XPath evaluation using XML schemas rather than XML documents themselves

Matthew J. Riggott

Master of Science

School of Informatics

University of Edinburgh

2003

Abstract

XPath, the XML path language, consists of expressions denoting paths that locate elements or attributes in an XML document tree. As an example, the expression `/a//b/c` picks out any *c* attribute of a *b* element at any depth under an *a* document root element. But what if the document is constrained by its document type—a DTD or an XML schema—so *b* elements don't have *c* attributes? Or if *a* elements don't have *b* descendents, at any depth? Then evaluation is pointless.

Taking this into consideration, this thesis sets out to evaluate XPath expressions against XML schemas, not against the documents they describe, to determine whether in principle they could be satisfied.

Acknowledgements

I would like to thank Dr. Henry Thompson for his help and supervision throughout this project; thanks also to Dr. Wenfei Fan of Bell Laboratories for making some suggestions on useful reading matter. On a personal level, this project would not have been possible without the unconditional support from my mother, my father, and Sarah—words cannot express my gratitude. And of course, many thanks to Suzie for proofreading what to her must have been an inordinately dull thesis.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Matthew J. Riggott*)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

XPath [11] is a language for addressing parts of an XML document, designed to be used by other languages for such a purpose and so provide a common syntax for document addressing. XML (Extensible Markup Language) [7] is a markup language that is used to create other markup languages, instances of which are stored as documents. XML came about after a concerted effort by the World Wide Web Consortium (commonly known as the W3C, the Web's governing body) to design a flexible language for use over the Internet. It is a sub-set of the enormous Standardized General Markup Language (SGML) [17], which as its name implies is a general markup language with huge capabilities. However the sheer complexity of SGML means it can be hard to learn and document processing requires much effort. A W3C working group agreed on a sub-set of SGML that meant processing would be much simpler, documents would be easy to create and human-readable, whilst still being flexible enough to 'support a wide variety of applications' [7].

Although a markup language such as XML is not a new idea, it has found great popularity due to well thought-out parts of its design, notably:

- **Data exchange**: computing has traditionally been hampered by proprietary, incompatible data formats, and the Web has been no exception. An XML document is stored as human-readable text using the Unicode standard [12], so if all else fails it can be edited by hand. XML is also an open standard that can be used freely without being restricted by copyright or patents.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <memorandum id="30df7">
3   <header>
4    <from>John Smith</from>
5    <to>Steve Blois</to>
6    <to>Joe Bloggs</to>
7    <to>Peter O'Neil</to>
8    <subject>Sales figures</subject>
9   </header>
10  <message>
11   Sales figures for August required by noon Friday.
12  </message>
13 </memorandum>
```

Figure 1.1: A simple XML document.

- **Customization**: by agreeing on an *application* (a markup language defined in XML) a group of people or organizations can transfer data in a set format and create programs to handle that data. Examples include Scalable Vector Graphics [16] and the mathematical markup language MathML [8].

- **Self-describing data**: figure 1.1 shows an XML document. The data in the document is surrounded by XML elements, the names of which are chosen to describe the data they hold. This allows XML to be self-describing to a point, and allows (in this case) any English-speaker to understand the meaning and structure of the document.

Of course the structure of the document in figure 1.1 is only apparent if one speaks English, which computers have a tendency not to do. To check whether the structure and syntax of an XML document is valid, XML allows for document type definitions (DTDs) to be stored either as part of the document itself, or referenced from a separate file. Although part of the XML specification, the DTD is a hangover from the days of SGML, and has a syntax all of its own. Once an XML document is well-formed (i.e. it complies with the XML syntax), it is also valid if it conforms to a given DTD. The

DTD defines what elements and attributes can appear in a document and where they can appear, along with having some control over what type of data can occur inside elements and attributes.

However, as XML became more popular and more widely used, DTDs were decried for being too complex; so the W3C set out to create a successor. Early 2001 saw the recommendation* of XML Schema [15, 30, 5], and XML-based syntax for defining XML documents†. Contrary to being simpler than DTDs, XML Schema is a very powerful specification that allows for much more control over XML documents, including: specifying data types for elements and attribute content; inheritance from other schemas; and minimum and maximum occurrences of elements (DTDs rely on regular expressions; XML Schema allows for limits to be set explicitly). XML Schema has superseded DTDs and is likely to become more dominant as awareness of it grows.

1.1 XML and its companion recommendations

XML was proverbially in the right place at the right time. It has become the *lingua franca* of the Internet and ideas for its use has gone beyond the original intentions. Attempts are being made to represent databases in it, and XML now sports among other things query languages (e.g. [13, 6]) and processing languages (e.g. [22]). To aid this development of XML technology—and also to control it in a benevolent fashion—the W3C publishes companion recommendations, i.e. standards that complement and augment XML. Two of these recommendations are XSLT (Extensible Style Language transformations) [10] and XPointer [29]. These are used to transform XML documents into XML documents of different types, and for specifying locations inside an XML document respectively.

Clearly both these languages require a method of addressing parts of an XML doc-

*The W3C is made up of constituent organizations, and based in three academic institutions across the world. As such it is not a government agency or a recognized international standards organization. To indicate this it uses the softer wording of ‘recommendation’ for what is essentially an international standard.

†Imagine a future where XML etc. has been lost and people come across an XML document which is described using an XML schema—which is itself an XML document. One can see this as being an equivalent of the chicken-or-the-egg argument: which came first, XML or XML Schema?

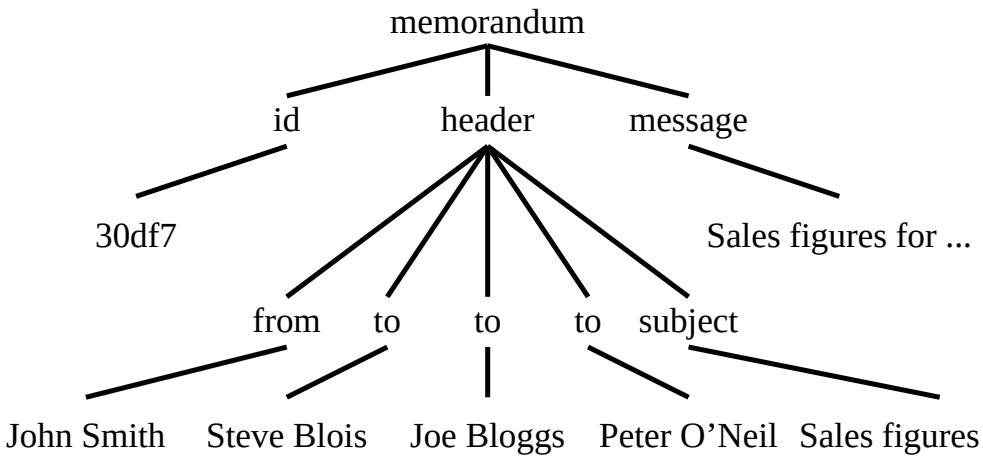

Figure 1.2: An XML document represented as a tree of nodes.

ument. So rather than duplicate the effort the W3C decided on creating a new language with the primary purpose of providing document addressing functionality, viz. XPath [11]. Along with its primary purpose XPath also provides basic string, number, and Boolean manipulation. XPath uses a non-XML syntax so that it can be used within XML attribute values in documents themselves. The most important construction in XPath is the expression, an example of which is given below.

```
/child::document/descendant::paragraph[attribute::type]
```

This expression looks for the *document* element as the root element (the element inside which the entire XML document is held), and returns any children or children of children that have the name *paragraph* and contain an attribute named *type*. (A large number of examples of XPath expressions can be found in the XPath specification [11]; further discussion of XPath can be found in this thesis in section 3.3 on page 16.) When evaluated, an expression will result in either a node set (a node being an element, an attribute, or one of a few other such parts of an XML document), Boolean, string, or number.

XPath models XML documents as unranked trees of nodes. Such trees are ordered, finite, and labelled, where nodes can have a arbitrary number of children and so each label is not associated with a fixed rank. Figure 1.2 has such a representation of the

document given in figure 1.1. In that view inner nodes correspond to elements which determine the structure of the document while the leaf nodes and the attributes provide the content. Using thirteen axes (relationships between nodes; *child* and *descendant* in the expression above are examples of axes), XPath can traverse the tree and locate certain elements, attributes. To take the expression

```
child::memorandum/descendant::to
```

(which finds any *to* nodes underneath the *memorandum* element in the tree) as an example and evaluate it against the XML document in figure 1.1, XPath would return the node set [*to*, *to*, *to*], each corresponding to one of the three *to* elements in the document.

In this instance the node set contains three elements, but it also possible that the node set will be returned empty. Evaluating the expression

```
child::to/descendant::memorandum
```

would return an empty node set because *memorandum* does not appear as a child of any *to* element. Similarly, what if the evaluation is looking for elements with certain attributes, when the elements don't have any attributes? What if the elements don't actually exist in the document?

With hindsight it would be obvious that an evaluation of this sort is pointless. On documents the size of that in figure 1.1 it makes little difference as it is quite simple. However, complex documents that are time-consuming to process (e.g. very flat trees with a large number of instances of an element) or very large documents (e.g. databases) bring to question this blind faith in XPath evaluation. Something as simple as a spelling mistake would cause unnecessary processing—and if the evaluation took place over multiple documents this could only add to the problem. Something is required to evaluate whether the evaluation of the XPath expression is feasible or necessary before the XML document itself is consulted.

1.2 Evaluating against document types

The one place the details of the document are found other than the document itself is in its type definition, in the form of either a DTD or an XML schema. From the document type one can find out what elements and attributes are allowed, what their data content is, where they can appear, and at what frequency. Indeed, by evaluating the XPath expression against the document type one can perform operations as simple as a spell-check and as complex as discovering whether the fifth *item* child of a *list* element that is somewhere below a *paragraph* element can have an attribute named *language* with the content *español*.

There are obvious advantages to this approach. In an extreme example an evaluation against a ten kilobyte schema might show evaluation against a five-hundred megabyte XML document database to be pointless. The hypothesis of this thesis then, is that introducing an intermediate step—before an XPath expression is evaluated against an XML document—where the expression is evaluated against the document type to prove the feasibility of a full evaluation, would be a sagacious measure. The objective of this project is to implement such a system to be used to provide this functionality.

1.3 Project overview

The original aim of this project was to design and implement a complete system for such abstract evaluation of XPath mentioned above. While a theory was produced for such a system (see chapter 3), XPath proved to be a more complex language than expected and only a partial implementation of the language was possible given the time limit of this project. The implementation concentrated on a particular type of XPath expression, essentially ignoring XPath's string, number, and Boolean manipulation facilities and instead concentrating on the language's most important function, the traversal of XML document trees. The occurrent system implemented six of the thirteen axes (specifically those that traverse the tree in a downward fashion) as proof that the system is feasible.

The system does not handle DTDs. A decision to concentrate solely on XML

schemas was made for two reasons, viz. that schemas are the more interesting of the two, but mainly because a DTD's functionality is a sub-set of XML Schema and it is a simple matter to transform a DTD into an XML schema*. Indeed tools exist to do this (the W3C has a Perl script, DTD2XSD, available from its Web site for example) and it would be trivial to add functionality to the system to take in a DTD, transform it into a schema and use it that way.

The system works as a black box. It takes in input, and based on that input returns output representing either a success or a failure of the evaluation. The input is an XML schema, an XPath expression, the document root element (the upper-most element in the document tree), along with some optional arguments. From here, the schema is modelled as a finite-state automaton and the XPath expression is used as an input language for the automaton. The evaluation is successful if the input language corresponds to the automaton, otherwise it fails†. This project builds upon an XML Schema parser and validation tool by Henry Thompson and Richard Tobin of the W3C and the University of Edinburgh [31]. As this is written in the Python programming language, the practical work in this project also uses Python.

Thesis outline The following chapter contains a review of related literature. Chapter 3 contains a discussion on the theory, design, and methods behind the implementation. The system itself and its workings are described in chapter 4, while analysis, tests, and future work follow in chapter 5. The main text concludes with a summary in chapter 6, after which come a small number of appendices and the bibliography.

*It is prudent to note here that *XML Schema* (as a proper noun) is used throughout this thesis to refer to the actual recommendation, while *schema* or *schemas* refers to specific instances of the recommendation.

†Finite-state automata and the theory used in this implementation are discussed in further detail in section 3.1.

Chapter 2

Literature review

To the best of my knowledge there has been no previous work on evaluating XPath expressions against document types. However, much research has focused separately on the three main parts of this project: automata theory, XML Schema, and XPath. This chapter details some of the work related to this thesis, and discusses its relevance.

The finite automaton came about during the middle years of the last century, based on Turing's model of algorithmic computation [33]. Turing's paper is considered by many to be the foundation of modern computing; in it he defined a machine with finite control and an input-and-output tape. The machine could, in one move, read a symbol from the tape, write a different symbol back onto the tape, change state, and move the position of the tape in either direction.

From this work came McCulloch and Pitts's model of a neuron [26]. This automaton-like binary device took excitatory and inhibitive input from other neurons and fired once its activation passed a given threshold. Based on this Kleene [24] defined finite automata (see section 3.1 for a definition) along with regular expressions, and proved them to be equivalent. From this comes the axiom that any regular language may be represented as a finite automaton.

The part of XML Schema that constrains the validity of elements and attributes, the content model, is itself a regular language. In [29], Thompson and Tobin show that by modifying Aho and Ulman's algorithm for converting regular expressions to finite-state automata [1], XML schemas can be represented as finite-state automata themselves. Thompson and Tobin use this to implement an XML Schema validation

tool, but it can also be used to evaluate an input language *against* the XML Schema, as will be seen in this thesis.

Recent database research has seen a shift away from traditional relational database management systems towards XML and semi-structured data, with XML Schema used for data description. In [3], Arenas et al. show that XML Schema has problems with the integrity constraints that are integral to database design (i.e. constraint checking is intractable and NP-hard). This, along with other problems (e.g. a lack of required restriction, mentioned in [29]), has seen an updated version of XML Schema drafted that will eventually supersede the current standard.

XPath has also seen much research interest since its publication by the W3C. [27] sees Neven introduce tree-automata theory for use with XML, and shows how to use such automata to parse and accept XPath expressions. XPath is a complex language, and some have realized that in many cases, a full XPath parser is not necessary. In [4], Benedikt et al. discuss several fragments of XPath (for instance only implementing either downward or upward axes), and concentrate on simplifying and optimizing XPath expressions.

On papers that discuss practical implementations rather than theory, there are a number of techniques for processing XPath expressions. In [18] and more compactly in [19], Gottlob et al. introduce algorithms for evaluating XPath expressions using top-down and bottom-up parsing techniques. Interestingly they note that the XPath axes (with the exception of *attribute* and *namespace*) can be defined using using two primitives, *firstchild* and *nextsibling*, and their inverses. Altinel and Franklin [2] describe XFilter, a system for evaluating large numbers of XPath expressions using what is essentially a highly-optimized non-deterministic finite-state automata. A similar system, XTrie, is described in [9] by Chan et al. XTrie is a more efficient system than XFilter; it identifies common sub-strings in the XPath expression and organizes them in a trie*, and also uses extra optimization techniques at run time. A related system is detailed by Green et al. [20]; This again models the XPath expression as a finite-state automaton—although here it is deterministic—and evaluates it against streamed XML documents.

*A tree for storing strings. There is one node for each common prefix, with the strings stored in leaf nodes.

These systems all evaluate against the XML document itself, and pay no attention to the related DTD or XML schema. The latter three also model the XPath expression as finite-state automata and use the document as the input language. This project will differ by using the document type and not the document, and by modelling the document type as finite-state automata with the XPath expression as the input language.

Chapter 3

Materials and methods

This chapter introduces the theory behind the project, discussing finite-state automata, XPath expression parsing, document navigation, and the problems associated with them. It also includes conceptual design matters and design decisions. The theory in this chapter is the basis for the implementation of the system discussed in the following chapter.

3.1 Finite-state automata

With the history of the finite-state automaton being covered in chapter 2 this section will concentrate on the underlying theory. To illustrate finite-state automata, this author will borrow from the 'sheep language' example found in [23, p. 34]. The language of sheep consists of strings like the following:

baa!

baaa!

baaaa!

baaaaa!

...

More formally, we can say that the language consists of strings that start with a *b* and is followed by an *a*, then one or more *a*s, and finally an exclamation mark. This can be represented with the regular expression `baa+!`. (The Kleene plus, +, indicates

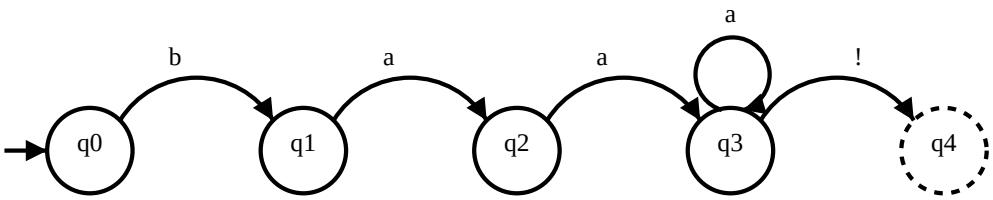

Figure 3.1: A finite-state automaton for a sheep language. The finishing state is represented as a dotted circle.

that the previous character must appear at least one time, up to an unbounded number of appearances.) Figure 3.1 shows an automaton that models this regular expression.

An automaton recognizes a set of strings in the same way a regular expression does. There are many ways to visually represent an automaton, but here it is as a directed graph. There are a finite set of nodes (represented by labelled circles) and a finite set of edges (directed links from one node to another, represented by arrows). The nodes represent states in the automaton. An automaton has a starting state (here q_0), a set of final states (here just one, q_4), and a set of transitions represented by the arrows. The automaton is used by giving it an input sentence. The FSA takes the input sentence and uses it to move through the states and along the transitions to the finishing state. The automaton starts in the start state and checks the first symbol of the input—in the sheep language each letter represents a symbol. If the automaton matches the symbol and an edge leaving the state, the automaton will follow it and move on to the next state and advance one symbol in the input sentence. If the automaton reaches the end of the input and is in a finishing state, then the machine has successfully recognized the sentence, and the sentence can be considered an instance of the automaton's grammar. The automaton may never get to a finishing state, either because there is no edge matching the current symbol, or because it finishes in a non-final state. In this case the input sentence is not recognized, and it is not a valid instance of the automaton's grammar.

As an example for this sheep language automaton, *baaaaaaaaa!* is a valid sentence because the automaton can keep looping from q_3 to q_3 until it reaches the exclamation mark, upon which it can move to the finishing state, q_4. However, *baba!* is not a valid sentence because the automaton has no edge from state q_2 labelled with the *b* symbol.

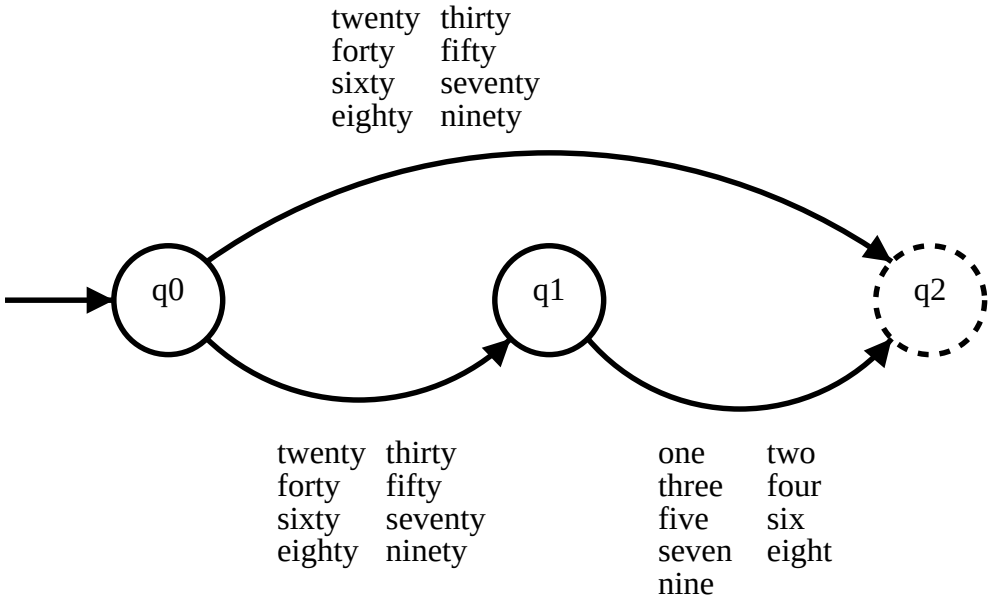

Figure 3.2: A non-deterministic finite-state automaton.

Formally, a finite-state automaton can be defined as needing the following five parameters:

- Q: a finite set of n states q_0, q_1, ..., q_n;
- Σ: a finite input alphabet of symbols;
- q_o: the start state;
- F: the set of final states, $F \subseteq Q$; and
- $\delta(q, i)$: the transition function between states. Given the state $q \in Q$ and the input symbol $i \in \Sigma$, the function returns a new state $q\prime \in Q$.

3.1.1 Non-determinism

The automata discussed so far have been deterministic, i.e. there has been no decision-making necessary since an automaton simply follows the correctly-labelled edges until it runs out of input, or gets stuck. However, figure 3.2 shows an automaton that requires

some decision making. The automaton recognizes English words for the numbers twenty through to ninety-nine. If the automaton takes as input a number that is a multiple of ten, it has two possible edges from the starting state. To allow for this an algorithm needs to be introduced that can handle non-determinism. There are three standard solutions to this:

- Put a marker at every choice point, and then follow a possible transition. When a wrong transition is taken, the algorithm backtracks to the last marker and tries another path.
- Look ahead in the input to help make a decision on which path to take.
- At each choice-point try every path in parallel and drop those that eventually prove to be a wrong choice.

The algorithm is not decided upon by the automaton; it is up to the programmer to use the automaton with the algorithm of his choice. Note also that it is possible to convert any non-deterministic automaton into a deterministic one, as proved by Rabin and Scott [28].

3.2 XML schemas and automata

Key to XML Schema is its content model. The content model is used to define complex types (not necessarily that complex since a complex type is any term in a schema that contains more than just text). The content model is described in terms of particles, which are terms in the schema grammar for element content. A particle has three properties:

- ***term***: one of an element declaration (specifying the element's name and namespace), a wildcard, or a sequence or choice of particles.
- ***min-occurs***: a positive integer denoting the minimum number of occurrences of the *term* that is acceptable. If it is zero the *term* is optional.

```
1  <element name='head'>
2   <complexType>
3    <sequence>
4     <element ref='t:title'/>
5     <element ref='t:author' maxOccurs='unbounded'/>
6     <choice minOccurs='0' maxOccurs='unbounded'>
7      <element ref='t:date'/>
8      <element ref='t:note'/>
9     </choice>
10   </sequence>
11  </complexType>
12 </element>
```

Figure 3.3: Extract from an XML schema.

- ***max-occurs***: a positive integer, or ∞, denoting the maximum number of occurrences of the *term* that is acceptable.

This list is deliberately brief; more details are available in [30].

The XML Schema content model is similar to—but more powerful than—regular expressions. Whereas regular expressions use the Kleene plus, Kleene star (∗, allowing for zero or more occurrences), and question mark (zero or one) to define the occurrence range of a symbol, XML Schema allows the range to be explicitly and precisely defined. The content model is still a regular language however, and can therefore be represented as a finite-state automaton. Each element or attribute declaration in a schema that is of a complex type can be modelled as an automaton, so a schema itself can be modelled as a collection of automata.

As an example, an extract of a schema is shown in figure 3.3. The extract defines an element named *head*, which has a complex type. Its content model defines a sequence of elements starting with *title*, followed by one or more occurrences of *author*[*], and then followed by zero or more occurrences of either *date* or *note*. Fundamentally, this

[*]Note that the *min-occurs* parameter is missing on line 5. When this happens, XML Schema defines the default minimum occurrence as one.

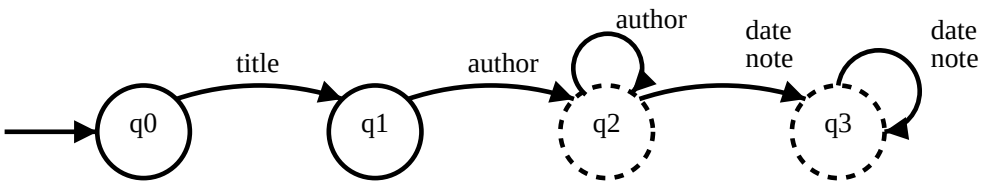

Figure 3.4: Automaton corresponding to the schema extract in figure 3.3.

allows the *head* element to contain a single document *title* and any number of *authors*, *dates*, and *notes*. A finite-state automaton representing this content model is shown as a directed graph in figure 3.4.

At a basic level, XSV [29]—an XML schema validation tool from the University of Edinburgh—works as above. It converts a schema to a collection of automata, and uses these to validate the schema against the XML Schema specification. The automata are actually augmented automata due to a need to cope with constraints XML Schema places on particles, so there are some differences. However the theory is sound, and the interested reader is directed towards [32].

XSV is used as the basis for this project. By parsing the XPath expression into suitable sub-strings and using these sub-strings as input sentences for each automata in the collection XSV produces, the collection of automata, it is possible to evaluate XPath expressions against XML schemas.

3.3 Parsing XPath expressions

XPath expressions were mentioned briefly in section 1.1 on page 4. They will now be discussed in more detail, with references to the Backus-Naur form grammar used to specify the language formally, as reproduced in appendix A on page 54. All instances of XPath take the form of an expression. An expression is a complex and powerful construct that is recursively defined (it is not at all uncommon to find that an expression contains an expression that contains an expression, etc.). There are several types of expression, the most important being the *location path*. This construct is the primary means of navigating through the document tree. Formally, it selects a set of nodes relative to the context node. A location path is made up of a number of *location steps*.

Each location step selects a set of nodes relative to the context node, and each node in that set is used as a context node for the following location step. The final step's sets of nodes are unioned together, and this is used as the result of the location path as a whole.

Steps in a location path are separated by solidi. If a location paths starts with a solidus it is absolute, and the preliminary context node is the root node*. If the path starts with a location step (i.e. without a prefixed solidus) it is relative and no context node is given explicitly in the path. Usually XPath is used in conjunction with another language, and this would supply the context node. However, in this project XPath is used in isolation so this implementation will need to include an algorithm to supply possible context nodes (see section 4.4.1).

As an example (taken from [11]), take the location path:

```
child::div/child::para
```

This selects the *para* children of the *div* children of the context node—in other words the *para* element grandchildren of the context node that have *div* element parents. In this example, the relationship between the context node and the current step is is given before the `::` delimiter, in this case *child*. This is known as an axis, and XPath defines thirteen axes as shown in table 3.1 on the following page. The axes overlap quite substantially and can be partitioned into two sub-sets, viz. forward axes and reverse axes. Forward axes include *child*, *descendant*, *following-sibling*, *following*, *attribute*, *namespace*, and *descendant-or-self*, and always move downwards in the document tree. The other axes are classed as reverse axes, and always move upwards in the document tree. The exception is the *self* axis, which can be classed as either since it doesn't move from the context node.

The axes are extremely important given the use of finite-state automata, as they add a great level of non-determinism. Take the *descendant* axis: this may involve moving along multiple transitions in one step, for instance if the resulting node set includes children of children. Since an XML schema is held as a collection of automata, it will be necessary to move not just from state-to-state, but from automaton-to-automaton. It

*The root node is a special node, denoted `/`, that sits above the document root element in the document tree. It cannot contain anything other than the root element.

Table 3.1: Axes defined by XPath.

Axis name	**Description**
child	Children of the context node.
descendant	Any node at a lower level of the tree, but within the same branch. Children, children of children, etc.
parent	The node directly above the context node.
ancestor	Inverse of *descendant*. Contains the parent node, its parent, and so on up the tree to the root node.
following-sibling	All the following siblings of a node (i.e. any node with the same parent that is to the right of the context node in the tree).
preceding-sibling	Siblings with the same parent but to the left of the context node in the tree.
following	Any node in the tree that follows the context node in document order, excluding descendants, attributes, and namespaces.
preceding	Any node in the tree that precedes the context node in document order, excluding ancestors, attributes, and namespaces.
attribute	Contains attributes of the context node. Empty unless the context node is an element.
namespace	Contains the namespace nodes of the context node. Empty unless the context node is an element.
self	The context node itself.
descendant-or-self	Contains the context node itself along with its descendants.
ancestor-or-self	Contains the context node itself along with its ancestors.

is clear then, than any algorithm employed will need to look ahead across states and automata to find acceptable transitions.

After an axis specifier in a location step come its two other component parts, a node test and a (possibly empty) set of predicates. The node test is fairly self explanatory, and is true only if the node being tested has the same namespace and local name. It is complicated slightly by the inclusion of wildcards which can match node types (e.g. elements, attributes, comments). In the most recent example, the node tests are *div* and *para*. These are testing against element names, and do not specify a namespace.

The predicate list is the most interesting part of a location path, simply because of its definition (line numbers equate to those in appendix A):

```
35  Predicate           ::= '[' PredicateExpr ']'
36
37  PredicateExpr       ::= Expr
```

A predicate can contain any type of XPath expression, meaning that a location path can if required contain any number of nested location paths. Successful parsing and handling of predicates is clearly the most complicated part of location paths.

The Backus-Naur form definition of XPath lends itself very well to an object-oriented implementation. XPath has been defined so that it builds upon itself as much as possible. By modelling the non-terminal symbols in the grammar as objects this should make implementing location paths and its recursive predicates somewhat easier.

3.3.1 Abbreviated syntax of location paths

Those location steps anticipated to be the most frequently appearing are allowed to be shortened as shown in table 3.2 on the next page. Any algorithm used to parse XPath expressions must be capable of handling both the abbreviated and unabbreviated syntax. This need not be complex and should be possible by parsing a given location path and replacing any abbreviated steps with their verbose equivalent, and then always working on a full unabbreviated path.

Table 3.2: XPath's abbreviated syntax. In the final two rows, any node test can be part of the abbreviated syntax but *type* and *para* are used as examples.

Abbreviation	**Verbose equivalent**
`//`	`descendant-or-self::*`
`.`	`self::*`
`..`	`parent::*`
`@type`	`attribute::type`
`para`	`child::para`

3.4 Location paths as automata input sentences

XPath was designed to be evaluated against an XML document tree. In this project the model is twisted slightly by evaluating against a document's XML schema, so the actual XML document tree is useless. So how does the evaluation take place? The evaluator must build feasible document trees that both conform to the schema and agree with the XPath expression. An evaluation will be successful if it creates a document tree from the schema that will allow the XPath expression to return a non-empty node set. It is this abstract property that sets this project aside from previous XPath evaluators. The discussion will now move onto parsing location paths for use with automata, and this focus on possible, rather than concrete, structures will be important to the choice of parsing method.

Just as the with the sheep language on page 11, the location path will need to be separated into a set of input symbols that can be used to move from state to state in a finite-state automata. In the sheep language it was individual characters, and with XPath it seems obvious from the previous section that these symbols should be location steps. The question now is how to use these steps with the finite-state automata to get the required result. The two options considered here are parsing from left-to-right in the path, and from right-to-left.

3.4.1 Left-to-right parsing

A left-to-right parser starts with the first location step and attempts to create a feasible document tree from that point. It is similar to a top-down parser, moving from the top of the tree, down to the bottom, but of course XPath allows location steps to move up or down a tree, so it is not entirely synonymous.

In its attempts to evaluate the location path successfully, the parser will have two major algorithms. The first will be concerned with forward axes (downwards through the tree), and the second with reverse axes (upwards through the tree). The forward algorithm will be moving through the automata as normal, from state to state as it finds children, descendants, etc. that conform to the location step. But the reverse algorithm will be concerned with the path already taken by the forward algorithm. For instance, the *ancestor* axis contains the parent of the context node, its parent, its parent's parent, and so on up to the root node. If the location path is absolute, then the path already taken will contain all these nodes. However, if the location path is relative it will come to a point when the parent is not specified. To cope with this the reverse algorithm will need to be augmented with a method of reversing through the automata to find complete the location step.

3.4.2 Right-to-left parsing

The right-to-left parser starts from the last location step and moves in reverse through the path. Whereas the left-to-right parser is attempting to find a feasible document tree from the source, this parser takes all the nodes that could be part of the resulting node set, and attempts to create a document tree from the bottom up that leads to the source context node. The algorithm needs to find all nodes that comply with the location step's node test, and then use the step's axis to create a document tree in reverse to get to nodes that comply with the location step to the left of it (i.e. the previous step in the path). This technique has many problems, especially when it comes to reverse axes. A reverse axis would require the algorithm to create a sub-tree that could connect to the top-level nodes in the current tree. For example, take the sub-path:

```
child::para//ancestor::div
```

```
1  <element name='code'>
2   <complexType mixed='true'>
3    <choice minOccurs='0' maxOccurs='unbounded'>
4     <element ref='eg:emph'/>
5     <element ref='eg:code'/>
6    </choice>
7   </complexType>
8  </element>
```

Figure 3.5: An example of recursive definitions in XML Schema.

As the parser is moving from right-to-left, it would look at the *ancestor* axis first. A tree would need to be created that ran from the *div* element to the *para* element (essentially the same as the left-to-right algorithm). The complexity of this technique, along with its potential for mistakes, led to it being discarded as an option. Instead, the left-to-right parsing technique was chosen.

3.4.3 Advantages and disadvantages of left-to-right parsing

The left-to-right parser has its own advantages and disadvantages. It never wastes time exploring trees that cannot begin with the context node, since it generates only those trees. However it may spend considerable time looking at trees that are not consistent with the following steps, arising from the need to generate trees as it moves through the location path.

It also suffers from problems with recursion. Consider the fragment of an XML schema in figure 3.5, defining an element *code*. On line 5, the schema segment allows the *code* element to contain a *code* element. This recursive definition means the evaluator could attempt to create an infinite tree in the search for a tree that matches the XPath expression. Clearly this problem will need to be resolved in the implementation, and it is discussed further in section 4.5.

A final problem is with the repeated parsing of sub-trees. In its effort to evaluate the XPath expression, the algorithm may end up searching through the same sub-trees again and again. The parser may build a valid sub-tree, for example for a regularly

used element and its children, only for it to be discarded because it doesn't match the location step—and then use it again in the next step. While this doesn't impact on the final result of the evaluation, it would be an efficiency improvement if sub-trees were to be reused.

3.4.4 A note on the Earley algorithm

The Earley algorithm [14] is a linguistic parsing technique for context-free grammars that removes the problems associated with top-down and bottom-up parsing, and reduces a possibly exponential-time problem to polynomial-time. The idea was briefly entertained that the algorithm could be used in this situation, to remove the problem of recursion from the parsing algorithm. After discussion however, this idea was forgotten.

3.5 Navigating the document tree

In [18, 19], Gottlob et al. show that ten of the thirteen axes in XPath can be defined using just two primitives and their inverses. These two primitives are *firstchild*, which returns the first child of a node, and *nextsibling*, which returns the next node in document order with the same parent. The definitions of the axes using these primitives are shown in table 3.3 on the next page. Some of the axes are defined in terms of others, but note that they are acyclic. Implementing these primitives rather than ten separate axes will reduce the complexity of the algorithm. Once the two primitives are in place, implementing the forward axes should not be as complex as it would have been to implement them all independently. Of course in the original papers, these primitives were designed for use on XML document trees, so the *firstchild* primitive would return the first child. This thesis is using document types, and so the *firstchild* primitive will need to be changed slightly to return a *set of nodes* that are *feasible* first children. Likewise, the *nextsibling* primitive and the inverses will not return definitive nodes, but sets of nodes containing possibilities. It will be up to the algorithm to decide which node to use, by attempting evaluations using each returned node in turn.

The three remaining axes are special cases and need to be dealt with separately.

Table 3.3: XPath axes in terms of two primitives and their inverses. Based on table on page 2 of [19].

Axis	**Definition in primitive terms**
child	$\mathit{firstchild}.\mathit{nextsibling}*$
parent	$(\mathit{nextsibling}^{-1})*.\mathit{firstchild}^{-1}$
descendant	$\mathit{firstchild}.(\mathit{firstchild} \cup \mathit{nextsibling})*$
ancestor	$(\mathit{firstchild}^{-1} \cup \mathit{nextsibling}^{-1})*.\mathit{firstchild}^{-1}$
descendant-or-self	$\mathit{descendant} \cup \mathit{self}$
ancestor-or-self	$\mathit{ancestor} \cup \mathit{self}$
following	$\mathit{ancestor\text{-}or\text{-}self}.\mathit{nextsibling}^{-1}.\mathit{nextsibling}*.\mathit{descendant\text{-}or\text{-}self}$
preceding	$\mathit{ancestor\text{-}or\text{-}self}.\mathit{nextsibling}^{-1}.(\mathit{nextsibling}^{-1})*.\mathit{descendant\text{-}or\text{-}self}$
following-sibling	$\mathit{nextsibling}.\mathit{nextsibling}*$
preceding-sibling	$(\mathit{nextsibling}^{-1})*.\mathit{nextsibling}^{-1}$

The *self* axis simply needs to return the context node, and *attribute* and *namespace* return their respective node sets.

Chapter 4

Implementation

This chapter covers the implementation of the theory that was espoused in the previous section as a Python program. As mentioned briefly in chapter 1, the system detailed here is not a complete implementation of XPath expressions—it should be thought of instead as a proof of concept. The system implements six forward axes, *self*, *child*, *descendant*, *descendant-or-self*, *following-sibling*, and *attribute*. It also supports namespaces, but not the *namespace* axis (a minor addition that time would not allow for). The program evaluates location path expressions, and can handle a sub-set of the predicate construct that includes evaluation of binary *or* and binary *and* expressions, and existence-checking. Note that in node tests only literal names are supported, and functions and wildcards such as *node()* and $*$ are not. Due to this lack of wildcard support, the XML Schema elements *any* and *anyAttribute*—which allow a document to include any element or attribute even if they are not defined in the schema—are not currently supported either.

4.1 Collecting unknown data

Some data are required to run the evaluation that are simply not available to the program, and must be supplied by the user, viz. the document root element and the schema's namespace prefixes. While an XML schema defines the valid structure of a document, it does not specify a document root element. Any element in the schema

```
1  <?xml version='1.0'?>
2  <schema xmlns:eg='http://example.org/'
3          targetNamespace='http://example.org/'
4          xmlns='http://www.w3.org/2001/XMLSchema'>
5
6   <element name='doc'>
7    <complexType>
8     <sequence>
9      <element ref='eg:head' minOccurs='0' maxOccurs='1'/>
10     <element ref='eg:body'/>
11    </sequence>
12   </complexType>
13  </element>
```

Figure 4.1: XML schema fragment showing the use of namespace prefixes.

can act as the root element in which all other elements are held as long as the document conforms to its content model. In practice, by defining an element in the schema but not referencing it in any other element's content model, most schemas are designed with a single element implicitly defined as the document root. (As it is unreferenced, it can only be used in the document as a container for all other elements, i.e. the document root.) However this is not necessarily so, and the only way to be sure of the document root is to have it specified by the user as a command-line argument.

Namespaces are referenced in an XML schema using prefixes. Figure 4.1 shows an schema fragment which includes the namespace *http://example.org/* that uses the prefix *eg* (line 2). The prefix rather than the full namespace is used throughout the schema (see lines 9 and 10). Prefixes are also used in XPath to denote namespaces, partly due to the fact that namespaces use solidi as delimiters (e.g. uniform resource identifiers such as *http://example.org/*), and so could cause problems parsing location paths, which also use solidi to delimit location steps.

The program itself does not directly access the XML schema. It passes the schema to XSV for validity checking, and uses the result set from that check to evaluate the

location paths (see below, section 4.2). Therefore, the system does not have access to the namespace prefixes defined in the schema. While XSV keeps track of the actual namespaces it does not keep the prefixes. Thus, in translation from schema to XSV result set, the namespace prefixes are lost. In order to retain the prefixes, they must be included along with the namespace they point to as a command-line argument. Note that the prefixes are only for use in the location path and so they need not be the same as those used in the schema.

Along with the document root element, the XML schema and the XPath location path make up the minimum data required to perform evaluation. Only if the XML schema contains namespaces are the prefixes required.

4.2 Obtaining the finite-state automata

XSV can be used as both a command-line tool and a utility for other programs. The system here uses it as a utility, hiding it from the user. The XML schema file name is passed by the user as a command-line argument, and it is then passed onto the XSV driver. The XSV driver performs a validation and restriction-check on the schema in the file, and returns an extensive result set. For the purposes of XPath evaluation most of it is superfluous, the exceptions being an XML document defining success or failure and XSV's internal model of the schema. If the XML document indicates success the system knows the schema is valid and evaluation can take place. From here the system can extract the schema's element table from the XSV result set.

The element table is XSV's method of storing the elements, attributes and their content model. The element table contains each element's name along with its namespace, content type, attribute declarations, and a finite-state automaton representing its content model. The element table also contains many more data and functions that are not needed for XPath evaluation, so the program separates the useful data into a custom element table. It is this that provides the basis for the evaluation.

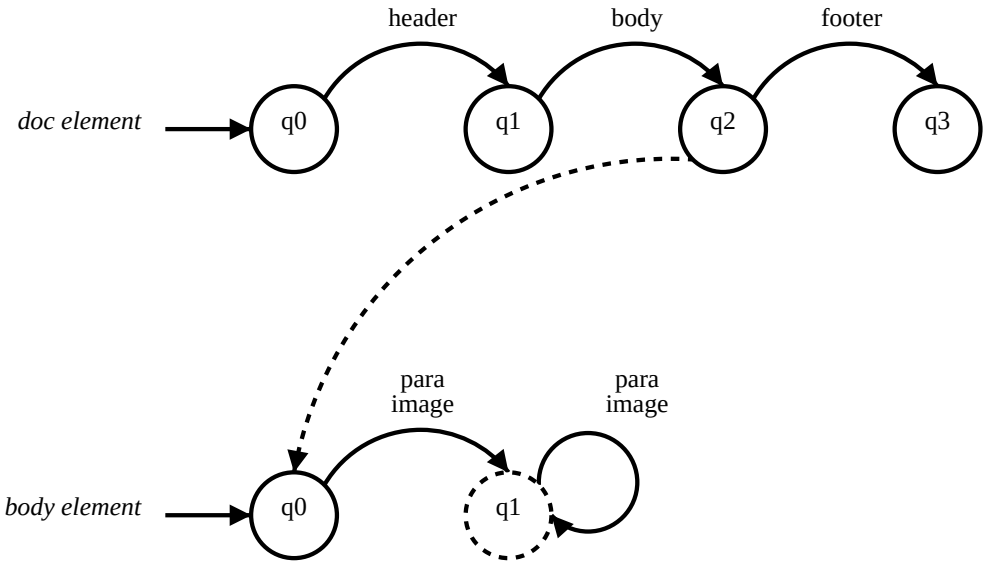

Figure 4.2: Two connected automata.

4.2.1 Using multiple automata

To perform the evaluation the automata need to be used together. For example, take the location path below (note the use of the *f* namespace prefix in the node tests, separated from the element name by a single colon):

```
/child::f:doc/descendant::f:image
```

Two automata representing a possible schema fragment for this location path are given in figure 4.2. To find out if the *doc* element can have *image* descendants, the system needs to traverse two automata, first for the *doc* element, then for the *body* element below that to come to an *image* descendant. To find that the location path is feasible the system moves from state q_2 in the first automaton to the starting state in the second automata. That state has an edge leaving it labelled *image*, proving that the location path can be evaluated successfully. Individual automata only list an element's children, and its children's siblings in that context. To find a child's children, the system moves to another automaton and continues through that until it finds a match, moves onto another automaton, or reaches a finishing state. Eventually it will reach a match or come to the end of the automata in which case it will fail.

Note that the system does not need to leave each automaton in a final state, because the location path may only specify a fragment of a full document. While the *doc* element may require *header*, *body*, and *footer* children, the location path is only interested in whether it can have an *image* descendant. The location path above is evaluated successfully while leaving the top automaton in state q_2.

4.2.2 Handling non-element nodes

The XSV automata only contain element data. Since all other node types are leaf data in the document tree, there is little point including them in the automata as the edges would all lead to the final state. Instead they are held in the element's declaration, outside its automaton. Attributes are held in their own table, and an existence-test can be done by simply checking whether the attribute exists in the element's attribute declaration table. The element's content type is used to see whether an element can contain content other than elements (i.e. text), and node types such as comments and processing instructions can occur anywhere within an XML document—although these are not supported in this implementation.

4.3 Handling XPath expressions

After the program is passed the location path, the first operation is to ensure all its location steps are in their unabbreviated form. This is done by splitting the path into steps, and checking their form. If it is abbreviated, then it is replaced by the verbose form. The location steps are then compiled into an unabbreviated location path, and this is used for the duration of the program. There is some inefficiency in this process since the next step in the program splits the location path again to convert each location step into an object-oriented form. A future version would do well to combine these two processes so to split the location path only once. Figure 4.3 on the following page contains a code fragment of the unabbreviating algorithm.

Each location step is split into its axis specifier, node test, and predicate list. In addition, the node test is split into its namespace prefix and name test, and the predicate list is split into individual predicates. All the component parts are held in an object-

```
1  if locationStep == "":
2   unabbreviatedLocationPath += "descendant-or-self::node()/"
3  elif locationStep == ".":
4   unabbreviatedLocationPath += "self::node()/"
5  elif locationStep == "..":
6   unabbreviatedLocationPath += "parent::node()/"
7  elif locationStep[0] == "@":
8   unabbreviatedLocationPath += "attribute::" + locationStep[1:] + "/"
9  elif locationStep.find("::") < 1:
10  unabbreviatedLocationPath += "child::" + locationStep + "/"
11 else:
12  unabbreviatedLocationPath += locationStep + "/"
```

Figure 4.3: Simple algorithm to unabbreviate location paths.

oriented manner, the most interesting of which is the predicate. One type of predicate supported is the binary expression, made up of two unary expressions separated by an operator—one of either *or*, or *and*. The unary expression is simply a wrapper for a location path. Thus a binary expression is a way of evaluating two location paths. These predicates can be used for existence-checking, as shown in the expression below.

```
/child::customers/child::customer[descendant::curracc or descendant::creditard]
```

The second location step has an *or* predicate, used to check if the *customer* element has either a *curracc* or *creditcard* descendant. If either of these descendants are feasible the predicate will evaluate to true. To evaluate predicates, a recursive evaluation is run on each location path in the expression. In the expression above the two location paths in the predicate are relative, not absolute. So that the evaluator knows that the context node is *customer*, the axis specifier and node test are prefixed as a separate location step. Thus, the two expressions evaluated would be:

```
child::customer/descendant::curracc
child::customer/descendant::creditcard
```

This leads to the obvious question ‘How does the evaluator know the context node of this new location path?’ The answer is that it doesn’t matter. If the *customer* element can contain a *curracc* descendant under any context node, then it can also contain it as the child of the *customers* element. This is due to the *customer* element being defined only once in the XML schema, and so its possible content will be the same regardless of its parent. The problem of finding the context node for a relative location path is dealt with in the following section.

4.4 Evaluation

This section describes the actual process of evaluation used in the program, including finding the context node for relative location paths and implementing the axes in terms of the *firstchild* and *nextsibling* primitives discussed in section 3.5.

4.4.1 Finding the context node

If the location path passed to the program is relative, the context node needs to be discovered before evaluation can take place. In fact, there may be more than one possible context node when evaluating against the document type. Given the first step in the location path, the context node could be any node that could contain the step, e.g. if the first step is `descendant::a`, the context node could be any node that allows *a* elements as their descendants.

The program needs to find all possible starting points for the evaluation. To focus briefly on element nodes, the first step is to find all occurrences of the node in the finite-state automata. Each automaton needs to be searched through fully to see if any edges are labelled with the element name. Each time an edge labelled with the element’s name is found in a finite-state automata the edge is added to a list. Once all the automata have been checked, this list contains all possible occurrences of the element in the document tree, i.e. valid starting points for the evaluation. From here possible context nodes could be found. For many XPath expressions, it may be satisfactory to find a starting point without knowing the actual context node. For example, the system detailed here supports only forward axes, and so once a starting point is found,

the document tree can be followed from there. The context node is important however when evaluating reverse axes, as the context node will show which nodes appear in a document tree before the first step. For the purposes of this system the actual identity of the context node is not important, and it remains anonymous. As long as a relative location path is evaluated successfully it shows the the expression is valid according to *some point* in the XML schema. If the user wishes to test for a specific context node they can make the location path absolute.

Of course, some relative location paths are essentially the same as some absolute location paths. There is little difference between these two expressions:

```
/descendant::element/attribute::type
```

and

```
child::element/attribute::type
```

While the top expression is absolute, it is merely saying 'The root has below it some descendant element named *element* with an attribute *type*.' The second, relative, expression is saying 'The context node has a child named *element*, with an attribute *type*.' The difference between the two to the program is that the evaluation of the first expression will start off at the root and find a path to *element*, while the evaluation of the second will start from each of the points that *element* can be at in a document, without knowing its relationship with the root node (cf. section 5.3).

A location path need not start with an element. The program also supports attributes, in which case it searches through the element table looking for elements that can contain such an attribute node. The program only supports literal names in node tests, but it is possible for a node test to include wildcards (* selects all element and attribute nodes, *node()* selects all nodes). In this case, the number of starting points could be huge—if the location path was absolute and started with `descendant::node()` then any node in the the document tree would be a feasible starting point. Note also that XPath has tests for processing instructions, comments, and text nodes. Without the actual XML document, the former two tests always evaluate to true, since a processing instruction and a comment can appear anywhere in a document as long as the syntax is valid. The latter test for text nodes needs to check an element's content model, and is not supported in this implementation (see section 4.6).

4.4.2 Location steps, axes, and primitives

Once the start points have been collected, the program starts an iterative execution, attempting to evaluate the location path from each start point in the automata. If the first location step is validated according to the schema (i.e. it can appear somewhere in the defined document), the program takes the first location step as the context node, and attempts to evaluate the next location step successfully. Once a location step is evaluated, it becomes the context node, and the program moves onto the next location step. If the program gets to the end of the location path with each location step evaluated successfully, then the entire path is considered feasible, and a successful result is returned.

The first step in evaluating a location step is to test whether it is valid according to the schema, regardless of the context node. This is analogous to a word processor's spell-checking function. The program checks to see if the element or attribute name in the location step's node test appears somewhere in the schema (similar to looking up a word in a dictionary). If it doesn't there is no need to go any further as it is clear that the expression will fail evaluation since the node cannot actually exist. If the element or attribute does exist in the schema, then the location step's axis specifier is examined, and a function is called depending upon which axis is specified.

Each axis has a corresponding function, which is called as necessary, and is expected to return a node set based upon the axis and the context node. Take the following location path as an example:

```
child::manufacturer/child::address
```

Firstly, all edges labelled with the *manufacturer* element are used as start points. This gives the program the positions in the document where *manufacturer* can be the child of a node. To find out if it can have a child named *address*, the function *getChildren* is called, and the finite-state automata equivalent to the *manufacturer* element's content model is passed as an argument. The function moves through the automata, with each edge equivalent to a possible child. Each edge is added to a list, and this list is returned as the node set containing the *manufacturer* element's children. Given an XML schema that contains the fragment shown in figure 4.4 and which results in

```
1  <element name='manufacturer'>
2   <complexType>
3    <sequence>
4     <element ref='eg:companyname' maxOccurs='1'/>
5     <element ref='eg:address' maxOccurs='unbounded'/>
6     <element ref='eg:companycontact' minOccurs='0' maxOccurs='unbounded'/>
7    </sequence>
8   </complexType>
9  </element>
```

Figure 4.4: XML schema fragment defining an element *manufacturer*.

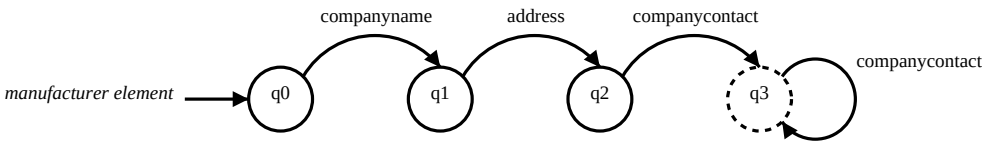

Figure 4.5: Finite-state automaton corresponding to the definition in figure 4.4.

a finite-state automaton as shown in figure 4.5, the function *getChildren* would return the node set [*address*].

4.4.2.1 Axis functions and the *firstchild* and *nextsibling* primitives

The six functions corresponding to the axes are defined in terms of the two primitives, *firstchild* and *nextsibling* as discussed in section 3.5 and shown in table 3.3 on page 24. To recapitulate the discussion, in this abstract evaluation *firstchild* takes a node as an argument and returns a set of nodes that could each occur in a document as the first child of the node. The other primitive, *nextsibling*, takes an automaton state and returns a set of possible next siblings. All the forward axes can be defined in terms of these two primitives. The implementation of these two primitives can be found in appendix B.

Thus, the function corresponding to the *child* axis calls the *firstchild* function, which returns a set of nodes that are possible first child nodes. Then, for each node in

the returned node set, the *nextsibling* function is called. This will result in a node set containing siblings for each of the possible first child nodes. For each node in the result set, possible siblings have to be found for those too. This highly recursive algorithm is run until all possibilities have been followed. The node sets are then unioned together, and a set of possible children are returned as a result set. (Note that the function does not create a document tree, but simply returns a set of possible children. This is done to neuter the recursion problem discussed in section 3.4.3, and is detailed below in section 4.5.) The function to get the children of a node is probably the most complicated axis function. In this implementation, it uses a nested function to achieve the required results as efficiently as possible. The code for the function is shown in figure 4.6 on the next page.

The function to get a node's descendants uses the function that gets a node's children. Each child node that the child axis function returns is passed recursively to the descendant axis function until all possible descendants have been found. The node sets are then unioned together and returned. The other axis functions work in a similar manner, each returning a node set corresponding to the axis they relate to. This node set is then used to evaluate whether the current location step is feasible or not. If the location step is possible, then the returned node set should contain the node's name. If it does, then the location step is successfully evaluated, and the program can make that the context node and attempt to evaluate the following location step. If the evaluation of the step fails, then the location path evaluation itself fails, but *only* according to the start point used in this particular iteration. In this case, the program tries the next possible start point in the automata. If the program runs through all the start points unsuccessfully then the XPath expression is not valid according to the given XML schema. If one particular start point yields a successful evaluation then the program stops. The implementation's main objective is to test whether location path is feasible or not. There may be multiple ways to successfully evaluate the location path, but once the program has found one, it exits successfully.

```
1  def getChildren(self, elementNamePair):
2   def getSiblings(self, fsmNode):
3    possSiblings = []
4    siblings = self.nextSibling(fsmNode)
5    for sibling in siblings:
6     possSiblings.append(sibling)
7     nextSiblings = getSiblings(self, sibling[1])
8     for nextSibling in nextSiblings:
9      if not nextSibling in possSiblings:
10      possSiblings.append(nextSibling)
11   return possSiblings
12  elist = []
13  firstChildren = self.firstChild(elementNamePair)
14  if firstChildren != None:
15   for child in firstChildren:
16    if not child in elist:
17     elist.append(child)
18    siblings = getSiblings(self, child[1])
19    for sibling in siblings:
20     if not sibling in elist:
21      elist.append(sibling)
22  return elist
```

Figure 4.6: Uncommented code for the *getChildren* function.

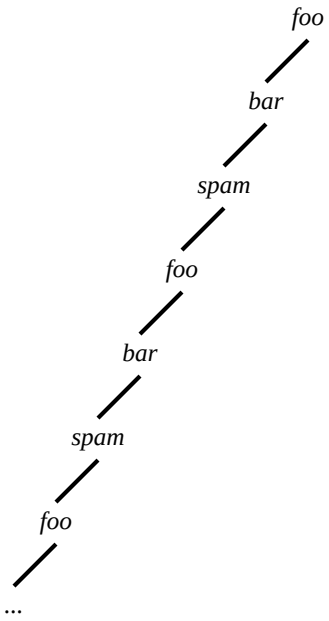

Figure 4.7: Left-recursion problem when creating a document tree. The tree can never be completed because an infinite number of possibilities are followed.

4.5 Removing the problem of recursion

The problem of recursion and infinite loops was mentioned in section 3.4.3. The problem could occur if an element could contain itself *at any depth*. For instance, a declaration for a *foo* element may allow for a *bar* child, which could contain a *spam* child that itself could have a *foo* child. Because of cyclic definition like this, building a document tree becomes an incomputable task as the document tree is extended infinitely as shown in figure 4.7.

The solution to the problem in this implementation is simply not to build a document tree. Instead, a node set of possibilities is used, as described previously. When a function returns a node set of possibilities, it is in the form of a one-dimensional array, not as a tree. Each time a function returns a node set, it is unioned with the current node set, and duplicates are removed. Each node in the set is evaluated one-by-one according to the needs of the axis, and its results are unioned with the node set, again with duplicates removed. The evaluation of each node continues, until each node in

the set has been evaluated. Once the node set has been fully evaluated, the set will contain one instance of each node that is contained within the axis. Because each node is unique, it is only evaluated once. Thus in the example above, the *foo* element will not be added a second time, and so the recursion problem will be solved.

Since this implementation only supports forward axes, this solution works very well. However, it will need to be augmented if it is to work with reverse axes also. The node set is one-dimensional, rather then being a tree. Because of this, the level that a given node is below the context node cannot be discerned. As an example, take the location path:

```
child::foo/descendant-or-self:foo
```

This location path is guaranteed to evaluate to true, since in the second location step the context node *foo* will always match the *self* axis. The returned node set would be [*foo*]. This tells the program that the location step is valid, but it doesn't tell it *how* it is valid. The location step might match to any of the document trees as shown in figure 4.8 on the next page. For the reverse nodes to work there will need to be a record of the path taken to get to the current position. This will need to be added to the solution before the reverse nodes can be implemented. It could be done by adding a list of possible routes to each node. This of course would reintroduce the recursion problem, and so perhaps some sort of sub-tree matching and reuse could be used.

4.6 Simple types

The discussion so far has centred on XML Schema's complex type, and has somewhat neglected its fellow type definition, the simple type. This section will attempt to redress the balance a little, by including a brief discussion of simple types. While more modest than their complex counterparts, the name is slightly misleading—simple types are not necessarily simple. An element defined with a simple type can indeed only contain text, however the text can be of a certain type. XML Schema contains definitions for numerous data types including strings, integers, Booleans, and times and dates, as well as allowing custom types to be defined within schema instances. Content can be

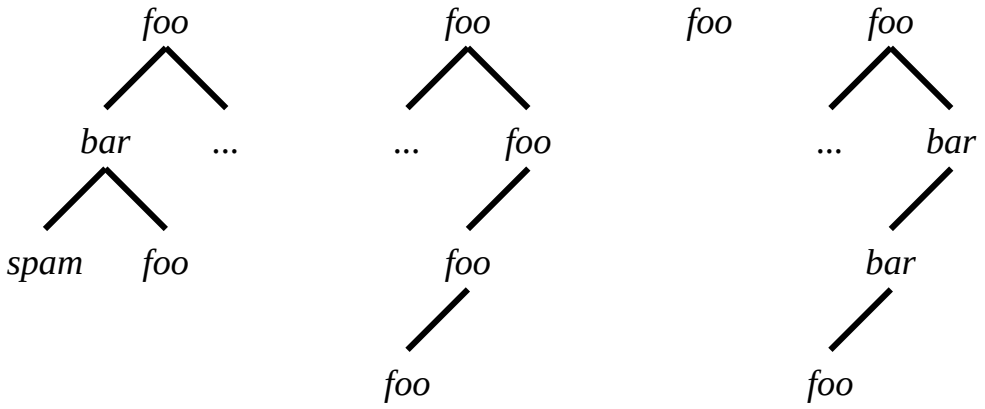

Figure 4.8: Examples of trees that could match the node set returned by the location path `child::foo/descendant-or-self::foo`.

restricted by adding facets to data types, and by requiring the data to match defined patterns.

Even with the capabilities that simple types add to XML Schema, they have no need for finite-state automata. They cannot contain attributes or other elements, and so there is simply no use for an automaton. This lack of an automaton means that simple types are regarded as leaf nodes in this implementation, whereas in an XML document tree the content of the type would sit below it as the leaf node. If the program comes upon a node defined using a simple type within a location path, it must be at the end for the evaluation to be successful (this would obviously change if the reverse axes were implemented). The XPath recommendation allows for the content of simple types to be tested (e.g. only selecting elements whose content is an integer), although this is not supported in this implementation due to the time constraints.

Attributes Note that attributes are always defined as simple types, which is why they do not need finite-state automata. An element can take an already defined attribute and augment it for its own uses, and therefore the program holds attribute declarations within an element declaration to ensure that an augmented attribute does not get mistaken for its base type.

Chapter 5

Analysis

This chapter covers a discussion of the working program. It includes examples of the program in action and tests to show the success and limitations of its location path evaluation. Future work is also discussed, including practical and speculative suggestions for further implementations.

5.1 Command-line interface

The program runs as a command-line utility, with all the advantages and disadvantages that this implies. While it is reasonably cryptic in its syntax and not particularly user-friendly, it can be used in conjunction with other programs. The program outputs a success or failure message for the user, but it also indicates the result of the evaluation with its exit code. An exit code of -1 indicates an unsuccessful evaluation; 0 indicates the evaluation did not take place; and 1 indicates successful evaluation. These exit codes can be used by other programs to manipulate the program and do evaluations of there own, using it as a slave. Examples of user input and program output can be found in appendix C.

In its current form the program is of little use in command-line piping constructions. The output is fairly useless in such a scheme, being just a human-readable success or failure string. For it to be of any use in a command pipe, the program should output possible paths the XPath expression could take through the theoretical XML

document.

5.2 Testing the program

As proof that the evaluator works in line with the XPath recommendation, this section contains tests for each of the supported axes. It also discusses the evaluation of location steps that include predicates, and some problems that were found with the implementation. Due to the terse—and quite frankly dull—program output it is not included in this section; the results of each test are given however.

Throughout this section only one schema is used to demonstrate the tests, and is listed in appendix D. The program was tested on multiple schemas during development, but it was felt for ease of reading only one was needed to demonstrate these tests. The schema uses a single namespace, *http://example.org/* which is synonymous with the prefix *eg* used in these tests. Two further details to note is that throughout the tests the *doc* element from the *http://example.org/* namespace is used as the document root element, and that the attributes defined in the schema have no namespace.

5.2.1 *Self* axis

The *self* axis is the simplest of all axes, because it contains only the context node. The program only matches name literals in a location step's node test, and so the evaluation here is simply to check whether the namespace prefix and the node name given in the location step match the namespace and node name of the context node. Using the program to evaluate the first two expressions in table 5.1, the first evaluates to true, while the latter evaluates to false. Note the unexpected result using the third expression. The *unreferenced* element is included in the schema (lines 202–208) to show up this bug. This element cannot be part of a document when the document root element is *doc*, as it is defined as sitting above *doc* in the document tree an is referenced by no other element. This bug occurs due to a mishandling of the first location step in a relative location paths (using any axis) and is covered in section 5.3.

Table 5.1: Results from evaluating the *self* axis.

Location path	Expected result	Actual result
`/eg:doc/self::eg:doc`	True	True
`descendant::eg:list/self::eg:item`	False	False
`self::eg:unreferenced`	False	True

5.2.2 *Child* axis

As mentioned in section 4.4.2.1, the *child* axis is the most complicated axis supported by the program. Although the *descendant* axis is seemingly more complex, it can be defined in terms of *child*, so it is essential the that axis works as intended. The results laid out in table 5.2 use the abbreviated form of location paths when using the *child* axis.

Table 5.2: Results from evaluating the *child* axis.

Location path	Expected result	Actual result
`/eg:doc/eg:body/eg:div/eg:list/eg:item`	True	True
`/eg:doc/eg:image`	False	False
`eg:unreferenced/eg:note`	False	False
`eg:code/eg:code`	True	True

5.2.3 *Descendant* axis

The *descendant* axis is the most interesting of the supported axes since it is the one that would suffer from the left-recursion problem discussed in sections 3.4.3 and 4.5. Table 3.1 on page 18 suggests that *descendant* axis should be implemented using the *firstchild* and *nextsibling* primitives. However, it can be simplified slightly by using the *child* axis function and the *nextsibling* instead.

The results shown in table 5.3 prove that the recursion problem has been solved in this implementation, purely on the basis that results are always returned. In the

schema used, both the *code* and *link* elements are defined so they can contain instances of themselves. The program handles these recursive definitions comfortably.

Table 5.3: Results from evaluating the *descendant* axis.

Location path	Expected result	Actual result
`eg:code/descendant::eg:code`	True	True
`eg:link/descendant::eg:link`	True	True
`/descendant::eg:image`	True	True
`eg:image/descendant::doc`	False	False

5.2.4 *Descendant-or-self* axis

The *descendant-or-self* axis is implemented by unioning the results of the *descendant* and *self* axes together. As such, if these two axes work the by definition so will the *descendant-or-self* axis. Compare the first two results in table 5.4: the first evaluates true against the context node (using the *self* axis), while the second fails because the *doc* element cannot contain itself at any level. The third result is essentially the same as the third result in table 5.3.

Table 5.4: Results from evaluating the *descendant-or-self* axis.

Location path	Expected result	Actual result
`eg:doc/descendant-or-self::eg:doc`	True	True
`eg:doc/descendant::eg:doc`	False	False
`/descendant-or-self::eg:image`	True	True

5.2.5 *Following-sibling* axis

The *following-sibling* axis is implemented by following all edges from a single state in a finite-state automaton. The state indicates the position of the context node, and each edge that can be followed from that state is the equivalent of one or more possible following siblings (an edge can be labelled with more than one node). Once these

edges have been collected and unioned together, the program has a list of possible following siblings. The program can then follow the general algorithm discussed in section 4.4 to evaluate the location step. This essentially consists of testing whether the node name and namespace given in the step's node test is in the set of edges, and results of such evaluations are shown in table 5.5. Note results three and four, which show how you can step through each child of the context node in turn.

Table 5.5: Results from evaluating the *following-sibling* axis. The ↘ symbol indicates the location path continues on the subsequent line.

Location path	**Expected result**	**Actual result**
`/eg:doc/eg:head/following-sibling::eg:body`	True	True
`eg:head/following-sibling::eg:body/` ↘		
`following-sibling::eg:body`	False	False
`eg:title/following-sibling::eg:div/` ↘		
`following-sibling::eg:p`	True	True
`/descendant::eg:p/child::eg:emph/` ↘		
`following-sibling::eg:code`	True	True

5.2.6 Attributes and predicates

Attributes are discussed here with predicates because it is there that they are the most interesting. Attributes can of course be part of a location path outside of its predicates, but they will always be leaf nodes. It is relatively simple to do an existence check on an attribute given the context node's attribute declarations.

Used inside a predicate binary expression however, they add an extra factor to location paths. For example, the following expression only includes *foo* elements if they include either a *type* or a *part* attribute:

```
foo[@type or @part]
```

(equivalent to)

```
child::foo[attribute::type or attribute::part]
```

As discussed in section 4.3, the program handles predicates by extracting them individually from the location step, reformatting them as one or more location paths which include the context node, and evaluating them as location paths in their own right. The single predicate above has two location paths inside it, separated by the *or* operator. Tables 5.6 and 5.7 show the steps involved in evaluating a location path with predicates. The former table successfully evaluates the path with an *or* predicate, while the latter unsuccessfully evaluates an *and* predicate. The subscript numbers used with the truth values are merely to differentiate between particular true and false values, and to show where each step is used in the evaluation.

Table 5.6: Steps in the successful evaluation of the location path `/eg:doc/descendant::eg:list[@term-width or @part]`.

Location path fragment	**Result**
`/child::eg:doc/descendant::eg:list`	True_1
`descendant::eg:list/attribute::term-width`	True_2
`descendant::eg:list/attribute::part`	False_1
`[`True_2 `or` False_1`]`	True_3
True_1 `[`True_3`]`	True (overall result)

Table 5.7: Steps in the unsuccessful evaluation of the location path `/eg:doc/descendant::eg:list[@term-width and @part]`.

Location path fragment	**Result**
`/child::eg:doc/descendant::eg:list`	True_1
`descendant::eg:list/attribute::term-width`	True_2
`descendant::eg:list/attribute::part`	False_1
`[`True_2 `or` False_1`]`	False_2
True_1 `[`False_2`]`	False (overall result)

The process is the same for a unary predicate, except only one nested location path is required to evaluate it. Additionally, an unlimited number of predicates* can be

*The XPath recommendation specifies an unlimited number of predicates, although in practice it would be hard to actually support an infinite predicate set due to memory constraints, etc.

appended to each location step, and they will be evaluated in the same manner, with each location step requiring the additional number of processes necessary to evaluate each predicate. Note also that as predicates can be any type of location step, they can also take a form such as

```
child::foo[descendant::image]
```

so the node set will only include *foo* nodes that have *image* descendants.

5.3 Bugs in the implementation

The tests described above brought to light two problems with the evaluator in its current form, each of which means in certain circumstances the evaluator would produce erroneous results. The first bug occurs due to the mishandling of relative location paths, and the second involves the handling of predicates.

When the location path passed to the program is relative, it does not attempt to create a path from the document root to the context node. The context node is, to all intents and purposes, ignored. As long as there is an element or attribute in the XML schema that corresponds to the node test in the first location step, that step is evaluated successfully and the context node is assumed to sit above that node at a level allowed by the axis (see section 4.4.1). This works well in most circumstances, with the exceptions.

The first is when starting with an element that only appears above the selected document root element. Take the schema is appendix D: the *unreferenced* element has been added to the original schema deliberately to illustrate this. The *unreferenced* element can only appear in a document above the *doc* element, i.e. it can only contain *doc* elements and it is not referenced in any other element's content model. Therefore if you run an evaluation on the schema using an location path that references the *unreferenced* element, and use *doc* as the document root element, you would expect the evaluation to fail. And indeed, using the expression

```
/descendant-or-self::eg:unreferenced
```

the evaluation does fail. However, if this expression is made relative and expressed as

```
descendant-or-self::eg:unreferenced
```

the evaluation is successful. Admittedly this problem is only likely to occur in exceptional circumstances—as mentioned previously, most schemas are implicitly designed to include only one possible document root element—but any flaw in the program will reduce a user's confidence in its capabilities and is simply inexcusable. The short-term solution is to make relative location paths that could suffer this problem into absolute paths, but in the long-term this problem should be eradicated entirely. The problems with the first location step in relative location paths can be solved by always finding a path to a valid context node. Just as an absolute location path starts at the root element, so to must a relative path. The system must be expanded so that when evaluating relative location paths the program must first find a path to a context node that allows the first location step to be successfully evaluated.

This bug rears its ugly head once more when a relative location path starts with a location step that uses *following-sibling* as its axis specifier. If a node exists as specified in the step's node test, it is assumed that a context node can be found to fit the axis. This is an erroneous assumption if, for example, the node in question can only appear as the document root element. Again using the schema in appendix D, and taking *doc* as the document root element a location path including

```
following-sibling::eg:doc
```

should always fail. And indeed, using expressions such as

```
/following-sibling::eg:doc
```

and

```
self::eg:doc/descendant::eg:body/following-sibling::eg:doc
```

the evaluation will fail. Unfortunately, if a relative location path starts with

```
following-sibling::eg:doc
```

then the evaluation will succeed (assuming of course that the following location steps are evaluated successfully).

It is important to note that these bugs affect only the first location step in a relative location path, although it is equally important to note that they are unacceptable. Future work should include the removal of these bugs as soon as possible.

5.4 Future work

This system described in chapter 4 is by no means complete, and much work could be done to improve or build upon it. The most obvious of this future work is to provide a full implementation of the XPath language. Currently six of the thirteen axes are supported. Missing are the final forward axes *following* and *namespace*, and the five reverse axes. Support for the two forward axes could be added using the current framework, but this would need to be augmented if it were to include the reverse axes.

As it stands the system does not keep a record of the path it has taken to get to the context node, and in order to evaluate reverse nodes such a record would be necessary. In addition, the system has made use of two primitives, *firstchild* and *nextsibling* to implement the six axes. This has worked well, and it would be a canny move to use this method for the reverse axis. The two primitives only allow movement forward in the document tree and to extend this to reverse movement inverse primitives would be necessary. These inverses, denoted $\mathit{firstchild}^{-1}$ and $\mathit{nextsibling}^{-1}$, would select a node's parent (if any) and previous sibling (if any) respectively. With these additions of path recording and primitive inverses the current system could be made to include all thirteen axes in XPath.

After the axis in a location path comes the node test. This also requires some work to do it full justice. Currently the system takes a node test and evaluates on name literals only without supporting wildcards. While the addition of the *comment()* and *processing-instruction()* functions would be fairly trivial (they can appear anywhere in a document, and so abstractly should always evaluate as true), others such as *node()* and * would add much complexity to the current algorithm. These complications could be eased somewhat if a system of recording paths taken, as that recommended for reverse axes, were used so backtracking were possible. This would allow the algorithm to attempt paths based on a given element and back out if it was eventually found not to be possible. Support for wildcards would go some way to implementing the *any* and *anyAttribute* elements of XML Schema, that allow for any element or attribute to be used, even if they are undefined in that particular schema.

Axes and node tests are of main concern to location paths, which are but one type of XPath expression (albeit the most prominent type). Further work should include sup-

porting the other types of expression such as functions and mathematical operations. Although some expressions are of minor use when involved in an abstract evaluation of XPath, it would be useful to implement them all. Some support is included for *or* and *and* expressions, and other expressions such as multiplication, subtraction, addition, etc. could use the binary expression system already in place. Other expressions, like functions, would need the algorithms to be extended specifically to include them. Each time an extra expression is supported, the capabilities of predicates would be extended also. And once all expression types were supported along with the measures laid out above the system could provide a full implementation of XPath.

5.4.1 A new technique

The technique discussed in chapter 4 uses one-dimensional node sets rather than document trees to evaluate location steps. The advantage of this is that it removes problems with left-recursion, but it carries the disadvantage of making it hard to keep track of which path has been taken while attempting an evaluation. The previous section proposes augmenting the current system with the capability to record the path taken. This section will propose a new way of evaluating XPath expressions. This method came about during the final stages of implementing the current system, when the possible limitations became apparent.

This theory proposes using document trees in some way, but modelling them as ordered cyclic graphs. Figure 5.1 on the next page models the schema in appendix D as an ordered cyclic graph, as usual with the *doc* element serving as the root element. (For ease of reading the figure does not include attributes or leaf nodes.) Note the use of sub-trees that are referenced by more than one element. The proposal is that the finite-state automata are used to traverse the schema, but the cyclic tree is used to keep track of the path taken while evaluating an expression. Recursion can be avoided by keeping a flag at each node that is set once the evaluator has passed through it. The flag can then be used to tell the evaluator whether it has been through the path already, and thus avoiding recursion. Using cyclic graphs rather than node sets makes it much easier both to keep track of the path taken by the evaluator, and to allow for back-tracking so wildcards can be supported. This theory is in an early stage so it is not certain to

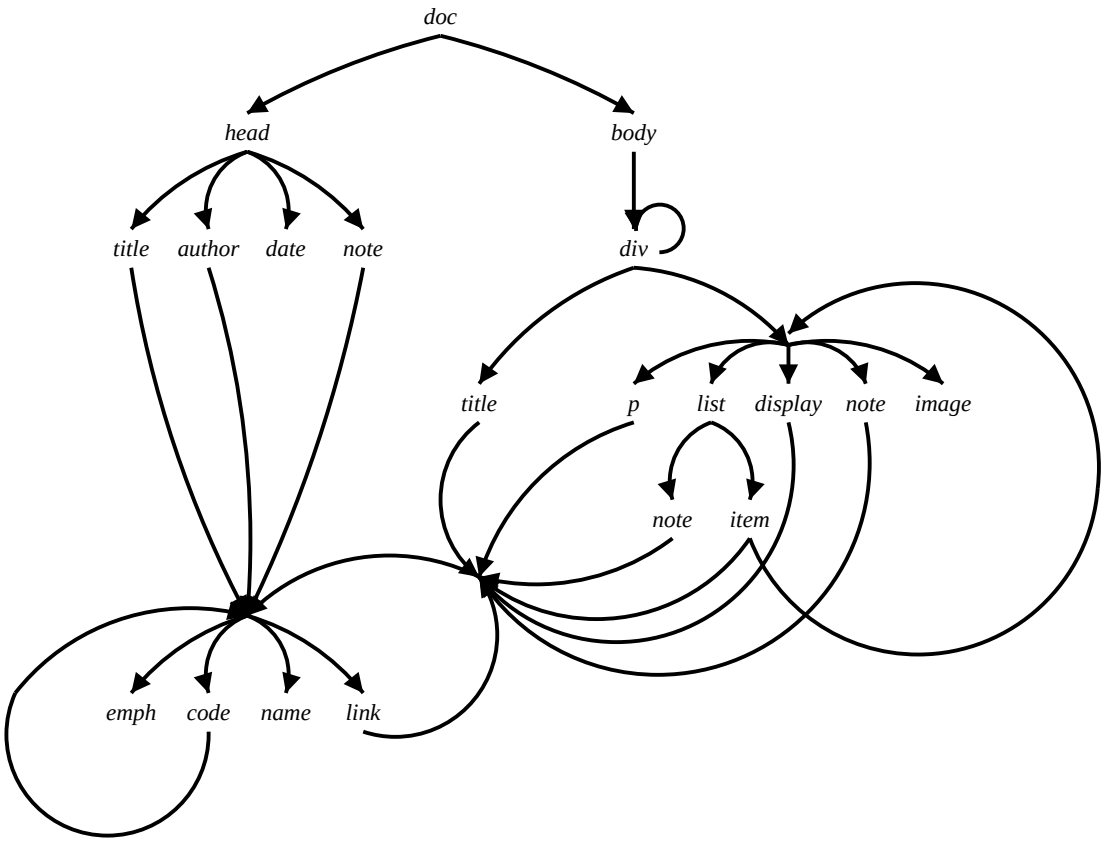

Figure 5.1: An ordered cyclic graph representing an XML schema.

be better than an node set–based implementation, but any future work should consider using such a system.

A final issue to include as future work is multiple evaluations. Currently the system exits successfully as soon as it finds a path through the schema that allows it to evaluate the XPath expression as true. A nice addition would be to continue until all possible paths have been found. This would allow the program to expand its output from a truth value to a set of all valid paths. The program could then interact with other programs (e.g. in command-line pipes) in a much more valuable way.

Chapter 6

Summary

This chapter looks at the conclusions that can be made from the preceding material. The hypothesis used as the basis of this work was given in the first chapter, and stated that introducing an intermediate step before an XPath expression is evaluated against an XML document instance would be a sagacious measure. This step would provide an abstract evaluation of the XPath expression against the document type—an XML schema—to determine whether a concrete evaluation against the XML document could, in principle, be satisfied.

In support of the hypothesis this thesis set out the method involved in converting any given XML schema into a form that could be used to evaluate an XPath expression. XSV, an XML Schema validation tool from the University of Edinburgh, was used to convert XML schemas into a collection of finite-state automata that represented the content models of the elements and attributes defined in a schema. The XPath expression can then be modelled as an input sentence for the automata. The input sentence is made up of symbols, in this case location steps, that are used to progress through the automata. Once the end of the input sentence has been reached, the XPath expression can be said to have been evaluated successfully, and therefore in principle could return a non-empty node set when evaluated against the XML document itself. If the end of the input sentence cannot be reached then the evaluation has failed and has thus proved that further evaluation against the XML document would be pointless.

The project was successful in providing a partial implementation of the most important type of XPath expression, the location path. XPath defines this construct as

having three components parts, the axis specifier, the node test, and a (possibly empty) set of predicates. The first of these contains one of thirteen axes, six of which have been implemented here. These six are all but two of the eight forward axes, i.e. those that allow movement downwards through the document tree. The remaining unsupported axes are those that allow movement upwards through the tree. By focusing on the forward axes the project could use a system of one-dimensional node sets in place of a document tree to evaluate the location paths. This system meant that possible problems with left-recursion in trees was avoided.

Node tests are supported when using node name literals. This allows the XPath expression to evaluate against specific elements and attributes, but not against wildcard structures. Multiple namespaces are fully supported in node tests by using corresponding prefixes passed by the user. Predicates are supported only in part, mainly due to the complexity of their structure. A predicate can contain any type of XPath expression, meaning that XPath expressions have a recursive design. A method to evaluate expressions recursively was devised, and certain additional expression types were implemented for use in predicates. Currently there is support for predicates that contain location paths, binary *or* expressions, binary *and* expressions, and unary expressions for existence-checking.

As is apparent from the previous paragraphs, a full XPath implementation was not created. Although this was the original intention of the project the time-span did not allow for it. XPath is a very complex language, and this was underestimated at the start of the project. In order to complete the XPath evaluator, the reverse axes are needed, along with wildcards in node tests, and support for the expression types other than location paths. It would be possible to augment the current system with the necessary additions to complete the implementation, the most important being to include a method of recording the path taken through the document tree in order to evaluate the expression. However a more interesting and possibly more efficient process would be to use ordered cyclic graphs as well as finite-state automata to keep track of the path taken through evaluation. It is suggested that any further work looks at expanding the system to include such graphs.

The system has shown that the evaluation of XPath expressions against document

types rather than the document itself is a feasible proposition. By evaluating expressions in such a way it is possible to decide whether the evaluation against the document is a worthwhile proposition or not. In doing so, this sort of evaluation can be used to filter out pointless and impossible evaluations before they take place. Inserting this sort of evaluation as an intermediate step before a concrete evaluation took place would be of some advantage to an XPath evaluator—providing efficiency gains for example. Such a measure should be considered worthwhile, which leaves the hypothesis satisfied.

Appendix A

XPath Backus-Naur form grammar

This appendix lists the Backus-Naur form grammar that is used to formally define the XPath language in the W3C's recommendation [11]. The grammar is spread throughout the recommendation, and so it is reproduced here for ease of reading. Some parts of the grammar (e.g. `QName` and `S`) are from the XML recommendation [7] and are not repeated here.

```
1  LocationPath          ::= RelativeLocationPath
2                           | AbsoluteLocationPath
3
4  AbsoluteLocationPath  ::= '/' RelativeLocationPath?
5                           | AbbrAbsoluteLocPath
6
7  RelativeLocationPath  ::= Step
8                           | RelativeLocationPath '/' Step
9                           | AbbrRelativeLocPath
10
11 Step                  ::= AxisSpecifier NodeTest Predicate*
12                          | AbbreviatedStep
13
14 AxisSpecifier         ::= AxisName '::'
15                          | AbbrAxisSpecifier
16
```

```
AxisName             ::= 'ancestor'
                         | 'ancestor-or-self'
                         | 'attribute'
                         | 'child'
                         | 'descendant'
                         | 'descendant-or-self'
                         | 'following'
                         | 'following-sibling'
                         | 'namespace'
                         | 'parent'
                         | 'preceding'
                         | 'preceding-sibling'
                         | 'self'

NodeTest             ::= NameTest
                         | NodeType '(' ')'
                         | 'processing-instruction' '(' Literal ')'

Predicate            ::= '[' PredicateExpr ']'

PredicateExpr        ::= Expr

AbbrAbsoluteLocPath  ::= '//' RelativeLocationPath

AbbrRelativeLocPath  ::= RelativeLocationPath '//' Step

AbbreviatedStep      ::= '.'
                         | '..'

AbbrAxisSpecifier    ::= '@'?

Expr                 ::=  OrExpr

```

```
PrimaryExpr          ::= VariableReference
                         | '(' Expr ')'
                         | Literal
                         | Number
                         | FunctionCall

FunctionCall         ::= FunctionName '(' ( Argument
                             ( ',' Argument )* )? ')'

Argument             ::= Expr

UnionExpr            ::= PathExpr
                         | UnionExpr '|' PathExpr

PathExpr             ::= LocationPath
                         | FilterExpr
                         | FilterExpr '/' RelativeLocationPath
                         | FilterExpr '//' RelativeLocationPath

FilterExpr           ::= PrimaryExpr
                         | FilterExpr Predicate

OrExpr               ::= AndExpr
                         | OrExpr 'or' AndExpr

AndExpr              ::= EqualityExpr
                         | AndExpr 'and' EqualityExpr

EqualityExpr         ::= RelationalExpr
                         | EqualityExpr '=' RelationalExpr
                         | EqualityExpr '!=' RelationalExpr

RelationalExpr       ::= AdditiveExpr
```

```
83                          | RelationalExpr '<' AdditiveExpr
84                          | RelationalExpr '>' AdditiveExpr
85                          | RelationalExpr '<=' AdditiveExpr
86                          | RelationalExpr '>=' AdditiveExpr
87
88  AdditiveExpr        ::= MultiplicativeExpr
89                          | AdditiveExpr '+' MultiplicativeExpr
90                          | AdditiveExpr '-' MultiplicativeExpr
91
92  MultiplicativeExpr  ::= UnaryExpr
93                          | MultiplicativeExpr MultiplyOperator
94                              UnaryExpr
95                          | MultiplicativeExpr 'div' UnaryExpr
96                          | MultiplicativeExpr 'mod' UnaryExpr
97
98  UnaryExpr           ::= UnionExpr
99                          | '-' UnaryExpr
100
101 ExprToken           ::= '(' | ')' | '[' | ']' | '.' | '..' |
102                             '@' | ',' | '::'
103                         | NameTest
104                         | NodeType
105                         | Operator
106                         | FunctionName
107                         | AxisName
108                         | Literal
109                         | Number
110                         | VariableReference
111
112 Literal             ::= '"' [^"]* '"'
113                         | "'" [^']* "'"
114
115 Number              ::= Digits ('.' Digits?)?
```

```
116                         | '.' Digits
117
118 Digits              ::= [0-9]+
119
120 Operator            ::= OperatorName
121                         | MultiplyOperator
122                         | '/' | '//' | '|' | '+' | '-' | '=' |
123                             '!=' | '<' | '<=' | '>' | '>='
124
125 OperatorName        ::= 'and' | 'or' | 'mod' | 'div'
126
127 MultiplyOperator    ::= '*'
128
129 FunctionName        ::= QName - NodeType
130
131 VariableReference   ::= '$' QName
132
133 NameTest            ::= '*'
134                         | NCName ':' '*'
135                         | QName
136
137 NodeType            ::= 'comment'
138                         | 'text'
139                         | 'processing-instruction'
140                         | 'node'
141
142 ExprWhitespace      ::= S
```

Appendix B

Implementation of the two primitives *firstchild* and *nextsibling*

This appendix lists the Python code used to implement the two functions corresponding to the *firstchild* and *nextsibling* primitives. These primitives are used to define the six supported XPath axes, and are discussed further in sections 3.5 on page 23 and 4.4.2.1 on page 34.

```
1  def firstChild(self, nodeName):
2   possibilities = []
3   if nodeName[1] == "/":
4    # If the node is the root then return a special node indicating the
5    # first (and only) child is the document root element.
6    return [[self.documentRoot.getName(), "*documentRoot"]]
7   if nodeName == "*documentRoot":
8    # Get the document root element's FSM.
9    fsm = \
10    self.elementTable.getElement(self.documentRoot.getName()).getFSM()
11  else:
12   # Otherwise get the element in question's FSM.
13   elementName = self.elementTable.getElement(nodeName)
14   if elementName:
15    fsm = elementName.getFSM()
```

```
16     else:
17      fsm = None
18    if fsm != None:
19     # If there is a finite-state machine (i.e. the element has a
20     # complex type), then look through it for possible first children.
21     startNode = fsm.startNode
22     for edge in startNode.edges:
23      newItem = [edge.label[0].pair, edge.dest]
24      if newItem not in possibilities:
25       possibilities.append(newItem)
26     return possibilities
27
28   def nextSibling(self, fsmNode):
29    siblings = []
30    if fsmNode == "/":
31     # Root cannot have any siblings...
32     return []
33    if fsmNode == "*documentRoot":
34     # ...neither can the document root...
35     return []
36    if fsmNode == "attribute":
37     # ...or attribute nodes.
38     return []
39    if fsmNode != None:
40     if len(fsmNode.edges) > 0:
41      # For each edge leaving this FSM node...
42      for edge in fsmNode.edges:
43       if edge.dest != fsmNode:
44        # ...add a new sibling.
45        newItem = [edge.label[0].pair, edge.dest]
46        if newItem not in siblings:
47         siblings.append(newItem)
48    return siblings
```

Appendix C

The program from the command-line

This appendix serves to show some examples of the user input and program output as it is used from the command-line.

C.1 Successful evaluation

User input

```
axe –e /t:doc/t:body/descendant::t:list[@type] –s htdtd.xsd
–r t:doc –n t=http://example.org/
```

Program output

```
Match found for the XPath expression in the schema.
```

C.2 Unsuccessful evaluation

User input

```
axe –e following-sibling::t:bdoy –s htdtd.xsd –r t:doc
–n t=http://example.org/
```

Program output

```
No match found for the XPath expression in the schema.
```

Appendix D

XML Schema used in testing

The schema below was used to carry out the tests detailed in section 5.2. The schema was converted from a DTD supplied by the project supervisor, Dr. Henry Thompson. Note the addition of a definition for an element named *unreferenced*. This element is used in chapter 5 to show up a bug in the program.

```
1  <?xml version='1.0'?>
2  <schema xmlns:eg='http://example.org/'
3          targetNamespace='http://example.org/'
4          xmlns='http://www.w3.org/2001/XMLSchema'>
5
6   <element name='doc'>
7    <complexType>
8     <sequence>
9      <element ref='eg:head' minOccurs='0' maxOccurs='1'/>
10     <element ref='eg:body'/>
11    </sequence>
12   </complexType>
13  </element>
14
15  <element name='head'>
16   <complexType>
```

```
17      <sequence>
18       <element ref='eg:title'/>
19       <element ref='eg:author' maxOccurs='unbounded'/>
20       <choice minOccurs='0' maxOccurs='unbounded'>
21        <element ref='eg:date'/>
22        <element ref='eg:note'/>
23       </choice>
24      </sequence>
25     </complexType>
26    </element>
27
28    <element name='title'>
29     <complexType mixed='true'>
30      <choice minOccurs='0' maxOccurs='unbounded'>
31       <element ref='eg:emph'/>
32       <element ref='eg:code'/>
33       <element ref='eg:name'/>
34       <element ref='eg:link'/>
35      </choice>
36     </complexType>
37    </element>
38
39    <element name='author'>
40     <complexType mixed='true'>
41      <choice minOccurs='0' maxOccurs='unbounded'>
42       <element ref='eg:emph'/>
43       <element ref='eg:code'/>
44       <element ref='eg:name'/>
45       <element ref='eg:link'/>
46      </choice>
47     </complexType>
```

```
48    </element>
49
50    <element name='date'>
51     <complexType mixed='true'>
52     </complexType>
53    </element>
54
55    <element name='body'>
56     <complexType>
57      <sequence>
58       <element ref='eg:div' maxOccurs='unbounded'/>
59      </sequence>
60     </complexType>
61    </element>
62
63    <element name='div'>
64     <complexType>
65      <sequence>
66       <element ref='eg:title'/>
67       <choice maxOccurs='unbounded'>
68        <element ref='eg:div'/>
69        <element ref='eg:p'/>
70        <element ref='eg:list'/>
71        <element ref='eg:display'/>
72        <element ref='eg:note'/>
73        <element ref='eg:image'/>
74       </choice>
75      </sequence>
76     </complexType>
77    </element>
78
```

```
<element name='p'>
 <complexType mixed='true'>
  <choice minOccurs='0' maxOccurs='unbounded'>
   <element ref='eg:emph'/>
   <element ref='eg:code'/>
   <element ref='eg:name'/>
   <element ref='eg:link'/>
  </choice>
 </complexType>
</element>

<element name='display'>
 <complexType mixed='true'>
  <choice minOccurs='0' maxOccurs='unbounded'>
   <element ref='eg:emph'/>
   <element ref='eg:code'/>
   <element ref='eg:name'/>
   <element ref='eg:link'/>
  </choice>
 </complexType>
</element>

<element name='note'>
 <complexType mixed='true'>
  <choice minOccurs='0' maxOccurs='unbounded'>
   <element ref='eg:emph'/>
   <element ref='eg:code'/>
   <element ref='eg:name'/>
   <element ref='eg:link'/>
  </choice>
 </complexType>
```

```
110    </element>
111
112    <element name='list'>
113     <complexType>
114      <choice maxOccurs='unbounded'>
115       <element ref='eg:note'/>
116       <element ref='eg:item'/>
117      </choice>
118      <attribute name='type' use='required'>
119       <simpleType>
120        <restriction base='string'>
121         <enumeration value='normal'/>
122         <enumeration value='enum'/>
123         <enumeration value='defn'/>
124         <enumeration value='naked'/>
125         <enumeration value='tdefn'/>
126        </restriction>
127       </simpleType>
128      </attribute>
129      <attribute name='term-width' type='string' use='required'/>
130      <attribute name='term-align' use='required'>
131       <simpleType>
132        <restriction base='string'>
133         <enumeration value='left'/>
134         <enumeration value='center'/>
135         <enumeration value='right'/>
136         <enumeration value='justify'/>
137         <enumeration value='char'/>
138        </restriction>
139       </simpleType>
140      </attribute>
```

```
141     </complexType>
142    </element>
143
144    <element name='emph'>
145     <complexType mixed='true'>
146      <attribute name='color' type='string' use='optional'/>
147     </complexType>
148    </element>
149
150    <element name='code'>
151     <complexType mixed='true'>
152      <choice minOccurs='0' maxOccurs='unbounded'>
153       <element ref='eg:emph'/>
154       <element ref='eg:code'/>
155       <element ref='eg:name'/>
156       <element ref='eg:link'/>
157      </choice>
158     </complexType>
159    </element>
160
161    <element name='name'>
162     <complexType mixed='true'>
163     </complexType>
164    </element>
165
166    <element name='item'>
167     <complexType mixed='true'>
168      <choice minOccurs='0' maxOccurs='unbounded'>
169       <element ref='eg:p'/>
170       <element ref='eg:list'/>
171       <element ref='eg:display'/>
```

```
172       <element ref='eg:note'/>
173       <element ref='eg:image'/>
174       <element ref='eg:emph'/>
175       <element ref='eg:code'/>
176       <element ref='eg:name'/>
177       <element ref='eg:link'/>
178      </choice>
179      <attribute name='term' type='string' use='optional'/>
180     </complexType>
181    </element>
182
183    <element name='link'>
184     <complexType mixed='true'>
185      <choice minOccurs='0' maxOccurs='unbounded'>
186       <element ref='eg:emph'/>
187       <element ref='eg:code'/>
188       <element ref='eg:name'/>
189       <element ref='eg:link'/>
190      </choice>
191      <attribute name='href' type='string' use='optional'/>
192      <attribute name='name' type='string' use='optional'/>
193     </complexType>
194    </element>
195
196    <element name='image'>
197     <complexType mixed='true'>
198      <attribute name='source' type='string' use='required'/>
199     </complexType>
200    </element>
201
202    <element name='unreferenced'>
```

```
203      <complexType>
204       <sequence>
205        <element ref='eg:doc' maxOccurs='unbounded'/>
206       </sequence>
207      </complexType>
208     </element>
209
210    </schema>
```

Bibliography

[1] A. Aho and J. Ullman. *Principles of Compiler Design*. Addison-Wesley, Reading, Mass., 1977.

[2] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. *VLDB Journal*, 9:53–64, 2000.

[3] M. Arenas, W. Fan, and L. Libkin. What's hard about XML Schema constraints? *Lecture Notes in Computer Science*, 2453:269–278, 2002.

[4] M. Benedikt, W. Fan, and G. M. Kuper. Structural properties of XPath fragments. *Lecture Notes in Computer Science*, 2572:79–95, 2003.

[5] P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes, 2001. http://www.w3.org/TR/xmlschema-2/.

[6] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language, 2002. http://www.w3.org/TR/xquery/.

[7] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition), 2000. http://www.w3.org/TR/2000/REC-xml-20001006.

[8] D. Carlisle, P. Ion, R. Miner, and N. Poppelier. Mathematical Markup Language (MathML) Version 2.0, 2001. http://www.w3.org/TR/MathML2/.

[9] C. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In R. Agrawal, K. Dittrich, and A. H. H.

Ngu, editors, *Proc. of the 18th Int'l Conf. on Data Engineering*, pages 235–244, San Jose, Calif., 2002. IEEE Comput. Soc.

[10] J. Clark. XSL Transformations (XSLT), 1999. http://www.w3.org/TR/xslt.

[11] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0, 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[12] The Unicode Consortium. *The Unicode Standard*. Addison-Wesley Developers Press, Reading, Mass., 2000.

[13] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A Query Language for XML, 2002. http://www.w3.org/TR/NOTE-xml-ql/.

[14] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 6(8):451–455, 1970.

[15] D. C. Fallside. XML Schema Part 0: Primer, 2001. http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/.

[16] J. Ferraiolo. Scalable Vector Graphics (SVG) 1.0 Specification, 2001. http://www.w3.org/TR/SVG/.

[17] International Organization for Standardization. ISO 8879:1986(E). Information processing – Text and office systems – Standard Generalized Markup Language (SGML), 1986. Geneva, Switzerland.

[18] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proc. of the 28th Int'l Conf. on Very Large Databases*, pages 95–106, Hong Kong, China, 2002. Morgan Kaufmann Publishers.

[19] G. Gottlob, C. Koch, and R. Pichler. XPath processing in a nutshell. *SIGMOD Record*, 32(1):12–19, 2003.

[20] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. *Lecture Notes in Computer Science*, 2572:173–189, 2003.

[21] S. Holzner. *Inside XML*. New Riders, Indianapolis, Ind, 2001.

[22] H. Hosoya and B. C. Pierce. XDuce: a typed XML processing language. In *Int'l Workshop on the Web and Databases*, Dallas, Tex., 2000.

[23] D. Jurafsky and J. H. Martin. *Speech and Language Processing*. Prentice-Hall, Upper Saddle River, NJ., 2000.

[24] S. C. Kleene. Representation of events in nerve nets and finite automata. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ., 1956.

[25] M. Lutz and D. Ascher. *Learning Python*. O'Reilly & Associates, Sebastopol, Calif., 1999.

[26] W. S. McCulloch and W. Pitts. A logical calculus of ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, pages 115–133, 1943. Reprinted in *Neurocomputing: Foundations of Research*, ed. by J. A. Anderson and E. Rosenfeld, MIT Press 1988.

[27] F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31(3):39–46, 2002.

[28] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

[29] H. S. Thompson. W3C XML Pointer, XML Base and XML Linking, 2003. http://www.w3.org/XML/Linking.

[30] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures, 2001. http://www.w3.org/TR/xmlschema-1/.

[31] H. S. Thompson and R. Tobin. Current status of XSV: Coverage, known bugs, etc., 2003. http://www.ltg.ed.ac.uk/~ht/xsv-status.html.

[32] H. S. Thompson and R. Tobin. Using finite state automata to implement W3C XML Schema content model validation and restriction checking. At the time of

writing this was unpublished. It was submitted for XML Europe 2003, and seen in June, 2003.

[33] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. of the London Mathematical Society*, 42:230–265, 1936. A correction was published in the following volume, pp. 544–546.