the

children of a node only after calling `started`() on that node, `started`(π) is called before `started`(ν).

If π is an ancestor of ν, then there is some chain of parents $\nu = p_0, p_1, \ldots, p_n = \pi$ leading from ν to π. We may apply the previous result inductively: `started`(ν) is called after `started`(p_1), which is called after `started`(p_2), and so on; in particular, `started`(ν) is called after `started`(π). □

Lemma 3.3.4 *In a single call to* ***`traverse`***() *on a GOG, if μ precedes B in the global order, then* ***`started`***(μ) *is called before* ***`started`***(B).

Proof: We proceed by induction on the greater of the depths of μ and ν; neither can have depth 0, as by orthogonality the root node does not precede any node. If μ and ν have depth 1, they are both children of the root node ρ (and only of ρ), with $\mu <_\rho \nu$. Hence μ is first visited before ν, and `started`() is called on each the first time it is visited, so `started`(μ) is called before `started`(ν).

Now take as the inductive hypothesis that, for each node α with depth less than i that precedes a node β with depth less than i, `started`(α) is called before `started`(β). Let μ be a node with depth $j \leq i$, and ν a with depth $k \leq i$, such that μ precedes ν. Consider the last parent π of μ under the global order. If π is a parent of ν, then $\mu <_\pi \nu$ so μ is visited for the last time before ν is visited for the last time; hence μ is started before ν. If π is not a parent of ν, then by Theorem 3.3.2 the last parent σ of ν follows or is a descendant of π. In the former case, σ is started after π by the inductive hypothesis; in the latter, it is started after π by Lemma 3.3.3. Since `started`(μ) is called from `visit`(π) and `started`(ν) is called from `visit`(σ), it follows that `started`(μ) is called before `started`(ν), concluding the induction. □

The converse of these two lemmata follows by the orthogonality constraint.

Corollary 3.3.2 *In a single call to* ***`traverse`***() *on a GOG,* ***`started`***(μ) *is called before* ***`started`***(ν) *only if μ precedes or is an ancestor of ν.*

Proof: If μ does not precede or ν and is not an ancestor of ν, then by orthogonality either $\mu = \nu$, ν precedes μ, or ν is an ancestor of μ. In the first case, `started`(μ) is the same call as `started`(ν), so does not precede it. In the second, we have by Lemma 3.3.4 that `started`(ν) is called before `started`(μ) (and hence, by Lemma 3.3.2, not after it); likewise in the third case, by Lemma 3.3.3, `started`(ν) is called

before `started`(μ). $\square$

As with `started`(), the procedure `ended`() is called once per node.

Lemma 3.3.5 *A call to* ***`traverse`***() *results in* ***`ended`***() *being called exactly once for each node in the document.*

PROOF: Define the **height** of a node as the length of the longest path from that node to a leaf. If π is an ancestor of ν, then π has a greater height than ν; leaf nodes have height zero.

If ν is a leaf of the document, then it has no children, so the loop on lines 6–15 of Algorithm 4 does not execute, and `ended`(ν) is called just after `started`(ν). Furthermore, ν is never be marked as the uplink of another node, so `ended`(ν) is not called other than immediately after `started`(ν). By Lemma 3.3.2, `started`(ν) is called exactly once; hence `ended`(ν) is also called exactly once.

Now suppose that `ended`() is called exactly once for each node of height $< h$, where $h > 0$. Let ν be a node with height h. Consider the last visit to ν, just after `started`(ν) is called. If `visit`() returns true for the last child χ of ν, then `ended`(ν) is called immediately, and ν is not the uplink of any node. Hence `ended`(ν) is called once. If instead `visit`(χ) returns false, ν is added to the uplink list of χ, and `ended`(ν) is called once for each call to `ended`(χ); by the inductive hypothesis, `ended`(χ) is called exactly once. $\square$

Lemma 3.3.6 *In a single call to* ***`traverse`***() *on a GOG, if* μ *precedes* ν *in the global order, then* ***`ended`***(μ) *is called before* ***`ended`***(ν).

PROOF: We first note that, by Lemma 3.3.4, `started`(μ) is called before `started`(ν).

We proceed by induction on the height of μ. If μ is a leaf (height zero), then `ended`(μ) is called immediately after `started`(μ), before any subsequent calles to `started`(). In particular, it is called before `started`(ν) and thus before `ended`(ν).

Now suppose that the condition holds when the height of μ is less than i. Let $\mu <_\pi \nu$ where μ has height $i > 0$ (and thus has at least one child), and consider the rightmost ($<_G$-maximal) children χ_μ and χ_ν of μ and ν respectively. When `ended`(χ_μ) is called, either μ is on the uplink list of χ_μ, or `visit`(χ_μ) was called from χ_μ. In either case, `ended`(μ) is called after `ended`(χ_μ), before any node that is not an ancestor of χ_μ; likewise for χ_ν and ν, if the former exists. If there is no χ_ν (because ν has no

children), then ν follows χ_μ and is ended after it by the inductive hypothesis; and since ν is not an ancestor of χ_μ, it is ended after μ.

If $\chi_\mu = \chi_\nu$ is a single node χ, then, because ν was started after μ, it follows μ on the uplink list of χ if it appears there at all; if it is not on the uplink list, this result is because it is the rightmost parent of χ, and thus the last to call `visited`(χ). Therefore when `ended`($\chi_m u$) is called, μ is ended before ν.

Otherwise, by Theorem 3.3.2, χ_ν follows or is an ancestor of χ_μ. If it follows χ_μ, then by the inductive hypothesis it is ended after χ_μ; and since it is not an ancestor of χ_μ, after μ. Hence ν is ended after μ. If instead χ_ν is an ancestor of χ_μ, it follows μ and is thus started after it; therefore when `ended`(χ_μ) is called, μ is in the uplink list of χ_μ before χ_ν or any of its descendants (if the latter is there at all), and is therefore ended before χ_ν and before ν. □

Lemma 3.3.7 *In a single call to* ***traverse***() *on a GOG, if χ is an descendant of ν, then* ***ended***(χ) *is called before* ***ended***(ν).

PROOF: There are three possibilities. If `visit`(χ) returns true when called from `visit`(ν), then `visit`(χ) called `ended`(χ) before returning and hence before the call to `ended`(ν).

If `visit`(χ) returns false when called from `visit`(ν), and χ is the last child of ν, then ν is added to the uplink of χ, and `ended`(ν) is called from `ended`(χ) (and hence after it).

Otherwise, the call to `visit`(χ) from `visit`(ν) returned false, but the final child μ of ν is not χ; by the previous cases, `ended`(ν) is called after `ended`(μ). By Lemma 3.3.6, `ended`(χ) is called before `ended`(μ) and thus before `ended`(ν). □

As before, the converse holds by orthogonality.

Corollary 3.3.3 *In a single call to* ***traverse***() *on a GOG,* ***ended***(μ) *is called before* ***ended***(ν) *only if μ precedes or is an ancestor of ν.*

PROOF: Analogous to the proof of Corollary 3.3.2. □

Taken together, Lemma 3.3.2 through Corollary 3.3.3 demonstrate the correctness of Algorithm 3.

Corollary 3.3.4 *After a call to* `traverse`(), *openorder is a list of all the nodes in the document, sorted by* $<^o$*; and closeorder is a list of all the nodes, sorted by* $<^c$*.*

The total runtime of the `traverse` algorithm may be easily derived. There is one call to `visit` for each edge in the document; in the last call on each node, the algorithm iterates over that node's children. Thus the total amount of time taken by `visit` is proportional to the number of edges, plus the sum over all nodes of the number of children of that node (also proportional to the number of edges). `started` and `ended` are each called once per node, performing a constant amount of work. Therefore the entire algorithm executes in time $\mathcal{O}(V + E)$ where V is the number of nodes and E the number of edges.

Finally, we note that Algorithms 2 and 3 may be used to verify that a GODDAG is in fact globally ordered. First, run Algorithm 3 on the input GODDAG G to obtain *openorder* and *closeorder*. Since each node is represented once in each list, the lists do in fact represent total orders $<^o$ and $<^c$; use Algorithm 2 to convert these lists back into a (globally ordered) GODDAG. If the resulting GODDAG G' is isomorphic to G, then by Theorem 3.3.7, G corresponds to a range GODDAG and is thus globally ordered. Although the general graph isomorphism problem is difficult, our algorithms allow us to determine the bijection between nodes of G and nodes of G'. It is therefore possible to check for isomorphism by comparing the arc and child order relations, in time $\mathcal{O}(V^2)$.

Copyright © Neil Moore, 2012. Portions of this chapter (portions of Sections 3.2.3 and 3.3) previously appeared in Moore, N. "Reconciling two models of multihierarchical markup", *WebDB '10: Proceedings of the 13th International Workshop on the Web and Databases*, © 2010 ACM, Inc. `http://doi.acm.org/10.1145/1859127.1859146`. Reprinted by permission.

Chapter 4 Fine-grained access control

We present a rule-based model for fine-grained access control of update operations on XML documents, based on models in the literature [27, 49] and the update operations described in Section 2.3. We describe the operations permitted by a policy, and consider a problem related to the the analysis of policies: to determine *a posteriori* whether a document could have been constructed under a given access control policy.

This problem, which we call PGen, is motivated by a number of database administration and collaborative editing scenarios. For example, an administrator might wish to audit a document by verifying that it was (or at least could have been) created under the existing policy; auditing is particularly important when logs of previous operations are missing or unavailable. Another application is to evaluate a proposed policy change, by verifying that existing documents could be reconstructed under the policy. Finally, an algorithm for PGen would allow administrators to verify that particular undesirable document states are disallowed: that is, that the policy does not generate particular trees. Therefore this problem has important implications for document security, verifiability, and provenance.

4.1 Definitions

We begin with a collection of definitions. We use the term "tree" to refer to a hierarchical document (Definition 2.1.3). We use $\mathrm{T}_\emptyset$ to represent the empty tree, containing no nodes. We consider the (simple) insert, delete, and rename operations from Section 2.3.

Definition 4.1.1 *The result of a sequence of operations* $\langle o_1, \ldots, o_n \rangle$ *on a tree* T *is the tree* $o_n(o_{n-1}(\cdots o_1(T)))$.

Remark: These operations are loosely based on those of XUpdate [48]. The rename and delete operations are unchanged from their counterparts in XUpdate. Insert serves as a combined version of the XUpdate operations InsertBefore, InsertAfter, and Append, though as a simple insert it can only add a single node, not an entire subtree as in XUpdate. More complicated insertions can be accomplished by a sequence of operations, allowing all intermediate steps to be checked by access control rules.

Rules in FGAC policies identify potential targets of operations by means of **path expressions**. Typically, path expressions are expressed in XPath [12] or some XPath

fragment such as Core XPath or XPattern [31]. Our rules use the $XP^{\{[],*,//\}}$ fragment of XPath, described in Section 2.2.3. To briefly review, a path expression consists of a sequence of location steps that select nodes along paths from the root to the target according to their labels. If a location step is preceded by /, it selects (appropriately-labelled) children of the preceding node; if preceded by //, it selects descendants of that node. A location step may specify a label from the set L of element labels, or one of three pseudo-labels: $*$ for any element regardless of label, `text()` for any text nodes, or `node()` for any node, text or element. A location step may be followed by a number of **predicates**, each a path expression surrounded by square brackets []; the location step selects a node ν only if each predicate selects a nonempty set of nodes when evaluated in context $\{\nu\}$. Some examples:

- $/*$ selects the root node, regardless of its label.
- //m selects every node labelled m.
- /w/x//y selects every y descendant of an x child of the root w node.
- //x[*/q]/z selects every z node that is the child of some x node that has a grandchild q.

With these preliminary definitions behind us, we may define the form of access control rules in our model.

Definition 4.1.2 *An **access control rule** over the set L of element labels is a tuple with the form $(s, \textsf{insert}, P, \tau)$, $(s, \textsf{delete}, P)$, or $(s, \textsf{rename}, P, \tau)$, where: s is either $+$ or $-$; P is a path expression over L, possibly the empty path expression ε; and τ is a node test (an element label or node type; Definition 2.2.2).*

*A rule is **positive** if s is $+$; otherwise it is **negative**. A rule is **simple** if its path expression contains no predicates.*

Definition 4.1.3 *A rule R **matches** the operation O on tree T, written $R \sim_T O$, if and only if one of the following holds:*

- $R = (s, \textsf{insert}, \varepsilon, \tau)$, $O = (\textsf{insert}, \top, \top, \ell)$, *$T$ is empty, and τ matches ℓ;*
- $R = (s, \textsf{insert}, P, \tau)$, $O = (\textsf{insert}, \pi, \nu, \ell)$, $\pi \in \mathcal{S}[[P]]_T$, *and τ matches ℓ.*
- $R = (s, \textsf{rename}, P, \tau)$, $O = (\textsf{rename}, \nu, \ell)$, $\nu \in \mathcal{S}[[P]]_T$, *$\tau$ matches ℓ; or*
- $R = (s, \textsf{delete}, P)$, $O = (\textsf{delete}, \nu)$, *and* $\nu \in \mathcal{S}[[P]]_T$.

Note that rules never match operations (such as inserting a root into a non-empty document) that would yield $\perp$ on the tree.

Definition 4.1.4 *A positive rule R with path expression P is* ***active*** *on tree T if it matches some possible operation on T: that is, if $\mathcal{S}[[P]]_T$ is not the empty set; or if T is the empty tree, $P = \varepsilon$, and R is an* insert *rule.*

Definition 4.1.5 *An access control* ***policy*** *is an unordered finite set of access control rules over some finite set L of labels. A policy is* ***positive*** *if it contains only positive rules;* ***simple*** *if it contains only simple rules;* ***delete-free*** *if it contains no positive* delete *rules;* ***rename-free*** *if it contains no positive* rename *rules; and* ***monotone*** *if it is both delete-free and rename-free.*

Definition 4.1.6 *The policy $\mathcal{P}$* ***permits*** *the operation O on tree T, written $\mathcal{P} \vdash_T O$, if there exists some positive rule $R \in \mathcal{P}$ such that $R \sim_T O$ and there does not exist a negative rule $R_- \in \mathcal{P}$ such that $R_- \sim_T O$.*

Remark: Our model uses "deny overwrites" and "default deny" conflict resolution [28, 33]: if an operation is matched by both positive and negative rules, or is matched by no rule, it is not permitted.

Lemma 4.1.1 *It may be verified whether policy $\mathcal{P}$ permits operation O on tree T in time $\mathcal{O}(|\mathcal{P}|\,|T|)$.*

Proof: Gottlob, Koch, Pichler [31] describe an algorithm for evaluating core XPath expressions that runs in time $\mathcal{O}(|P|\,|T|)$, where P is the path expression and T the tree. This algorithm allows us to test whether a single rule R matches an operation O in time $\mathcal{O}(|R|\,|T|)$. To determine whether $\mathcal{P}$ permits the operation O, we test whether each rule matches O and return true if some positive rule, and no negative rule, matched; this operation requires time $\sum_{R\in\mathcal{P}} \mathcal{O}(|R|\,|T|) = \mathcal{O}(|\mathcal{P}|\,|T|)$. □

Remark: Although we have defined the result of the operation $(\mathsf{insert}, \varepsilon, \ell)$ on non-empty trees (Definition 4.1.1), Definition 4.1.3 ensures that no rule ever matches such an operation on a non-empty tree. Hence every permitted insert operation adds precisely one node to the tree, a fact that will be important later.

Remark: In order to facilitate analysis of fine-grained access control rules and policies, we have made some simplifying assumptions. First, we disregard subjects in our model of policies. Even when multiple users are present, many important

questions can be expressed in terms of subject-less policies, either by considering only rules governing a particular subject, or by considering all rules regardless of subject. Secondly, our rules ignore the issue of document order: If a rule permits inserting a new node as a child of an existing node, the new node may be inserted anywhere in the child list of the existing node.

Finally, we extend the definition of permission to sequences of operations.

Definition 4.1.7 *If $\mathcal{S} = \langle o_1, \ldots, o_n \rangle$ is a finite sequence of operations, we say that $\mathcal{P}$ permits $\mathcal{S}$ on tree T ($\mathcal{P} \vdash_T \mathcal{S}$) if $\mathcal{P} \vdash_{T_{i-1}} o_i$ for each $1 \leq i \leq n$, where $T_0 = T$ and $T_i = o_i(T_{i-1})$.*

We use these definitions to formally define the problem PGen.

Definition 4.1.8 *The language $\mathrm{L}_{\mathcal{T}}(\mathcal{P})$ generated by a policy $\mathcal{P}$ from the set $\mathcal{T}$ of trees is the set of trees T such that there exists a sequence of operations $\mathcal{S}$ and a tree $T_0 \in \mathcal{T}$ such that $\mathcal{P} \vdash_{T_0} \mathcal{S}$ and $\mathcal{S}(T_0) = T$. We write $\mathrm{L}(\mathcal{P})$ for $\mathrm{L}_{\{\mathrm{T}_\emptyset\}}(\mathcal{P})$, the language of trees generated by $\mathcal{P}$ from the empty tree.*

Definition 4.1.9 *The problem* PGen *is: Given a pair $(\mathcal{P}, T)$, where $\mathcal{P}$ is an access control policy and T is a tree, is $T \in \mathrm{L}(\mathcal{P})$?*

We also define restricted forms of this problem. PGen_+ is the restriction of PGen to positive policies; PGen_s the restriction to simple policies; PGen^i the restriction to monotone policies; $\mathrm{PGen}^{i,d}$ the restriction to rename-free policies; and $\mathrm{PGen}^{i,r}$ the restriction to delete-free policies. These restrictions may be combined to give a 4×4 table of subproblems; for example $\mathrm{PGen}^{i,d}_{+s}$ is the restriction of PGen to simple, positive, rename-free policies. We demonstrate lower and/or upper bounds on the computational complexity of each of these subproblems, as indicated in Tables 4.1 and 4.2.

4.2 Sub-problems with NP algorithms

4.2.1 Positive monotone policies

Recall from Definition 4.1.5 that a **monotone** policy is one that is both delete-free and rename-free; that is, where every positive rule specifies the insert operation. Since an insert operation adds exactly one node to the tree, any permitted sequence that yields a tree T consists of exactly $|T|$ insert operations. This fact significantly limits the decision tree for PGen^i and PGen^i_+.

Table 4.1: Subproblems of PGEN and lower bounds on their complexities. N is the size of the tree and M is the total size of the policy's rules.

	PGEN^i	PGEN^{i,r}	PGEN^{i,d}	PGEN
PGEN$_{+s}$	$\Omega(N+M)$ (input size)	NP-hard (Sec. 4.3.1)	$\Omega(N+M)$ (input size)	NP-hard (Sec. 4.3.1)
PGEN$_s$	$\Omega(N+M)$ (input size)	NP-hard (Sec. 4.3.1)	$\Omega(N+M)$ (input size)	NP-hard (Sec. 4.3.1)
PGEN$_+$	$\Omega(N+M)$ (input size)	NP-hard (Sec. 4.3.1)	PSPACE-hard (Sec. 4.3.2)	PSPACE-hard (Sec. 4.3.2)
PGEN	$\Omega(N+M)$ (input size)	PSPACE-hard (Sec. 4.4.6)	undecidable (Sec. 4.4.5)	undecidable (Sec. 4.4)

Table 4.2: Subproblems of PGEN and upper bounds on their complexities. N is the size of the tree and M is the total size of the policy's rules.

	PGEN^i	PGEN^{i,r}	PGEN^{i,d}	PGEN
PGEN$_{+s}$	$\mathcal{O}(N^3M)$ (Sec. 4.2.1)	$\in$ PSPACE (Sec. 4.3.1)	$\in$ P (Sec. 4.2.3)	$\in$ PSPACE (Sec. 4.3.1)
PGEN$_s$	$\in$ NP (Sec. 4.2.2)	$\in$ PSPACE (Sec. 4.3.1)	$\in$ NP (Sec. 4.2.3)	$\in$ PSPACE (Sec. 4.3.1)
PGEN$_+$	$\mathcal{O}(N^3M)$ (Sec. 4.2.1)	$\in$ PSPACE (Sec. 4.3.1)		
PGEN	$\in$ NP (Sec. 4.2.2)	$\in$ PSPACE (Sec. 4.3.1)		

Our algorithm for PGEN$^i_+$ (Algorithm 4.1) determines whether $\mathcal{P}$ generates T by attempting to construct T. In doing so, we keep track of the current tree S and the set C of **candidate** nodes: those nodes of T that have not yet been inserted into S, but whose parents have been. At a given step in the algorithm, the candidate nodes are those that can be inserted into S with a single (not necessarily permitted) operation.

Lemma 4.2.1 *In Algorithm 4.1, the following four invariants hold when the condition of the **while** loop on line 6 is being evaluated:*

1. $\mathrm{nodes}(S) \subseteq \mathrm{nodes}(T)$;
2. *for each non-root node* $\mu \in T$, *if* $\mu \in S$ *then* $\mathrm{parent}(\mu) \in S$;
3. $C \subseteq \mathrm{nodes}(T) \setminus \mathrm{nodes}(S)$
4. C *consists precisely of those nodes* $\mu \in \mathrm{nodes}(T) \setminus \mathrm{nodes}(S)$ *such that either* $\mu = \mathrm{root}(T)$ *or* $\mathrm{parent}(\mu) \in S$.

Figure 4.1: Algorithm for deciding PGEN^i_+ (positive monotone policies)

Require: T is a tree over the set L of labels
Require: $\mathcal{P}$ is a positive monotone policy over L

1: $S \leftarrow \mathrm{T}_\emptyset$ *// the tree generated so far*
2: **if** $T = \mathrm{T}_\emptyset$ **then**
3:    **return true**
4: **end if**
5: $C \leftarrow \{\mathrm{root}(T)\}$ *// the candidate nodes*
6: **while** $C \neq \emptyset$ **do**
7:    *found* $\leftarrow$ **false**
8:    **for all** $\nu \in C$ **do**
9:       $\pi \leftarrow \mathrm{parent}(\nu)$, or ε if $\nu = \mathrm{root}(T)$
10:       $o \leftarrow (\mathsf{insert}, \pi, \mathrm{label}(\nu))$
11:       **if** $\mathcal{P} \vdash (o, S)$ **then**
12:          $C \leftarrow (C \cup \mathrm{children}(\nu)) \setminus \{\nu\}$
13:          $S \leftarrow o(S)$
14:          *found* $\leftarrow$ **true**
15:       **end if**
16:    **end for**
17:    **if** *found* = **false then**
18:       **return false**
19:    **end if**
20: **end while**
21: **return true**

Furthermore, in each iteration, S contains more nodes than in the previous iteration.

PROOF: If T is empty, the algorithm terminates (line 3) before reaching line 6, vacuously satisfying the lemma. Suppose then that T is not empty.

On entry to the while loop, $S = \mathrm{T}_\emptyset$, satisfying invariants 1 and 2; and C contains the root node of T, satisfying invariants 3 and 4. Now suppose the invariants held on iteration i of the **while** loop and that $C \neq \emptyset$ (i.e., the loop continues); we wish to show that the invariants hold on iteration $i + 1$.

Since $C \neq \emptyset$, the **for** loop (lines 8–16) executes at least once. Each time, it may modify S and C, by removing a node ν from C, adding ν's children to C (line 12), and adding the node ν to S. Let C_i and S_i be the values of C and S before these lines are executed, and C_{i+1} and S_{i+1} their values after these lines are executed.

- Because $S_i \subseteq \mathrm{nodes}(T)$ (invariant 1) and the added node ν is in $C_i \subseteq \mathrm{nodes}(T)$ (invariant 4), $\mathrm{nodes}(S_{i+1}) \subseteq \mathrm{nodes}(T)$. Hence invariant 1 is preserved.

- Either $\nu = \text{root}(T)$ or $\text{parent}(\nu) \in S_i$ (invariant 4). If the former is true, invariant 2 continues to be satisfied, as no non-root nodes have been added to S and no nodes removed. If instead $\text{parent}(\nu) \in S_i$, then $\text{parent}(\nu) \in S_{i+1}$, and invariant 2 continues to be satisfied.

- The nodes added to C_{i+1} are all children of $\nu \in C_i$, which was not present in S_i by invariant 3. By invariant 2, then, these added nodes were not in S_i and are thus not in S_{i+1}. Furthermore, the single node ν that was added to S was removed from C in line 12. Hence $C_{i+1} \cap S_{i+1}$ is empty, and invariant 3 continues to be satisfied.

- Each node in C_{i+1} was either in C_i (hence having a parent in $\text{nodes}(S_i) \subseteq \text{nodes}(S_{i+1})$), or is a child of $\nu \in S_{i+1}$, satisfying the forward condition of invariant 4: every node in C_{i+1} is the root of T or has a parent in S.

  Conversely, suppose μ is a node in $T \setminus S_{i+1}$ such that $\text{parent}(\mu) \in S_{i+1}$ (μ cannot be the root because S_{i+1} is non-empty and hence contains the root of T by invariant 2). Then either $\text{parent}(\mu) \in S_i$ and $\mu \neq \nu$; or $\text{parent}(\mu) = \nu$. In the former case, $\mu \in C_i$, and since only ν was removed from C, $\mu \in C_{i+1}$; in the latter, μ was added to C on line 12, so $\mu \in C_{i+1}$. Hence invariant 4 is satisfied.

Therefore each iteration of the **for** loop either leaves S and C unchanged or adds a single node to S and preserves the invariants. Hence, if the invariants hold on iteration i of the **while** loop, they continue to hold on iteration $i+1$. Furthermore, if no nodes were added to S, then *found* is **false** and the algorithm terminates (line 18). Hence we furthermore have that, in each iteration, S is strictly larger than in the previous. □

Corollary 4.2.1 *When the condition on line 6 is evaluated, $C = \emptyset$ if and only if $S = T$.*

PROOF: If $S = T$, then since $C = \text{nodes}(T) \setminus \text{nodes}(S)$ by invariant 3 of Lemma 4.2.1, $C = \emptyset$.

Now suppose $C = \emptyset$. Then by invariant 4, there are no nodes in $\text{nodes}(T) \setminus \text{nodes}(S)$ that are either the root of T or have a parent in S. Hence $\text{root}(T) \in S$. Now let ν be a node of T, and let $\text{parent}(\nu) = \pi_1, \pi_2, \ldots \pi_k = \text{root}(T)$ be its chain of ancestors. Since $\pi_k \in S$ and no nodes in $\text{nodes}(T) \setminus \text{nodes}(S)$ have a parent in S, π_{k-1} is also in S, as is the entire chain and ν itself. This argument demonstrates that

every node in T is in S; along with invariant 1, we have that $T = S$. □

Theorem 4.2.1 *Algorithm 4.1 terminates in time $\mathcal{O}\left(|T|^3 |P|\right)$, returning* true *if $T \in \mathrm{L}(\mathcal{P})$, and false otherwise. Hence it is a correct polynomial-time algorithm for PGEN^i_+ (and its subproblem PGEN^i_{+s}).*

PROOF: In each iteration of the **while** loop (lines 6–20), either one or more nodes is added to S (line 13) and *found* is **true**; or no nodes are added, *found* is **false**, and the algorithm terminates in line 18. Each iteration of the **while** loop either adds one or more nodes to S from nodes(T) \ nodes(S) or terminates; because the algorithm terminates when S has the same (finite) size as T, the while loop runs for at most $|T|$ iterations before terminating. In each iteration, the inner **for** loop (lines 8–16) executes $|C| \leq |T|$ times, each time checking whether $\mathcal{P}$ contains a rule permitting a particular operation. Since this last task can be solved in time $\mathcal{O}(|T|\,|P|)$ (Lemma 4.1.1), the algorithm terminates in time $\mathcal{O}\left(|T|^3 |P|\right)$.

If the algorithm returns **true**, then $S = T$ (line 6 and Corollary 4.2.1). Since $\mathcal{P}$ permitted adding each node in S (line 11), $S = T \in \mathrm{L}(\mathcal{P})$.

If the algorithm returns **false**, then there is some tree S with nodes(S) $\subsetneq$ nodes(T), such that $\mathcal{P}$ does not allow adding to S any node from $D =$ nodes(T) \ nodes(S). Because $\mathcal{P}$ is positive, removing nodes from S can only reduce the set of active rules, and therefore does not permit adding nodes from D. Hence it is not possible to add a node from D to any tree that contains only nodes from nodes(T) \ D. Furthermore, since $\mathcal{P}$ is monotone, no tree that contains a node not in nodes(T) can possibly yield the tree T. Hence it is not possible to obtain the tree T from the empty tree by a sequence of permitted operations, and $T \notin \mathrm{L}(\mathcal{P})$. □

4.2.2 Monotone policies

If $\mathcal{P}$ is monotone but not positive, Algorithm 4.1 will not necessarily be successful. In particular, a negative rule with predicates might establish a constraint on the order of insertion of two nodes. For example, if the policy contains the rule ($-$, insert, /a[b], $*$), we must be careful not to insert the node b until every other child of a has been inserted.

Still, if T is generated by $\mathcal{P}$, there is some permitted sequence of $|T|$ insertions that results in T. The nondeterministic Algorithm 4.2 decides whether such a sequence exists.

Figure 4.2: Nondeterministic algorithm for deciding $\textsc{PGen}^i$ (monotone policies)

Require: T is a tree over the set L of labels
Require: $\mathcal{P}$ is a monotone policy over L
1: $S \leftarrow \mathrm{T}_\emptyset$
2: $\mathcal{N} \leftarrow$ nondeterministically choose a permutation of nodes(T)
3: **for all** $\nu \in \mathcal{N}$ **do**
4: $\quad \pi \leftarrow$ parent(ν), or ε if $\nu =$ root(T)
5: $\quad o \leftarrow (\mathsf{insert}, \pi, \mathrm{label}(\nu))$
6: $\quad$ **if** $\mathcal{P} \vdash_S o$ **then**
7: $\quad\quad S \leftarrow o(S)$
8: $\quad$ **else**
9: $\quad\quad$ **return false**
10: $\quad$ **end if**
11: **end for**
12: **return true**

We begin by nondeterministically choosing a permutation of the nodes of T (line 2). We then test whether $\mathcal{P}$ permits inserting the nodes in that order (lines 3–11). If this test succeeds for some permutation, then $\mathcal{P}$ generates T (line 12); otherwise it does not.

Theorem 4.2.2 *Nondeterministic Algorithm 4.2 always terminates in time polynomial in $|T|$ and $|\mathcal{P}|$, returning **true** if $T \in \mathrm{L}(\mathcal{P})$ and **false** otherwise. It is thus a correct algorithm for* $\textsc{PGen}^i$ *and its subproblem* $\textsc{PGen}^i_s$.

PROOF: The loop that tests whether the nodes of T may be inserted in a particular order (lines 3–11) runs for $|T|$ iterations. In each iteration we check whether a particular insertion is permitted; this check may be performed in polynomial time (Lemma 4.1.1). Hence, for any choice of $\mathcal{N}$, the loop terminates after a polynomial amount of time.

Since $\mathcal{P}$ is monotone, if $T \in \mathrm{L}(\mathcal{P})$, then T can be obtained by some sequence of $|T|$ insert operations. In this case, for the corresponding $\mathcal{N}$ all insertions are be permitted, the **for** loop completes, and the algorithm returns **true** on line 12.

If $T \notin \mathrm{L}(\mathcal{P})$, no sequence of operations results in T. Hence for each permutation of nodes(T), some node cannot be inserted, and each execution path of Algorithm 4.2 returns **false**. $\square$

4.2.3 Simple policies with delete

An important feature of simple rules is that whether or not a node is matched by such a rule depends only on the labels of that node and its ancestors:

Definition 4.2.1 *The* ***label path****,* $\mathrm{LP}(\nu)$*, of a node* ν *is the sequence of labels* $\langle \ell_k, \cdots, \ell_0 \rangle$*, where* k *is the depth of* ν*,* ℓ_0 *is the label of* ν*, and* ℓ_i *is the label of the* i*th ancestor of* ν*.*

Lemma 4.2.2 *Let* $\nu_1 \in T_1$ *and* $\nu_2 \in T_2$ *be two nodes, possibly from different trees, with the same label path; and let* P *be a predicate-free path expression. Then* $\nu_1 \in P(T_1)$ *if and only if* $\nu_2 \in P(T_2)$*.*

PROOF: A predicate-free path expression is of the form $a_1 t_1 \ldots a_n t_n$, where each a_i is either / or //, and each t_i is either a label ℓ or the symbol $*$. Such an expression selects a node ν if and only if $\mathrm{LP}(\nu)$ is of the form $p[o_1]p[a_1] \ldots p[o_n]p[a_n]$, where $p[/]$ is the empty sequence of labels; $p[//]$ is an arbitrary sequence of labels; $p[\ell]$ is the label ℓ; and $p[*]$ is a single arbitrary label. Since ν_1 and ν_2 have the same label path, P selects either both nodes or neither. $\square$

Another important property of simple policies is that delete rules may be ignored: any permitted sequence of operations may be converted into a permitted sequence of non-delete operations that results in the same tree.

Lemma 4.2.3 *Let* $\mathcal{P}$ *be a simple policy. Define* $\mathcal{P}_M$ *as the policy obtained by removing all* delete *rules from* $\mathcal{P}$*:*

$$\mathcal{P}_M = \mathcal{P} \cap \Big(\{+, -\} \times \{\textit{insert}, \textit{rename}\} \times \mathrm{XP}^{\{[],*,//\}}\Big) \quad ,$$

Then $\mathrm{L}(\mathcal{P}_M) = \mathrm{L}(\mathcal{P})$*.*

PROOF: If $\mathcal{P}_M$ permits some sequence of operations, that sequence consists only of insert and rename operations, so $\mathcal{P}$ permits the same sequence. Therefore, $\mathrm{L}(\mathcal{P}_M) \subseteq \mathrm{L}(\mathcal{P})$.

Now consider the converse; suppose there is some sequence of operations $\mathcal{Q}$ such that $\mathcal{P} \vdash \mathcal{Q}$. For each delete operation $o \in \mathcal{Q}$, remove that operation as well as every preceding operation that inserted the target ν_o of o or a descendant of ν_o. The resulting sequence $\mathcal{Q}_M$ of operations produces the same tree as $\mathcal{Q}$. Furthermore, by Lemma 4.2.2, any of the remaining operations that was permitted by $\mathcal{P}$ is also

permitted by $\mathcal{P}_M$, as its target is outside the deleted subtrees and the path expressions in the policy's rules do not contain predicates. Hence $\mathcal{P} \vdash \mathcal{Q}$, and since $\mathcal{Q}$ contains no delete operations, $\mathcal{P}_M \vdash \mathcal{Q}$; thus $\mathrm{L}(\mathcal{P}) = \mathrm{L}(\mathcal{P}_M)$.

□

As a result, any instance of $\textrm{PG}\textsc{en}_s^{i,d}$ (resp. $\textrm{PG}\textsc{en}_{+s}^{i,d}$) may in polynomial time be converted into an instance of $\textrm{PG}\textsc{en}_s^{i}$ (resp. $\textrm{PG}\textsc{en}_{+s}^{i}$); and any instance of $\textrm{PG}\textsc{en}_s$ (resp. $\textrm{PG}\textsc{en}_{+s}$) into an instance of $\textrm{PG}\textsc{en}_s^{i,r}$ (resp. $\textrm{PG}\textsc{en}_{+s}^{i,r}$). We can therefore extend the results of Theorems 4.2.1 and 4.2.2.

Corollary 4.2.2 $\textrm{PG}\textsc{en}_{+s}^{i,d}$ *is in* P*, and* $\textrm{PG}\textsc{en}_s^{i,d}$ *is in* NP*.*

4.3 Lower bounds for more complicated subproblems

4.3.1 Delete-free policies

Delete-free policies extend the monotone policies by adding rules that permit rename operations. The presence of such rules allows modifying generated trees by changing the labels of nodes, even internal nodes. This ability makes solving $\textrm{PG}\textsc{en}_+^{i,r}$ more difficult than solving $\textrm{PG}\textsc{en}_+^{i}$. We show that, in fact, the problem is NP-hard, by reduction from the classic vertex cover problem.

Definition 4.3.1 *The* ***vertex cover*** *problem* VC *is: Given a graph $G = (V, E)$ and a natural number k, is there a set $C \subset V$ of size $\leq k$ such that, for each $(v_i, v_j) \in E$, at least one of v_i, v_j is in C?*

Given an instance $(G = (V = \langle v_1, \ldots, v_n \rangle, E), k)$ of the vertex cover problem, we construct a simple positive delete-free policy $\mathcal{P}_{G,k}$ over the language $V \cup E \cup \{\mathsf{root}, \mathsf{vtx}\}$ of element labels.

$$\mathcal{P}_{G,k} = (+,\ \mathsf{insert},\ \varepsilon,\ \mathsf{root}) \tag{4.1}$$

For each $v_i \in V$:

$$(+,\ \mathsf{insert},\ //*,\ v_i) \tag{4.2}$$

$$(+,\ \mathsf{rename},\ //v_i,\ \mathsf{vtx}) \tag{4.3}$$

For each $e_j = (u, v) \in E$:

$$(+,\ \mathsf{insert},\ //u,\ e_j) \tag{4.4}$$

$$(+,\ \mathsf{insert},\ //v,\ e_j) \tag{4.5}$$

$$(+,\ \mathsf{insert},\ //u//*,\ e_j) \tag{4.6}$$

$$(+,\ \mathsf{insert},\ //v//*,\ e_j) \tag{4.7}$$

We ask if this policy generates the tree $T_{G,k}$ containing a root node labelled root, a chain of k descendants labelled vtx, and for each $e \in E$, one child of the kth descendant labelled e (Figure 4.3(c)).

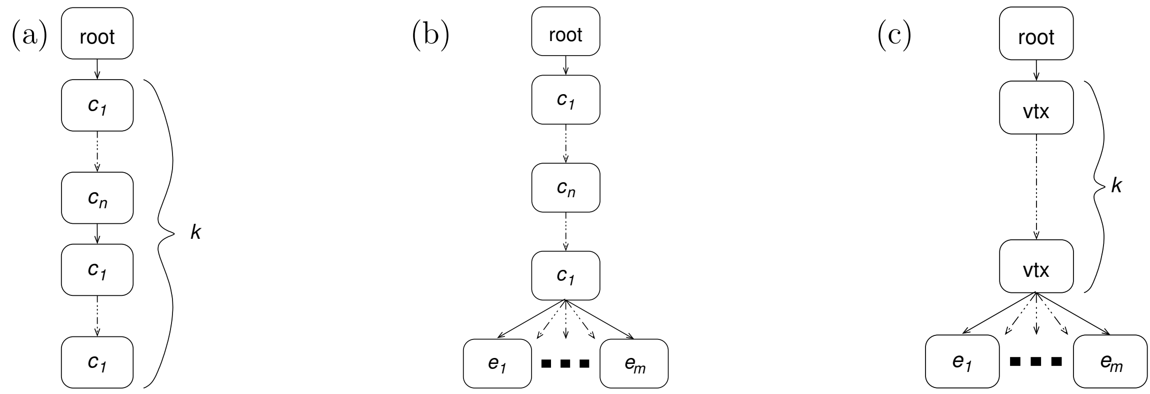

Figure 4.3: Steps of the vertex cover simulation. (a) Rules 4.1 and 4.2 allow inserting a root root and a chain of k vertex nodes. (b) Rules 4.4–4.7 permit adding nodes for covered edges. (c) Rule 4.3 permits renaming vertex nodes, yielding $T_{G,k}$.

Theorem 4.3.1 *$T_{G,k} \in \mathrm{L}(\mathcal{P}_{G,k})$ if and only if $(G, k) \in \mathrm{VC}$.*

Proof: First suppose $(G, k) \in \mathrm{VC}$, so G has a cover $C = \langle c_1, \ldots, c_n \rangle$ of size $n \leq k$. The following sequence of operations generates $T_{G,k}$ from the empty tree:

$$\begin{aligned}
\nu_0 &= (\mathsf{insert}, \varepsilon, \mathsf{root}) && \\
\zeta_1 &= (\mathsf{insert}, \nu_0, c_1) && \\
\zeta_i &= (\mathsf{insert}, \zeta_{i-1}, c_i) && \text{for } i = 2 \ldots n \\
\zeta_j &= (\mathsf{insert}, \zeta_{j-1}, c_1) && \text{for } j = n \ldots k \\
& \quad (\mathsf{insert}, \zeta_k, e_i) && \text{for } e_i \in E \\
& \quad (\mathsf{rename}, \zeta_i, \mathsf{vtx}) && \text{for } i = 1 \ldots k.
\end{aligned}$$

The first operation is permitted by Rule 4.1. The next k operations are permitted by instances of Rule 4.2, and yield the tree in Figure 4.3(a). Since C is a vertex cover of G, at least one endpoint of each edge is contained in C, so the next $|E|$ operations are permitted by instances of Rules 4.4–4.7, yielding the tree in Figure 4.3(b). Finally, the k rename operations are all permitted by instances of Rule 4.3, and yield the tree $T_{G,k}$ in Figure 4.3(c). Thus this sequence of operations is permitted, and $T_{G,k} \in \mathrm{L}\left(\mathcal{P}_{G,k}\right)$.

Conversely, suppose $\mathcal{P}_{G,k}$ generates the tree $T_{G,k}$. For each edge $e \in E$, this tree contains a node labelled e. This node could only have been created by an instance of Rules 4.4–4.7; hence some predecessor tree contained a node labelled with one of the endpoints of e. Since nodes labelled with vertices cannot be removed, only inserted or renamed to vtx, and since $T_{G,k}$ contains no nodes with vertex labels, each such endpoint node is reflected in one of the vtx nodes in $T_{G,k}$. Because there are k such V nodes, at most k different vertex nodes belonged to the collection of predecessor trees. Hence there is some subset of k or fewer vertices that covers every $e \in E$, and $(G, k) \in \mathrm{VC}$. □

Together with the fact that the instance $(\mathcal{P}_{G,k}, T_{G,k})$ may be constructed in polynomial time, and that VC is NP-complete [44], we have:

Corollary 4.3.1 $\mathrm{PG{\scriptsize EN}}^{i,r}_{+s}$ *(and thus* $\mathrm{PG{\scriptsize EN}}^{i,r}_{s}$*,* $\mathrm{PG{\scriptsize EN}}^{i,r}_{+}$*,* $\mathrm{PG{\scriptsize EN}}_{+s}$*, and* $\mathrm{PG{\scriptsize EN}}_{s}$*) is* NP*-hard.*

There remains the question of whether this bound is tight; that is, whether $\mathrm{PG{\scriptsize EN}}^{i,r}_{+s}$ is in NP. There is some evidence that it may not be: we can construct a simple positive delete-free policy and tree such that the tree is generated by the policy, but only after exponentially many operations. The following policy generates every tree with labels from the set $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, but generating a tree of height h requires $\mathcal{O}\left(2^h\right)$ operations.

$$
\begin{aligned}
\mathcal{P}_{\mathrm{exp}} = &(+,\ \mathsf{insert},\ \varepsilon,\ \mathsf{a}) && (+,\ \mathsf{insert},\ //\mathsf{c},\ \mathsf{a}) \\
&(+,\ \mathsf{rename},\ /\mathsf{a},\ \mathsf{b}) && (+,\ \mathsf{rename},\ //\mathsf{c}/\mathsf{a},\ \mathsf{b}) \\
&(+,\ \mathsf{rename},\ /\mathsf{b},\ \mathsf{c}) && (+,\ \mathsf{rename},\ //\mathsf{b}/\mathsf{b},\ \mathsf{c}) \\
&(+,\ \mathsf{rename},\ /\mathsf{c},\ \mathsf{a}) && (+,\ \mathsf{rename},\ //\mathsf{a}/\mathsf{c},\ \mathsf{a})
\end{aligned}
$$

The existence of such policies eliminates the possibility of an NP algorithm that uses as a witness of membership the sequence of operations generating the tree. However, we can establish an upper bound by noting that no intermediate tree can grow larger

than the final tree. This bound allows us to decide the general delete-free case with a polynomial-space algorithm.

Algorithm 7 Nondeterministic algorithm for deciding $\text{PG}\textsc{en}^{i,r}$

Require: T is a tree over the set L of labels
Require: $\mathcal{P}$ is a delete-free policy over L

```
 1: S ← T_∅
 2: while |S| ≤ |T| ∧ S ≠ T  do
 3:    ℓ ← nondeterministically choose a label from L
 4:    if S = T_∅ then
 5:       o ← (insert, ε, ℓ)
 6:    else
 7:       ν ← nondeterministically choose a node of S
 8:       x ← nondeterministically choose insert or rename
 9:       o ← (x, ν, ℓ)
10:    end if
11:    if P ⊬_S o then
12:       return  false
13:    end if
14:    S ← o(S)
15: end while
16: return  S = T
```

Lemma 4.3.1 *Nondeterministic Algorithm 7 requires space polynomial in the size of its input, and returns* ***true*** *if and only if* $T \in \mathrm{L}(\mathcal{P})$.

Proof: First, note that this algorithm requires only space to store the policy $\mathcal{P}$, the current intermediate tree S, and a single operation on this tree; and to check whether the operation is permitted. The number of nodes in S is bounded by the size of the input tree T and the operation can be checked in polynomial time (and hence polynomial space); therefore the algorithm requires space polynomial in the size of the input.

Suppose that $T \in \mathrm{L}(\mathcal{P})$. Then there is some finite permitted sequence $\mathcal{Q}$ of insert and rename operations such that $\mathcal{Q}(\mathrm{T}_\emptyset) = T$, and a corresponding sequence of trees $\mathcal{S}$. The first such operation must be an insert operation with target ε; each subsequent operation must have as its target a node in the corresponding tree. Therefore it is possible to nondeterministically select that operation in (lines 3–10). Since $\mathcal{Q}$ is permitted, the condition in line 11 is never true, and in each iteration line 14 produces the next tree in $\mathcal{S}$. After applying the last such operation, we have $S = T$, thus ending

the **while** loop (line 2) and returning **true** (line 16). Therefore there is an accepting computation path.

Now suppose that $T \notin \mathrm{L}(\mathcal{P})$. Then there is no such permitted sequence $\mathcal{Q}$; since non-permitted operations are excluded by lines 11–12, we can never produce the final tree T in line 14. Hence there is no accepting computation path. □

Finally, we note that Algorithm 7 need not terminate: a sequence of rename operations may leave S unchanged after one or more iterations of the loop, thus resulting in an infinite computation path. However, the space upper bound allows us to limit the number of iterations. Let $s(\mathcal{P}, T) \in \mathcal{O}\left(|(\mathcal{P}, T)|^k\right)$ be the number of bits of space required by the algorithm. Then, if any computation path succeeds, one does so after no more than $2^{s(\mathcal{P},T)}$ iterations. Hence, after this many iterations we can be certain that the algorithm will never return **true**; the algorithm can thus terminate and return **false**. This argument gives us:

Theorem 4.3.2 $\mathrm{PG{\scriptsize EN}}^{i,r}$ *and its subproblems* $\mathrm{PG{\scriptsize EN}}_{+}^{i,r}$, $\mathrm{PG{\scriptsize EN}}_{s}^{i,r}$, *and* $\mathrm{PG{\scriptsize EN}}_{+s}^{i,r}$ *are in* PSPACE.

Lemma 4.2.3 allows us to extend the result for $\mathrm{PG{\scriptsize EN}}_{s}^{i,r}$ to policies that also include delete operations:

Corollary 4.3.2 $\mathrm{PG{\scriptsize EN}}_{s}$ *and its subproblem* $\mathrm{PG{\scriptsize EN}}_{+s}$ *are in* PSPACE.

4.3.2 Positive policies

We saw in Lemma 4.2.3 that, in simple policies, delete operations can be ignored entirely; however, the presence of predicates in more general classes of policies requires that we take such operations into account. As a result, generating a desired tree might require constructing a much larger intermediate tree, making the problem $\mathrm{PG{\scriptsize EN}}_{+}$ substantially more difficult than $\mathrm{PG{\scriptsize EN}}_{+}^{i}$.

We show that $\mathrm{PG{\scriptsize EN}}_{+}^{i,d}$ (and thus $\mathrm{PG{\scriptsize EN}}_{+}$) is PSPACE-hard by showing a polynomial-time reduction from a well-known PSPACE-complete problem. We choose the **true quantified Boolean formula** problem (TQBF) for this purpose. TQBF is the problem of determining whether a first-order sentence over a finite set of Boolean variables (a so-called **quantified Boolean formula** or QBF) is true. We assume that the formula is presented in prenex negation-normal form (PNNF) [29]. That is, the formula consists of a number of quantifiers governing an unquantified propositional formula ϕ; and in ϕ the Boolean negation operator always has a single atom as its

operand. By repeatedly lifting quantifiers, renaming clashing variables, and applying De Morgan's laws, it is always possible to convert a QBF into an equivalent PNNF formula in polynomial time [29]. Hence this assumption does not affect the run-time complexity of TQBF.

Let $F = Q_1x_1 \ldots Q_kx_k\phi(x_1, \ldots, x_k)$ be a QBF in PNNF. Write X for the set $\{x_1, \ldots, x_k\}$ of variables. We construct a policy $\mathcal{P}_F$, with size polynomial in the size of F, that generates the tree $/\mathsf{F}/\mathsf{proved}$ if and only if F is true.

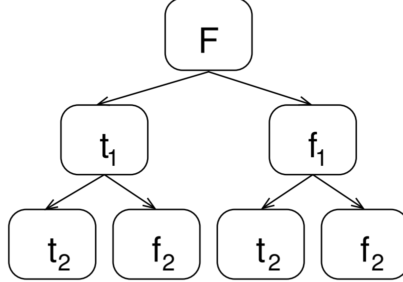

Figure 4.4: Semantic tree for a QBF on two variables.

First, the policy $\mathcal{P}_F$ allows constructing a tree that represents certain partial valuations of X. This **semantic tree** corresponds to the decision tree used by many QBF solvers [29]. In this tree, the node t_i represents that variable x_i is true in this particular computation path, and f_i that it is false. As an example, the semantic tree for a formula on two variables (and hence with two quantifiers) is shown in Figure 4.4. The portion of the policy that constructs this semantic tree, $\mathcal{P}_T$, consists of $3 + 4k$ rules:

$$\mathcal{P}_T = \begin{cases} (+,\ \mathsf{insert},\ \varepsilon,\ \mathsf{F}) \\ (+,\ \mathsf{insert},\ /\mathsf{F},\ \mathsf{t}_1) \\ (+,\ \mathsf{insert},\ /\mathsf{F},\ \mathsf{f}_1) \\ \text{For each } i, \quad 1 < i \leq k: \\ \quad (+,\ \mathsf{insert},\ //\mathsf{t}_{i-1},\ \mathsf{t}_i) \\ \quad (+,\ \mathsf{insert},\ //\mathsf{t}_{i-1},\ \mathsf{f}_i) \\ \quad (+,\ \mathsf{insert},\ //\mathsf{f}_{i-1},\ \mathsf{t}_i) \\ \quad (+,\ \mathsf{insert},\ //\mathsf{f}_{i-1},\ \mathsf{f}_i) \end{cases}$$

The semantic tree algorithm for TQBF operates by recursively splitting the QBF into two subproblems with fewer quantifiers: $Q_ix_i\,F'$ becomes $F'[x_i = 0]$ and $F'[x_i = 1]$. The nodes generated by $\mathcal{P}_T$ correspond to these subproblems.

Definition 4.3.2 *Let ν be a node in some tree generated by the policy $\mathcal{P}_T$. The* ***node formula*** *of ν, $\text{form}(\nu)$, is defined recursively.*

- *If ν is the root node (labelled F), then $\text{form}(\nu) = F$.*
- *Otherwise, ν has the label t_i or f_i, for some $1 \leq i \leq k$. Let v_ν be 0 (false) if the label is f_i and 1 (true) if the label is t_i. Furthermore, let G be the node formula of ν's parent; this formula has the form $Q_i x_i \cdots Q_k x_k\, \phi_G$, where ϕ_G is an unquantified formula over the variables $x_i, \ldots, x_k$. Then the node formula of ν is*

  $$Q_{i+1} x_{i+1} \ldots Q_k x_k \phi_G[x_i = v_\nu].$$

  It may be verified that this formula has the required form.

We additionally have a subpolicy that allows us to perform unquantified deduction at the leaves of the semantic tree. These leaves, labelled t_k and f_k, have unquantified node formulae. We perform structural induction on these formulae, inserting a node corresponding to a subformula only if that formula can be proved from the valuation or from already-proved subformulae. This unqualified deduction policy $\mathcal{P}_\phi$ consists of at most $2m + 3$ rules, where m is the size of ϕ (the number of subformulae):

$$\mathcal{P}_\phi = \begin{cases} (+,\ \mathsf{insert},\ //\mathsf{t}_k,\ \mathsf{phi}) \\ (+,\ \mathsf{insert},\ //\mathsf{f}_k,\ \mathsf{phi}) \\ (+,\ \mathsf{insert},\ //*[\mathsf{phi}/\mathsf{psi}_1],\ \mathsf{proved}) \\ R(i) \quad \text{for each subformula } \psi_i \text{ of } \phi \end{cases}$$

Where, for each subformula ψ_i of ϕ, $R(i)$ consists of one or two rules:

$$R(i) = \begin{cases} (+,\ \mathsf{insert},\ //\mathsf{t}_j//\mathsf{phi},\ \mathsf{psi}_i) & \psi_i = x_j \\ (+,\ \mathsf{insert},\ //\mathsf{f}_j//\mathsf{phi},\ \mathsf{psi}_i) & \psi_i = \neg x_j \\ \big(+,\ \mathsf{insert},\ //\mathsf{phi}[\mathsf{psi}_h][\mathsf{psi}_j],\ \mathsf{psi}_i\big) & \psi_i = \psi_h \wedge \psi_j \\ (+,\ \mathsf{insert},\ //\mathsf{phi}[\mathsf{psi}_h],\ \mathsf{psi}_i)\,, & \\ \quad \big(+,\ \mathsf{insert},\ //\mathsf{phi}[\mathsf{psi}_j],\ \mathsf{psi}_i\big) & \psi_i = \psi_h \vee \psi_j \end{cases}$$

Lemma 4.3.2 *Let ν be a leaf node of the semantic tree constructed by policy $\mathcal{P}_T$. The policy $\mathcal{P}_\phi$ permits inserting a child node of ν labelled* proved *if and only if $\text{form}(\nu)$ is true.*

Proof: By Definition 4.3.2, the node formula of a leaf node is an unquantified formula $\phi[V_\nu]$, where V_ν is a complete valuation. This valuation is determined by the label path of ν: $V(x_i) = 0$ if f_i appears in the label path, or 1 if t_i appears. It may be verified that exactly one of these two labels is present in the label path of each leaf node ν. Furthermore, since ν is a leaf node, its label is t_k or f_k, so $\mathcal{P}_\phi$ permits inserting a child labelled phi.

Since ϕ is in negation-normal form, each subformula is either a variable, the negation of a variable, or the conjunction or disjunction of two other subformulae. A subformula consisting of a variable x_i is true if and only if that variable is true in the node formula of ν: that is, if t_i appears as an ancestor. Likewise the subformula $\neg x_i$ is true if and only if f_i appears. The rules of $R(i)$ allow us to insert a node representing the subformula ψ_i in precisely these cases. Furthermore, a conjunction is true if and only if both of its subformulae are true; and a disjunction if and only if at least one of its subformulae is true. Hence $R(i)$ allows us to insert a node corresponding to subformula ψ_i if and only if ψ_i is true in valuation V_ν. In particular, we can insert the node psi_1 if and only if $\psi_1 = \phi$ is true in valuation V_ν.

Finally, the third rule of $\mathcal{P}_\phi$ allows us to insert a proved child of ν when and only when we have inserted the psi_1 node. Therefore, we can insert this node if and only if $\phi[V_\nu] = \text{form}(\nu)$ is true. □

As an example, consider the QBF $F = \forall x_1 \exists x_2 (x_1 \vee \neg x_2)$. Figure 4.4 demonstrates the semantic tree for F, which may be constructed following $\mathcal{P}_T$. The subpolicy $\mathcal{P}_\phi$ allows inserting nodes labelled phi as children of each leaf of the semantic tree, and children of these phi nodes representing the unquantified subformulae x_1, $\neg x_2$, and $x_1 \vee \neg x_2$ that are true in those branches of the tree (Figure 4.5). For example, a node corresponding to $\neg x_2$ may be inserted only in those parts of the semantic tree where x_2 is false: that is, as a descendant of f_2.

Once we have proved some of the unquantified node formulae at the t_k and f_k nodes, we can perform logical inference up the tree, following the semantic tree algorithm for TQBF. The presence of a child node named proved indicates that the node formula of a node ν has been proved. In Figure 4.5, for example, a proved node may be inserted as the child of each shaded node, in bottom-up order. The quantified deduction subpolicy $\mathcal{P}_Q$ consists of between k and $2k$ rules, depending on the quantifiers of F.

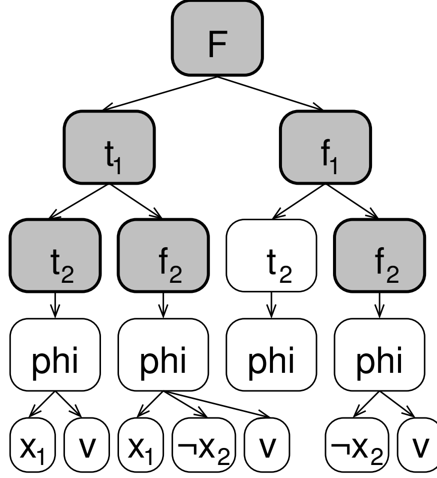

Figure 4.5: Tree for a QBF and its subformulae. The formula is $F = \forall x_1 \exists x_2 (x_1 \vee \neg x_2)$; shaded nodes correspond to true quantified formulae.

$$\mathcal{P}_Q = \begin{cases} \text{For each } Q_i, \quad 1 < i \leq k\text{:} \\ \text{If } Q_i = \forall\text{:} \\ \quad (+,\ \mathsf{insert},\ //*[\mathsf{t}_i/\mathsf{proved}][\mathsf{f}_i/\mathsf{proved}],\ \mathsf{proved})\,; \\ \text{otherwise, } Q_i = \exists\text{:} \\ \quad (+,\ \mathsf{insert},\ //*[\mathsf{t}_i/\mathsf{proved}],\ \mathsf{proved}) \\ \quad (+,\ \mathsf{insert},\ //*[\mathsf{f}_i/\mathsf{proved}],\ \mathsf{proved}) \end{cases}$$

Lemma 4.3.3 *In the policy $\mathcal{P}_M = \mathcal{P}_T \cup \mathcal{P}_\phi \cup \mathcal{P}_Q$, it is possible to insert a child* proved *of a node ν only if the node formula of ν is a true QBF.*

Proof: We saw in Lemma 4.3.2 that the statement holds for each node ν with an unquantified node formula. This result establishes a basis for our induction.

Now suppose ν has a node formula with $j > 0$ quantifiers, and that our statement holds for every node formula with fewer than j quantifiers. Then form(ν) has the form $Q_{k+1-j} x_{k+1-j} \Phi$, where Q_{k+1-j} is a quantifier and Φ is a possibly quantified formula with the free variable x_{k+1-j}. Furthermore, ν's children have the node formulae $\Phi[x_i = 0]$ and $\Phi[x_i = 1]$.

If $Q_i = \forall$, then $Q_i x_i \Phi$ is true if and only if $\Phi[x_i = 0]$ and $\Phi[x_i = 1]$, the node formulae of ν's children, are both true. By hypothesis, each of these children can contain a proved node if and only if its node formula is true; hence the first rule schema of $\mathcal{P}_Q$ allows us to insert a proved child of ν only if form(ν) is true.

Otherwise, $Q_i = \exists$; then form(ν) is true if and only if at least one of the children's node formulae is true. The second and third rule schemata of $\mathcal{P}_Q$ allow inserting a proved node only if one of the children contains a proved node. Again, since by our inductive hypothesis these children can contain proved nodes if and only if they are true, the same is true of form(ν). Therefore, ν can contain a proved child if and only if form(ν) is true. □

We have seen that the positive monotone policy $\mathcal{P}_M$ allows constructing a tree that models the decision tree and inference rules of a QBF solver—and hence the proof of a QBF F. In particular, this policy allows constructing a tree containing a node with label path F proved if and only if F is a true QBF. However, the tree in question may be huge: in the worst case, it has size exponential in the number of quantifiers in F. We add a single delete rule that permits paring down this tree to a fixed size.

$$\mathcal{P}_D = (+,\ \mathsf{delete},\ /\mathsf{F}[\mathsf{proved}]//*)$$

Theorem 4.3.3 $\text{PG}\textsc{en}_+^{i,d}$ *(and thus* $\text{PG}\textsc{en}_+$*) is* PSPACE*-hard.*

Proof: Let F be an instance of TQBF. The policy $\mathcal{P}_F$ consists of a number of rules polynomial in the number of subformulae of F. If F is not true, we cannot insert a proved child of F, and $\mathcal{P}_D$ has no effect. If F is true, then such a node can be generated by Lemma 4.3.3, and the remaining nodes can be deleted by $\mathcal{P}_D$. Hence the tree T_F consisting of a root node F with a single child proved can be generated by $\mathcal{P}_F$ if and only if F is a true quantified Boolean formula. This construction establishes a polynomial-time many-to-one reduction from TQBF to $\text{PG}\textsc{en}_+^{i,d}$, so the latter problem is PSPACE-hard. □

4.4 General policies

The general problem PGEN combines the features studied in previous sections: insert, rename, and delete operations, positive and negative rules, and predicates. The combination of these features (or, as we shall see in Section 4.4.5, all of these features except rename operations) results in a much more complex language of generated trees. In fact, we prove that PGEN is undecidable. As in Section 4.2, we proceed by reduction from a problem of known complexity, in this case the halting problem for

deterministic Turing machines. We show how to encode a Turing machine M and initial tape S as an access control policy $\mathcal{P}_{M,S}$ such that $\mathcal{P}_{M,S}$ generates a particular tree T_{halt} if and only if M halts on input S.

We represent a Turing machine as a 7-tuple of a set of states, an initial state, a set of final (accepting) states, a tape alphabet, an input alphabet, a blank symbol, and a transition function.

$$M = (Q, q_0 \in Q, Q_F \subseteq Q, \Gamma, \Sigma \subset \Gamma, b \in \Gamma \setminus \Sigma, \delta)$$

We consider deterministic Turing machines with left and right moves only, where no transitions are permitted from final states: that is, the transition function δ maps from $(Q \setminus Q_F) \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$. Furthermore, we assume the tape is bounded on the left.

Definition 4.4.1 *A* ***configuration*** *is a tuple (q, t, p) of a state $q \in Q$, a tape $t \in \Gamma^*$, and a tape position $1 \leq p \leq |t|$. We write $\mathcal{C}_0^{M,S}$ for the initial configuration of Turing machine M on input S, namely $(q_0, S, 1)$. We write Δ for the function taking a configuration to the successive configuration.*

4.4.1 Modelling Turing configurations as trees

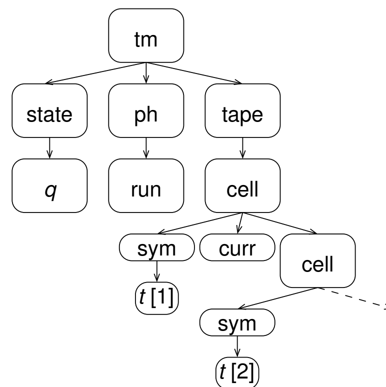

Figure 4.6: A configuration tree

A tree generated by $\mathcal{P}_{M,S}$ represents a configuration of M, as well as additional bookkeeping information necessary to the simulation. Figure 4.6 shows an example of (part of) such a configuration tree.

Definition 4.4.2 *Let $\mathcal{C} = (q, t, p)$ be a configuration of the Turing machine $M = (Q, q_0, Q_F, \Gamma, b, \Sigma, \delta)$. Write K for the larger of the input tape size and the index of the rightmost visited cell. The* ***configuration tree*** conftree($\mathcal{C}$) *is the tree with:*

- *a root node labelled* tm*;*
- *three children, labelled* ph*,* state*, and* tape*;*
- *one child of the* ph *node, labelled* run*;*
- *one child of the* state *node, labelled q;*
- *one child C_1 of the* tape *node, labelled* cell*;*
- *each* cell *node C_i $(1 \leq i \leq K)$ having:*
  - *one child labelled* sym*, itself having a child labelled with the tape symbol $t[i]$;*
  - *if $i < K$, one child C_{i+1} labelled* cell*;*
  - *if $i = p$, one child labelled* curr*.*

Cells to the right of cell K all have label b; they are omitted from the tree, but will be added if the cell is subsequently visited.

4.4.2 The policy $\mathcal{P}_{M,S}$

Let $M = (Q, q_0, Q_F, \Gamma, \Sigma, b, \delta)$ be a Turing machine, and $S = S_1S_2 \ldots S_n \in \Sigma^*$ an initial tape for M. The policy $\mathcal{P}_{M,S}$ simulates the action of M on input S. In the following presentation of the policy we use the notation $(x)^n$ to represent n repetitions of the path expression fragment x. $\mathcal{P}_{M,S}$ consists of the subpolicies $\mathcal{P}_{\text{struct}}$, $\mathcal{P}_{\text{init}}$, $\mathcal{P}_{\text{build}}$, $\mathcal{P}_{\text{b}\to\text{r}}$, $\mathcal{P}_{\text{run}}$, $\mathcal{P}_{\text{write}}$, $\mathcal{P}_{\text{move}}$, and $\mathcal{P}_{\text{cleanup}}$.

We begin with a subpolicy $\mathcal{P}_{\text{struct}}$ consisting of a number of negative rules that enforce certain constraints on the tree. These **structural rules** ensure that there is only one copy of each top-level node (ph, state, and tape); that the state and ph nodes have only a single child each; that no tape cell contains two symbols or two successor tape cells; and that no cell is marked as "current" or "new" twice.

$$
\mathcal{P}_{\text{struct}} = \begin{array}{llll}
(-,\ \mathsf{insert},\ \mathsf{/tm[ph]},\ \mathsf{ph}) & (4.8) & (-,\ \mathsf{insert},\ \mathsf{//{*}[cell]},\ \mathsf{cell}) & (4.13) \\
(-,\ \mathsf{insert},\ \mathsf{/tm[state]},\ \mathsf{state}) & (4.9) & (-,\ \mathsf{insert},\ \mathsf{//cell[curr]},\ \mathsf{curr}) & (4.14) \\
(-,\ \mathsf{insert},\ \mathsf{/tm[tape]},\ \mathsf{tape}) & (4.10) & (-,\ \mathsf{insert},\ \mathsf{//cell[new]},\ \mathsf{new}) & (4.15) \\
(-,\ \mathsf{insert},\ \mathsf{/tm/ph[*]},\ *) & (4.11) & (-,\ \mathsf{insert},\ \mathsf{//cell[sym]},\ \mathsf{sym}) & (4.16) \\
(-,\ \mathsf{insert},\ \mathsf{/tm/state[*]},\ *) & (4.12) & (-,\ \mathsf{insert},\ \mathsf{//sym[*]},\ *) & (4.17)
\end{array}
$$

4.4.3 Simulation phases

Our Turing machine simulation proceeds from one configuration tree to the next by proceeding through a number of **phases**. The tree's current phase is indicated by the

label of the child of the ph node. In each phase, a particular sequence of operations is permitted, ending with a transition into the successor phase. There are three independent phases build, run, and cleanup; and $2|Q \setminus Q_F| \cdot |\Gamma|$ phases that depend on the simulated machine's state q and the contents γ of its current tape cell; these phases are labelled write-q-γ and move-q-γ.

The build phase

At the beginning of our simulation, the tree is empty, and hence does not have a phase. In this stage, the subpolicy $\mathcal{P}_{\text{init}}$ constructs enough of the tree to enter the build phase:

$$\mathcal{P}_{\text{init}} = \begin{array}{ll} (+,\ \textsf{insert},\ \varepsilon,\ \textsf{tm}) & (4.18) \\ (+,\ \textsf{insert},\ \textsf{/tm},\ \textsf{ph}) & (4.19) \\ (+,\ \textsf{insert},\ \textsf{/tm/ph},\ \textsf{build}) & (4.20) \end{array}$$

Because every other positive rule in $\mathcal{P}_{M,S}$ requires the existence of a child of the /tm/ph node, only the rules in this subpolicy are active at this point. We therefore have the following:

Lemma 4.4.1 *Any sufficiently long sequence of operations that is permitted on the null tree produces an intermediate tree containing exactly: a* tm *root, one* ph *child of the root, and one* build *child of that node.*

Remark: Because an operation that would insert a root node is not permitted on a non-empty tree, Rule 4.18 can be used only once. Likewise, structural Rules 4.8 and 4.11 prevent Rules 4.19 and 4.20 from being activated again as long as the tree contains a ph node. Hence we shall disregard $\mathcal{P}_{\text{init}}$ in the following lemmas.

In the build phase, we complete the initial configuration tree, including the state and tape nodes and their descendants. Subpolicy $\mathcal{P}_{\text{build}}$ consists of $2|S| + 5$ rules:

$$(+,\ \textsf{insert},\ \textsf{/tm[ph/build]},\ \textsf{state}) \quad (4.21)$$

$$(+,\ \textsf{insert},\ \textsf{/tm[ph/build]},\ \textsf{tape}) \quad (4.22)$$

$$(+,\ \textsf{insert},\ \textsf{/tm[ph/build]/state},\ q_0) \quad (4.23)$$

$$(+,\ \textsf{insert},\ \textsf{/tm[ph/build]/tape},\ \textsf{cell}) \quad (4.24)$$

$$(+,\ \textsf{insert},\ \textsf{/tm[ph/build]/tape/cell},\ \textsf{curr}) \quad (4.25)$$

$$(+,\ \textsf{insert},\ \textsf{/tm[ph/build]/tape//cell},\ \textsf{sym}) \quad (4.26)$$

$$(+,\ \textsf{insert},\ \textsf{/tm[ph/build]/tape/cell/sym},\ S_1) \quad (4.27)$$

For each i, $1 < i \leq |S|$:

$$(+, \text{insert}, /\text{tm[ph/build]/tape(/cell)}^{i-1}, \text{cell}) \tag{4.28}$$

$$(+, \text{insert}, /\text{tm[ph/build]/tape(/cell)}^{i-1}\text{[sym/}*\text{]/cell/sym}, S_i) \tag{4.29}$$

At the end of this phase, once the entire initial configuration tree is constructed, we permit transitioning to the run phase:

$$\mathcal{P}_{b \to r} = \Big(+, \text{rename}, /\text{tm[state/}q_0\text{][tape/(/cell)}^{|S|}\text{/sym/}S_{|S|}\text{]/ph/build}, \text{run}\Big) \tag{4.30}$$

Lemma 4.4.2 *Any sufficiently long sequence of operations permitted on the null tree produces the tree* conftree($\mathcal{C}_0^{M,S}$) *as an intermediate step.*

PROOF: By Lemma 4.4.1, a sufficiently long sequence of operations permitted on the null tree produces as an intermediate step a tree with a /tm/ph/build node, ph and tm ancestors, and no other nodes. On such a tree, only Rules 4.21 and 4.22 permit further operations.

Once a state node has been inserted by Rule 4.21, Rule 4.23 permits inserting a q_0 child; Rules 4.9 and 4.12 ensure that at most one state node and one child of that node are inserted.

After applying Rule 4.22 to insert a tape node, Rules 4.24–4.27 allow inserting a cell child; curr and sym grandchildren; and a S_1 child of the sym node. Once the cell node has been inserted, the instances of Rule 4.28 allow inserting $i-1$ cell descendants, each a child of the previous node. Rule 4.13 ensures that neither the tape node nor the cell nodes may contain multiple cell children; and Rule 4.14 prevents applying Rule 4.25 again.

For each of the cell nodes, Rules 4.26 and 4.29 allow inserting sym children, and grandchildren labelled with the appropriate symbol. Furthermore, the [sym/$*$] predicate in the latter rule ensures that no symbol is inserted until the previous symbol (and hence all preceding symbols) are inserted.

Finally, Rule 4.30 allows renaming the build node to run, but only after the q_0 node and last cell's symbol (and hence all $|S|$ symbols) have been inserted.

Since all other positive rules contain predicates that require that the ph node contain some child other than build, these are the only rules that permit operations before Rule 4.30 is applied. Along with the structural rules, this result ensures that any sequence of $3 + 2 + 1 + 4 + 3|S - 1| = 3|S| + 7$ operations results in a tree where Rule 4.30 is the only applicable positive rule that is not blocked by negative rules. Applying this rule results in precisely the tree conftree($\mathcal{C}_0^{M,S}$). Hence any

permitted sequence of length $3|S| + 8$ yields this tree, and any longer permitted sequence produces this tree as an intermediate result. □

The run phase

The run phase represents a Turing machine configuration. This phase can be entered from the build phase (Rule 4.30), or from the cleanup phase (Rule 4.46). The only operations permitted in this phase are to rename the run node to the next phase, one of the write phases (Rule schema 4.31); to insert a finished node if the configuration is a final one (Rule schema 4.32); and to delete most of the tree when the finished node is present (Rule 4.33), obtaining a fixed tree with two nodes (a tm root and a deleted child). Subpolicy $\mathcal{P}_{\text{run}}$ consists of $|\Gamma|(|Q| - |Q_F|) + |Q_F| + 1$ rules:

For each non-final state $q \in Q \setminus Q_F$ and each tape symbol γ:

$$(+,\ \mathsf{rename},\ \mathsf{/tm[state/}q\mathsf{][tape//cell[curr][sym/}\gamma\mathsf{]]/ph/run},\ \mathsf{write\text{-}}q\text{-}\gamma) \tag{4.31}$$

For each final state $q_f \in Q_F$:

$$(+,\ \mathsf{insert},\ \mathsf{/tm[ph/run][state/}q_f\mathsf{]},\ \mathsf{finished}) \tag{4.32}$$

One single rule:

$$(+,\ \mathsf{delete},\ \mathsf{/tm[finished]//}*) \tag{4.33}$$

Lemma 4.4.3 *If $\mathcal{C}$ is not a final configuration (that is, its state is not a member of Q_F), then exactly one operation is permitted on* conftree($\mathcal{C}$)*, resulting in a tree T_w, otherwise identical to* conftree($\mathcal{C}$)*, but with a* write-q-γ *node replacing the* run *node, where q is the state of $\mathcal{C}$ and γ is the symbol in the current tape cell of $\mathcal{C}$.*

PROOF: Other than Rules 4.18–4.20, which are blocked by structural rules, the only positive rules whose predicates are satisfied by a tree in the run phase without a finished node are instances of Rule schemata 4.32 and 4.31; only the latter is active for a non-final configuration. This rule transforms the tree into precisely T_w. □

If the simulation has reached a final configuration, a finished node is inserted, and the tree is pared down to just the tm and finished nodes.

Lemma 4.4.4 *If $\mathcal{C} = (q_f, t, p)$ is a final configuration (that is, $q_f \in Q_F$), there is a sequence of operations permitted on $T =$* conftree($\mathcal{C}$) *that yields the tree T_{halt} containing a root node labelled* tm*, a child node labelled* finished*, and no other nodes.*

PROOF: The tree T contains nodes with paths /tm/ph/run and /tm/state/q_f; hence Rule 4.32 is active and it is permitted to insert a finished node. Once this node has been added, Rule 4.33 becomes active, allowing any non-root node in the tree to be deleted. In particular, it is permitted to delete the ph, state, and tape nodes, leaving precisely T_{halt}. □

The write phases

For each transition, that is to say each pair $(q, \gamma) \in (Q \setminus Q_F) \times \Gamma$, the phase write-$q$-$\gamma$ allows changing the configuration tree's state and current tape symbol. It is followed by the move-q-γ and cleanup phases, where the position of the tape head is adjusted. Subpolicy $\mathcal{P}_{\text{write}}$ consists of $3|\Gamma|(|Q| - |Q_F)$ rules. For each transition:

$$\big(+,\ \text{rename},\ \text{/tm[ph/write-}q\text{-}\gamma\text{]/tape//cell[curr]/sym/}\gamma,\ \gamma'\big) \tag{4.34}$$

$$\big(+,\ \text{rename},\ \text{/tm[ph/write-}q\text{-}\gamma\text{]/state/}q,\ q'\big) \tag{4.35}$$

$$\big(+,\ \text{rename},\ \text{/tm[state/}q'\text{][tape//cell[curr]/sym/}\gamma'\text{]ph/write-}q\text{-}\gamma,\ \text{move-}q\text{-}\gamma\big) \tag{4.36}$$

Lemma 4.4.5 *Let $\mathcal{C}$ be a Turing machine configuration with state q and current symbol γ; let T_w be the tree resulting from Lemma 4.4.3; and let $(q', \gamma', D) = \delta(q, \gamma)$. Any permitted sequence of operations on T_w that results in a tree without a* write-q-γ *node produces as an intermediate step a tree T_m that differs from T_w in that: the child of the* ph *node is labelled* move-q-γ*; the child of the* state *node is labelled q'; and the child node of the* sym *sibling of the* curr *node is labelled γ'.*

PROOF: In this phase, only Rules 4.34–4.36 are active. Only the last of these rules permits renaming the write-q-γ node. This rule's predicates allow it to be used only if the state node has a child labelled q' and the current tape cell node ζ has a grandchild labelled γ'. These are precisely the state and tape value specified by $\delta(q, \gamma)$.

Rule 4.35 permits renaming the state node q to q'; and Rule 4.34 permits renaming the γ grandchild of ζ to γ'; since these are the only three active rules, no other operations are permitted in this phase. Hence, immediately after applying Rule 4.36, the tree is precisely the described T_m. □

REMARK: Unlike in the other phases, it is not necessarily the case that any sufficiently long sequence of operations leads to the next phase. If a cell is being re-written with the same symbol, or the transition does not change the Turing machine state, it

is possible to apply Rule 4.34 or 4.35 an arbitrary number of times.However, since in this case the rule in question does not alter the tree, the result does not differ from applying the rule only once.

This difficulty could be avoided by maintaining a second marked set of labels representing the tape symbols, using the marked version of q' in Rules 4.34 and 4.36, and renaming the node with the non-marked label in the cleanup phase. Because our proof is not adversely affected by the presence of such cycles, we do not complicate the policy in this way.

The move phases

When the move-q-γ phase is entered, the tree's state and current tape symbol have been updated. It remains to move the tape head, adding a new cell if necessary. This operation is accomplished in two phases: in the move phase, we mark the updated tape head position as "new" and remove the "current" mark. Then, in the cleanup phase, we change the "new" mark to "current". This separation of phases allows us to ensure that the tape head is moved exactly one step. First, subpolicy $\mathcal{P}_{\text{move}}$ consists of between $4|\Gamma|(|Q| - |Q_F|)$ and $7|\Gamma|(|Q| - |Q_F|)$ rules. For each transition $\delta(q, \gamma) = (q', \gamma', D)$:

$$(+,\ \mathsf{delete},\ \mathsf{/tm[ph/move\text{-}}q\text{-}\gamma\mathsf{][tape//new]/tape//curr}) \tag{4.37}$$

$$(+,\ \mathsf{rename},\ \mathsf{/tm[tape//cell/new]/ph/move\text{-}}q\text{-}\gamma,\ \mathsf{cleanup}) \tag{4.38}$$

$$(-,\ \mathsf{rename},\ \mathsf{/tm[tape//cell/curr]/ph/move\text{-}}q\text{-}\gamma,\ *) \tag{4.39}$$

If $D = L$:

$$(+,\ \mathsf{insert},\ \mathsf{/tm[ph/move\text{-}}q\text{-}\gamma\mathsf{]/tape//cell[cell/curr]},\ \mathsf{new}) \tag{4.40}$$

Otherwise, $D = R$:

$$(+,\ \mathsf{insert},\ \mathsf{/tm[ph/move\text{-}}q\text{-}\gamma\mathsf{]/tape//cell[curr]},\ \mathsf{cell}) \tag{4.41}$$

$$(+,\ \mathsf{insert},\ \mathsf{/tm[ph/move\text{-}}q\text{-}\gamma\mathsf{]/tape//cell},\ \mathsf{sym}) \tag{4.42}$$

$$(+,\ \mathsf{insert},\ \mathsf{/tm[ph/move\text{-}}q\text{-}\gamma\mathsf{]/tape//cell[curr]/cell/sym},\ b) \tag{4.43}$$

$$(+,\ \mathsf{insert},\ \mathsf{/tm[ph/move\text{-}}q\text{-}\gamma\mathsf{]/tape//cell[curr]/cell[sym/*]},\ \mathsf{new}) \tag{4.44}$$

Lemma 4.4.6 *Let $\mathcal{C} = (q, t, p)$ be a Turing machine configuration; let T_m be the tree resulting from Lemma 4.4.5; and let D be the directional component of $\delta(q, \gamma)$.*

If the $p = 1$ and $D = L$ (a ***hanging configuration****), no operations are permitted on T_m. Otherwise, let ζ be the* cell *node of T_m containing a* curr *child. Any sufficiently long sequence of permitted operations of T_m produces as an intermediate step the tree*

T_c that is otherwise identical to T_m, but without a curr *node, with the* move-q-γ *node replaced with* cleanup *node, and with changes depending on D and p:*

- *If $D = L$ and $p > 1$, then the parent of ζ in T_c contains a child labelled* new*.*
- *If $D = R$ and $p = |t|$, then ζ in T_c contains a new* cell *child, containing two children labelled* new *and* sym*, with the latter having a child node with the blank cell label b.*
- *Otherwise, $D = R$ and $p < |t|$; then the* cell *child of ζ in T_c contains a child labelled* new*.*

PROOF: The positive rules active in this phase are instances of Rules 4.37 and 4.38; if $D = L$, Rule 4.40; and if $D = R$, Rules 4.41–4.44. Furthermore, the negative Rules 4.13, 4.15, 4.16, 4.17, and 4.39 are relevant to the operation of this phase. Rules 4.37 and 4.38 are not active on T_m, because the tree contains no new node.

We consider separately the two directions in which the tape head may move. If $D = L$, only Rule 4.40 is active on T_m. This rule permits inserting a new node as a child of the parent cell node of ζ. If $p = 1$, the parent of ζ is the tape node; hence no operations are permitted. Otherwise, after inserting the new node, Rule 4.15 prohibits inserting another such node; and Rule 4.37 becomes active. Once this rule is used to delete the curr node, Rule 4.39 is no longer active, leaving Rule 4.38 as the only active rule; applying this rule renames the move-q-γ node to cleanup, yielding T_c. Hence, any permitted sequence of three operations yields T_c.

If $D = R$, there are two cases to consider. If $p < |t|$, only Rules 4.41 and 4.44 are active; and because ζ has a cell child, Rule 4.13 prohibits inserting a node with the former rule. Hence the only permitted operation is to insert a new child of the cell child of ζ. After this insertion, the argument from the $D = L$ case applies: the next operation must be to delete the curr node and deactivate Rule 4.39, then to rename the move-q-γ node. Hence any permitted sequence of three operations yields T_c.

If $p = |t|$, Rule 4.44 is not active, because ζ has no cell children. Thus the only permitted operation is to insert a new cell child of ζ by Rule 4.41; Rule 4.13 prohibits inserting more than one such node. After this node has been inserted, Rule 4.42 permits inserting a sym node (but only once because of Rule 4.16). Then Rule 4.43 allows inserting a blank symbol node b as a child of this node, but only once because of Rule 4.17. Once these three operations have been performed, Rule 4.44 is active and the other rules inactive; then we may proceed as in the $p < |t|$ case. Hence any

permitted sequence of six operations yields T_c. □

The cleanup phase

After the write and move phases, the tree has almost completely been updated to reflect the new Turing machine configuration. The only remaining steps are to replace the new marker node with a curr node, and to enter the run phase again, yielding the tree for the successor configuration.

$$\mathcal{P}_{\text{cleanup}} = (+,\ \mathsf{rename},\ \mathsf{/tm[ph/cleanup]//new},\ \mathsf{curr}) \quad (4.45)$$

$$(+,\ \mathsf{rename},\ \mathsf{/tm[tape//curr]/ph/cleanup},\ \mathsf{run}) \quad (4.46)$$

Lemma 4.4.7 *Let $\mathcal{C}$ be a Turing machine configuration and let T_c be the tree resulting from applying Lemmas 4.4.3, 4.4.5, and 4.4.6 to* conftree($\mathcal{C}$)*. Any sufficiently long sequence of operations permitted on T_c produces as an intermediate result the tree T', otherwise the same as T_c, but with a* curr *node in place of the* new *node, and a* run *node in place of the* cleanup *node.*

Proof: In the cleanup phase, positive Rules 4.45 and 4.46 are active; the latter is not active on T_c because it contains no curr node. Hence the only permitted operation is to rename the new node to curr by the former rule. Performing this operation deactivates Rule 4.45, because there is no longer a new node. Hence the second operation must be to rename the cleanup node to run by Rule 4.46, resulting in the desired tree T'. Thus any permitted sequence of two operations on T_c yields T'. □

4.4.4 Correctness of simulation

Lemma 4.4.8 *Let $\mathcal{C}$ be a non-final Turing machine configuration, and let Q be a non-empty sequence of operations permitted on $T =$* conftree($\mathcal{C}$) *such that Q(*conftree($\mathcal{C}$)*) contains a node labelled* /tm/ph/run*. Then Q produces* conftree($\Delta(\mathcal{C})$) *as an intermediate result, representing the successor configuration $\Delta(\mathcal{C})$. Furthermore, if $\mathcal{C}$ is not a hanging configuration, such a sequence Q exists.*

Proof: After applying Lemmas 4.4.3 through 4.4.7 to the tree $T =$ conftree($\mathcal{C}$), we have a tree T' that is otherwise identical to T, except that: its state is q'; the symbol of the previous tape cell ζ (the cell that was current in conftree($\mathcal{C}$)) is γ'; the

parent (if $D = L$) or child (if $D = R$) cell of ζ is marked as current; and if ζ had no cell children and $D = R$, a new child cell with symbol b has been inserted. These are precisely the changes necessary to transform conftree($\mathcal{C}$) into conftree($\Delta(\mathcal{C})$). If any of these changes had not been performed, the resulting tree could not have a /tm/ph/run node. □

Figure 4.7 demonstrates the sequence of phases for a typical transition involving a

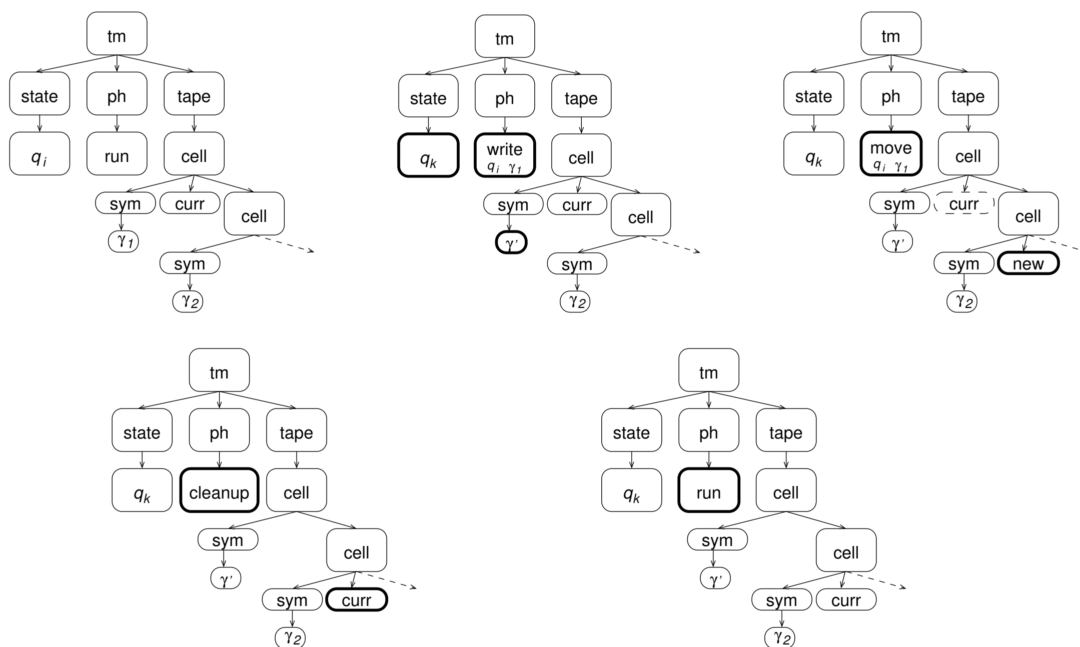

Figure 4.7: Simulating a Turing machine transition. Shown is the Turing machine transition $\delta(q_i, \gamma_1) = (q_k, \gamma', R)$. First row: a configuration tree in phase run, and the tree at the end of phases write, move. Second row: the tree at the end of phase cleanup, and the successor configuration tree.

rightwards move.

Theorem 4.4.1 *Let M be a Turing machine and $S \in \Sigma^*$ an initial tape of M; and let T_{halt} be the tree containing only a* tm *root and a* finished *child. Then $\mathcal{P}_{M,S}$ generates T_{halt} if and only if M halts on input S.*

Proof: By Lemmas 4.4.1 and 4.4.2, $\mathcal{P}_{M,S}$ generates the tree $T_1 = \text{conftree}(\mathcal{C}_0^{M,S})$, and any sufficiently long permitted sequence of operations produces this tree as an intermediate step. If M halts on input S after k steps, then k applications of Lemma 4.4.8 produces a configuration tree for a final configuration. From such a tree, by Lemma 4.4.4 there is a permitted sequence of operations that yields T_{halt}.

Now suppose M does not halt on input S. Then either M eventually enters a hanging configuration, or there is an infinite sequence of configurations $\langle \mathcal{C}_0, \mathcal{C}_1, \ldots \rangle$ such that each $\mathcal{C}_{i+1} = \Delta(\mathcal{C}_i)$. In the former case, by Lemma 4.4.6 the simulation does not advance beyond the hanging configuration, and thus does not produce T_{halt}. Consider then the second case, where there is an infinite sequence of successive configurations. Suppose $\mathcal{S}$ were a permitted sequence of operations resulting in T_{halt}. By Lemma 4.4.8, $\mathcal{S}$ produces as an intermediate result the tree conftree($\mathcal{C}_1$). Repeated applications of Lemma 4.4.8 demonstrates that $\mathcal{S}$ subsequently produces intermediate results conftree($\mathcal{C}_2$), conftree($\mathcal{C}_3$), and in general conftree($\mathcal{C}_i$) for each $i \in \mathbb{N}$, none of which permit inserting a finished node. This result means that $\mathcal{S}$ is an infinite sequence, contradicting the assumption that it is a permitted sequence. Hence no such sequence exists, and $\mathcal{P}_{M,S}$ does not generate T_{halt}. □

Corollary 4.4.1 PGEN *is undecidable.*

PROOF: Given a Turing machine M and input tape S, it is possible to construct the policy $\mathcal{P}_{M,S}$ algorithmically (in fact, in polynomial time). Combined with Theorem 4.4.1, this construction establishes a many-to-one reduction from the halting problem to PGEN. Since the halting problem is undecidable, so too is PGEN. □

4.4.5 Policies without rename

In fact, we do not need the full generality of PGEN for undecidability. It is possible to construct a modified version of $\mathcal{P}_{M,S}$ that excludes rename rules, yet still correctly simulates a Turing machine. We begin by noting that, in the trees in L $(\mathcal{P}_{M,S})$, no node that can be the target of a permitted rename operation can have children. Furthermore, in all cases but one (Rule 4.45), there exist structural rules that prohibit the renamed node from having a sibling. We take advantage of these facts to produce a policy that permits sequences of insert and delete operations (and only those sequences) that simulate the rename operations being eliminated. We do so by inserting additional **signal nodes**, indicating that a particular rename operation is being simulated, and prohibiting all operations on other parts of the tree while the signal node exists. Once the simulated rename is complete, we permit the signal node to be removed.

A significant portion of the rename-free policy is identical with $\mathcal{P}_{M,S}$. In particular, $\mathcal{P}_{\text{struct}}$, $\mathcal{P}_{\text{init}}$, and $\mathcal{P}_{\text{build}}$ are used unmodified. The subpolicies for the other simulation phases, however, contain rename rules that must be replaced.

We replace Rule 4.30 with the subpolicy $\mathcal{P}^{i,d}_{\text{b}\to\text{r}}$:

$$\Big(+,\ \mathsf{insert},\ \mathsf{/tm[state/}q_0\mathsf{][tape/(/cell)}^{|S|}\mathsf{/sym/}S_{|S|}\mathsf{][ph/build]},\ \mathsf{sig\text{-}build\text{-}run}\Big) \tag{4.47}$$

$$(+,\ \mathsf{delete},\ \mathsf{/tm[sig\text{-}build\text{-}run]/ph/build}) \tag{4.48}$$

$$(+,\ \mathsf{insert},\ \mathsf{/tm[sig\text{-}build\text{-}run]/ph},\ \mathsf{run}) \tag{4.49}$$

$$(+,\ \mathsf{delete},\ \mathsf{/tm[ph/run]/sig\text{-}build\text{-}run}) \tag{4.50}$$

$$(-,\ \mathsf{insert},\ \mathsf{/tm[sig\text{-}build\text{-}run]},\ *) \tag{4.51}$$

In subpolicy $\mathcal{P}^{i,d}_{\text{run}}$ we maintain Rules 4.32 and 4.33, while replacing Rule 4.31 with:

$$(+,\ \mathsf{insert},\ \mathsf{/tm[state/}q\mathsf{][tape//cell[curr][sym/}\gamma\mathsf{]][ph/run]},\ \mathsf{sig\text{-}write\text{-}}q\text{-}\gamma) \tag{4.52}$$

$$(+,\ \mathsf{delete},\ \mathsf{/tm[sig\text{-}write\text{-}}q\text{-}\gamma\mathsf{]/ph/run}) \tag{4.53}$$

$$(+,\ \mathsf{insert},\ \mathsf{/tm[sig\text{-}write\text{-}}q\text{-}\gamma\mathsf{]/ph},\ \mathsf{write\text{-}}q\text{-}\gamma) \tag{4.54}$$

$$\big(+,\ \mathsf{delete},\ \mathsf{/tm[ph/write\text{-}}q\text{-}\gamma\mathsf{]/sig\text{-}write\text{-}}q\text{-}q'\big) \tag{4.55}$$

$$(-,\ \mathsf{insert},\ \mathsf{/tm[sig\text{-}write\text{-}}q\text{-}\gamma\mathsf{]},\ *) \tag{4.56}$$

Subpolicy $\mathcal{P}^{i,d}_{\text{write}}$ differs the most from its counterpart in $\mathcal{P}_{M,S}$, as we we replace all three of its rules (4.34–4.36):

$$\big(+,\ \mathsf{insert},\ \mathsf{/tm[ph/write\text{-}}q\text{-}\gamma\mathsf{][tape//cell[curr]/sym/}\gamma\mathsf{]},\ \mathsf{sig\text{-}}\gamma\text{-}\gamma'\big) \tag{4.57}$$

$$\big(+,\ \mathsf{delete},\ \mathsf{/tm[sig\text{-}}\gamma\text{-}\gamma'\mathsf{]/tape//cell[curr]/sym/}\gamma\big) \tag{4.58}$$

$$\big(+,\ \mathsf{insert},\ \mathsf{/tm[sig\text{-}}\gamma\text{-}\gamma'\mathsf{]/tape//cell[curr]/sym},\ \gamma'\big) \tag{4.59}$$

$$\big(+,\ \mathsf{delete},\ \mathsf{/tm[tape//cell[curr]/sym/}\gamma'\mathsf{]/sig\text{-}}\gamma\text{-}\gamma'\big) \tag{4.60}$$

$$\big(-,\ \mathsf{insert},\ \mathsf{/tm[sig\text{-}}\gamma\text{-}\gamma'\mathsf{]},\ *\big) \tag{4.61}$$

$$\big(+,\ \mathsf{insert},\ \mathsf{/tm[ph/write\text{-}}q\text{-}\gamma\mathsf{][state/}q\mathsf{]},\ \mathsf{sig\text{-}}q\text{-}q'\big) \tag{4.62}$$

$$\big(+,\ \mathsf{delete},\ \mathsf{/tm[sig\text{-}}q\text{-}q'\mathsf{]/state/}q\big) \tag{4.63}$$

$$\big(+,\ \mathsf{insert},\ \mathsf{/tm[sig\text{-}}q\text{-}q'\mathsf{]/state},\ q'\big) \tag{4.64}$$

$$\big(+,\ \mathsf{delete},\ \mathsf{/tm[state/}q\mathsf{]/sig\text{-}}q\text{-}q'\big) \tag{4.65}$$

$$\big(-,\ \mathsf{insert},\ \mathsf{/tm[sig\text{-}}q\text{-}q'\mathsf{]},\ *\big) \tag{4.66}$$

$$(+,\ \textsf{insert},\ \textsf{/tm[state/}q'\textsf{][tape//cell[curr]/sym/}\gamma'\textsf{][ph/write-}q\textsf{-}\gamma\textsf{]},\ \textsf{sig-move-}q\textsf{-}\gamma) \tag{4.67}$$

$$(+,\ \textsf{delete},\ \textsf{/tm[sig-move-}q\textsf{-}\gamma\textsf{]/ph/write-}q\textsf{-}\gamma) \tag{4.68}$$

$$(+,\ \textsf{insert},\ \textsf{/tm[sig-move-}q\textsf{-}\gamma\textsf{]/ph},\ \textsf{move-}q\textsf{-}\gamma) \tag{4.69}$$

$$(+,\ \textsf{delete},\ \textsf{/tm[ph/move-}q\textsf{-}\gamma\textsf{]/sig-move-}q\textsf{-}\gamma) \tag{4.70}$$

$$(-,\ \textsf{insert},\ \textsf{/tm[sig-move-}q\textsf{-}\gamma\textsf{]},\ *) \tag{4.71}$$

$$(-,\ \textsf{insert},\ \textsf{/tm[sig-move-}q\textsf{-}\gamma\textsf{]/tape//}*,\ *) \tag{4.72}$$

$$(-,\ \textsf{delete},\ \textsf{/tm[sig-move-}q\textsf{-}\gamma\textsf{]/tape//}*) \tag{4.73}$$

In subpolicy $\mathcal{P}^{i,d}_{\text{move}}$ we preserve Rules 4.37 and 4.40–4.44, replacing the positive Rule 4.38 and the negative Rule 4.39 with:

$$(+,\ \textsf{insert},\ \textsf{/tm[tape//cell/new][ph/move-}q\textsf{-}\gamma\textsf{]},\ \textsf{sig-cleanup-}q\textsf{-}\gamma) \tag{4.74}$$

$$(+,\ \textsf{delete},\ \textsf{/tm[sig-cleanup-}q\textsf{-}\gamma\textsf{]/ph/move-}q\textsf{-}\gamma) \tag{4.75}$$

$$(+,\ \textsf{insert},\ \textsf{/tm[sig-cleanup-}q\textsf{-}\gamma\textsf{]/ph},\ \textsf{cleanup}) \tag{4.76}$$

$$(+,\ \textsf{delete},\ \textsf{/tm[ph/cleanup]/sig-cleanup-}q\textsf{-}\gamma) \tag{4.77}$$

$$(-,\ \textsf{insert},\ \textsf{/tm[sig-cleanup-}q\textsf{-}\gamma\textsf{]},\ *) \tag{4.78}$$

$$(-,\ \textsf{insert},\ \textsf{/tm[sig-cleanup-}q\textsf{-}\gamma\textsf{]/tape//}*,\ *) \tag{4.79}$$

$$(-,\ \textsf{delete},\ \textsf{/tm[sig-cleanup-}q\textsf{-}\gamma\textsf{]/tape//}*) \tag{4.80}$$

$$(-,\ \textsf{insert},\ \textsf{/tm[tape//cell/curr]},\ \textsf{sig-cleanup-}q\textsf{-}\gamma) \tag{4.81}$$

And finally, we replace both Rules 4.45 and 4.46, yielding subpolicy $\mathcal{P}^{i,d}_{\text{cleanup}}$:

$$(+,\ \textsf{insert},\ \textsf{/tm[ph/cleanup]//cell[new]},\ \textsf{curr}) \tag{4.82}$$

$$(+,\ \textsf{delete},\ \textsf{/tm[ph/cleanup]//cell[curr]/new}) \tag{4.83}$$

$$(-,\ \textsf{insert},\ \textsf{/tm[tape//cell[curr][new]]},\ *) \tag{4.84}$$

$$(+,\ \textsf{insert},\ \textsf{/tm[tape//curr][ph/cleanup]},\ \textsf{sig-cleanup-run}) \tag{4.85}$$

$$(+,\ \textsf{delete},\ \textsf{/tm[sig-cleanup-run]/ph/cleanup}) \tag{4.86}$$

$$(+,\ \textsf{insert},\ \textsf{/tm[sig-cleanup-run]/ph},\ \textsf{run}) \tag{4.87}$$

$$(+,\ \textsf{delete},\ \textsf{/tm[ph/run]/sig-cleanup-run}) \tag{4.88}$$

$$(-,\ \textsf{insert},\ \textsf{/tm[sig-cleanup-run]},\ *) \tag{4.89}$$

First we consider the more common case, exhibited by Rules 4.30, 4.31–4.36, 4.38, and 4.46. Each of these rules permits a leaf node of the tree to be renamed. In each

case, a structural rule (Rule 4.11, 4.12, or 4.17) prohibits inserting a sibling of the node in question. Furthermore, the parent of the renamed node (the target of the eventual insert) can be located even after the original node has been deleted. This combination of facts allows us to simulate rename with a deletion then insertion, and to construct a set of rules that permits this sequence of operations, and disallows all others until the sequence is complete. As a concrete example, consider Rule 4.34.

$$(+,\ \mathsf{rename},\ /\mathsf{tm}[\mathsf{ph}/\mathsf{write}\text{-}q\text{-}\gamma]/\mathsf{tape}//\mathsf{cell}[\mathsf{curr}]/\mathsf{sym}/\gamma,\ \gamma')$$

For each instance of this rule schema, we introduce a signal node sig-γ-γ' (Rule 4.57). When this signal node is present, we permit removing the target γ node (Rule 4.58) and inserting a node labelled γ' in its place (Rule 4.59). Rule 4.60 ensures that the new node is present when the signal node is deleted, and the structural Rule 4.17 ensures that the new node had no siblings. Finally, Rule 4.61 ensures that, until the sig-γ-γ' node is deleted, no other signal nodes can be inserted. Since only rename operations are permitted in a write phase, this prohibition on signal nodes is enough to ensure that no operations interrupt the simulated rename. In other cases, such as Rules 4.36 and 4.38, other insertions are allowed in the relevant phases, and we must prevent those as well. As a result, multiple negative rules (4.71–4.73 and 4.78–4.80, respectively) are required in the replacement of the rename rule.

Now we consider the exceptional case, Rule 4.45. This rule, active in the cleanup phase, permits the new marker to be replaced by a curr marker. Unlike the rules considered above, it is not sufficient here to first delete the new node then insert a curr node, because the presence of the new node indicates where curr may be placed. Instead, we take advantage of the fact that our structural rules do not prohibit a single node from having both curr and new children, and simulate the rename by inserting the curr node then removing the new node.

The only other positive rule active in the cleanup phase is Rule 4.46, which permits us to enter the run phase. This rule is a rename rule, and is thus itself simulated with a signal node (Rule 4.85). Therefore, to prevent other operations from being interspersed between the insert and delete, it suffices to prevent a signal node from being inserted while both a curr and node node are present. As a result we can in this case dispense with the signal node entirely (Rules 4.82–4.84).

Finally, with all positive rename rules accounted for, we turn our attention to the the negative Rule 4.39. This rule acts to prohibit Rule 4.38 (a rename rule) from firing before the curr node has been removed from the tree. We can gain the same effect in our rename-free policy by prohibiting insertion of the corresponding signal

node while curr exists; see Rule 4.81.

The resulting policy $\mathcal{P}_{M,S}^{i,d}$ generates all the trees generated by $\mathcal{P}_{M,S}$, as well as additional trees containing signal nodes. In particular, $\mathcal{P}_{M,S}^{i,d}$ generates T_{halt} if and only if $\mathcal{P}_{M,S}$ does. Therefore we have the following.

Theorem 4.4.2 *The problem* PGEN^{i,d}*, the subset of* PGEN *where the policy contains no* rename *rules, is undecidable.*

4.4.6 Policies without delete

It is also possible to produce a delete-free policy that simulates a Turing machine. However, the absence of delete operations means that the final tree is no smaller than the largest configuration tree produced by the simulation, which could be arbitrarily large. As a result, it is not possible to specify a target tree that would allow us to reduce the halting problem to PGEN^{i,r}. If we limit the size of the tape, the size of the largest configuration tree is bounded and it is possible to perform the reduction from the simpler halting problem for linear bounded automata (LBAs), a PSPACE-complete problem [36].

Suppose M is a linear bounded Turing machine and S an input tape. It is possible to obtain a linear reduction in the number of tape cells used by constructing a new machine with more tape symbols; and this transformation can be done in time polynomial in $|M|$ [36]. Hence we may without loss of generality assume that M's constant factor is 1; that is, it requires no tape cells beyond the input. Thus any reachable configuration contains exactly the same number of cells as the input tape.

As with our rename-free policy, we can use $\mathcal{P}_{\text{struct}}$ and $\mathcal{P}_{\text{init}}$ unaltered; we can also carry over $\mathcal{P}_{b\to r}$, $\mathcal{P}_{\text{write}}$, and $\mathcal{P}_{\text{cleanup}}$ without changes, as they consist entirely of rename rules.

In $\mathcal{P}_{\text{build}}^{i,r}$ we make two changes to ensure uniformity of our tree, requiring that every cell node have an child labelled curr, new, or inactive. The new Rule 4.90 allows us to insert an inactive child beneath each cell node we construct, other than the first (which has a curr child). Furthermore, we replace Rule schema 4.29 with Rule schema 4.91, which requires these inactive nodes to be present before we insert the tape symbol children of the same cells. As a result, Rule 4.30 allows us to leave the build phase only when each cell node has a curr or inactive child.

$$(+,\ \mathsf{insert},\ \mathsf{/tm[ph/build]/tape/cell//cell},\ \mathsf{inactive}) \tag{4.90}$$

$$\big(+,\ \mathsf{insert},\ \mathsf{/tm[ph/build]/tape(/cell)}^{i-1}\mathsf{[sym/}{*}\mathsf{]/cell[inactive]/sym},\ S_i\big) \tag{4.91}$$

In the subpolicy $\mathcal{P}_{\text{move}}$ we first note that, because M requires no cells beyond the input tape, we can eliminate Rules 4.41–4.43, which were responsible for extending the tape. Next, it is necessary to replace instances of Rule 4.37, which delete the curr node. Instead of deleting this node, the policy renames it to inactive (Rule schema 4.92). We also modify instances of Rules 4.40 and 4.44 so that we do not produce cells with both inactive and new children. Rather than inserting the new node, we rename the inactive node, resulting in Rule schemata 4.93 and 4.94:

$$(+,\ \mathsf{rename},\ \mathsf{/tm[ph/move\text{-}}q\text{-}\gamma\mathsf{][tape//new]/tape//curr},\ \mathsf{inactive}) \tag{4.92}$$

$$(+,\ \mathsf{rename},\ \mathsf{/tm[ph/move\text{-}}q\text{-}\gamma\mathsf{]/tape//cell[cell/curr]/inactive},\ \mathsf{new}) \tag{4.93}$$

$$(+,\ \mathsf{rename},\ \mathsf{/tm[ph/move\text{-}}q\text{-}\gamma\mathsf{]/tape//cell[curr]/cell[sym/}*\mathsf{]/inactive},\ \mathsf{new}) \tag{4.94}$$

Our resulting policy, then, cannot add a new curr, new, or inactive node after the build phase, but can only rename them. Hence each cell has exactly one such child at any point after the build phase. In fact, after these modifications, the only node that can be inserted outside of the build phase is the finished node (Rule 4.32).

Finally, we replace the sole remaining delete Rule 4.33 in subpolicy $\mathcal{P}_{\text{run}}$, resulting in the new Rule 4.95:

$$(+,\ \mathsf{rename},\ \mathsf{/tm[finished]//}*,\ \mathsf{finished}) \tag{4.95}$$

Instead of deleting every node other than the finished node, this replacement rule renames them to finished. Therefore, if M terminates on input S, it is possible to produce a tree with root tm and with every other node labelled finished. Since only the finished node can be inserted after the build phase, we know the shape of the final tree: It is the tree $T_{\mathsf{fin}}^{|S|}$ obtained by relabelling each non-root node in $\text{conftree}(\mathcal{C}_0^{M,S})$ to finished, and adding one additional finished child of the root. This tree has size linear in $|S|$ and in fact only depends on $|S|$, not any other feature of the input tape, the Turing machine, or the accepting computation. Furthermore, by a simple variant of Theorem 4.4.1, it is only possible to generate this tree if M halts on input S. We therefore have the result:

Theorem 4.4.3 PGEN^{i,r} *is* PSPACE-*complete.*

PROOF: We have shown that, from an instance (M, S) of the halting problem for LBAs (HPLBA), we can in polynomial time create an instance $\left(\mathcal{P}_{M,S}^{i,r}, T_{\mathsf{fin}}^{|S|}\right)$ of PGEN^{i,r}; and that this instance is in PGEN^{i,r} if and only if M halts on input S. This construction establishes a reduction from HPLBA to PGEN^{i,r}. Since the former problem is PSPACE-complete [36], the latter is PSPACE-hard. Furthermore, we

showed in Theorem 4.3.2 that this problem is in PSpace. Therefore, $\text{PGen}^{i,r}$ is PSpace-complete. □

4.5 Fine-grained access control for multihierarchical markup

As discussed in Chapter 3, hierarchical structures are not always enough to represent complex text documents, particularly those with multiple layers of analysis. We therefore proposed in that chapter a model of multihierarchical markup. Since multihierarchical documents are often edited the product of collaborative editing efforts [34, 45, 46], it is natural to ask whether fine-grained access control can be applied to such documents.

We consider the three update operations from in Section 3.3.3: splice-in, splice-out, and rename. These operations correspond to the insert, delete, and rename operations on hierarchical documents; however, they have properties that suggest some changes to the forms of access control rules and the definition of matching between rules and operations. Significantly, the splice-in operation targets not a single parent node, but rather two nodes that may or may not be children of the target. Furthermore, the splice-in operation can add nodes that become ancestors of existing nodes; in order to maintain invariants on the parent-child relationships of elements, it is therefore necessary to constrain the operation based on the descendants, and not just the ancestors, of the new node. This constraint is already possible with delete and rename rules, as it is possible to check the children of the existing target nodes with a predicate on the path expression's final location step. However, since a node being spliced in is not yet part of the document, an XPath expression on the document cannot easily refer to the nodes that will become its children (or parents). To solve this problem, we add to the insert rules an additional list of predicates that should be satisfied on the new node in the updated document.

Definition 4.5.1 *An **multihierarchical access control rule** over the set L of labels is a tuple with the form $(s, \textsf{splice-in}, P, \tau, Q)$, $(s, \textsf{splice-out}, P)$, or $(s, \textsf{rename}, P, \tau)$, where s is either + or −; P is a multihierarchical path expression over L, possibly the empty path expression ε; L is a node test (Definition 2.2.2); and Q is a predicate-list (from Grammar 2.2.2).*

*A rule is **positive** if s is +; otherwise it is **negative**. A rule is **simple** if its path expression contains no predicates and, if it is a splice-in operation, Q is empty.*

Rules with splice-out and rename operations correspond respectively to delete and rename rules in hierarchical documents, and have similar semantics.

Definition 4.5.2 *A splice-out rule* $X = (s, \textsf{splice-out}, P)$ ***matches*** *the operation* O *on document* G*, written* $X \sim_G O$*, if* $O = (\textsf{splice-out}, \nu)$ *and* $\nu \in \mathcal{S}_{MH}[[P]](G)$

Definition 4.5.3 *A rename rule* $R = (s, \textsf{rename}, P, \tau)$ ***matches*** *the operation* O *on document* G*, written* $R \sim_G O$*, if* $O = (\textsf{rename}, \nu, \ell)$*,* $\nu \in \mathcal{S}_{MH}[[P]](G)$*, and* τ *matches* ℓ*.*

However, splice-in rules are more difficult to specify. As with insert rules on hierarchical documents, a rule's path expression is checked against the parents of the new node. However, since the splice-in operation is not specified in terms of the node's parents, and since splice-in rules contain an additional list of predicates on the newly-inserted node, the matching criterion is somewhat more complex. We begin by identifying the existing nodes that will be ancestors and parents of the new node.

Definition 4.5.4 *The potential ancestor set of a range* $(\alpha, \beta) \in T \cup E \cup \{\infty\})$*, written* $\mathrm{pa}(\alpha, \beta)$*, is the set of nodes* $\nu \in T \cup E$ *such that*

- α *is* ∞ *or* $\nu <^o \alpha$*; and*
- β *is* ∞ *or* $\nu >^c \beta$*.*

The potential parent set of (α, β)*, written* $\mathrm{pp}(\alpha, \beta)$*, is the set of nodes* $\nu \in \mathrm{pa}(\alpha, \beta)$ *such that there is no node* $\chi \in \mathrm{pa}(\alpha, \beta)$ *with* $\nu \to^+ \chi$*.*

The potential ancestor and potential parent sets of a splice-in operation $O = (\textsf{splice-in}, \alpha, \beta, \ell)$ *are the potential ancestor and parent sets of* (α, β)*.*

The potential ancestors of a range are the nodes that open before the beginning of the range (treating ∞ as following the last-opening node) and close before the end of the range (treating ∞ as preceding the first-closing node). These nodes will be ancestors of the new node, which will immediately precede α in $<^o$ and immediately follow β in $>^c$. The potential parents are the potential ancestors that do not contain other potential ancestors; these nodes would hence directly contain the new node. Unfortunately, although it is possible to define the potential children of the new node in a similar manner, this definition does not allow us to easily apply the predicate list Q; instead, we apply relative expressions in those predicates in the updated tree.

Definition 4.5.5 *A splice-in rule* $S = (s, \textsf{splice-in}, P, \tau, Q)$ ***matches*** *the operation* O *on document* G*, written* $S \sim_G O$*, if:*

- $O = (\textsf{splice-in}, \alpha, \beta, \ell)$*;*
- *either* $P = \varepsilon$ *and* $\mathrm{pp}(\alpha, \beta) = \emptyset$*, or* $P \neq \varepsilon$ *and there exists some node* $\pi \in \mathrm{pp}(\alpha, \beta)$ *such that* $\pi \in \mathcal{S}_{MH}[[P]]\,(G)$*;*
- τ *matches* ℓ*; and*
- *for each predicate* $q \in Q$ *consisting of a relative path expression, the new node* $\xi \in \mathcal{S}^{\mathrm{Q}}_{MH}[[q]]_{O_G}$*; and for each predicate* $q' \in Q$ *consisting of an absolute path expression,* $\mathcal{S}^{\mathrm{Q}}_{MH}[[q']]_G$ *is not empty.*

The ***matched parents*** *of operation* O *with respect to rule* S *are the potential parents that are selected by the rule's path expression.*

$$\mathrm{mp}(O, S) = \mathrm{pp}(\alpha, \beta) \cap \mathcal{S}_{MH}[[P]]\,(G)$$

Note that we treat relative and absolute path expressions in the predicate list Q differently. Relative path expressions are evaluated in the updated document, since they refer to children and descendants of the new node. Absolute path expressions, on the other hand, are evaluated in the original document, so that they may assert the presence of an existing node without being fooled by the new node. In particular, this observation allows a rule ($-$, splice-in, ε, $*$, $[/*]$) that prohibits inserting a root node into a document that already contains one.

At the cost of some complexity in defining different semantics for the predicate list Q and its components, we could eliminate the references to $O(G)$ in this definition. The bottom-up semantics of predicates by Gottlob *et al.*[31] would assist in these definitions.

Before moving from individual rules to complete policies, we note one additional complexity. Unlike the hierarchical case, where nodes have at most one parent and one path from the root, a multihierarchical node may have many parents. It may be overly permissive to allow the operation whenever an operation is matched through just one of many parents, because this ability would allow adding children to nodes that are not referenced by any positive rule, a violation of the principle of least surprise. Instead we require that every potential parent of the operation be matched by a positive rule (and none by a negative rule). Doing so allows us to add splice-in rules for particular types of parent nodes without being concerned that these rules allow manipulation of other nodes that happen to overlap the desired ones.

Definition 4.5.6 *A multihierarchical access control policy is a finite set of multihierarchical access control rules over some set L of labels.*

The policy $\mathcal{P}$ ***permits*** *the operation O on globally ordered GODDAG G, written $\mathcal{P} \vdash_G O$, if there is no negative rule $R_- \in \mathcal{P}$ such that $R_- \sim_G O$, and either:*

- *O is a* rename *or* splice-out *operation and there exists some positive rule $R_+ \in \mathcal{P}$ such that $R_+ \sim_G O$; or*
- *O is a* splice-in *operation, $\mathcal{R}$ is the set of positive rules matching O in G, $\mathcal{R}$ is non-empty, and $\mathrm{pp}(O) = \bigcup_{r \in \mathcal{R}} \mathrm{mp}(O, r)$.*

We may ask about the extension PGEN_{MH} of the problem PGEN to multihierarchical documents.

Definition 4.5.7 *The language $\mathrm{L}_{\mathcal{G}}(\mathcal{P})$ generated by a multihierarchical access control policy $\mathcal{P}$ from the set $\mathcal{G}$ of globally ordered GODDAGs is the set of globally ordered GODDAGs G such that there exists a sequence of operations $\mathcal{S}$ and a GODDAG $G_0 \in \mathcal{G}$ such that $\mathcal{P} \vdash_{G_0} \mathcal{S}$ and $\mathcal{S}(G_0) = G$. We write $\mathrm{L}(\mathcal{P})$ for $\mathrm{L}_{\{\mathrm{G}_\emptyset\}}(\mathcal{P})$, the language of globally ordered GODDAGs generated by $\mathcal{P}$ from the empty tree.*

Definition 4.5.8 *The problem PGEN_{MH} is: given a pair $(\mathcal{P}, G)$, where $\mathcal{P}$ is a multihierarchical access control policy and G is a globally ordered GODDAG, is $T \in \mathrm{L}(\mathcal{P})$?*

As might be expected, this version of the problem is also undecidable. We begin by constructing a multihierarchical version of the policy $\mathcal{P}_{M,S}$ from Section 4.4.

Definition 4.5.9 *Let $M = (Q, q_0, Q_F, \Gamma, \Sigma, b, \delta)$ be a Turing machine and $S \in \Sigma^*$ an initial tape of M. The multihierarchical access control policy $\mathcal{P}^{\mathrm{MH}}_{M,S}$ is given by*

$$\mathcal{P}^{\mathrm{MH}}_{M,S} = \mathcal{P}_{\mathrm{hier}} \cup \bigcup_{R \in \mathcal{P}_{M,S}} \mathrm{MH}(R),$$

where $\mathcal{P}_{\mathrm{hier}}$ contains the rules:

$$(-,\ \textit{splice-out},\ //*[*]) \tag{4.96}$$

$$(-,\ \textit{splice-in},\ //*,\ *,\ [*]) \tag{4.97}$$

$$(-,\ \textit{splice-in},\ \varepsilon,\ *,\ [/*]), \tag{4.98}$$

and where MH*, transforming hierarchical rules into multihierarchical rules, is defined as*

$$\mathrm{MH}\,(s, \textsf{\textit{insert}}, P, \ell) = (s, \textsf{\textit{splice-in}}, P, \ell, \varepsilon)$$
$$\mathrm{MH}\,(s, \textsf{\textit{delete}}, P) = (s, \textsf{\textit{splice-out}}, P)$$
$$\mathrm{MH}\,(s, \textsf{\textit{rename}}, P, \ell) = (s, \textsf{\textit{rename}}, P, \ell).$$

Rename rules have essentially the same semantics in hierarchical and multihierarchical documents, and hence remain unchanged. Rules for delete and splice-out rules have different semantics, as the former remove entire subtrees while the latter remove only a single node; however, if the target of a splice-out rule is a leaf node, the semantics are similar. In $\mathcal{P}_{M,S}$ there are two delete rules, Rule 4.37 and Rule 4.33. The former targets nodes labelled curr, which can only occur as leaves in documents generated by $\mathcal{P}_{M,S}$. The latter rule becomes active only after a final configuration is reached; at this point in the simulation, it is important only that the nodes of the document other than the root and finished can be deleted, not the order in which they are deleted; in particular, the nodes can be deleted in a bottom-up order, so that delete operations always target leaf nodes. Rule 4.96 ensures that, in fact, only leaf nodes may be deleted, as every non-leaf node is selected by $//*[*]$.

The additional Rules 4.97 and 4.98 control splice-in operations, ensuring that no node can be added as a parent of an existing node or as a new root of a non-empty document. As a result, any non-empty document constructed under $\mathcal{P}^{\mathrm{MH}}_{M,S}$ is rooted, and must be constructed in a top-down fashion, with each node spliced into the tree as a leaf. The insert rules themselves are translated into splice-in rules that permit adding rules in the same contexts; in particular, since $\mathcal{P}_{\mathrm{hier}}$ prohibits splicing in nodes as parents of existing nodes, these translated rules may have empty predicate lists. Furthermore, permitted sequences of operations generate only trees

Lemma 4.5.1 *Every document generated by $\mathcal{P}^{\mathrm{MH}}_{M,S}$ is a tree, possibly the empty tree, with identical structure to a tree generated by $\mathcal{P}_{M,S}$.*

Proof: Let $\mathcal{S}$ be a sequence of operations permitted by $\mathcal{P}^{\mathrm{MH}}_{M,S}$. We show by induction that $D = \mathcal{S}(\mathrm{T}_\emptyset)$ is a tree. As our base case, we note that, if $\mathcal{S}$ is empty, D is the empty tree.

Assume now that every document produced by a sequence of n operations permitted by $\mathcal{P}^{\mathrm{MH}}_{M,S}$ is a tree generated by $\mathcal{P}_{M,S}$. Consider a sequence of operations $\mathcal{S} = \langle o_1, \ldots, o_n, o_{n+1}\rangle$ permitted by $\mathcal{P}^{\mathrm{MH}}_{M,S}$. By the inductive hypothesis, $D_n = o_n(\cdots o_1(\mathrm{T}_\emptyset))$ is a tree.

If o_{n+1} is a rename operation, then $D = o_{n+1}(D_n)$ has the same graph structure as D_n, but with different labels, and is hence a tree itself; furthermore, $\mathcal{P}_{M,S}$ permitted the identical operation on D_n, so D is generated by $\mathcal{P}_{M,S}$. If o_{n+1} is a splice-out operation, then by Rule 4.96 it targets a leaf node. Splicing out a leaf node from a tree results in a tree, so $D = o_{n+1}(D_n)$ is a tree; and the same node may be deleted from D_n under $\mathcal{P}_{M,S}$, so D is generated by $\mathcal{P}_{M,S}$.

Now consider the case where o_{n+1} is a splice-in operation. As demonstrated by Lemmas 4.4.1–4.4.7, at most one positive insert rule is active in D_n, with at most a single permissible target. As a result, only the corresponding rule of $\mathcal{P}^{\mathrm{MH}}_{M,S}$ is active, and it likewise matches at most one target node. Since a splice-in operation is permitted only if every parent of the new node is selected by some positive splice-in rule, the operation o_{n+1} splices in a node with at most one parent. Since Rules 4.97 and 4.98 furthermore prohibit splicing in a node except as a leaf node, the document $D = o_{n+1}(D_n)$ consists of a tree with a new leaf node added as a child of a single node; hence it is the tree resulting from the corresponding insert operation under $\mathcal{P}_{M,S}$, concluding the induction. □

Finally, we may use the correspondence between documents generated by $\mathcal{P}^{\mathrm{MH}}_{M,S}$ and those generated by $\mathcal{P}_{M,S}$ to extend Theorem 4.4.1 to multihierarchical policies.

Theorem 4.5.1 *There exists a reduction from the halting problem for Turing machines to* $\textsc{PGen}_{MH}$.

PROOF: By Lemma 4.5.1, the documents generated by $\mathcal{P}^{\mathrm{MH}}_{M,S}$ are those trees generated by $\mathcal{P}_{M,S}$. In particular, T_{halt} is generated by $\mathcal{P}^{\mathrm{MH}}_{M,S}$ if and only if it is generated by $\mathcal{P}_{M,S}$ (so, by Theorem 4.4.1, if and only if M halts on input S). Furthermore, $\mathcal{P}^{\mathrm{MH}}_{M,S}$ can be constructed by a straightforward linear-time transformation of $\mathcal{P}_{M,S}$. Therefore, the function mapping each (M, S) to $\mathcal{P}^{\mathrm{MH}}_{M,S}$ is a reduction from the halting problem to $\textsc{PGen}_{\mathrm{MH}}$. □

We can therefore extend our main complexity result for $\textsc{PGen}$ to multihierarchical policies.

Corollary 4.5.1 $\textsc{PGen}_{MH}$ *is undecidable.*

Copyright © Neil Moore, 2012. Portions of this chapter (Sections 4.1–4.4) are reprinted from *Journal of Information and Computation* **209**:3, Moore, N., "Computational complexity of the problem of tree generation under fine-grained access control policies", pp. 548–567, Copyright © 2011, with permission from Elsevier.

Chapter 5 Conclusions and future research directions

5.1 Thesis contributions

We have made several contributions dealing with problems of text encoding and collaborative editing for complex documents. Our results have focused on the topics of modelling multihierarchical markup and fine-grained access control, as well as issues that arise in the intersection of these topics.

We have shown that augmenting the GODDAG [61] structure with an ordering of nodes meeting certain requirements results in a model of markup, the globally ordered GODDAG, that reflects precisely those documents that can be serialized in text-and-tags form without spurious overlap. This structure provides an alternative to the approach of Marcoux [52], while also permitting us to simply define the semantics of update operations on such documents. We furthermore introduced a variant of core XPath [12, 31] that supports the globally ordered GODDAG, allowing queries based on the expanded range of relationships found in a multihierarchical document.

We also investigated fine-grained access control for updates of XML documents, in particular access control policies that use XPath expressions to select the nodes that may or may not be affected by different kinds of updates. We showed that, even if such expressions are restricted to limited subset $\mathrm{XP}^{\{[],*,//\}}$ of XPath, the problem PGen of determining whether a document could be constructed under the access control policy is undecidable. We additionally showed that even restricted variants of this problem, limiting the permitted of operations and the subset of XPath even further, have often high lower bounds on their complexity, although a polynomial time algorithm exists for at least one simple variant.

Finally, we have introduced a model for fine-grained access control of updates to multihierarchical documents in our globally ordered GODDAG. Such a model can allow for tighter management of collaborative editing scenarios, with administrators defining the types of markup and portions of the document that may be edited by particular users.

5.2 Future work

Although we have contributed several results, our work also raises new questions related to both multihierarchical markup and fine-grained access control.

We have shown that globally ordered GODDAGs and range GODDAGs are equivalent, and that updates to such a document may be specified in terms of the range GODDAG orders. However, it is likely to be prohibitively expensive to convert between the two forms of markup every time the document is updated. Future research should address the problem of efficiently maintaining the two representations side-by-side. Dynamic transitive closure and transitive reduction algorithms [59] may help to maintain both the arcs of the globally ordered GODDAG and the order relations of the range GODDAG.

Because the GOG faithfully represents overlapping markup with a text-and-tags serialization, it shares the limitations of such serializations. Further work is required to extend the GOG to represent discontinuous features [62] such as damaged regions of a manuscript [43]. Furthermore, some documents, such as palimpsests or bidirectional texts, are best represented with several order relationships rather than a single global order. Effectively representing such documents requires us to extend, or possibly even abandon, the text-and-tags serialization.

The results on the complexity of PGen, and in particular the undecidability of the full problem, suggests the need for models of fine-grained access control for XML updates that are less expressive, and thus better suited to analysis, while remaining practical and flexible. Two possibilities we have considered are positive insert-only and simple rename-free policies. However, although these policy languages do make PGen tractable, their expressiveness is greatly lacking: The former cannot account for documents that change in any way other than the accumulation of new text and markup, and the latter is unable to express constraints on sibling nodes such as "A section may be assigned a copyright only if it has an author". It may be necessary, then, to abandon the use of XPath altogether in fine-grained access control for XML updates. Two proposed models appear promising and warrant further investigation: schema-based access control [7, 8] and multi-level security of trees [11].

Adapting schema-based access control to multihierarchical markup will present its own set of challenges. Although there has been some research into schemata for multihierarchical markup [25, 38], such schemata are typically associated with the concurrent hierarchies model (Section 3.2.1), where the document's markup is partitioned into a number of separate hierarchies. Some work has considered the interaction of nodes in these separate hierarchies [20, 60], but further work is required to specify schemata for documents with self-overlap and other situations that prevent modelling documents as overlapping trees.

Copyright © Neil Moore, 2012.

Bibliography

[1] Early Manuscripts at Oxford University. http://image.ox.ac.uk/, 2000.

[2] Wikipedia: The free encyclopedia. http://www.wikipedia.org/, Accessed 8 February 2012.

[3] Wikipedia: Why create an account? http://en.wikipedia.org/?title=Wikipedia:Why_create_an_account%3F, Accessed 8 February 2012.

[4] Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Arnaud Le Hors, Gavin Nicol, Jonathan Robie, Peter Sharpe, Bill Smith, Jared Sorensen, Robert Sutor, Ray Whitmer, and Chris Wilson. Document Object Model (DOM) level 1 specification, version 1.0. http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/, 1998. World Wide Web Consortium Recommendation.

[5] Elisa Bertino, M. Braun, Silvana Castano, Elena Ferrari, and Marco Mesiti. Author-X: A Java-based system for XML data protection. In *Proceedings IFIP TC11/WG11.3 Fourteenth Annual Working Conference on Database Security: Data and Application Security, Development and Directions*, pages 15–26, 2000.

[6] Elisa Bertino, Barbara Catania, Elene Ferrari, and Paolo Perlasca. A logical framework for reasoning about access control models. *ACM Trans. Inf. Syst. Secur.*, 6(1):71–127, 2003.

[7] Loreto Bravo, James Cheney, and Irini Fundulaki. Repairing inconsistent XML write-access control policies. In Marcelo Arenas and Michael I. Schwartzbach, editors, *DBPL*, volume 4797 of *LNCS*, pages 97–111. Springer, 2007.

[8] Loreto Bravo, James Cheney, and Irini Fundulaki. ACCOn: checking consistency of XML write-access control policies. In Alfons Kemper, Patrick Valduriez, Noureddine Mouaddib, Jens Teubner, Mokrane Bouzeghoub, Volker Markl, Laurent Amsaleg, and Ioana Manolescu, editors, *EDBT*, volume 261 of *ACM International Conference Proceeding Series*, pages 715–719. ACM, 2008.

[9] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Extensible markup language (XML) 1.1. http://www.w3.

org/TR/2004/REC-xml11-20040204/, Feb 2004. World Wide Web Consortium Recommendation.

[10] Bogdan Cautis, Serge Abiteboul, and Tova Milo. Reasoning about XML update constraints. In *PODS '07: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 195–204, New York, NY, USA, 2007. ACM.

[11] SungRan Cho, Sihem Amer-Yahia, Laks V.S. Lakshmanan, and Divesh Srivastava. Optimizing the secure evaluation of twig queries. In *Proceedings 28th VLDB Conference*, 2002.

[12] James Clark and Steve DeRose. XML path language (XPath), version 1.0. http://www.w3.org/TR/1999/REC-xpath-19991116, 1999. World Wide Web Consortium Recommendation.

[13] Unicode Consortium. *The Unicode Standard*. The Unicode Consortium, Mountain View, California, 2012.

[14] Ernesto Damiani, Sabrina de Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Securing XML documents. In *Proceedings 2000 International Conference on Extending Database Technology (EDBT2000)*, 2000.

[15] Ernesto Damiani, Majirus Fansi, Alban Gabillon, and Stefania Marrara. Securely updating XML. In Bruno Apolloni, Robert J. Howlett, and Lakhmi C. Jain, editors, *KES (3)*, volume 4694 of *LNCS*, pages 1098–1106. Springer, 2007.

[16] A. Dekhtyar and I. E. Iacob. A framework for management of concurrent XML markup. *Data Knowl. Eng.*, 52(2):185–208, 2005.

[17] A. Dekhtyar and I. E. Iacob. xTagger: a new approach to authoring document-centric XML. In *Proceedings of the 5th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL '05)*, pages 44–45, June 2005.

[18] A. Dekhtyar, I. E. Iacob, J. W. Jaromczyk, K. Kiernan, N. Moore, and D. C. Porter. Database support for image-based electronic editions. In *Proceedings of the 10th International Workshop on Multimedia Information Systems (MIS 2004)*, Aug 2004.

[19] A. Dekhtyar, I. E. Iacob, J. W. Jaromczyk, K. Kiernan, N. Moore, and D. C. Porter. Support for XML markup of image-based electronic editions. *International Journal on Digital Libraries*, 6(1):55–69, 2006.

[20] A. Dekhtyar, I. E. Iacob, and W. Zhao. XPath extension for querying concurrent XML markup. In *Proceedings 30th VLDB Conference*, 2004.

[21] S. DeRose. Markup overlap: a review and a horse. In *Proceedings of Extreme Markup Languages*, 2004.

[22] S. J. DeRose, D. Durand, E. Mylonas, and A. Renear. What is Text, Really? *Journal of Computing in Higher Education*, 1(2):3–26, 1990.

[23] Alin Deutsch and Val Tannen. Containment of regular path expressions under integrity constraints. In *Knowledge Representation Meets Databases*, 2001.

[24] Denise Draper, Peter Fankhauser, Mary Fernández, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 formal semantics. `http://www.w3.org/TR/2007/REC-xquery-semantics-20070123/`, 2007. World Wide Web Consortium Recommendation.

[25] P. Durusau and M.B. O'Donnell. Coming down from the trees: Next step inthe evolution of markup? In *Proceedings of Extreme Markup Languages*, 2002.

[26] Ben Dushnik and E. W. Miller. Partially ordered sets. *American Journal of Mathematics*, 63(3):600–610, 1941.

[27] Irini Fundulaki and Sebastian Maneth. Formalizing XML access control for update operations. In Volkmar Lotz and Bhavani M. Thuraisingham, editors, *SACMAT*, pages 169–174. ACM, 2007.

[28] Irini Fundulaki and Maarten Marx. Specifying access control policies for XML documents with XPath. In *Proceedings 9th ACM Symposium on Access Control Models and Technologies*, pages 61–69, 2004.

[29] Mohammed GhasemZadeh, Volker Klotz, and Christoph Meinel. Representation and evaluation of QBFs in prenex-NNF. In *Proceedings of AI-METH2004: Recent Developments in Artificial Intelligence Methods*, pages 115–120, November 2004.

[30] C. F. Goldfarb. *The SGML Handbook*. Clarendon Press, Oxford, 1990.

[31] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.

[32] Paul Grosso, Eve Maler, Jonathan Marsh, and Normal Walsh. XPointer framework. `http://www.w3.org/TR/2003/REC-xptr-framework-20030325/`, 2003. World Wide Web Consortium Recommendation.

[33] Satoshi Hada and Michiharu Kudo. XML Access Control Language: Provisional authorization for XML documents. `http://www.trl.ibm.com/projects/xml/xacl/xacl-spec.html`, 2000. Technical Report, Tokyo Research Laboratory, IBM Research.

[34] K. C. Hawley and K. Kiernan. An Image-Based Electronic Edition of Alfred the Great's Old English Version of Boethius's Consolation of Philosophy. In *Proceedings of the Joint International ALLC-ACH Conference*, 2003.

[35] M. Hilbert, O. Schonefeld, and A. Witt. Making CONCUR work. In *Proceedings of Extreme Markup Languages*, 2005.

[36] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[37] C. Huitfeldt and C. M. Sperberg-McQueen. TexMECS: An experimental markup meta-language for complex documents. `http://mlcd.blackmesatech.com/mlcd/Publications.html`.

[38] I. E. Iacob, A. Dekhtyar, and K. Kaneko. Parsing concurrent xml. In *Proceedings of the 6th ACM International Workshop on Web Information and Data Management (WIDM)*, pages 23–30, November 2004.

[39] ISO 8879. *Information Processing—Text and Office Systems–Standard Generalized Markup Language (SGML).* International Organization for Standardization, Geneva, Switzerland, 1986.

[40] Florent Jacquemard and Michael Rusinowitch. Rewrite-based verification of XML updates. In *PPDP '10: Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, pages 119–130, New York, NY, USA, 2010. ACM.

[41] J. W. Jaromczyk, M. Kowaluk, and N. Moore. On visualization of complex markup. In *Proceedings of the International Conference on Computer Vision and Geometry*, Sep 2004.

[42] J. W. Jaromczyk, M. Kowaluk, and N. Moore. A web interface to image-based concurrent markup using image maps. In *Proceedings of the 6th ACM International Workshop on Web Information and Data Management (WIDM)*, Nov 2004.

[43] J. W. Jaromczyk and N. Moore. Geometric data structures for multihierarchical XML tagging of manuscripts. In *Proceedings of the 20th European Workshop on Computational Geometry*, Mar 2004.

[44] R. Karp. Reducibility among combinatorial problems. In R. Miller and J. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[45] K. Kiernan, J. Jaromczyk, A. Dekhtyar, D. Porter, K. Hawley, S. Bodapati, and I. E. Iacob. The ARCHway project: Architecture for research in computing for humanities through research, teaching, and learning. *Literary and Linguistic Computing*, 2004.

[46] K. Kiernan, D. C. Porter, A. Dekhtyar, I. E. Iacob, J. W. Jaromczyk, and N. Moore. The Edition Production Technology (EPT) and the ARCHway and Electronic Boethius projects. In *Proceedings of the 17th Joint International Conference of the Association for Computers and the Humanities and the Association for Literary and Linguistic Computing (ACH/ALLC)*, Jun 2005.

[47] Lazaros Koromilas, George Chinis, Irini Fundulaki, and Sotiris Ioannidis. Controlling access to XML documents over XML native and relational databases. In Willem Jonker and Milan Petkovic, editors, *Secure Data Manag.*, volume 5776 of *LNCS*, pages 122–141. Springer, 2009.

[48] Andreas Laux and Lars Martin. XUpdate—XML update language. `http://xmldb-org.sourceforge.net/xupdate/xupdate-wd.html`, 2000. Working Draft.

[49] Chung-Hwan Lim, Seog Park, and Sang H. Son. Access control of XML documents considering update operations. In *Proceedings 2003 ACM Workshop on XML Security*, 2003.

[50] Bo Luo, Dongwon Lee, Wang-Chien Lee, and Peng Liu. QFilter: Fine-grained run-time XML access control via NFA-based query rewriting. In *Proceedings*

of 13th ACM Conference on Information and Knowledge Management (CIKM), pages 543–552, Nov 2004.

[51] Tze-Heng Ma and Jeremy Spinrad. Transitive closure for restricted classes of partial orders. *Order*, 8:175–183, 1991. 10.1007/BF00383402.

[52] Y. Marcoux. Graph characterization of overlap-only TexMECS and other overlapping markup formalisms. In *Proceedings of Balisage: The Markup Conference*, volume 1 of *Balisage Series on Markup Technologies*, 2008.

[53] Gerome Miklau and Dan Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1):2–45, 2004.

[54] Neil Moore. The halting problem and undecidability of document generation under access control for tree updates. In Adrian Horia Dediu, Armand-Mihai Ionescu, and Carlos Martín-Vide, editors, *Proceedings of the 3rd International Conference on Language and Automata Theory and Applications (LATA '09)*, volume 5457 of *Lecture Notes in Computer Science*, pages 601–613, Berlin, Heidelberg, 2009. Springer-Verlag.

[55] Neil Moore. Reconciling two models of multihierarchical markup. In *Proceedings of the 13th International Workshop on the Web and Databases (WebDB '10)*, New York, NY, USA, 2010. ACM.

[56] Neil Moore. Computational complexity of the problem of tree generation under fine-grained access control policies. *J. Information and Computation*, 209(3):548–567, 2011.

[57] G. Nicol. Core range algebra: Toward a formal model of markup. In *Proceedings of Extreme Markup Languages*, 2002.

[58] A. Renear, E. Mylonas, and D. Durand. Refining our Notion of What Text Really Is: The Problem of Overlapping Hierarchies. *Research in Humanities Computing*, 1996. N. Ide and S. Hockey, (Eds.).

[59] Liam Roditty. A faster and simpler fully dynamic transitive closure. *ACM Transactions on Algorithms*, 4(1), 2008.

[60] O. Schonefeld and A. Witt. Towards validation of concurrent markup. In *Proceedings of Extreme Markup Languages*, 2006.

[61] C. M. Sperberg-McQueen and Claus Huitfeldt. GODDAG: A data structure for overlapping hierarchies. In Peter King and Ethan V. Munson, editors, *DDEP/PODDP*, volume 2023 of *Lecture Notes in Computer Science*, pages 139–160. Springer, 2000.

[62] C. M. Sperberg-McQueen and Claus Huitfeldt. Markup discontinued: Discontinuity in TexMecs, Goddag structures, and rabbit/duck grammars. In *Proceedings of Balisage: The Markup Conference*, volume 1 of *Balisage Series on Markup Technologies*, 2008.

[63] M. Stührenberg and D. Jettka. A toolkit for multi-dimensional markup—the development of SGF to XStandoff. In *Proceedings of Balisage: The Markup Conference*, volume 3 of *Balisage Series on Markup Technologies*, 2009.

[64] TEI Consortium (eds.). 20. Non-hierarchical structures. In *TEI P5: Guidelines for Text Encoding and Interchange* [65].

[65] TEI Consortium (eds.). *TEI P5: Guidelines for Text Encoding and Interchange*. TEI Consortium, 2012.

[66] J. Tennison and W. Piez. The layered markup and annotation language (lmnl). In *Proceedings of Extreme Markup Languages*, 2002.

Vita

Name: Neil Moore

Date and place of birth: 11 February 1981, Pikeville, Kentucky, United States

Education: B.S., Computer Science, University of Kentucky, 2002
B.S., Mathematics, University of Kentucky, 2002

Employment History:

- Sep 2009–Present: IS technical support specialist IV, University of Kentucky.
- Jan 2008–Sep 2009: Research assistant, University of Kentucky
- Jan 2007–May 2008: Teaching assistant, University of Kentucky
- Jul 2003–Aug 2006: Research assistant, University of Kentucky
- Sep 2001–Aug 2003: Lead developer, BigTrends.com
- Nov 1998–May 2002: Undergraduate researcher, University of Kentucky
- Jun 1998–Aug 1998: Teaching assistant, Institute for the Academic Advancement of Youth

Honors:

- Excellence in Contribution to the Computer Science Enrichment Programs award, University of Kentucky, April 2007.
- First place graduate student talk, "A Geometric Approach to Efficient Implementation of Concurrent XML Markup". Eastern Kentucky University *18th Annual Symposium in Mathematical, Statistical, and Computer Sciences*, April 2004.
- Member of the University of Kentucky Programming Team, which advanced to the World Finals of the 25th annual ACM International Collegiate Programming Contest, Vancouver, BC, Canada.

Selected Publications:

1. R. A. Finkel, V. W. Marek, N. Moore and M. Truszczyski. Computing Stable Models in Parallel. In *Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*, pp. 72–76. AAAI Press, Palo Alto, CA, 2001.

2. J. W. Jaromczyk and N. Moore. Geometric data structures for multihierarchical XML tagging of manuscripts. In *Proceedings of the 18th European Workshop on Computational Geometry*, March 2004.

3. A. Dekhtyar, I. E. Iacob, J. W. Jaromczyk, K. Kiernan, N. Moore, and D. C. Porter. Database support for image-based electronic editions. In *Proceedings of the International Workshop on Multimedia Information Systems (MIS 2004)*, August 2004.

4. J. W. Jaromczyk, M. Kowaluk, N. Moore. On visualization of complex image-based markup. In *Proceedings of the International Conference on Computer Vision and Geometry*, September 2004.

5. J. W. Jaromczyk, M. Kowaluk, N. Moore. A web interface to image-based concurrent markup using image maps. In *Proceedings of the 6th ACM International Workshop on Web Information and Data Management (WIDM 2004)*, November 2004.

6. A. Dekhtyar, I. E. Iacob, J. W. Jaromczyk, K. Kiernan, N. Moore, D. C. Porter. Support for XML markup of image-based electronic editions. *International Journal on Digital Libraries* 6(1):55–69, 2006.

7. K. Kiernan, D. Porter, A. Dekhtyar, I. E. Iacob, J. W. Jaromczyk, N. Moore. The Edition Production Technology (EPT) and the ARCHway and Electronic Boethius projects. In *Proceedings of the 17th Joint International Conference of the Association for Computers and the Humanities and the Association for Literary and Linguistic Computing (ACH/ALLC)*, June 2005.

8. E. G. Arnaoudova, P. J. Bowens, R. G. Chui, R. D. Dinkins, U. Hesse, J. W. Jaromczyk, M. Martin, P. Maynard, N. Moore, C. L. Schardl. Visualizing and sharing results in bioinformatics projects: GBrowse and GenBank exports. *BMC Bioinformatics* 10(S-7), 2009.

9. N. Moore. The halting problem and undecidability of document generation under access control for tree updates. In Adrian Horia Dediu, Armand-Mihai

Ionescu, and Carlos Martín-Vide, eds, *LATA '09: Proceedings of the 3rd International Conference on Language and Automata Theory and Applications*, *Lecture Notes in Computer Science* 5457:601–613, April 2009.

10. E. G. Arnaoudova, J. W. Jaromczyk, N. Moore, C. L. Schardl, R. Yoshida. Phylotree—a toolkit for computing experiments with distance-based methods for genome coevolution. *BMC Bioinformatics* 11(S-4): P6, 2010.

11. E. G. Arnaoudova, D. C. Haws, P. Huggins, J. W. Jaromczyk, N. Moore, C. L. Schardl, R. Yoshida. Statistical phylogenetic tree analysis using differences of means. *Frontiers in Neuroscience* 4:47, 2010.

12. N. Moore. Reconciling two models of multihierarchical markup. In *Proceedings of the 13th International Workshop on the Web and Databases (WebDB '10)*, June 2010.

13. N. Moore. Computational complexity of the problem of tree generation under fine-grained access control policies. *J. Information and Computation* 209(3):548–567, 2011.

14. N. Moore, J. W. Jaromczyk. Finding a longest open reading frame of an alternatively spliced gene. *Proceedings of the IEEE International Conference on Bioinformatics and Biomedicine (BIBM), Workshops*, November 2011.