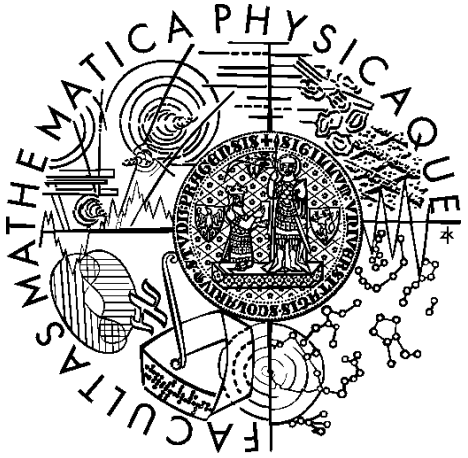
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Dušan Rychnovský

Generating of Synthetic XML Data

Department of Software Engineering

Supervisor of the master thesis: RNDr. Irena Holubová, Ph.D.

Study programme: Informatics

Specialization: Software Engineering

Prague 2013

I would like to express my gratitude to my supervisor RNDr. Irena Holubová, Ph.D., for her helpful advices, guidance and corrections.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Generování syntetických XML dat

Autor: Dušan Rychnovský

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Irena Holubová, Ph.D.

Abstrakt: V této diplomové práci jsme se zaměřili na vývoj algoritmu pro generování syntetických XML dokumentů vhodných k použití jako testovací případy pokrývající daný XPath dotaz. Značný důraz byl kladen na jednoduchost konfigurace. Součástí byla také implementace prototypu, optimalizace jeho výkonu a demonstrace jeho chování nad ukázkovými daty.

Klíčová slova: XML, DTD, XPath, generování dat, syntetické dokumenty

Title: Generating of Synthetic XML Data

Author: Dušan Rychnovský

Department: Department of Software Engineering

Supervisor: RNDr. Irena Holubová, Ph.D.

Abstract: The aim of the thesis was to develop an algorithm to generate synthetic XML documents to cover the resolution of an XPath query with test cases, with a special emphasis on the ease of configuration. We have also implemented a prototype, optimized its performance and demonstrated its properties by running it on sample data.

Keywords: XML, DTD, XPath, data generating, synthetic documents

Contents

Chapter 1

Introduction

Today, XML is widely used as a technology for representation, storage and exchange of structured data. With the increasing emphasis on automated software testing, the need for generators of suitable synthetic XML documents as test cases rises. Many different tools are already available for this purpose. From the perspective of users, two requirements seem especially important - the ability to adjust the generated documents to conform to the expectations of the target applications and the ease of configuration. Unfortunately, these two aspects contradict with each other. The aim of the thesis is to find a suitable compromise and come up with an algorithm that would allow to customize the results in detail in a user-friendly way.

Designing a general-purpose generator which could be adapted for the needs of applications of any sort and yet would be easy to work with is very hard, if not impossible. To reach our goal, we decided to limit our focus on applications which use XPath queries. XPath is a popular language for extracting data represented in the form of XML documents. It is simple yet powerful and forms the basis of more complex technologies, such as XQuery and XSLT. The following are examples of applications that rely on it.

- XML databases,
- applications configured via XML documents,
- applications exchanging messages encoded as XML documents.

For these cases, to expect an XPath query on the input of our algorithm seems like a promising idea. We expect to be able to extract a lot of information about the way the resulting documents should look like from it. Moreover, target users will typically already have a query which they need to get covered by testing documents and thus providing it to the generator as a configuration argument should not constitute an obstacle.

1.1 Structure of the Thesis

The thesis is structured as follows.

- Chapter 1 introduces the goal of the thesis and describes its structure.

- Chapter 2 defines some basic terms, which are used throughout the rest of the thesis. It also briefly introduces the XML, XML Schema and XPath technologies.
- Chapter 3 compares existing generators, with a special emphasis on parameters that they accept.
- Chapter 4 forms the main part of the thesis. This is where we describe the conclusions of our theoretical research and propose the solution.
- In Chapter 5, some important implementation details about the prototype are specified.
- Chapter 6 contains the experimental part of the thesis. It includes results of running the prototype on sample data and introduces performance optimizations.
- Chapter 7 contains the conclusion and suggestions for future work.

Chapter 2

Used Technologies and Definitions

In this chapter we will first give a few basic terms which will be used in definitions later in the thesis to keep them clear and concise and then briefly introduce the family of XML technologies.

2.1 Basic Definitions

2.1.1 Graphs

Definition 1. *Suppose we have a directed graph* $G = (V, E)$ *and its node* $v \in V$. *Then*

- $Edges(v) = \{e \in E; v \in e\}$ *is the set of all edges* incident *with* v,
- $ChildNodes(v) = \{w \in V; (v, w) \in E\}$ *is the set of all* child nodes *of* v,
- $GrandChildNodes(v) = \{u \in V; \exists w \in V : (v, w) \in E \wedge (w, u) \in E\}$ *is the set of all* grand-child nodes *of* v.

Moreover, when $|Edges(v)| = 1$, *i.e.* $ChildNodes(v) = \{u\}$ *for some node* $u \in V$, *then*

- $ChildNode(v) = u$ *is the only child node of* v.

2.1.2 Sequences

Definition 2. *Let* X *be a finite set of symbols and* $w = (x_1, x_2, ..., x_n)$ *a sequence in* X. *We call* x_i *a* member *of* w *for each* $i \in \{1...n\}$. *We denote the fact that* $x \in X$ *is a member of* w *as* $x \in w$.

Definition 3. *Let* X *be a finite set of symbols and* $v_1 = (x_1, ..., x_n)$ *and* $v_2 = (y_1, ..., y_m)$ *two sequences in* X. *We say that* $v = v_1 \odot v_2$ *is a* concatenation *of* v_1 *and* v_2 *if and only if* $v = (x_1, ..., x_n, y_1, ..., y_m)$.

Definition 4. *Let* X *be a finite set of symbols and* V_1 *and* V_2 *two sets of sequences in* X. *Then we call* $V = V_1 \odot V_2$ *a* concatenation *of* V_1 *and* V_2 *if and only if* $V = \{v_1 \odot v_2; v_1 \in V_1, v_2 \in V_2\}$.

2.2 XML Technologies

XML is a set of technologies maintained by the W3C Consortium [12] which includes languages such as XML, DTD, XPath, XML Schema, XSLT and other. In the rest of this chapter we will briefly describe those that are related to the aim of the thesis and which we will therefore refer to in the rest of it. We will only focus on those features of individual technologies that are important with regards to our purpose. Describing these technologies in detail is beyond the scope of the thesis.

2.3 XML

XML (Extensible Markup Language, [13]) is a technology designed for data storage and exchange. It is a markup language which means that XML documents are basically plain text documents augmented by certain marks that provide the structure and indicate the meaning of individual parts.

Let us show an example. Figure 2.1 contains a simple email message encoded as an XML document. Such a message has an address of the sender and the receiver, a subject and the main body and might optionally contain one or more attachments.

```
1  <?xml version="1.0" ?>
2  <message>
3    <sender>bob@gmail.com</sender>
4    <receiver>josh@gmail.com</receiver>
5    <subject>Hello from Thailand!</subject>
6    <body>
7      Hi Josh!
8
9      I am greeting you from the wonderful land of Thailand! We
10     are having a really great time here. I'm sending you some
11     photos as attachments so that you can see the beautiful
12     country yourself.
13
14     Looking forward to seeing you again,
15     Bob
16   </body>
17   <attachment file="photo-01.png" />
18   <attachment file="photo-02.png" />
19 </message>
```

Figure 2.1: The message XML document

2.3.1 Elements

The marks in an XML document are called *tags*. Every tag has a name and always starts with an opening (<) and ends with a closing (>) parenthesis. Tags always occur in pairs and each pair together denotes an element.

An *element* is a part of the document between (and including) an opening tag and a closing tag. Both tags must have the same name and the closing one must contain a slash (/). Each element has a name, which is the same as the name of

its tags, and a content, which is anything that is placed between its opening and closing tags.

Figure 2.2 contains a sample XML element. The name of the element is *sender* and its content is *bob@gmail.com*.

```
1  <sender>bob@gmail.com</sender>
```

Figure 2.2: The sender element

Based on the content type, XML elements can be divided into four groups:

Empty content

These are elements that do not contain any characters between their tags. A sample empty content-like element is depicted on Figure 2.3.

```
1  <married></married>
```

Figure 2.3: An element with empty content

The XML definition allows an abbreviated syntax for this kind of elements - the closing tag is omitted and the opening one has a slash appended before the closing parenthesis. Figure 2.4 shows an equivalent notation for the *married* element.

```
1  <married />
```

Figure 2.4: An element with empty content in the abbreviated form

Element content

An element of this type contains a sequence of other elements, so called *sub-elements*. It does not directly contain any non-element characters.

Figure 2.5 contains a sample element with element content. Note that only the *name* element has element content here - the *firstname* and *surname* elements have text contents.

```
1  <name>
2    <firstname>John</firstname>
3    <surname>Sinclair</surname>
4  </name>
```

Figure 2.5: An element with element content

Text content

Elements with text content contain strings of characters directly, i.e. not within sub-elements. A sample element with text content is depicted in Figure 2.2 above.

Mixed content

Elements with mixed content contain both sub-elements and non-element text characters, intermixed with each other. Element *p* in Figure 2.6 has mixed content type.

```
1    <p>
2      <b>Touching</b> is an important form of communication among
3      <i>elephants</i>. Individuals greet each other by
4      <b>stroking</b> or <b>wrapping their trunks</b>; the latter
5      also occurs during <b>mild competition</b>.
6    </p>
```

Figure 2.6: An element with mixed content

We can observe that the sample email message XML document in Figure 2.1 contains seven different elements - one (named *message*) with element content, four (named *sender*, *receiver*, *subject*, *body*) with text contents and two (both named *attachment*) with empty contents.

2.3.2 Attributes

Each element might optionally contain one or more *attributes*. An attribute consists of a name and a value and forms basically a key-value pair. Figure 2.7 shows a sample element with two attributes - it depicts a t-shirt of size XL and with blue colour.

```
1    <t-shirt colour="blue" size="XL" />
```

Figure 2.7: An element with two attributes

In the XML example in Figure 2.1 only the *attachment* elements have attributes - a single attribute each - with $file$ as names and $photo-01.png$ and $photo-02.png$ as values.

2.3.3 Prolog

Every XML document starts with a prolog. It is the first line of the document with a special syntax and contains the following attributes.

- **version** (mandatory) - denotes the version of the XML specification that the document conforms to,
- **encoding** (optional) - indicates the file encoding (UTF-8 by default).

Figure 2.8 shows a sample prolog of an XML file. It prescribes the 1.0 specification version and the $UTF-8$ encoding.

```
1    <?xml version="1.0" encoding="UTF-8" ?>
```

Figure 2.8: An example of an XML prolog

2.3.4 Well-Formed XML Documents

Every XML document must satisfy a few conditions in order to be syntactically correct. We call such documents *well-formed*. A list of these conditions follows.

1. Each XML document must start with a prolog.
2. Every element must be delimited by a start and end tag pair (except for abbreviated empty content elements, see Section 2.3.1) with matching names (in a case sensitive way).
3. Elements must not overlap, i.e. the pairs of tags must not cross.
4. The whole document must be enclosed in a single element, so called *root element*.

The sample XML document in Figure 2.1 meets all the criteria and is therefore well formed.

2.3.5 Common XML Data-Model

For the sake of simplicity we usually use trees to represent XML documents in memory for further processing. Such a tree has elements of the document as nodes and there are edges between nodes if and only if they represent an element - sub-element relationship. Additionally, each node has a (possibly empty) set of element attributes and a content.

Figure 2.9 contains a tree that represents the XML document from Figure 2.1. Nodes are represented by ellipses where the corresponding element name is written in bold face and the attached content and attributes are depicted below using the standard font. Edges are displayed using arrows.

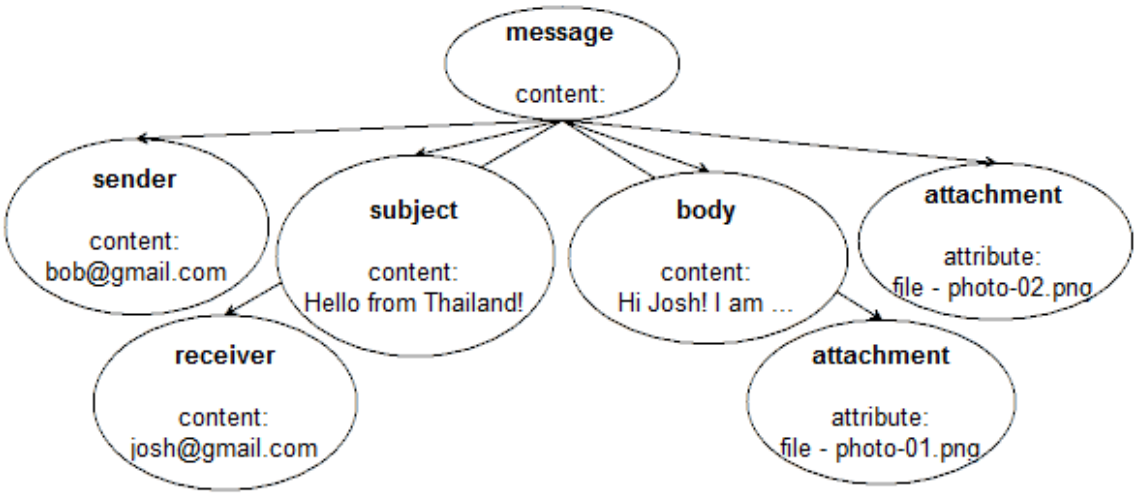


Figure 2.9: The message XML file data model

Definition 5. *The* depth *of an XML document doc (denoted as Depth(doc)) is the number of edges on its longest path from the root node to a leaf node in its data-model graph.*

Definition 6. *The* size *of an XML document doc (denoted as Size(doc)) is the total number of nodes in its data-model graph.*

2.4 DTD

The set of all well-formed documents is very broad and we often need to constraint it in order to get a (more or less) narrow set of documents convenient for describing a certain domain. Many languages were designed for this purpose. We will use DTD in this thesis.

DTD (Document Type Definition, [13]) is a language that allows to describe the structure and content of XML documents using regular expressions. It is used to denote sets of documents which conform to the descriptions. We say that such documents are *valid*.

Definition 7. *The set of all documents valid against a schema S is denoted as* $Valid(S)$.

Figure 2.10 shows a sample DTD schema. It describes a set of XML documents to represent XML messages. For one of the set of all valid documents see Figure 2.1.

```
1  <!ELEMENT message (sender,receiver,subject,body,attachment*)>
2  <!ELEMENT sender (#PCDATA)>
3  <!ELEMENT receiver (#PCDATA)>
4  <!ELEMENT subject (#PCDATA)>
5  <!ELEMENT body (#PCDATA)>
6  <!ELEMENT attachment EMPTY>
7  <!ATTLIST attachment file CDATA #REQUIRED>
```

Figure 2.10: A sample DTD definition

We will postpone a more detailed description of the DTD language to Chapter 4.

2.5 XPath

XPath (XML Path Language, [14]) is a language that allows to select specific parts of XML documents. To refer to the particular document parts it defines a dedicated query language. Such a query describes how to navigate through the document to locate them.

The sample XPath query given in Figure 2.11 could be used to reference the *subject* element of the document in Figure 2.1.

```
1 /message/subject
```

Figure 2.11: A sample XPath query

We will give a more in-depth description of the XPath language in Chapter 4.

Chapter 3

Existing Approaches

This chapter presents an overview of existing approaches to generating synthetic XML documents. We are primarily interested in parameters that individual generators accept to affect the resulting documents. There are generators that can be given:

- a schema,
- a query,
- other parameters.

3.1 Key Representatives

In this section we will introduce selected key representatives.

3.1.1 Partition-based Approach

In [1], C. de la Riva, J. García-Fanjul and J. Tuya present a solution based on the category-partition method for generating testing data, which was proposed in [2]. The intention is to generate a set of test cases in accordance with a given query and then discard those that do not conform to a given schema. Only some very basic XPath syntax elements are supported - the child, self, parent, and descendant-or-self axes plus elementary predicates.

The authors also outline two different possible implementation techniques - using model checking (described in [3]) and using ToXgene (in [4]). They do not, however, offer a full and ready implementation.

3.1.2 Schema-based Partitioning

Another approach inspired by the category-partition method is introduced in [5]. Unlike in the previous article, this time a given schema instance alone is used to extract test-cases. The authors also propose applying weights to individual parts of the schema to reduce the size of the results. In order to make the generation more flexible, the solution also allows users to decide if they want to get a fixed number of resulting documents or a fixed functional coverage (or a compromise between both).

3.1.3 Oxygen

A different representative of approaches that generate XML documents using a schema instance is the *Generate sample XML files* tool in Oxygen [6]. In addition to a schema, it also accepts several other parameters, such as (among others):

- name of the root element,
- expected total number of documents,
- preferred number of element repetitions,
- maximum recursion level,
- maximum text content length.

3.1.4 ToXGene

The ToXGene generator [7] accepts a configuration file, which is basically an XML schema definition augmented with special annotations. These are mainly probability distributions that influence the numbers of occurrences of elements and attributes in the generated documents and also the text contents. The tool supports some more advanced annotations, which can be used to enforce additional integrity constraints (such as IDs and IDREFs).

ToXGene allows users to adjust the generated documents very closely, but at the cost of a difficult and time-demanding configuration.

3.1.5 Synthetic Benchmark

In [8], C. Dyreson and H. Jin propose a generator of XML documents and then use it to compare several different in-memory and persistent engines.

The generated documents are synthetic, which means that the aim was to choose those that would highlight the impact of different factors to querying efficiency as much as possible, rather than to mimic real documents with reasonable semantics. This effectively means that all elements at the same level of depth have the same name, which allows to construct queries that descend into a certain depth of the documents. Moreover, some elements (so called magic elements) can have a fixed suffix of *Magic* appended to their names. By changing the amount of magic elements in the documents one can change the size of the result of queries which locate them. We do not get to know anything about naming of attributes or generating text content.

The authors propose a group of factors to control the structure and content of the generated documents, such as (among others):

- document depth,
- the number of children per element,
- the number of attributes per element,
- the length of the text content,
- tree density (i.e. the percentage of nodes relative to the number of nodes in a complete tree),

- the level with magic nodes,
- the frequency of magic nodes at the specified level.

To reduce the number of the resulting documents, the authors only vary each single control factor while keeping all others fixed. This also allows them to isolate the impact of this single factor and determine how it affects the performance of query evaluation in various implementations.

3.1.6 XPathMark

In [9], M. Franceschet proposes an XPath benchmark to compare the functional completeness, correctness, efficiency and scalability of different XML engines. He uses XMark (see [10]), a tool to generate random XML documents which adhere to a single fixed schema, to create input data. He also designs a fixed set of XPath queries intended to cover all the various aspects of the language. The queries are designed with the schema of the input documents in mind. The author also proposes a methodology to evaluate XPathMark on a given engine and demonstrates the idea on two well known representatives.

As XMark is used to generate the documents, the only parameter that can be affected is the expected size.

3.2 Summary

The algorithms and tools that we have mentioned in the previous sections can be compared from many different perspectives. Following are those that are relevant with regards to the aim of the thesis.

Type

We can divide the approaches into two basic groups - tools (Oxygen, ToXGene, XPathMark) and papers (Partition-based Approach, Schema-based Partitioning, Synthetic Benchmark). The former one contains well defined algorithms for which there are robust implementations available. The latter one is formed by scientific papers in which the proposed methods are only outlined. They also mostly contain results of using prototypical implementations for benchmarking.

Configurability and configuration complexity

These two aspects depend on each other inversely. On the one hand there is XPathMark, for which only the size of the generated documents can be affected. ToXGene represents the other extreme - it can be configured in detail, but at the cost of a very demanding set-up and a steep learning curve. The rest of the approaches are in between.

Completeness

Many of the described representatives, especially those that belong to the group of scientific papers, do not deal with the full range of the task. The authors have

typically chosen just a sub-set of the syntax offered by the underlying technologies (XML, XPath, etc). Details have already been mentioned.

Table 3.1 shows an overview of the types of parameters expected by individual solutions.

	Query	**Schema**	**Other**
Partition-based Approach	Yes	Yes	No
Schema-based Partitioning	No	Yes	Yes
Oxygen	No	Yes	Yes
ToXGene	No	Yes	Yes
Synthetic Benchmark	No	No	Yes
XPathMark	No	No	Yes

Table 3.1: Key Representatives

The area of generating documents specifically for the purpose of covering the resolution of XPath queries has not been properly explored yet. Particularly close to our aim is Partition-based Approach. However, this paper offers just a basic idea, rather than a complete algorithm.

Chapter 4

Our Approach

The aim of the thesis is to find an algorithm to generate synthetic documents for the purpose of testing XML applications. Apparently, the set of all well-formed XML documents is too large (infinitely large). Moreover, we are usually only interested in testing applications on documents with certain structure and content. Hence, the algorithm must return a reasonably small set of suitable documents. It is therefore necessary to find a convenient set of constraints that will narrow the set of generated documents. We have decided to focus on the following types of constraints:

- constraints induced by the schema,
- constraints induced by the query,
- other constraints.

4.1 DTD

There are a number of languages available that can be used to describe the schema of a set of documents. We have chosen DTD as it has been found to be the most commonly used one (see [11]).

Let us suppose that we are given a schema defined in the DTD syntax on the input of the algorithm. What information can we extract from the schema? How can we characterize the set of all valid documents? What constraints does the schema induce on the documents?

4.1.1 Syntax of the DTD Language

Let us show the relevant snippets of the DTD grammar as defined in the specification by W3C (see [13]). Features that are beyond the scope of this thesis (such as XML entities) were omitted for clarity.

As we can see in Figure 4.1, a DTD schema is a sequence of element and attribute-list declarations.

```
1  <DTD_SCHEMA> ::= (<ELEMENT_DECL> | <ATTLIST_DECL>)*
```

Figure 4.1: DTD grammar - part 1 (the schema)

Figure 4.2 shows that each element declaration consists of the name of the declared element and its content type.

```
1 <ELEMENT_DECL> ::= "<!ELEMENT" <NAME> <CONTENT_SPEC> ">"
2 <CONTENT_SPEC> ::= "EMPTY" | "ANY" | <TEXT> | <MIXED> | <CHILDREN>
3 <TEXT> ::= "(#PCDATA)"
4 <MIXED> ::= "(#PCDATA" ("|" <NAME>)+ ")*"
5 <CHILDREN> ::= (<CHOICE> | <SEQUENCE>) ("?" | "*" | "+")?
6 <CP> ::= ( <NAME> | <CHOICE> | <SEQUENCE> ) ("?" | "*" | "+")?
7 <SEQUENCE> ::= "(" <CP> ("," <CP>)+ ")"
8 <CHOICE> ::= "(" <CP> ("|" <CP>)+ ")"
```

Figure 4.2: DTD grammar - part 2 (element declarations)

Finally, Figure 4.3 shows that an attribute-list declaration contains a reference to an element (its name) and a list of attribute declarations. Each of them consists of the name of an attribute, its content type and its default value declaration.

```
1 <ATTLIST_DECL> ::= "<!ATTLIST" <NAME> <ATT_DECL>* ">"
2 <ATT_DECL> ::= <NAME> <ATT_TYPE> <DEFAULT_DECL>
3 <ATT_TYPE> ::= <STRING_TYPE> | <TOKEN_TYPE> | <ENUM_TYPE>
4 <STRING_TYPE> ::= "CDATA"
5 <TOKEN_TYPE> ::= "ID" | "IDREF" | "IDREFS"
6 <ENUM_TYPE> ::= "(" <NAME> ("|" <NAME>)* ")"
7 <DEFAULT_DECL> ::= "#REQUIRED" | "#IMPLIED" | (("#FIXED")? <VALUE>)
```

Figure 4.3: DTD grammar - part 3 (attribute-list declarations)

Note that an attribute-list declaration is in fact a sequence of attribute declarations for the same element. Also note that a schema always binds each attribute declaration to a specific element, which we will further call the *target (or related) element* of the declared attribute.

Definition 8. *A* DTD schema S *is a pair* (El, At)*, where* El *is the set of all element declarations and* At *is the set of all attribute declarations.*

Definition 9. *For a schema* $S = (El, At)$*, given any attribute declaration* $a \in At$*, let* $Element(a)$ *be the target element declaration.*

4.1.2 Element Declarations

We will now take a closer look at the syntax of element declarations as defined in Figure 4.2.

Definition 10. *For a schema* $S = (El, At)$*, given any element declaration* $e \in El$*, the* set of all valid sub-element sequences for e *is denoted as* $ElSequences(e)$*.*

The content type of an element is in fact a set of valid sub-element sequences. These sets can be written in the schema definition in five different ways.

Children

The content type can be declared as a (simplified) regular expression. In that case, for an element declaration e with an expression r, $ElSequences(e)$ contains any sequence that conforms to r.

Definition 11. *Given a regular expression r we denote* the set of all sequences valid against r *as $Valid(r)$.*

For a regular expression r, the set of all valid sequences $Valid(r)$ can be determined by induction as described in Table 4.1, where a stands for a single element, alpha and beta for sub-expressions and λ for an empty sequence.

The expression r	**The set of valid sequences** $Valid(r)$
a	$\{a\}$
$alpha \vert beta$	$Valid(alpha) \cup Valid(beta)$
$alpha, beta$	$Valid(alpha) \odot Valid(beta)$
$alpha?$	$Valid(alpha) \cup \{\lambda\}$
$alpha+$	$\bigcup_{i=1}^{\infty} \bigodot_{j=1}^{i} Valid(alpha)$
$alpha*$	$\bigcup_{i=0}^{\infty} \bigodot_{j=0}^{i} Valid(alpha)$

Table 4.1: Valid sequences

In accordance with Definition 11 it holds for an element declaration e of this type with an expression r that $ElSequences(e) = Valid(r)$.

Empty

In this case only the empty sequence is valid. I.e. for an element declaration e, $ElSequences(e) = \{\lambda\}$.

Text

As in the previous case the only valid sub-element sequence is the empty sequence.

Mixed

It turns out (see [11]) that in reality this case is used quite rarely, especially for XML documents designed to be processed by a computer without any direct human intervention. We have therefore decided not to consider the mixed content type in our thesis.

Any

This content type does not induce any constraints on the structure of valid documents (the enumeration contains all sub-element sequences) and therefore does not bring in any information. We will ignore it for the rest of the thesis.

4.1.3 The Repetitive Operators

The + and $*$ operators always lead to infinitely large sets of sequences and thus to documents with unlimited sizes. It would therefore seem convenient for our purposes to choose an appropriate constant MC (max cardinality) and understand these operators in the following sense:

- the + operator as 1 to MC,
- the * operator as 0 to MC.

We will accept the value of MC as an input argument of the algorithm. That way users can customize the results according to their needs and potential.

Another option would be to let users set the value separately for each occurrence of these operators. This would allow for a more detailed configuration but at the cost of a more demanding setup by users.

4.1.4 Root Element Declaration

In a DTD schema itself there is no information about which element declaration should correspond to the root element of the generated documents. We could ask the user to specify the name of the root element as an additional parameter. In order to minimize configuration complexity, though, we prefer the following convention - the element declaration that comes first in the DTD definition corresponds to the root element of the generated documents.

Definition 12. *The* root element declaration *of a schema S, denoted as $RootElement(S)$, is the element declaration that comes first in the schema definition.*

4.1.5 Attribute Declarations

Let us now examine attribute declarations as defined in Figure 4.3. In this section we will only look at them with regards to the structure of the generated documents and ignore constraints on the content.

Definition 13. *Suppose we have a schema $S = (El, At)$ and an attribute declaration $a \in At$. We say that a represents a* required *attribute (denoted as $Required(a) = true$), if and only if, for an XML document to be valid against S, a must be specified for all target elements. Otherwise a represents an* optional *attribute (denoted as $Required(a) = false$).*

We will now analyse the default-value part of attribute declarations. There are four possible cases.

Required

As the name implies, attributes declared with the $\#REQUIRED$ flag are required, i.e. have to be specified for all target elements. For an example of a declaration of a required attribute refer to attribute *title* in Figure 4.4.

Implied

Attributes declared as implied (using the $\#IMPLIED$ flag) are optional. Attribute *sex* in Figure 4.4 is an example of an implied attribute.

Default

If an attribute is neither required nor implied, it has to be declared with a default value. It is then considered optional, meaning that an XML document is valid even if the attribute is omitted for one or more target elements. Attribute *lang* in Figure 4.4 falls into this category.

Fixed

The default value in an attribute declaration might be preceded by a $\#FIXED$ flag. Such an attribute is optional (as in the previous case), but, when it is specified for a target element, it can only be assigned the fixed default value. Attribute *read* in Figure 4.4 is a sample fixed attribute.

4.1.6 Attributes with Default Values

If an attribute is declared with a default value, it is considered optional, meaning that an XML document is valid even if it is omitted for one or more target elements. On the other hand, according to the specification [13], when an XML processor encounters an element without the specification of an attribute with a default value, it must behave as if the attribute was in fact specified with the declared default value. This also applies to XPath resolvers. There would be a point in suggesting that we treat default attributes as required in our algorithm, for their presence does not affect the result of applying an XPath query to an XML document. Moreover, it could lead to a more-or-less significant reduction of the number of generated documents. We will still consider these attributes optional, though, as we perceive this as a way to detect some resolvers that do not behave in exact accordance with the specification.

4.1.7 Attribute Sequences

According to W3C [13] the order in which attributes are specified for an element is irrelevant. We can therefore choose the ordering of attributes for the generated XML documents freely. We have decided to use the declaration order. From this perspective it makes sense to think about (ordered) sequences of attributes for individual elements with regards to the final XML documents (instead of unordered sets). When we will talk about sequences of attributes later in this section and further on, we mean sets of attributes sorted in the declaration order.

Definition 14. *For a schema $S = (El, At)$ and an element declaration $e \in El$ let $Attributes(e)$ be a sequence of all attribute declarations of S related to e sorted in the declaration order.*

Definition 15. *For a schema $S = (El, At)$ given any element declaration $e \in El$ we call $AttrSequences(e)$ the* set of all attribute sequences valid *for e.*

For an element declaration e, $AttrSequences(e)$ is defined by induction in the following sense.

1. *Suppose that $|Attributes(e)| = 0$. Then $AttrSequences(e) = \{\lambda\}$.*
2. *Suppose now that $Attributes(e) = (a_1, ..., a_n)$ and let e' be an element such that $Attributes(e') = (a_1, ..., a_{n-1})$. Let $AttrSequences(e') = K$. Then*
   (a) *$AttrSequences(e) = K \odot (a_n)$ if $Required(a_n)$ and*
   (b) *$AttrSequences(e) = K \cup (K \odot (a_n))$ if not $Required(a_n)$.*

An attribute sequence *as* valid for an element e is a set of attributes (each with e as the target element) sorted in the declaration order such that an XML document where only attributes from *as* are specified for e is valid.

4.1.8 Example

Given the sample declaration in Figure 4.4, the following list holds all attribute sequences valid for the *book* element. Note that the attributes in each of them are indeed sorted in the declaration order.

To distinguish attribute and sub-element sequences, we will now and further on prefix attribute names with the @ character in our examples.

- @author_id, @title,
- @author_id, @title, @lang,
- @author_id, @title, @read,
- @author_id, @title, @lang, @read.

4.1.9 Schema Graph

In this section we will define the *schema graph* data structure. We will use it as a convenient in-memory representation of any given DTD schema. The graph will hold information about the set of valid documents in a relatively compact yet easy-to-work-with way.

Definition 16. *For a schema $S = (El, At)$ and an element declaration $e \in El$ let the* set of all content sequences of element *e be $Sequences(e) = AttrSequences(e) \odot ElSequences(e)$.*

Definition 17. *For a schema $S = (El, At)$ and an attribute declaration $a \in At$ let the* set of all content sequences of attribute *a be $Sequences(a) = \{\lambda\}$.*

Attributes are not allowed to have sub-elements (or sub-attributes) and thus, for any attribute declaration, the only valid content sequence is the empty sequence.

Definition 18. *Suppose we are given a schema $S = (El, At)$ in the DTD syntax. The* graph of the schema *G is a quadruple (r, V, W, E) such that*

- *r is a node for which $r \notin V$ and $r \notin W$,*

- *V and W are disjoint sets of nodes where*
  - $V = El \cup At$,
  - $W = \{(RootElement(S))\} \cup \bigcup_{v \in El \cup At} Sequences(v)$,
- *and $E \subseteq (\{r\} \times W) \cup (V \times W) \cup (W \times V)$ is a multi-set of edges such that for all $v \in V$ and $w \in W$ it holds that*
  - $(r, w) \in E \Leftrightarrow w = (RootElement(S))$,
  - $(v, w) \in E \Leftrightarrow w \in Sequences(v)$ *and*
  - $(w, v) \in E \Leftrightarrow v \in w$.

A list of a few more things to observe follows:

- r is an artificial node which does not correspond to any part of any XML document valid against the represented schema. Its purpose is to simplify resolution of XPath expressions, as we will see later. We will call it the *document node*.
- V is the set of all declared elements and attributes. We can therefore call a node $v \in V$ *element node* (if $v \in El$) or *attribute node* (if $v \in At$).
- W is formed by all valid content sequences of all declared elements and attributes. It also contains sequence $(RootElement(S))$. We will call nodes of W *sequence nodes*.
- There is always a single child node of r - the node which corresponds to sequence $(RootElement(S))$.
- For every $w \in W$, $ChildNodes(w)$ is the multi-set of elements/attributes corresponding to all members of sequence w. Note that more than one edge might exist between w and a particular element node as the same sub-element might appear inside a single content-sequence more than once.
- For all $v \in V$ such that $v \in El$, $ChildNodes(v)$ contains nodes that correspond to valid content sequences of element v.
- For all $v \in V$ such that $v \in At$, $ChildNodes(v) = \{\lambda\}$.

Definition 19. *A schema S is* recursive *if the corresponding graph G contains at least one directed cycle. Otherwise it is* non-recursive.

We will use a common technique to display schema graphs in our examples. For a schema graph $G = (r, V, W, E)$ we will draw r using an indented rectangle, nodes of V as ellipses, nodes of W as regular rectangles and edges of E as arrows between incident nodes.

4.1.10 Example 1

Let us show an example. We will use a schema definition as depicted in Figure 4.4. It describes a set of documents which could be used to store information about books in a library. Each document contains a set of authors and a set of books divided into genres.

The related schema graph follows in Figure 4.5. Note that it does not contain any directed cycles and thus the corresponding schema is non-recursive.

```
<!ELEMENT library (authors, genres)>
<!ELEMENT authors (author+)>
<!ELEMENT author EMPTY>
<!ATTLIST author
    id ID #REQUIRED
    sex (male|female) #IMPLIED
    name CDATA #REQUIRED>
<!ELEMENT genres (genre*)>
<!ELEMENT genre (books)>
<!ATTLIST genre name CDATA #REQUIRED>
<!ELEMENT books (book+)>
<!ELEMENT book EMPTY>
<!ATTLIST book
    author_id IDREF #REQUIRED
    title CDATE #REQUIRED
    lang (en|cz) "en"
    read CDATA #FIXED "true">
```

Figure 4.4: The library schema definition

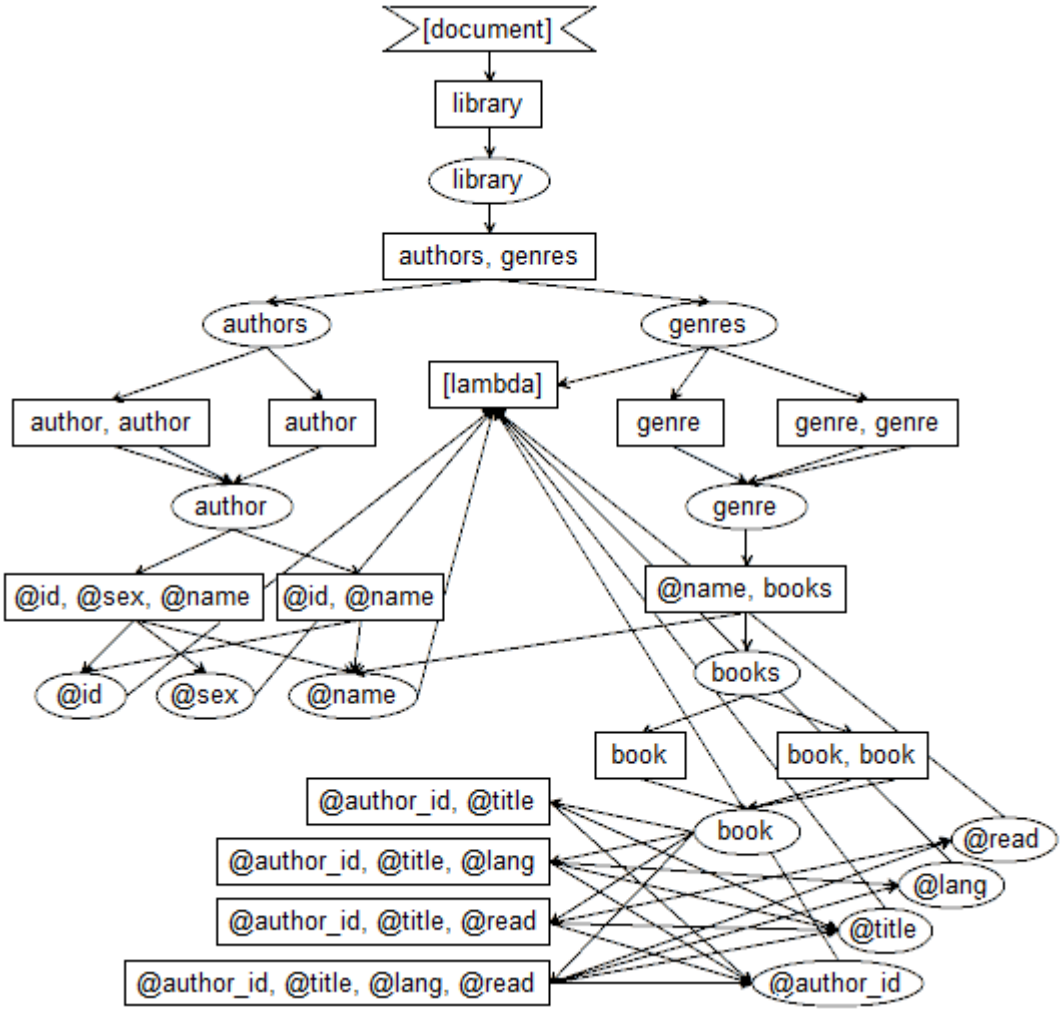


Figure 4.5: The library schema graph

4.1.11 Example 2

We will show one more example to illustrate the concept. The following schema (see Figure 4.6) describes documents which hold information about the structure of a company. Such a company has a CEO (chief executive officer) and one or more departments. Every department has at least one employee and might be further divided into sub-departments.

```
1   <!ELEMENT company (ceo, department+)>
2   <!ELEMENT ceo (name, phone+)>
3   <!ELEMENT department (manager, staff, subdepartments)>
4   <!ELEMENT manager (name, phone)>
5   <!ELEMENT staff (employee+)>
6   <!ELEMENT employee (name, salary)>
7   <!ATTLIST employee lang CDATA 'en'>
8   <!ELEMENT subdepartments (department*)>
9   <!ELEMENT name EMPTY>
10  <!ELEMENT phone EMPTY>
11  <!ELEMENT salary EMPTY>
```

Figure 4.6: The company schema definition

Figure 4.7 displays the corresponding schema graph. This time note that it contains a directed cycle on nodes *department*, *subdepartments*, *department*, which implies that the schema is recursive.

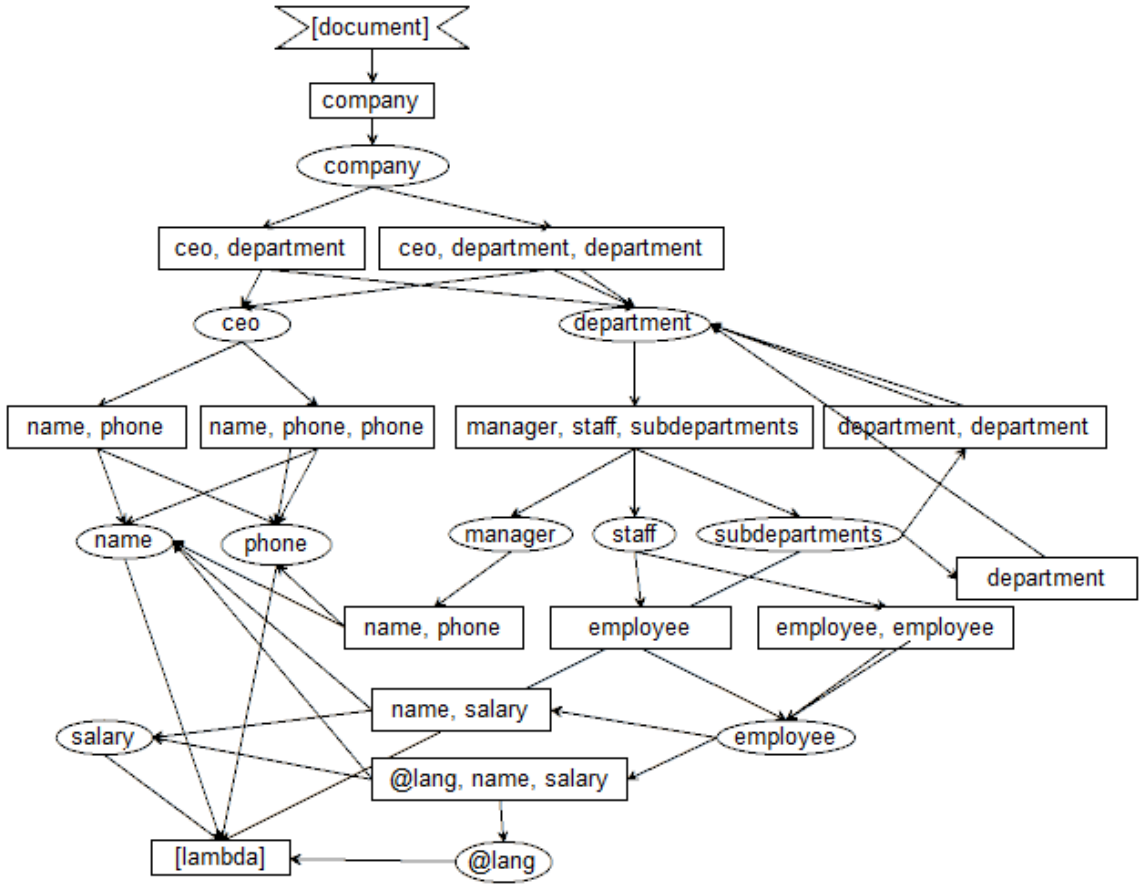


Figure 4.7: The company schema graph

4.1.12 Basic Characteristics of Valid Documents

Let us start with examining non-recursive schemas and show algorithms that can be used to find out characteristics that a schema imposes on valid documents.

The Number of Valid Documents

Suppose we have a non-recursive schema S with a graph $G = (r, V, W, E)$. An algorithm to find the number of all distinct documents which are valid against S (denoted as $Count(S)$) is as follows in Algorithm 1.

1 $\forall w \in W, |Edges(w)| = 0 : Count(w) = 1$
2 $\forall w \in W, |Edges(w)| > 0 : Count(w) = \prod_{u \in ChildNodes(w)} Count(u)$
3 $\forall v \in V : Count(v) = \sum_{u \in ChildNodes(v)} Count(u)$
4 $Count(r) = Count(ChildNode(r))$
5 $Count(S) = Count(r)$

Algorithm 1: The Count function

Minimum Document Depth

Let $S = (El, At)$ be a non-recursive schema with graph $G = (r, V, W, E)$. The *minimum document depth* $MinDepth(S) = min_{doc \in Valid(S)} Depth(doc)$ can be computed in accordance with Algorithm 2.

1 $\forall w \in W, |Edges(w)| = 0 : MinDepth(w) = 0$
2 $\forall w \in W, |Edges(w)| > 0 : MinDepth(w) = \max_{u \in ChildNodes(w)} MinDepth(u)$
3 $\forall v \in V, v \in At : MinDepth(w) = 0$
4 $\forall v \in V, v \in El : MinDepth(v) = \min_{u \in ChildNodes(v)} MinDepth(u) + 1$
5 $MinDepth(r) = MinDepth(ChildNode(r))$
6 $MinDepth(S) = MinDepth(r)$

Algorithm 2: The MinDepth function

Maximum Document Depth

The way to determine the *maximum document depth* for a schema S is analogous to the previous case.

Existence of a Valid Document with the Given Depth

Let us have a non-recursive schema $S = (El, At)$ with graph $G = (r, V, W, E)$ and a natural number d. The description of an algorithm to determine the set of all valid document depths $Depths(S) = \{Depth(doc); doc \in Valid(S)\}$ with regards to S follows in Algorithm 3. A valid document with depth d exists if and only if $d \in Depths(S)$.

1 $\forall w \in W, |Edges(w)| = 0 : Depths(w) = \{0\}$
2 $\forall w \in W, |Edges(w)| > 0 : Depths(w) = \{max_{z \in Z}; Z \in \prod_{u \in ChildNodes(w)} Depths(u)\}$
3 $\forall v \in V, v \in At : Depths(w) = \{0\}$
4 $\forall v \in V, v \in El : D(v) = \bigcup_{u \in ChildNodes(v)} \{d + 1; d \in Depths(u)\}$
5 $\forall v \in V : D(v) = \bigcup_{u \in ChildNodes(v)} \{d + 1; d \in Depths(u)\}$
6 $Depths(r) = Depths(ChildNode(r))$
7 $Depths(S) = Depths(r)$

Algorithm 3: The Depths function

Possible Implementation Techniques

All of these algorithms traverse the schema graph and do the respective calculations on its nodes. We describe them by giving a formulae of calculation for every node. Dynamic algorithms (bottom-to-top traversal) or recursive algorithms (top-to-bottom traversal with intermediate results stored in memory) would be good choices of implementation techniques.

4.1.13 Example

Let us demonstrate the preceding algorithms to find information about the depths of valid documents using a sample schema. We have chosen the schema listed in Figure 4.8 as it is reasonably small to fit on the page and yet complex enough for the demonstration purposes.

The schema describes a small set of documents which could be used to store information about peripherals connected to a computer. There might be a printer at a computer and a headset and the headset, which always contains speakers at least, might also contain a microphone and/or a subwoofer.

```
1  <!ELEMENT computer (printer?, headset?)>
2  <!ELEMENT printer EMPTY>
3  <!ATTLIST printer resolution #REQUIRED>
4  <!ELEMENT headset (speakers, microphone?)>
5  <!ELEMENT speakers (subwoofer?)>
6  <!ELEMENT subwoofer EMPTY>
7  <!ATTLIST subwoofer wattage #IMPLIED>
8  <!ELEMENT microphone EMPTY>
```

Figure 4.8: The computer schema definition

Figure 4.9 depicts the corresponding schema graph.

The small squares with numbers are the result of the algorithm to find the set of depths of valid documents. There is one for each node, as the algorithm prescribes. We could start by evaluating the leaf node and continue in iterations with nodes which have all child-nodes evaluated. Eventually we would get to the root node.

The figure also displays the results of the algorithms to find the minimal (and maximal) document depths. For each node it is the smallest (greatest resp.) number in the square adjusted to it.

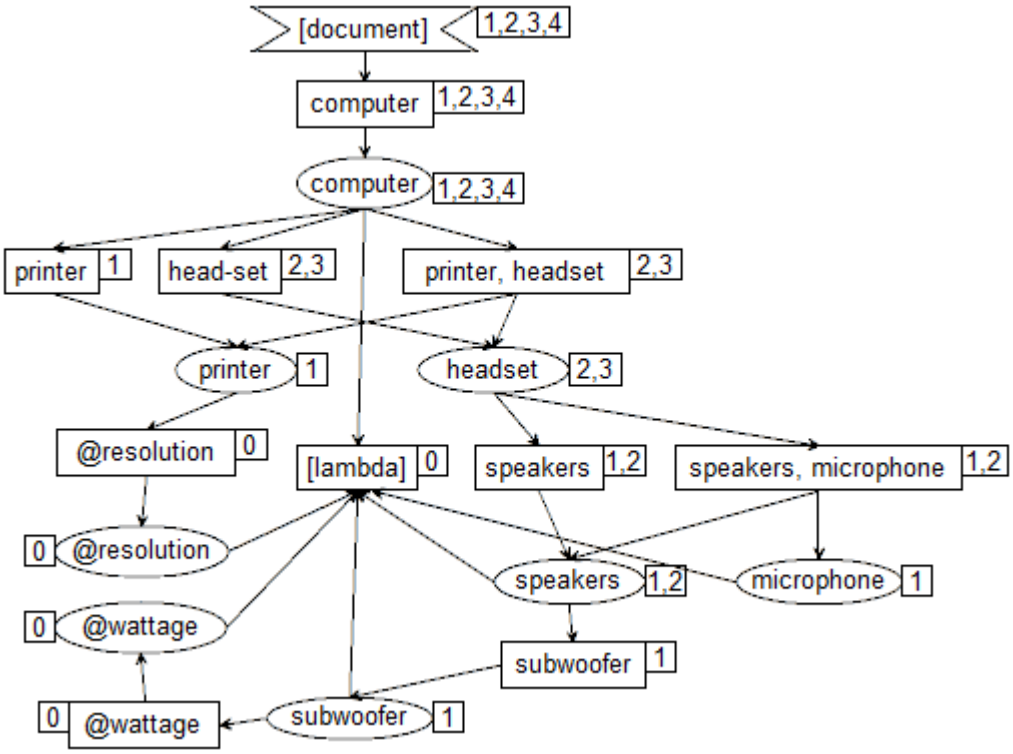


Figure 4.9: The computer schema graph

We can see that there exist valid documents with depths 1, 2, 3 and 4 (see the square at the root node), which implies that the minimal document depth is 1 and the maximal one is 4.

4.1.14 The Recursive Case

As defined in Definition 19, a schema is called recursive if its graph contains at least one cycle. In that case, the set of all valid XML documents is infinitely large and contains, among others, infinitely deep documents. Such a set is too large for our purpose and needs to be constrained.

We could ask the user for the expected minimum and maximum depths of the generated documents and then only take into account documents which satisfy this condition. This should enable us to work with a set of non-recursive schema graphs instead of the original recursive one. We could then apply all of the methods described in this thesis for recursive schemas as well.

The conversion of a recursive schema graph into a set of non-recursive ones would probably be inefficient for all but the simplest instances (at the very least it would have to deal with the set of all cycles of the schema graph, which is hard to even find). However, it turns out (see [11]) that in reality XML documents typically only embody very simple recursion cases. The algorithm should therefore be good enough for most of the real-world situations.

Despite this simplification, we will only deal with non-recursive schemas in this thesis and leave the conversion process as a suggestion for a future work.

4.1.15 Content of Attributes

Let us now look at what features DTD provides to constraint the values of attributes.

Definition 20. *Let $S = (El, At)$ be a schema, $a \in At$ one of its attribute declarations and s an arbitrary text string. We say that s is a* valid value *for a (denoted as $ValidValue(a, s) = true$) if and only if there exists an XML document valid against S in which s stands as the value of an attribute corresponding to a.*

Definition 21. *For a schema $S = (El, At)$ and an attribute declaration $a \in At$ let $ContentType(a) = \{string\ s; ValidValue(a, s)\}$ be the* basic content type *of a.*

The basic content type of an attribute declaration is the set of all values which are valid for attributes that the declaration represents. A DTD schema can additionally induce constraints on the relationships of values of pairs (or sets) of attributes, which these definitions do not reflect. We will deal with them later.

In accordance with the schema snippet in Figure 4.3 there are three ways to declare the basic content type of an attribute.

String

An attribute declared with the string type may take any literal string as value. In other words, $ContentType(a) = \{string\ s\}$ for an attribute declaration a of this type.

Attribute *title* in Figure 4.4 is an example of an attribute that falls into this category.

Enum

The enum type declares a (finite) set of string values that the attribute is permitted to contain. For an attribute declaration a with an enumeration of values $v_1, ..., v_n$, $ContentType(a) = \{v_1, ..., v_n\}$.

For an example see the declaration of attribute *lang* in Figure 4.4. It holds that $ContentType(lang) = \{en, cz\}$.

Fixed

If an attribute is declared with the $\#FIXED$ flag it can only be assigned its default value. Therefore, for an attribute declaration a declared with string s as a fixed default value it holds that $ContentType(a) = \{s\}$.

See attribute *read* in Figure 4.4 as an example of this declaration type. $ContentType(read) = \{true\}$.

Note that in general, attribute values may contain any characters but for <, & (except where part of a valid character or entity reference) and the quote character used around the value in the XML document (i.e. ′ or ″). For the sake of simplicity we have decided to only allow the space and single-quote characters plus all alpha-numerical ones. As a rule, we will use the double-quote character to surround values of attributes in the generated XML documents.

4.1.16 The ID and IDREF Constraints

The last feature that DTD provides to constraint the values of attributes that we will deal with in this thesis is the concept of IDs and IDREFs.

The ID keyword serves as a global uniqueness constraint. For an XML document *doc* to be valid against a schema S, every attribute that appears in *doc* and is declared using the ID keyword in S must be assigned a value which is unique among all other such attributes.

Definition 22. *For a schema S let $IDs(S)$ be the set of all attribute declarations which are defined using the ID keyword.*

While the ID construct could be seen as an equivalent to the primary key concept from the world of relational databases from some point of view, the IDREF construct would be an analogy to foreign keys.

For a document *doc* to be valid against a schema S, each attribute that appears in *doc* and is declared as IDREF in S can only be assigned a value that some other attribute of type ID in *doc* contains.

Definition 23. *For a schema S let $IDREFs(S)$ be the set of all attribute declarations which are defined using the $IDREF$ keyword.*

In addition to IDREF, the DTD language also supports the IDREFS keyword. An attribute declared as IDREFS is in fact just a multi-value IDREF attribute. Handling these is not much different and we will not deal with them in this thesis.

4.1.17 Content of Elements

We can see in Figure 4.2 that the features that DTD provides with regards to constricting the content of elements are just a subset of what it allows for attributes. Basically an element either can or cannot have text content assigned - there is no way to constraint it any further.

We have decided to ignore the text content of elements in this thesis. In all of the generated XML documents no element will have any text content assigned.

Note that the documents will still be valid against the corresponding schema - there is no way you could enforce an element to have a non-empty text content in DTD.

4.2 XPath

Unfortunately the number of all XML documents valid against a DTD schema is still very large. The set of generated documents therefore needs to be further constrained. In this section we will look at how we can narrow it to contain only documents which are useful for testing the evaluation process of an XPath query.

4.2.1 Syntax of the XPath Language

Supporting the full XPath syntax is beyond the scope of the thesis. We will only take into account the child and attribute axes and node tests given as element

names. We will also only support some basic forms of predicates. The (simplified) snippet of the XPath grammar as defined by W3C [14] follows in Figure 4.10.

```
1  <QUERY> ::= <ABS_QUERY> | <REL_QUERY>
2  <ABS_QUERY> ::= "/" <REL_QUERY>
3  <REL_QUERY> ::= <STEP> ("/" <STEP>)*
4  <STEP> ::= <AXIS_SPEC> <NODE_TEST> (<PREDICATE>)*
5  <AXIS_SPEC> ::= (<AXIS_NAME> "::") | <ABBR_AXIS_SPEC>
6  <AXIS_NAME> ::= "child" | "attribute"
7  <ABBR_AXIS_SPEC> ::= "@"?
8  <NODE_TEST> ::= <NAME>
9  <PREDICATE> ::= "[" <EXPR> "]"
10 <EXPR> ::= <QUERY> | <QUERY> "=" <VALUE> | <QUERY> "=" <QUERY>
```

Figure 4.10: XPath grammar

We can observe that an XPath expression can be either absolute or relative. We will only support absolute queries on the input of our algorithm. Both types of queries can be used inside predicates.

We can further see that an XPath expression is composed of a (non-empty) sequence of steps, where each step consists of an axis, a node test and a (possibly empty) sequence of predicates. Note that the specification allows for some abbreviations - the child axis is assumed by default (i.e. if we omit the axis specification for a step) and the @ character represents the attribute axis.

Definition 24. *For an XPath query Q let $Steps(Q)$ denote the* sequence of steps *that Q consists of and let $Step(Q, i)$ be the i-*th step *of Q.*

Definition 25. *For an XPath query Q and its step $S_i \in Steps(Q)$ let $Axis(S_i)$ denote the* axis*, $NodeTest(S_i)$ the* node test *and $Predicates(S_i)$ the* sequence of predicates *of S_i.*

4.2.2 Predicates

Let us now look at what the syntax described in Figure 4.10 offers us with regards to XPath predicates.

Definition 26. *Let P be a predicate that occurs in an XPath query Q. Then $Type(P) \in \{q, qv, qq\}$ is the* type of P *and $Queries(P)$ is the* sequence of XPath queries *that P consists of sorted in the declaration order.*

In situations when the predicate consists of just one sub-query we will use the following short-cut to refer to it. The intention is to make our definitions less verbose.

Definition 27. *Let P be a predicate and Q' the only sub-query that P consists of. Then $Query(P) = Q'$ denotes the* sub-query of predicate P.

Definition 28. *Let Q be an XPath query and P one of its predicates of type qv. $Value(P)$ is the* literal value *that appears in P.*

We will only deal with the following types of predicates. In all three cases both absolute and relative sub-queries are supported.

Query

In the simplest case a predicate might be formed by just a single sub-query. See predicate $[finished]$ in Figure 4.11 for an example.

For a predicate P of this type formed by a sub-query Q' it holds that $Type(P) = q$ and $Query(P) = Q'$.

Query equals value

This type represents predicates which compare a sub-query to a literal value. For such a predicate P with a sub-query Q' and a value v it holds that $Type(P) = qv$, $Query(P) = Q'$ and $Value(P) = v$.

For the sake of simplicity we will only consider string literals and allow only all alpha-numerical, the single-quote and the space characters. Support for other types can be addressed in future.

Predicate $[@label = "SP14"]$ in Figure 4.11 falls into this category.

Query equals query

Predicates of this type compare one sub-query to another. Let P be such a predicate and Q'_1 and Q'_2 the compared sub-queries. Then $Type(P) = qq$ and $Queries(P) = (Q'_1, Q'_2)$.

For an example see predicate $[@project\_id = /data/project[@label = "SP14"] /sub-project[finished]/@id]$ in Figure 4.11.

Figure 4.11 shows a query that could be used to retrieve the list of all customers associated with finished sub-projects of a project labelled "SP14" from a suitable XML data file.

```
1 /data/customer[@project_id = /data/project[@label = "SP14"]/sub-
      project[finished]/@id]
```

Figure 4.11: The customers query

4.2.3 The Element Axis

Since we have decided to leave the text content of all elements empty and only deal with values of attributes, the last step of sub-queries for all predicates of type qv and qq can only be of the attribute axis. The evaluation of predicates of type q depends just on the structure of the target document and we can therefore have these unconstrained. Note that all predicates in Figure 4.11 satisfy this criterion.

4.2.4 Document Trees

A schema graph represents all valid documents at once. In order to find and work with documents which are relevant to a given XPath query, we need a way to represent a single valid document. That is the purpose of the *document tree* structure, as defined in Definitions 29 and 30.

Definition 29. *Let S be a non-recursive schema with graph $G = (r, V, W, E)$. A* (partial) document tree *over S is an acyclic connected directed graph $G' = (r', V', E')$ with a (not necessarily one-to-one) mapping function $m : V' \to \{r\} \cup V \cup W$ such that:*

- *$r' \in V'$ is the root node of the tree,*
- *$m(r') = r$,*
- *for every path P on nodes $v_1, v_2, ..., v_n$ where $v_1 = r'$ and v_n is a leaf node in G', $m(v_1), m(v_2), ..., m(v_n)$ is a path in G,*
- *$|Edges(r')| = 1$ and $m(ChildNode(r')) = ChildNode(r)$,*
- *$\forall w' \in V', m(w') \in W : ChildNodes(m(w')) = \{m(u'); u' \in ChildNodes(w')\}$,*
- *$\forall v' \in V', m(v') \in V : |Edges(v')| \leq 1 \wedge (|Edges(v')| = 1 \Rightarrow m(ChildNode(v')) \in ChildNodes(m(v')))$.*

Definition 30. *$G' = (V', E')$ is a* full document tree *over S if it satisfies the conditions for a partial document tree plus the following one.*

- *$\forall v' \in V', m(v') \in V : |Edges(v')| = 1$.*

The document tree is a tree-like structure similar to the common XML document data model. But, instead of directly representing elements and attributes, each node is just a pointer to the corresponding node in the schema graph. That way the structure represents an XML document and, at the same time, it allows an easy access to the related schema nodes and sub-trees.

The difference between a partial and a full document tree is that while the full tree represents a complete XML document, the partial one might miss some sub-trees.

We can divide nodes of a document tree into groups analogously to what we did for schema graphs. Suppose a schema $S = (El, At)$ with graph $G = (r, V, W, E)$ and a corresponding document tree $G' = (r', V', E')$ with mapping m. We can distinguish the following types of nodes:

- r' is the *document node* as $m(r') = r$,
- $v' \in V'$ such that $m(v') \in El$ are *element nodes*,
- $v' \in V'$ such that $m(v') \in At$ are *attribute nodes* and
- $w' \in V'$ where $m(w') \in W$ are *sequence nodes*.

4.2.5 Example

We will illustrate the concept of document trees with an example. Figure 4.12 shows a graph of a schema which could be used to describe books. Every book has an author (with an id, a firstname and a surname), a title and a content, which is divided into chapters with headlines. For the sake of simplicity, we will only consider books with up to two chapters.

In addition to the schema graph, Figure 4.12 also displays a single document tree over the same schema. We use dashed arrows to represent the mapping function. Each arrow points from a node in the tree to its counterpart in the schema graph.

Note that this tree is partial, as it violates the full-tree condition at nodes *author* and *title*.

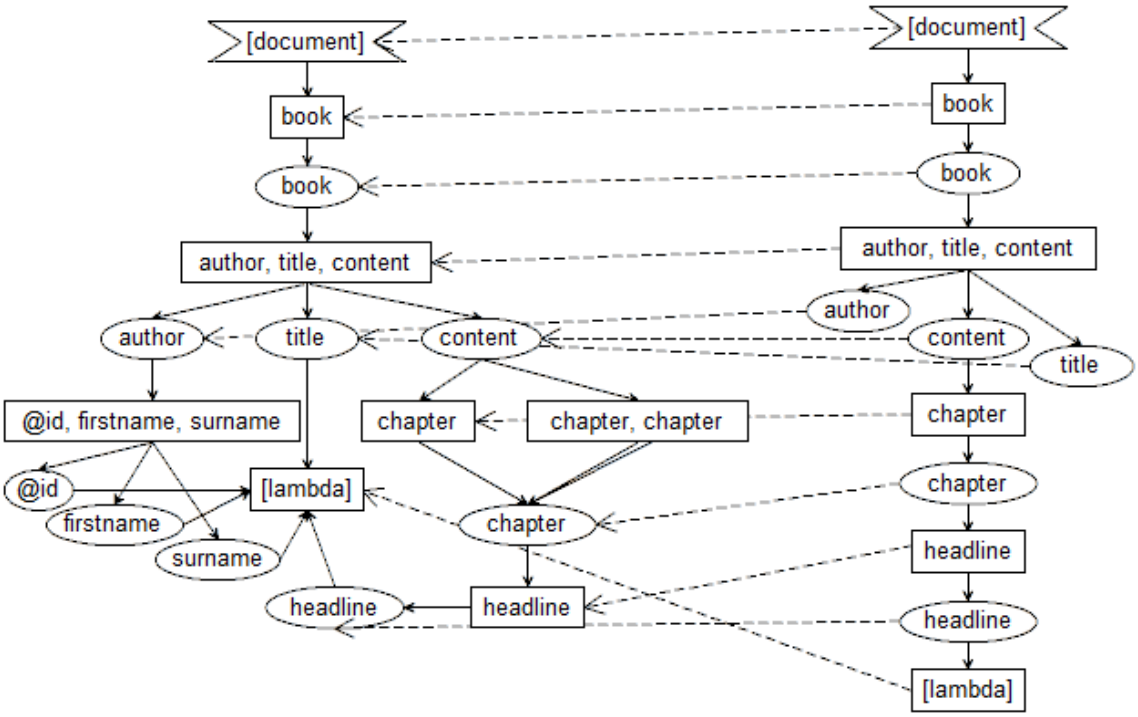


Figure 4.12: The book schema graph and a partial document tree

4.2.6 The Labelling Function

The document tree structure itself as defined in the previous sections represents only the structure of an XML document. Both the schema (the content type of attributes, the ID and IDREF constraints) and the query (predicates of type *qv* and *qq*) allow for restrictions on its content as well. We can capture these constraints using the labelling function defined here.

Definition 31. *Let $G' = (r', V', E')$ be a document tree and $U' \subseteq V'$ the set of all its attribute nodes. Then $l \subseteq \{eq, neq\} \times U' \times \{string\ s\} \cup \{same, different\} \times \binom{U'}{2}$ is a* labelling function *for G'.*

We can view this as a set of labels that can be extracted from the schema and from the query and then used as a guide when choosing correct values for individual attributes. Each of the labels captures a constraint of one of four different kinds. Let $G' = (r', V', W')$ be a document tree, U' the set of its attribute nodes, $u' \in U'$ and $v' \in U'$ two of them and s a string value. Then

- (eq, u', s) means that the attribute corresponding to u' should be assigned value s,

- (neq, u', s) means that the attribute represented by u' should be assigned a value different from s,
- $(same, u', v')$ means that the attributes represented by nodes u' and v' should be assigned the same value and
- $(different, u', v')$ means that the attributes corresponding to u' and v' should be assigned different values.

In our examples we will depict l as labels attached to individual attribute nodes of the tree.

4.2.7 Example

Figure 4.14 contains a sample document tree with labels. As we can see the depicted labelling function contains the following three items:

- $(different, v'_6, v'_8)$ and
- $(eq, v'_7, King)$,
- $(neq, v'_9, King)$.

4.2.8 The Step Criterion

For each step of an XPath query the axis and node test together form a criterion which separates schema graph nodes into two groups - those that satisfy it and those that do not.

Definition 32. *Suppose we have a non-recursive schema $S = (El, At)$ with graph $G = (r, V, W, E)$ and a node $v \in V$. Suppose we also have a query Q with a step $S_i \in Steps(Q)$. We say that:*

- v satisfies the node test $NodeTest(S_i)$ *if and only if* $v = NodeTest(S_i)$,
- v belongs to the axis $Axis(S_i)$ *if and only if* $v \in At$ *in case of the attribute axis and* $v \in El$ *in case of the child axis,*
- v satisfies the step S_i *(denoted as* $Satisfies(S_i, v) = true$*) if and only if* v *belongs to the axis* $Axis(S_i)$ *and satisfies the node test* $NodeTest(S_i)$.

Note that eventual predicates are irrelevant with regards to the satisfaction criterion as defined above as it only takes into account the axis and the node test of the query step. Unlike predicates these characteristics only involve the respective node itself and thus to determine if they are satisfied or not we do not need to traverse any other graph nodes.

Definition 33. *For a non-recursive schema graph $G = (r, V, W, E)$ and its sequence node $w \in W$ and a query Q with a step $S_i \in Steps(Q)$ we say that w* satisfies *S_i (denoted as $Satisfies(S_i, w) = true$) if and only if there exists a child node $v \in ChildNodes(w)$ such that $Satisfies(S_i, v)$ holds.*

Definition 34. *Suppose a non-recursive schema graph $G = (r, V, W, E)$ with a node $v \in V$ and a query Q with a step $S_i \in Steps(Q)$. We call*

- $Satisfying(S_i, v) = \{w \in ChildNodes(v); Satisfies(S_i, w)\}$ *the* set of all satisfying *and*
- $NonSatisfying(S_i, v) = ChildNodes(v) \setminus Satisfying(S_i, v)$ *the* set of all non-satisfying content sequences.

For a given element node v of a schema graph and a given step S_i of a query, $Satisfying(S_i, v)$ is the set of all valid content sequences of v which contain at least one element or attribute that satisfies S_i. $NonSatisfying(S_i, v)$ is the set of those valid content sequences that do not contain any such elements and attributes.

4.2.9 Example

Let us show an illustrative example. Figure 4.13 shows a snippet of a simple schema graph which represents a person.

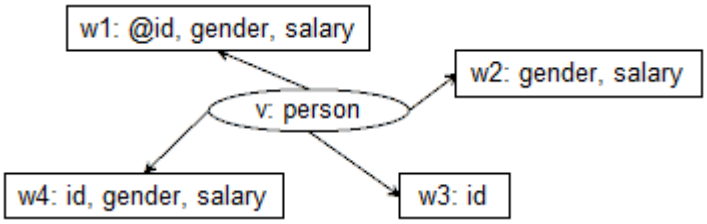


Figure 4.13: The person schema graph snippet

Let us analyse the following cases.

- Let $S_1 = @id$ (attribute axis, node test "id"). Then $Satisfying(S_1, v) = \{w_1\}$.
- Let $S_2 = id$ (element axis, node test "id"). Then $Satisfying(S_2, v) = \{w_3, w4\}$.
- Let $S_3 = gender$ (element axis, node test "gender"). Then $Satisfying(S_3, v) = \{w_1, w_2, w_4\}$.

4.2.10 Predicate Evaluation

The semantics of the equals operator, as prescribed by the XPath specification ([14]), are somewhat non-trivial and we will better describe them here in detail. For our purpose it suffices to consider the following two cases.

A Node Set and a String Value

Suppose a predicate P of type qv. As we will see in the next section, given a document tree and a base node, the sub-query $Q' \in Queries(P)$ can be evaluated to a set of attribute nodes C. We consider P satisfied if and only if there exists at least one node in C whose value equals to $Value(P)$.

When C contains more than one node then there is more than one situation which leads to accepting the predicate as satisfied. We will use labels of type eq and neq to capture all of them (not just a single positive and a single negative case), as each of them represents a different situation that a resolver must deal with.

Two Sets of Nodes

Suppose a predicate P of type qq for which the sub-queries evaluate to sets C_1 and C_2. Then P is satisfied if and only if there exists at least one attribute in C_1 and at least one attribute in C_2 with equal values.

As in the previous case we will capture all positive and negative situations, this time using labels $different$ and $same$.

4.2.11 Context Nodes

In this section we will define the concept of context nodes.

Definition 35. *Let $G' = (r', V', E')$ be a document tree with mapping m and a labelling function l over a schema graph $G = (r, V, W, E)$ with no cycles and Q a query with steps $(S_1, ..., S_n)$. Let $C \subseteq V'$ be a set of nodes such that $\forall v' \in C : m(v') \in \{r\} \cup V$. For each $i \in \{0, ..., n\}$ we define the* set of context nodes after executing step S_i *(or* before executing step S_1 *in case of $i = 0$)* when starting in C, *denoted as $ContextNodes(G', Q, C, i)$, by induction over i as follows.*

- *For $i = 0$ it holds that*
  - *$ContextNodes(G', Q, C, 0) = \{r'\}$ if Q is absolute,*
  - *$ContextNodes(G', Q, C, 0) = C$ if Q is relative.*
- *For $i > 0$ let $C' = ContextNodes(G', Q, C, i-1)$. Then $ContextNodes(G', Q, C, i) = \bigcup_{u' \in C'} \{v' \in GrandChildNodes(u'); Satisfies(S_i, m(v')) \wedge \forall P \in Predicates(S_i) : Holds(P, v')\}$.*

  Let $P \in Predicates(S_i)$ be a predicate and $v' \in V'$ a tree node. For each $Q'_i \in Queries(P)$ let $C'_i = ContextNodes(G', Q'_i, \{v'\}, |Steps(Q'_i)|)$. We say that P holds for *v' (and denote the fact as $Holds(P, v') = true$) if and only if the following conditions are all satisfied:*
  - *for all $Q'_i \in Queries(P)$ it holds that $|C'_i| > 0$,*
  - *if $Type(P) = qv$ there is a node $c \in C'_1$ such that $(eq, c, Value(P)) \in l$,*
  - *if $Type(P) = qq$ then there exist nodes $c_1 \in C'_1$ and $c_2 \in C'_2$ such that $(same, c_1, c_2) \in l$.*

For each step of a query the context set contains those grand-children of nodes from the context set of the previous step that fulfil the step criteria and satisfy all predicates. From a certain point of view this function traces the work of an XPath resolver on an XML document that corresponds to the particular document tree.

For a predicate to hold for a tree node all the queries that it consists of must evaluate to non-empty node sets and, in case of predicates of type qv and qq, the equality operator must be satisfied.

Definition 36. *Let $G' = (r', V', E')$ with mapping m be a document tree over a schema graph $G = (r, V, W, E)$ with no cycles and $b' \in V'$ its node such that $m(b') \in \{r\} \cup V$. Let Q be a query. Then $NonContextNodes(G', Q, b') = \{v' \in V'; m(v') \in V \wedge \forall i \in \{0...|Steps(Q)|\} \forall u' \in ContextNodes(G', Q, \{b'\}, i) : (v' \neq u' \wedge \forall P \in Predicates(Q) : v' \in NonContextNodes(G', P, u'))$ is the* set of all non-context nodes when starting in b'.

Definition 37. *For document tree* $G' = (r', V', E')$ *and a query* Q *let the* set of all non-context nodes *be* $NonContextNodes(G', Q) = NonContextNodes(G', Q, r')$.

The set of non-context nodes contains nodes of the document tree corresponding to element/attribute nodes of the schema graph which do not belong to the context set of any of the query steps of either the main query or any of the sub-queries of individual predicates.

4.2.12 Example

Figure 4.14 shows a (partial) document tree G' with a labelling function l. It represents two authors with ids and names. The ids should be mutually different and the name of the left one should be King while the name of the right one should be different from King.

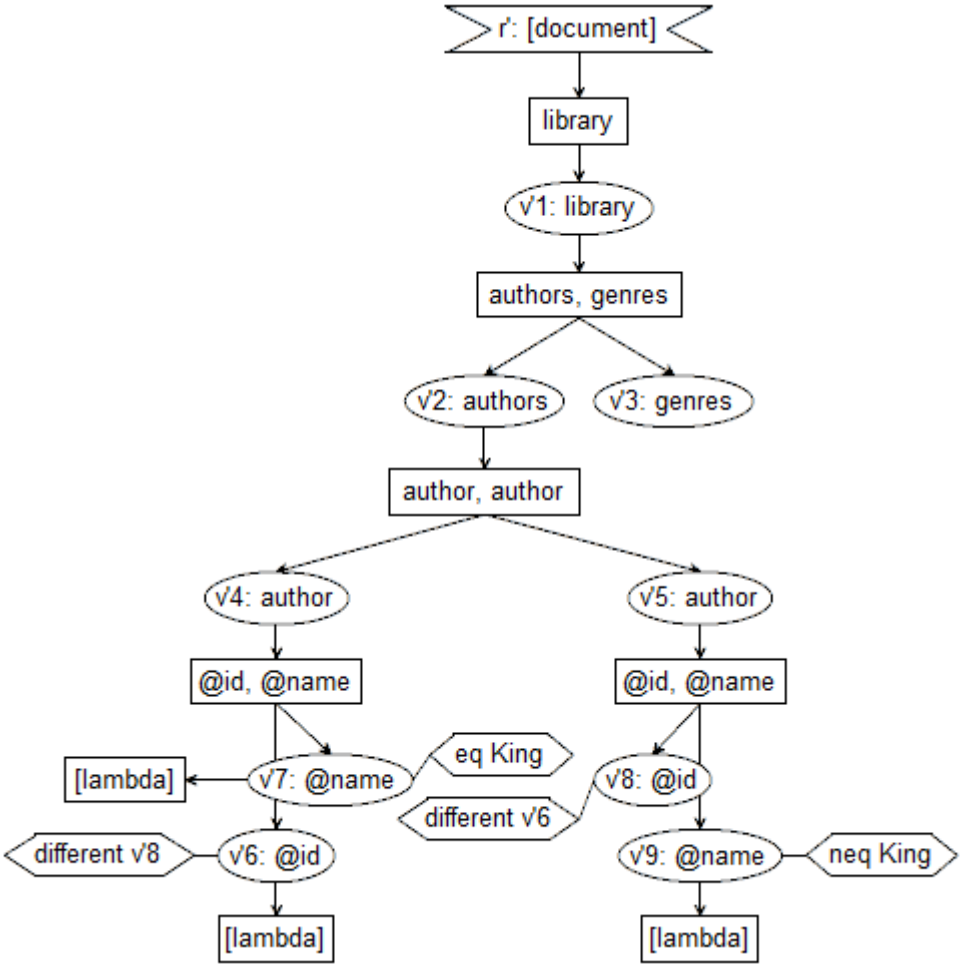


Figure 4.14: A relevant document tree

Suppose a query $Q = /library/authors/author[@name = "King"]/@id$. We distinguish the following sets of context nodes:

- $ContextNodes(G', Q, \{r'\}, 0) = \{r'\}$ as Q is an absolute query,
- $ContextNodes(G', Q, \{r'\}, 1) = \{v'_1\}$,
- $ContextNodes(G', Q, \{r'\}, 2) = \{v'_2\}$,

- $ContextNodes(G', Q, \{r'\}, 3) = \{v'_4\}$, because, as we can see below, only the first of the two *author* nodes satisfies the predicate,
- $ContextNodes(G', Q, \{r'\}, 4) = \{v'_6\}$.

Let us now focus on the predicate query $Q' = @name$ and on v'_4 as a base node. Then:

- $ContextNodes(G', Q', \{v'_4\}, 0) = \{v'_4\}$ for Q' is relative,
- $ContextNodes(G', Q', \{v'_4\}, 1) = \{v'_7\}$.

Finally, if we consider v'_5 as a base node instead of v'_4, then:

- $ContextNodes(G', Q', \{v'_5\}, 0) = \{v'_5\}$ and
- $ContextNodes(G', Q', \{v'_5\}, 1) = \{v'_9\}$.

We can see that the predicate holds for node v'_4, as $(eq, v'_7, King) \in l$, but not for v'_5, because $(eq, v'_9, King) \notin l$.

Also note that the only nodes that do not belong to any of the sets for either the main query or the sub-query are v'_3 and v'_8 and so $NonContextNodes(G', Q) = \{v'_3, v'_8\}$.

4.2.13 Relevant Document Trees

In order to keep the results reasonably small, our algorithm should only generate documents that are relevant (interesting) with regards to the given query. The aim is to focus on documents with a high potential to differentiate XPath resolvers that work correctly and efficiently from those that do not. In this section we will describe what such documents look like.

Definition 38. *For a given schema graph $G = (r, V, W, E)$ with no cycles with node $v \in V$ and a given query Q with step $S_i \in Steps(Q)$ let $Relevant(S_i, v) = Satisfying(S_i, v) \cup \{shortest, longest\}$ where*

- $shortest = \{w \in NonSatisfying(S_i, v); |w| = min_{u \in NonSatisfying(S_i, v)}|u|\}$,
- $longest = \{w \in NonSatisfying(S_i, v); |w| = max_{u \in NonSatisfying(S_i, v)}|u|\}$.

For a given element node v and a given step S_i of a query, $Relevant(S_i, v)$ contains those valid child-nodes of v which are considered relevant with regards to evaluating step S_i (irrespective of eventual predicates - they will be dealt with separately). That means nodes corresponding to all satisfying sequences and to the longest and the shortest of the non-satisfying ones.

Definition 39. *Let $G' = (r', V', E')$ be a document tree and $b' \in V'$ one of its nodes. Let P be a predicate. We say that a set of labels l* covers predicate P when evaluated from b', *and denote that as $CoversPredicate(G', b', P, l) = true$, if and only if the following conditions are satisfied.*

- *Suppose that $Type(P) = qv$, $Q' \in Queries(P)$ and $v = Value(P)$. Let $C = ContextNodes(G', Q', \{b'\}, |Steps(Q')|)$. There exists a $C' \subseteq C$ such that:*

  - $(eq, c', v) \in l$ *for all* $c' \in C'$ *and*
  - $(neq, c, v) \in l$ *for all* $c \in C \setminus C'$.

- *Suppose now that* $Type(P) = qq$ *and* $Queries(P) = (Q'_1, Q'_2)$. *Let* $C_i = ContextNodes(G', Q'_i, \{b'\}, |Steps(Q'_i)|)$ *for* $i \in \{1, 2\}$. *For each* $c_1 \in C_1$ *there is a* $C'_2 \subseteq C_2$ *such that:*

  - $(same, c_1, c'_2)$ *for all* $c'_2 \in C'_2$ *and*
  - $(different, c_1, c_2)$ *for all* $c_2 \in C_2 \setminus C'_2$.

This definition is based on how resolvers evaluate XPath predicates, which was described in Section 4.2.10. Let us add a brief description to make our intentions clear.

- We say that a set of labels l covers a predicate P of type qv if there exists a sub-set of the node-set that P evaluates to such that included items are labelled to match the $Value(P)$ and all excluded items are labelled not to match. Every such sub-set corresponds to a different situation with which a resolver has to deal with when evaluating P.
- A set of labels l covers a predicate P of type qq if for every node v' that the first sub-query evaluates to there is a sub-set of the node-set that the second sub-query evaluates to such that v' is labelled to match all included nodes and not to match any excluded node.

Definition 40. *Suppose a document tree* $G' = (r', V', E')$ *with labelling* l *and mapping* m *over a schema graph* $G = (r, V, W, E)$ *with no cycles and a query* Q *with steps* $(S_1, ..., S_n)$. *Let* $v' \in V'$ *be a node such that* $m(v') \in \{r\} \cup V$. *We say that* G' *is* relevant with regards to Q and v', *denoted as* $Relevant(G', Q, v') = true$, *if and only if* l *is the smallest set such that the following conditions are satisfied.*

- *For all* $i \in \{1, ..., n\}$ *and* $u' \in ContextNodes(G', Q, \{v'\}, i-1)$ *it holds that:*

  - $|ChildNodes(u') = 1|$ *and* $m(ChildNode(u')) \in Relevant(S_i, m(u'))$,
  - *for each* $P \in Predicates(S_i)$ *and each* $u'' \in GrandChildNodes(u')$ *such that* $Satisfies(S_i, u'')$:
    - $Relevant(G', P, u'')$ *and*
    - $CoversPredicate(G', u'', P, l)$.

- *For all* $u' \in NonContextNodes(G', Q, v')$ *it holds that* $|ChildNodes(u')| = 0$.

Definition 41. *A document tree* $G' = (r', V', E')$ *is* relevant with regards to query Q *if and only if* $Relevant(G', Q) = Relevant(G', Q, r') = true$.

A document tree is relevant with regards to a query if both its structure and its content satisfy the conditions listed above. Which means that:

- for each context node of each step of the query (and each sub-query) there is a child node and it corresponds to a relevant sequence node,
- non-context nodes do not have any children at all,
- all predicates are covered by the labelling function.

As a labelling function we want the smallest set of labels that conforms to the criteria. The reason is to avoid irrelevant content constraints. This way a relevant labelling function contains only labels which are necessary in order to cover all predicates.

4.2.14 Example 1

Let us return to the sample schema depicted in Figure 4.13. We can observe the following facts.

- For $S_1 = @id$ (attribute axis, node test "id"), $Relevant(S_1, v) = \{w_1, w_3, w_4\}$ (w_1 is the only satisfying and w_3 is the shortest and w_4 the longest of the non-satisfying sequences).
- Let $S_3 = gender$ (element axis, node test "gender"). Then $Relevant(S_1, v) = \{w_1, w_2, w_3, w_4\}$, as w_1, w_2 and w_4 are the satisfying sequences and w_3 is both the longest and the shortest of the non-satisfying ones.

4.2.15 Example 2

Let us focus on the document tree in Figure 4.14 and verify that, consistently with Definition 39, the depicted labelling function l covers predicate $P = [@name = "King"]$ when either v'_4 or v'_5 is taken as a base node. Node that $Type(P) = qv$.

- In case of v'_4 the sub-query $@name$ evaluates to $C = \{v'_7\}$. For $C' = C$ both the conditions hold. The first one due to the fact that $(eq, v'_7, King) \in l$ and the second one trivially because $C \setminus C' = \emptyset$.
- In case of v'_5 we have $C = \{v'_9\}$. This time both conditions hold for $C' = \{\}$. The first one trivially and the second one because $(neq, v'_9, King) \in l$.

4.2.16 Example 3

The document tree shown in Figure 4.14 is relevant with regards to query $Q_1 = /library/authors/author[@name = "King"]/@id$ as it satisfies all conditions. Note especially that the only non-context nodes are v'_3 and v'_8 and that consequently these are the only nodes for which a child-node was omitted.

On the other hand the same tree would not be relevant for some different queries, such as $Q_2 = /library/genres/genre/books/book$. In case of Q_2 not even the constraints on structure would be satisfied, due to the following defects:

- node v'_3 belongs to context nodes after the second step and should have a child-node and
- node v'_2 is a non-context node and should not be assigned any children.

4.2.17 Schema-Based Content Constraints

Besides XPath queries we can also extract text content restrictions from the schema. We will now look at how we can express the ID and IDREF constraints in terms of the labelling function.

Definition 42. *Let S be a schema and $G' = (r', V', E')$ with mapping m and a labelling function l a document tree over its graph. We say that l* covers all ID constraints*, and denote that as $CoversIDs(G', l) = true$, if and only if for each $u' \in V'$ and each $v' \in V'$, where $m(u') \in IDs(S)$ and $m(v') \in IDs(S)$, it holds that $(different, u', v') \in l$.*

Definition 43. *Let S be a schema and $G' = (r', V', E')$ with mapping m and a labelling function l a document tree over its graph. We say that l* covers all IDREF constraints*, and denote that as $CoversIDREFs(G', l) = true$, if and only if for each $u' \in V'$ where $m(u') \in IDREFs(S)$ there exists a $v' \in V'$ such that $m(v') \in IDs(S)$ and $(same, u', v') \in l$.*

4.2.18 Example

We can easily verify that the labelling function l displayed in Figure 4.14 covers all ID and IDREF constraints of the schema defined in Figure 4.4. Let us make the following two observations.

- Only nodes v'_6 and v'_8 correspond to elements with ID constraints and $(different, v'_6, v'_8) \in l$.
- There are no nodes that would correspond to elements with IDREF constraints and thus the IDREFs coverage condition is trivially satisfied.

4.2.19 The Value-Assignment Function

To be able to transform a document tree into an XML document we have to choose a single value for each of its attribute nodes. The value-assignment function, as defined in this section, denotes the chosen values.

Definition 44. *Let $G' = (r', V', E')$ be a document tree and $U' \subseteq V'$ the set of all its attribute nodes. For each $u' \in U'$ let $v(u') \in \{string\ s\}$ denote the* value assigned to node u'.

In order for the value-assignment function to be valid it must adhere to the content types of individual attributes and satisfy all constraints induced by the set of labels of the corresponding document tree.

Definition 45. *Let $G' = (r', V', E')$ with mapping m and labelling l be a document tree and $U' \subseteq V'$ the set of all its attribute nodes. We say that a value-assignment function v is* valid for G' and l*, and denote that as $Valid(G', l, v) = true$, if and only if the following conditions are all satisfied for all $u' \in U'$.*

- $v(u') \in ContentType(m(u'))$,
- $v(u') = s$ *for all string values* s *such that* $(eq, u', s) \in l$,

- $v(u') \neq s$ *for all string values* s *such that* $(neq, u', s) \in l$,
- $v(u') = v(v')$ *for all nodes* $v' \in U'$ *such that* $(same, u', v') \in l$ *and*
- $v(u') \neq v(v')$ *for all nodes* $v' \in U'$ *such that* $(different, u', v') \in l$.

Sometimes there is no valid value-assignment function (e.g. when there are two or more constraints of type *eq* for the same node, each with a different string value). In that case the document tree is invalid, even though relevant, and cannot be transformed to an XML document.

On the other hand there are situations when multiple different value-assignment functions are valid. In that case any one of them can be chosen as they all mean very similar situations for an XPath resolver to solve.

4.2.20 Example

Let us return once more to the document tree G' with mapping m and labelling l introduced in Figure 4.14. Suppose that:

- $ContentType(m(v'_6)) = ContentType(m(v'_8)) = \{string\ s\}$ and
- $ContentType(m(v'_7)) = ContentType(m(v'_9)) = \{King, Poe, Tolkien\}$.

We will show examples of valid and invalid value-assignment functions. We can observe that:

- $\{v'_6 => 1, v'_7 => King, v'_8 => 2, v'_9 => King\}$ is valid.
- $\{v'_6 => 1, v'_7 => King, v'_8 => 2, v'_9 => Rowling\}$ is not, as the value of node v'_9 does not belong to its content type, which breaks condition no. 1.
- $\{v'_6 => 1, v'_7 => Poe, v'_8 => 1, v'_9 => King\}$ is not either. Nodes v'_6 and v'_8 have the same value and thus break the fourth condition and node v'_7 was assigned a value different from King which breaks condition no. 2.

Note that the value of v'_9 is not constrained by the labelling function in any way and therefore any value from its content type would be valid.

4.3 The Algorithm

We will now describe an algorithm that, given a non-recursive schema S with graph G and a query Q on input, generates the set of all relevant document trees and converts them into the corresponding XML documents which it returns on output. The algorithm runs in five phases, as depicted in Algorithm 4.

Non-deterministic Approach

The algorithm is quite complex and so, to achieve reasonable clarity, we have decided to only describe it here in a non-deterministic form. That is the reason why it returns just one (non-deterministically chosen) relevant XML document, instead of the set of all of them. A deterministic counterpart can be derived using backtracking.

When describing individual parts of the algorithm, we will point out each non-deterministic choice.

Let us now take a closer look at each of these phases.

```
1  Function Main(schema graph G, query Q) : an XML document
2    // Phase 1 − generate a relevant (partial) document tree.
3    Let G' = Generate(G, Q).
4    // Phase 2 − complete it into a full document tree.
5    Call Complete(G').
6    // Phase 3 − cover the ID and IDREF constraints.
7    Call CoverIDs(G') and CoverIDREFs(G').
8    // Phase 4 − find a suitable value−assignment function.
9    Call Evaluate(G').
10   // Phase 5 − convert it to an XML document.
11   Let Doc = Convert(G').
12   Return Doc.
13 End.
```

Algorithm 4: The Main Function

4.3.1 Phase 1 - Partial Document Trees

Let us suppose to be given a non-recursive schema S with an acyclic graph G and a query Q. In this section we will introduce a method to determine the set of all partial document trees over S relevant to Q.

The Algorithm

The algorithm, as depicted in Algorithms 5 and 6, was too long to fit on a single page and therefore had to be split into two parts.

It is not much different from what Definition 40 prescribes. First of all we create a basic document tree which consists of just the root node and an empty labelling function. Then we iterate over all steps of the query, extending the tree with new branches and labels. For each step and each context node we choose one of the relevant sequence nodes and append it to the tree together with all its child nodes. Then we process each predicate of the step, use recursion to deal with sub-queries and add new labels so that the predicate gets covered. Finally we update the set of context nodes and continue with the next iteration. When the last step of the query is processed, the tree has been extended into a relevant document tree.

Absolute Predicate Queries

We have decided to support both relative and absolute queries inside predicates. Unlike relative sub-queries, which only affect the sub-tree rooted by the current context-node, absolute sub-queries can affect the whole document tree. We need to be aware of the fact that the same tree node might appear in the context sets of two (or more) different steps. Apparently it is not possible to choose each time a different child node (see Definition 40 where we state that, in a relevant document tree, each context node might only have a single child node assigned). As we can see on lines 56 - 63, if we are processing a node that has already been processed in the past, instead of choosing a new sequence node we stick with the one that was chosen previously.

Non-deterministic Choices

See lines 35 and 45 where we choose the correct sub-sets and line 65 where we choose the appropriate sequence node - all of these choices are non-deterministic.

4.3.2 Phase 2 - Full Document Trees

After we resolve the set of all relevant (partial) document trees, as described in the previous section, it is necessary to complete them to full document trees, one at a time. The process of completion is described below.

Motivation and Intent

According to Definition 40 the non-context nodes of a relevant document tree do not contain any children. In order for the generated documents to be valid against the given schema, these nodes have to be assigned valid sub-trees as well. That is the purpose of the following algorithm. The intention is to choose the smallest possible valid sub-tree for each of them, for which the precomputed values of $MinDepth$ are relied upon.

Text Content

Each tree node involved in predicate evaluation is a context node and thus the partial trees as a result of the previous phase already contain all of them. In this phase we only finish the non-relevant parts of the document (those that a typical XPath resolver would not even visit when evaluating the query). For this reason we do not need to add any new labels and so the labelling function stays unchanged.

The Algorithm

Let $G' = (r', V', E')$ with mapping m be a partial document tree over $G = (r, V, W, E)$. In order to transform G' into a full document tree, we need to satisfy the condition from Definition 30, which means to complete all its in-complete nodes. The algorithm is described in Algorithm 7.

4.3.3 Phase 3 - ID and IDREF constraints

Now that the have completed each relevant tree into a full tree G' we must update the labelling function l to cover the ID and IDREF constraints as declared in the schema S. The algorithm, as described in Algorithm 8, directly follows Definitions 42 and 43.

Non-deterministic Choices

Refer to line 10 where we choose a node of type ID for each node of type IDREF in a non-deterministic way.

```
1  Function Generate(schema graph G = (r, V, W, E), query Q): a document
       tree with a labelling function
2    Let G' = (r', {r'}, {}) where r' is a new node such that m(r') = r.
3    Let l = {}.
4    ExtendTree(G', l, Q, r').
5    Return (G', l).
6  End.
7
8  Function ProcessQuery(document tree G' = (r', V', E') with mapping m,
       labelling function l, query Q, node v' ∈ V') : a set of nodes
9    Assert that m(v') ∈ {r} ∪ V.
10   Let C = {v'}.
11   For each S_i ∈ Steps(Q) do
12     Let C' = {}.
13     For each v' ∈ C do
14       Let w' = ChooseNode(G', S_i, v').
15       If w' ≠ NULL then
16         For all u' ∈ ChildNodes(w') such that Satisfies(S_i, m(u')) do
17           Let ok = true.
18           For each P ∈ Predicates(S_i) do
19             Let ok = ok ∧ ProcessPredicate(G', l, P, u').
20           End
21           If ok then let C' = C' ∪ {u'}.
22         End
23       End
24     End
25     Let C = C'.
26   End
27   Return C.
28 End.
29
30 Function ProcessPredicate(document tree G' = (r', V', E') with mapping m,
       labelling function l, predicate P, node u' ∈ V'): a boolean value
31   Assert that m(v') ∈ V.
32   If Type(P) = q then return |ProcessQuery(G', l, Query(P), u')| > 0.
33   Else if Type(P) = qv then
34     Let C = ProcessQuery(G', l, Query(P), u').
35     Choose a sub−set C' ⊆ C.
36     For each c' ∈ C' let l = l ∪ {(eq, c', Value(P))}.
37     For each c ∈ C \ C' let l = l ∪ {(neq, c, Value(P))}.
38     Return |C'| > 0.
39   End
40   Else if Type(P) = qq then
41     Assume that (Q_1, Q_2) = Queries(P).
42     Let C_1 = ProcessQuery(G', l, Q_1, u') and C_2 = ProcessQuery(G', l, Q_2, u').
43     Let satisfied = false.
44     For each c_1 ∈ C_1 do
45       Choose a sub−set C'_2 ⊆ C_2.
46       If |C'_2| > 0 then let satisfied = true.
47       For each c'_2 ∈ C'_2 let l = l ∪ {(same, c_1, c'_2)}.
48       For each c_2 ∈ C_2 \ C'_2 let l = l ∪ {(different, c_1, c_2)}.
49     End
50     Return satisfied.
51   End
52   Assert false.
53 End.
```

Algorithm 5: The Generate Function (Part 1)

```
54 Function ChooseNode(document tree G' = (r', V', E') with mapping m, query
       step S_i, node v' ∈ V') : a tree node or NULL
55   Assert that m(v') ∈ {r} ∪ V.
56   If |Edges(v')| > 0 then
57     Assert that |Edges(v')| = 1.
58     Let w' = ChildNode(v').
59     If Relevant(S_i, m(w')) then
60       Return w'.
61     Else
62       Return NULL.
63     End
64   Else
65     Choose a node w ∈ ChildNodes(m(v')) such that Relevant(S_i, w).
66     Let w' be a new node such that m(w') = w.
67     Let V' = V' ∪ {w'} and E' = E' ∪ {(v', w')}.
68     For each (w, v) ∈ E do
69       Let u' be a new node such that m(u') = u.
70       Let V' = V' ∪ {u'} and E' = E' ∪ {(w', u')}.
71     End
72     Return w'.
73   End
74 End.
```

Algorithm 6: The Generate Function (Part 2)

```
1  Function Complete(document tree G' = (r', V', E'))
2    For all v' ∈ V' such that m(v') ∈ V and |E(v')| = 0:
3      Complete(v')
4    End
5  End.
6
7  Function Complete(node v' of a document tree G')
8    Let v = m(v').
9    Assert that |Edges(v)| > 0.
10   Find w ∈ ChildNodes(v) such that MinD(w) = min_{z∈ChildNodes(v)} MinD(z).
11   Let w' be a new node such that m(w') = w.
12   Let V' = V' ∪ {w'} and E' = E' ∪ {(v', w')}.
13   For all u ∈ ChildNodes(w) do
14     Let u' be a new node such that m(u') = u.
15     Let V' = V' ∪ {u'} and E' = E' ∪ {(w', u')}.
16     Call Complete(u').
17   End
18 End.
```

Algorithm 7: The Complete function

```
1  Function CoverIDs(full document tree G' = (r', V', E') with mapping m and
       labelling l)
2    For all v'_1 ∈ V' and v'_2 ∈ V' such that m(v'_1) ∈ IDs(S) and m(v'_2) ∈ IDs(S) do
3      Let l = l ∪ {(different, v'_1, v'_2)}.
4    End
5  End.
6
7  Function CoverIDREFs(full document tree G' = (r', V', E') with mapping m
       and labelling l)
8    For all v'_1 ∈ V' such that m(v'_1) ∈ IDREFs(S) do
9      Assert that {v' ∈ V'; m(v') ∈ IDs(S)} ≠ ∅.
10     Choose a node v'_2 ∈ V' such that m(v'_2) ∈ IDs(S).
11     Let l = l ∪ {(same, v'_1, v'_2)}.
12   End
13 End.
```

Algorithm 8: The CoverIDs and CoverIDREFs functions

4.3.4 Phase 4 - Value-Assignment Function

Suppose that we have a full document tree $G' = (r', V', W')$ with a (final) labelling function l that corresponds to schema $S = (El, At)$. Algorithm 9 shows an algorithm to produce a valid value-assignment function v (if it exists).

Intention

Our aim is to reduce this problem onto graph colouring. For that we need to build a suitable graph and determine a domain of possible values (colours) for each of its nodes. Any valid colouring should then give us a valid value-assignment function.

Algorithm

We start with a new (non-directed) graph on the set of all attribute nodes of G' with no edges and take the basic content types of corresponding attributes as initial domains.

Then we deal with labels of each type, one at a time, updating the initial graph. To capture *eq* and *neq* constraints it suffices to update the domains of individual attributes. For labels of type $different$ and *same* we need to modify the structure of the graph - add new edges and merge nodes.

It is important that the merging takes place at the end of the graph construction, when all other constraints are already reflected.

Merging Nodes

To ensure that two nodes will get the same colour we merge them into a single node. Note that we cannot merge nodes which are directly connected. There are two other things to observe about the newly created node.

- It takes over all original edges.
- Its domain is the intersection of the original domains.

This way all original constraints are preserved.

Graph Colouring

The algorithm to colour a graph has been omitted. Any standard technique can be used, such as a greedy colouring strategy with backtracking.

```
1  Function Evaluate(document tree G' = (r', V', E') with mapping m and
       labelling l): a value-assignment function
2    Let U' = {v' ∈ V'; m(v') ∈ At}.
3    For each u' ∈ U' do
4      Let dom(u') = ContentType(m(u')).
5      For each s such that (eq, u', s) ∈ l let dom(u') = dum(u') ∩ {s}.
6      For each s such that (neq, u', s) ∈ l let dom(u') = dum(u') \ {s}.
7    End
8    Let G'' = (U', E'') where E'' = ∅ be a new non-directed graph.
9    For all (different, u', v') ∈ l let E'' = E'' ∪ {(u'', v'')}.
10   For all (same, u', v') ∈ l do
11     Assert that {u', v'} ∉ E''.
12     Call Merge(G'', u', v').
13   End
14   Return a valid graph colouring for G''.
15 End.
16
17 Function Merge(non-directed graph G'' = (V'', E''), nodes v''_1 ∈ V'' and
       v''_2 ∈ V'')
18   Let v'' be a new node in G'' and V'' = (V'' \ {v''_1, v''_2}) ∪ {v''}).
19   Let dom(v'') = dom(v''_1) ∩ dom(v''_2).
20   For each {v''_1, u''} ∈ E'' let E'' = (E'' \ {{v''_1, u''}}) ∪ {{v'', u''}}.
21   For each {v''_2, u''} ∈ E'' let E'' = (E'' \ {{v''_2, u''}}) ∪ {{v'', u''}}.
22 End.
```

Algorithm 9: The Evaluate function

4.3.5 Phase 5 - XML Documents

Let us have a non-recursive schema $S = (El, At)$ with graph $G = (r, V, W, E)$ and a full document tree $G' = (r', V', E')$ over G with values v and mapping m. Note that the tree already contains all information about the corresponding document. In order to do the transformation, we traverse the tree recursively, as described in Algorithm 10.

4.3.6 Example

For an example see the following two figures. Figure 4.15 contains a sample full document tree which depicts a car with a coachwork, a motor and three wheels with tyres. Figure 4.16 shows the corresponding XML document.

```
1  Function Convert(full document tree G' = (r', V', E') with values v) : XML
       document
2    Let R = Convert(GrandChildNode(r'), v).
3    Return (P, R) where P is the XML prolog.
4  End.
5
6  Function Convert(tree node v' such that m(v') ∈ El, value function v) :
       XML element
7    Create a new element e named v'.
8    If |Edges(v')| > 0 then
9      Assert that |Edges(v')| = 1.
10     Let w' = ChildNode(v').
11     For all u' ∈ ChildNodes(w') do
12       If u' ∈ At then specify attribute named u' with value v(u') for e.
13       Otherwise add e' as a sub−element of e for e' = Convert(u', v).
14     End
15   End
16   Return E.
17 End.
```

Algorithm 10: The Convert function

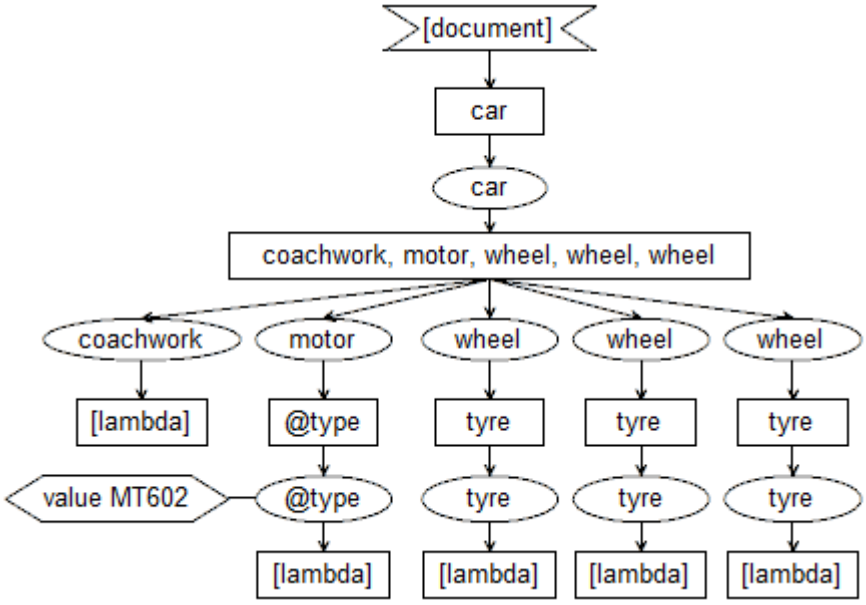


Figure 4.15: The car document tree

```
1  <?xml version="1.0" ?>
2  <car>
3    <coachwork />
4    <motor type="MT602" />
5    <wheel>
6      <tyre />
7    </wheel>
8    <wheel>
9      <tyre />
10   </wheel>
11   <wheel>
12     <tyre />
13   </wheel>
14 </car>
```

Figure 4.16: The car XML document

4.3.7 Evaluation

It is important to realize that, for a non-recursive schema S and an XPath query Q on input, the set of XML documents R as the result of our algorithm satisfies the following requirements:

1. R contains only documents valid against S.
2. R contains only documents with a high potential to exercise the resolver in situations relevant with regards to evaluating Q.

Let us now focus on each of these in more depth.

Validity

The fact that the structure of the generated documents adheres to the schema follows from the way we build the schema graph and the way we traverse it and build the relevant document trees. As already mentioned, the fact that we assign empty text content to all elements does not get us in conflict with any DTD schema definition. And finally, the content we assign to attributes is also valid, because it belongs to the declared basic content types and satisfies all ID and IDREF constraints.

Relevance

It follows from the way we choose relevant document trees and the way we complete them into full trees that the resultant XML documents are only developed in areas that an XPath resolver is supposed to visit when evaluating the query. For other areas we choose the simplest possible valid sub-trees and thus ignore documents that would differ in parts which a resolver would not visit at all.

The same applies for content of attributes. We choose values to expose different situations considering evaluation of individual predicates. For attributes that are not referenced by any of them we pick just a single valid value.

Therefore each of the generated documents means a different situation for the resolver to deal with.

4.4 Similar Document Trees

The amount of documents valid against a given schema and relevant to a given query is still very large. Moreover, many of them have a very similar structure and differ only in the ordering of elements at individual tree levels. We will now define a new relation called *similarity* which will help us recognize similar documents.

Definition 46. *Documents* $G'_1 = (r'_1, V'_1, E'_1)$ *with mapping* m_1 *and labelling* l_1 *and* $G'_2 = (r'_2, V'_2, E'_2)$ *with mapping* m_2 *and labelling* l_2 *are* similar *if and only if there exist bijective functions* $M : V'_1 \to V'_2$ *and* $L : l_1 \to l_2$ *such that:*

- $M(r'_1) = r'_2$,
- *for each* $v'_1 \in V'_1$ *it holds that:*
  - $m_1(v'_1) = m_2(M(v'_1))$ *and*
  - *for each* $w'_1 \in ChildNodes(v'_1)$ *there is a* $w'_2 \in ChildNodes(M(v'_1))$ *such that* $M(w'_1) = w'_2$,
- *for each* $l \in l_1$*:*
  - *if* $l = (t, v', s)$ *for* $t \in \{eq, neq\}$, $v' \in V'_1$ *and a string* s *then* $L(l) = (t, M(v'), s)$,
  - *if* $l = (t, v'_1, v'_2)$ *for* $t \in \{different, same\}$, $v'_1 \in V'_1$ *and* $v'_2 \in V'_1$ *then* $L(l) = (t, M(v'_1), M(v'_2))$.

In other words, two trees are similar if it is possible to meet the following requirements by changing the order of child nodes for one of them.

- The trees have the same structure. This means that for each node in the first tree there is a corresponding node in the second tree, which is at the same level and position. The corresponding nodes also have the same number of child nodes.
- Each node in the first tree is mapped to the same schema graph node as the corresponding node in the second tree.
- The trees have equivalent sets of labels, meaning that there exists a mapping between these sets such that for each pair of corresponding labels the related nodes correspond to each other.

Lemma 1. *The similarity relation is an equivalence on the set of all documents valid against a given schema* S *and relevant to a given query* Q*.*

The similarity relation is an equivalence and we can therefore use it to divide the set of all relevant document trees into disjoint groups, where all trees in each group are mutually similar. We will only keep one tree per group.

4.4.1 Example

To illustrate the notion of the similarity relationship we refer to the following figures. Figure 4.17 contains a (partial) document tree which represents a book, which is divided into chapters and sections. Note that the second chapter comprises of two sections, while the first and the last chapters contain just one.

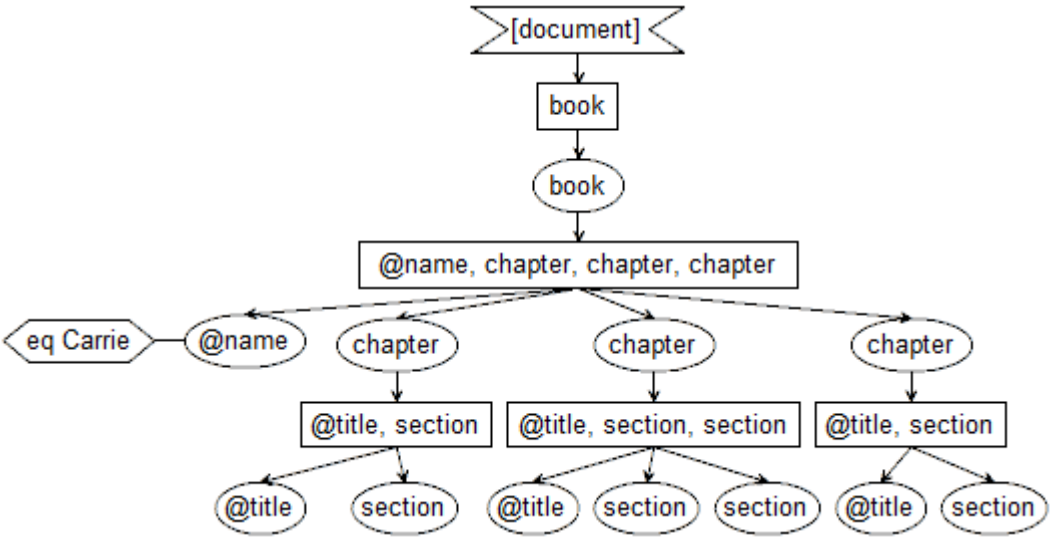


Figure 4.17: The book document tree no. 1

Figure 4.18 contains another document tree. It is not identical to the former one - this time the first chapter contains two sections and the other two just one. It is, however, similar to it as it satisfies all the criteria. If we swap the first two chapters we get a tree that has the same structure and labels as the original one.

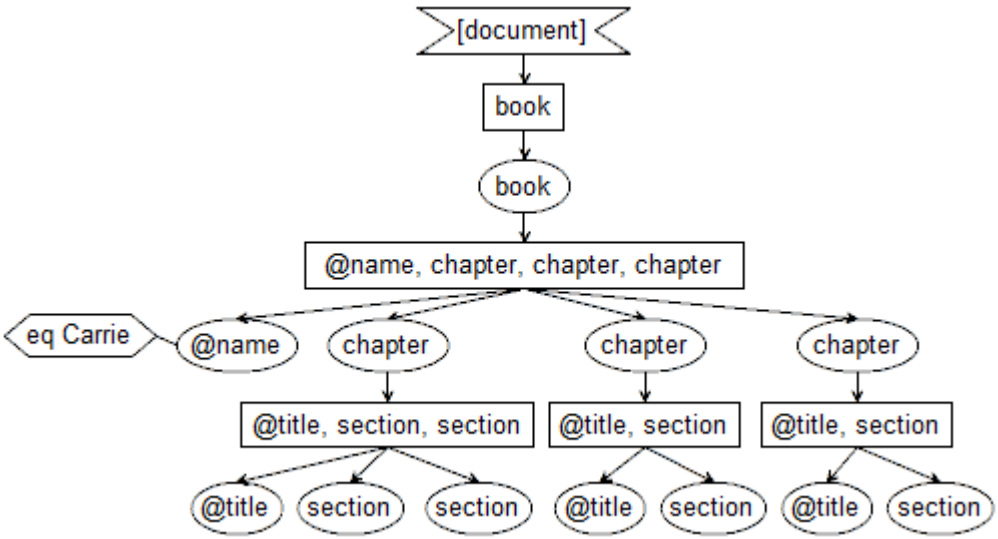


Figure 4.18: The book document tree no. 2

Figure 4.19 contains yet another document tree. This one is not similar to the previous trees. We can see that each of the chapters contains just one section and thus no ordering of child nodes would get us a tree with an identical structure.

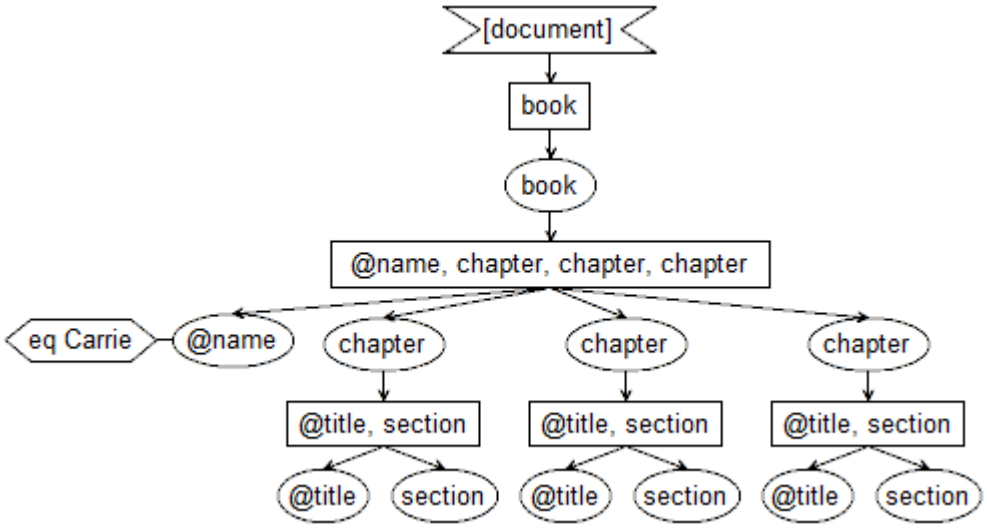


Figure 4.19: The book document tree no. 3

Figure 4.20 contains one last sample document tree. This one is not similar to the first tree either, even though its structure is identical. While the first tree contains just a single label, this time we have two. Clearly there is no bijective mapping between the two sets.

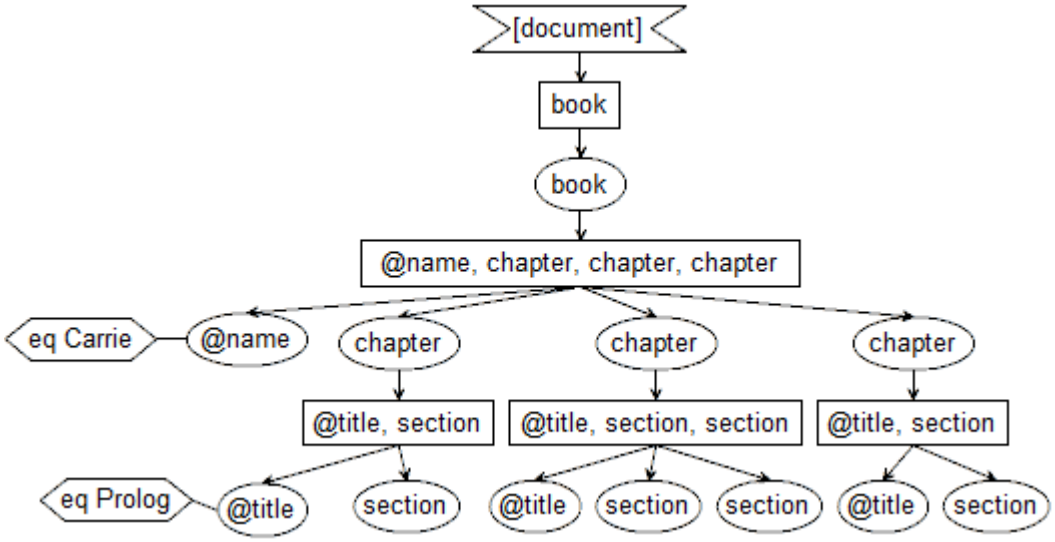


Figure 4.20: The book document tree no. 4

Chapter 5

Implementation

We have implemented a prototype of the algorithm as a part of the thesis to help us further demonstrate and validate our intentions. In this chapter we will briefly describe its architecture and how to use it.

5.1 Third Party Tools and Technologies

The application is based on the Java platform. It was written in the Eclipse 4.3.1 IDE [15] using JDK 1.7 [16]. The Maven 3.1 project management tool [17] was used as a build and dependency management system and the JUnit 4.1 framework [18] for unit and integration tests. We chose the JavaCC tool [19] to generate compilers for DTD schemas and XPath expressions. We used the JDOM 2 API [20] to work with XML documents and the Apache Commons library [21] to handle collections, files, command-line arguments, etc. The Log4J 1.2 framework [22] was used as a logging facility and Visual VM [23] as a profiling tool. GIT [24] served as a version control system.

5.2 Overall Architecture

The application consists of three basic modules:

- the schema parser,
- the query parser,
- the pipeline.

Let us now introduce each of them.

5.2.1 The Schema Parser

This module represents a parser that can be used to convert XML schema definitions written in DTD (or, more precisely, in the subset of the DTD language supported in our algorithm, as described in the previous chapter) into the corresponding instances of the schema graph structure (see Definition 18).

5.2.2 The Query Parser

Analogously to the previous case, this module consists of a parser for XPath queries.

5.2.3 The Pipeline

This is where the main part of the application, the algorithm itself, resides. It is implemented in the form of a pipeline, which accepts a schema graph and a query on input and gives a list of XML documents on output. Inside the pipeline the data flow through five stages, which correspond to different parts of the algorithm (as was described in Section 4.3). Each of them is implemented as a separate sub-module. There are the following five stages:

- the generator (generates relevant partial document trees),
- the completioner (completes them into full document trees),
- the coverer (covers the ID and IDREF constraints),
- the assigner (chooses suitable value-assignment functions),
- the converter (converts the trees into XML documents).

The algorithms for all phases have already been described in pseudo-code. In the assigner we use a greedy strategy to find valid colourings. To make it faster, we sort all nodes by the number of permissible values at the beginning of the search.

Similarity

The similarity relation is implemented as a standalone sub-module of the pipeline. It provides an interface to decide whether two document trees are similar or not, and, by extension, to choose a set of dissimilar representatives from a set of trees.

5.3 Usage

The prototype is a standard Java application with a command-line interface and can be run using the following command:

```
1 java -jar xmlgen-1.0.jar [parameters]
```

Let us now introduce the supported parameters:

- **schemaFile** (mandatory) - path to the file which contains the DTD schema definition (absolute, or relative to the installation directory),
- **query** (mandatory) - the XPath query,
- **outputDir** (mandatory) - path to a folder to which the resulting documents should be saved (absolute, or relative to the installation directory),
- **mc** (optional, defaults to 2) - value of MC (max cardinality, see Section 4.1.3),

- **stepNo** (optional) - index of an interesting step (will be explained in Section 6.5).

Chapter 6

Experiments

In this chapter we will present the results of running our implementation on some sample data using a computer with the following configuration:

- a 2-core 2.30GHz CPU,
- 2GB of RAM for Java heap.

We will use the library DTD schema listed in Figure 4.4 and the following XPath queries.

Addresses of all publishers (Q1)

Query */library/publishers/publisher/address* is the most trivial one, as it does not conform to the schema definition. There is no *publishers* sub-element under *library* and so the application should return just one empty document (or, rather, a minimal valid document).

Id of author named King (Q2)

Query */library/authors/author*[@*name* = "*King*"]/@*id* is the second simplest one. It references just the *authors* sub-tree of the schema graph and so this sub-tree should be the only part of the schema covered by the resulting documents. For the *genres* sub-tree we should again get the minimal valid content.

Titles of books of genre named fantasy (Q3)

Query */library/genres/genre*[@*name* = "*fantasy*"]/*books/book*/@*title* is moderately complex, as it covers the deeper and wider of the two sub-trees, the *genres* sub-tree.

Titles of books written by author named King (Q4)

The last query we will use is */library/genres/genre/books/book*[@*author_id* = */library/authors/author*[@*name* = "*King*"]/@*id*]/@*title*. It is the most complex one. It references both sub-trees of the schema graph and we should therefore get the biggest number of generated documents.

6.1 A Naive Implementation

We will start with an implementation which follows very closely the definitions and pseudo-codes described in Chapter 4, without any intentional optimizations.

Table 6.1 holds application run-times (in milliseconds) and numbers of generated documents (in parentheses) for the different queries and different values of MC (see Section 4.1.3). Each value was obtained as an average of five consecutive runs. There are two special values:

- OOM (out of memory) - the application failed on an Out of memory exception and
- TLE (time limit exceeded) - the application did not finish in an hour.

	Q1	**Q2**	**Q3**	**Q4**
MC 1	155 (1)	181 (4)	185 (6)	313 (17)
MC 2	154 (1)	237 (14)	1949 (136)	OOM
MC 3	145 (1)	531 (34)	OOM	—
MC 4	160 (1)	1643 (69)	—	—
MC 5	152 (1)	23593 (125)	—	—
MC 6	165 (1)	812509 (209)	—	—
MC 7	150 (1)	TLE	—	—

Table 6.1: Naive Implementation

We examined the generated sets of documents for all cases and confirmed that the results were correct. As we can see, though, the performance of the current implementation is very unsatisfactory. We can take the following actions to improve it:

- relieve our expectations on relevant documents,
- take performance optimizations.

6.2 Algorithm Modifications

We have modified the algorithm itself in two ways, as a trade-of between complete correctness and performance.

6.2.1 Relevant Child Nodes

In the original implementation (an in accordance with Definition 38) we regarded every satisfying sequence node (plus the longest and the shortest of the non-satisfying ones) as relevant, because each of them meant a different situation to a resolver. When examining the generated documents we noticed that in fact some of them represent situations that are very similar to each other and decided to modify this criterion in order to decrease the amount of generated documents

and gain performance. This trade-of comes at a low cost, because we only only loose documents that are not very different from some other documents that we keep.

Definition 47. *Let $G = (r, V, W, E)$ be a non-recursive schema graph and v its node. Then $SatC(S_i, v) = |Satisfying(S_i, v)|$ is the* satisfiability coefficient *of node v for step S_i.*

The satisfiability coefficient of a node v for a step S_i is given as the number of child-nodes of v which satisfy S_i.

Definition 48. *Let $G = (r, V, W, E)$ be a non-recursive schema graph and v one of its nodes. Let S_i be a step of a query and $k \in \mathbb{N}$ a natural number. Then $Satisfying(S_i, v, k) = \{w \in Satisfying(S_i, v); SatC(S_i, w) = k\}$ is the set of all* satisfying child-nodes with coefficient *k.*

While $Satisfying(S_i, v)$ is the set of all child-nodes of v that satisfy step S_i, $Satisfying(S_i, v, k)$ is a sub-set which contains just nodes for which the satisfiability coefficient equals k.

Definition 49. *For a non-recursive schema graph $G = (r, V, W, E)$, its node v and a natural number $k \in \mathbb{N}$ we define the set of all* relevant sequence-nodes with coefficient *k, denoted as $Relevant(S_i, v, k)$, as follows.*

- *Let $Relevant(S_i, v, 0) = \{shortest, longest\}$ where*
  - *$shortest = \{w \in NonSatisfying(S_i, v); |w| = min_{u \in NonSatisfying(S_i, v)}|u|\}$,*
  - *$longest = \{w \in NonSatisfying(S_i, v); |w| = max_{u \in NonSatisfying(S_i, v)}|u|\}$*

  and
- *$Relevant(S_i, v, k) = \{v \in Satisfying(S_i, v, k); |v| = min_{u \in Satisfying(S_i, v, k)}|u|\}$ for $k > 0$.*

Definition 50. *Let $G = (r, V, W, E)$ be a non-recursive schema graph and v its node. Let S_i be a step of a query. Then $\underline{Relevant}(S_i, v) = \cup_{k \in \mathbb{N}} Relevant(S_i, v, k)$.*

As we can see in the new definition, only one satisfying node is included among relevant child-nodes for each valid satisfiability coefficient.

6.2.2 Example

Let us illustrate the concept with an example. Figure 6.1 contains a snippet of a schema graph to represent books. There is a single *book* element node and five different sequence child-nodes.

We will now compare the old and the new definition. Suppose that $S_i = chapter$. Then:

- $Relevant(S_i, v) = \{w_1, w_2, w_3, w_4, w_5\}$ as w_1, w_2, w_3 and w_5 are satisfying and w_4 is the only non-satisfying child-node.
- $\underline{Relevant}(S_i, v) = \{w_2, w_3, w_4\}$ because w_2 is the shortest sequence node with coefficient 1 and w_3 with coefficient 2.

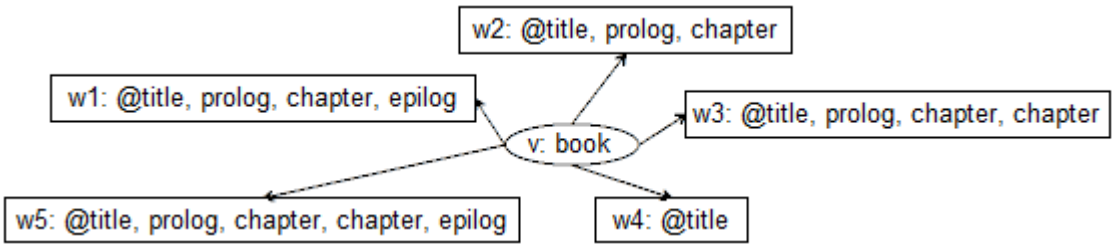


Figure 6.1: The book schema graph snippet

6.2.3 Covering Predicates

The way we define predicate coverage in Definition 39 has a performance drawback. It means that when processing a predicate we have to generate a new document tree for every sub-set of the final context-set of one of the sub-queries. Many of these trees are similar and are discarded later, during the similarity check. For larger sets the number of sub-sets grows very fast, which causes an explosion of intermediary trees.

We have decided to only process one sub-set of each size instead of the whole power-set. This reduces the number of cases that we have to deal with, but, on the other hand, causes a loss of some relevant documents for some complex queries.

6.2.4 Example

Let us demonstrate such a situation with an example. Suppose we have the following two predicates in a single query - $P_1 = [Q_1 = "King"]$, where evaluating the Q_1 sub-query results in nodes $\{v_1, v_2, v_3\}$, and $P_2 = [Q_2 = "King"]$, where Q_2 results in $\{v_2, v_3\}$.

We will now show what happens for sub-sets of size 1. For Q_1 the sub-set we choose is $\{v_1\}$ and we add the following labels to the tree.

- $(eq, v_1, King)$,
- $(neq, v_2, King)$ and
- $(neq, v_3, King)$.

For Q_2 the sub-set of size 1 is $\{v_2\}$, which means that we add the following labels:

- $(eq, v_2, King)$,
- $(neq, v_3, King)$.

There is no correct value-assignment function, because labels for node v_2 conflict with each other, and so we loose the document. On the other hand, if we chose some other pair of sub-sets of size 1, such as sub-set $\{v_2\}$ for both P_1 and P_2, we could get a valid set of labels. Each of the predicates is processed separately, though, and there is no way to guess the right pair of sub-sets without processing them all, which is what we wanted to avoid in the first place.

6.3 Performance Optimizations

We have deduced (and confirmed our conclusions using a profiler) that the main spots worth optimizing are the generator and the similarity sub-modules. Let us now describe each major optimization in more detail.

6.3.1 Fail Fast for Invalid Document Trees

In generator the relevant document trees are built gradually, starting with an empty tree and adding new branches for individual steps of the query. A lot of work can be saved if we can recognize invalid intermediary trees and immediately discard them.

We have added checks for the following basic combinations of contradicting labels:

- (eq, v, s) plus (neq, v, s) for some node v and a string s,
- (eq, v, s_1) plus (eq, v, s_2) for some node v and strings s_1 and s_2 such that $s_1 \neq s_2$,
- $(different, v_1, v_2)$ plus $(same, v_1, v_2)$ for some nodes v_1 and v_2.

6.3.2 Do the Similarity Checks in the Generator

Another way to prune the computation of the generator is to move the similarity checks inside and filter intermediary trees after each step (in the original implementation, the similarity checks were done only after all relevant trees have been fully generated). This reduces the number of cases that we have to deal with, but increases the importance of optimizing the similarity checks themselves, as they are executed more often.

6.3.3 Optimize the Similarity Checks

The most important optimizations were performed on the similarity relation. In the original code we first generated all mappings that were valid with regards to the structure of the trees and then checked if for any of them labels matched as well. For larger trees the total number of structurally correct mappings is very high. In the optimized implementation, conformance of labels is verified already during the construction process, which prunes the computation greatly. We traverse the trees, try to reorder the child-nodes for one of them, and stop invalid computation branches right away, which is achieved via the following checks. Two trees are not similar unless:

- the total counts of labels per type match and
- for each pair of matching nodes:
  - the target schema nodes match,
  - the sets of incident labels match,
  - the sub-trees have the same depth and sizes of individual levels.

6.4 The Optimized Implementation

We run the same test cases using the optimized implementation. The results are available in Table 6.2, which is in the same format as before.

	Q1	**Q2**	**Q3**	**Q4**
MC 1	137 (1)	143 (2)	148 (3)	159 (3)
MC 2	139 (1)	161 (5)	194 (10)	1300 (48)
MC 3	133 (1)	203 (9)	458 (35)	OOM
MC 4	140 (1)	348 (14)	2475 (126)	—
MC 5	137 (1)	601 (20)	TLE	—
MC 6	140 (1)	1180 (27)	—	—
MC 7	139 (1)	7714 (35)	—	—

Table 6.2: Optimized Implementation

Even though the time and memory savings are significant, the results are still not satisfying, especially for more complex queries. Also note the reduction of the numbers of generated documents. It is caused by the two algorithm modifications presented in Section 6.2.

6.5 Steps Priority

Apparently we are unable to fully process data of a certain level of complexity with reasonable demands on time and memory. What we can do instead in these cases is to allow users to give a relative priority for each step of the query. The algorithm would then reduce the number of different cases relevant with regards to steps with lower priorities in favour of steps with higher. This could even be used as a means to deliberately delimit the processing of uninteresting steps in complex queries to avoid being overwhelmed by very large numbers of generated documents.

We will only deal with a simplified case in this thesis and leave the research on other options as a suggestion for future work. We will allow users to (optionally) mark one of the steps of the query as important. The denoted step will be processed fully and all other steps will be suppressed. For suppressed steps only a single satisfying child-node will be regarded as relevant. This should cause a significant reduction in the number of generated documents and hence also in the application run-time.

6.6 Example

Let us return to the schema graph snippet shown in Figure 6.1 and step $S_i = \mathit{chapter}$. If S_i was regarded as the interesting step, the set of relevant child-nodes would stay the same as in the original example. Otherwise, we would only accept node w_2 as relevant, because it is the shortest of the satisfying child-nodes.

6.7 The Final Implementation

We have chosen a few interesting steps for the two more difficult queries (i.e. $Q3$ and $Q4$), ran the tests for them and presented the results in Table 6.3. The format is again same as before.

The numbers in square brackets next to the query identifiers denote the indices of the relevant steps. For example, $Q3$ [3] means the third step of query $Q3$. In other words:

- $Q3$ [3] stands for $genre[@name = "Fantasy]$,
- $Q4$ [3] for $genre$,
- $Q4$ [5] for $book[@author\_id = /library/authors/author[@name = "King"]/@id]$ and
- $Q4$ [6] for $@title$.

The more complex a step is, the more demanding the computation and also the greater the number of resulting documents. This is especially true for steps with sub-queries.

	Q3 [3]	**Q4 [3]**	**Q4 [5]**	**Q4 [6]**
MC 1	162 (3)	166 (3)	161 (2)	182 (2)
MC 2	180 (6)	195 (5)	343 (14)	189 (2)
MC 3	223 (10)	286 (7)	1603 (53)	186 (2)
MC 4	372 (15)	540 (9)	108910 (157)	187 (2)
MC 5	675 (21)	1051 (11)	TLE	182 (2)
MC 6	1269 (28)	1926 (13)	—	187 (2)
MC 7	8612 (36)	8270 (15)	—	186 (2)

Table 6.3: Final Implementation

While still not ideal, these results are far better than what we got with our initial implementation. The optimization obstacles are partially inherent to the problem itself, as the space of potentially interesting documents is large and grows rapidly with increasing MC values and query complexity.

6.8 Comparison

The measurements presented in the previous sections are graphically summarized on the following bar charts. Figure 6.2 depicts application run-times per different MC parameter values and implementation versions for query Q3. Figure 6.3 depicts the same data for query Q4.

6.9 Advanced Priority Settings

Let us yet shortly outline and discuss some of the options for more advanced steps priority settings. The ideas captured in this section were not incorporated into the implementation.

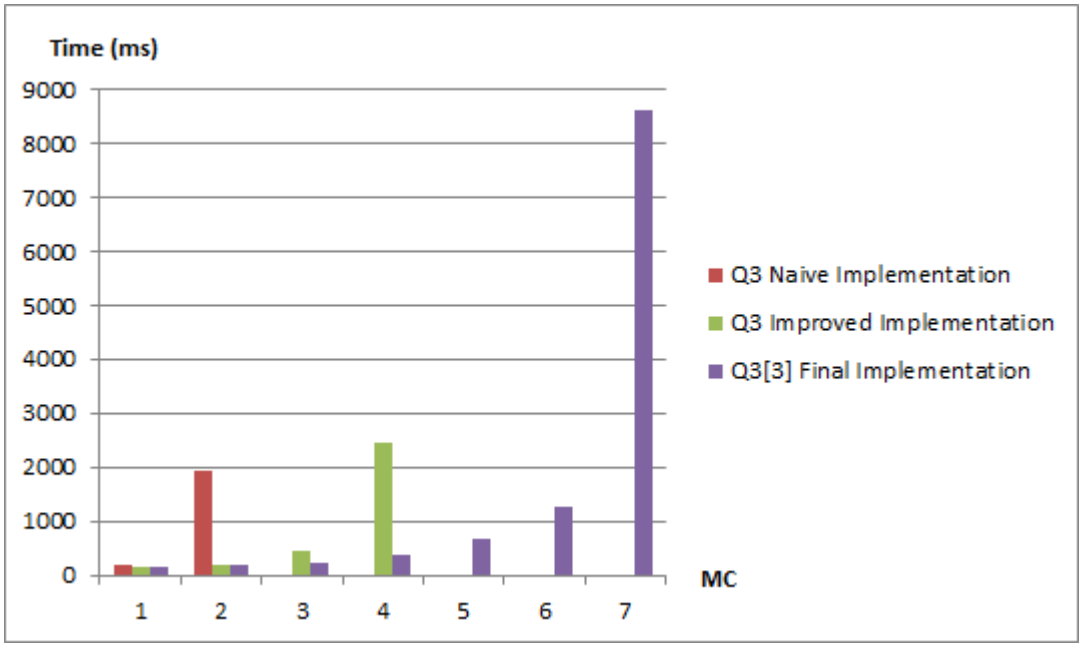


Figure 6.2: Application run-times for query Q3

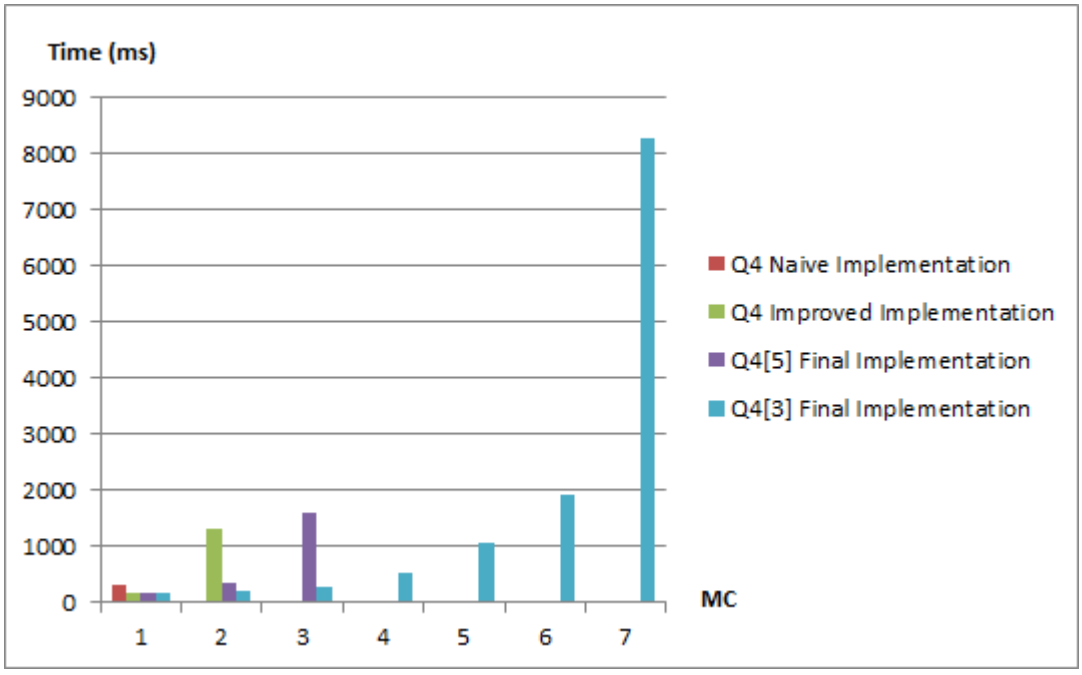


Figure 6.3: Application run-times for query Q4

More than one important step

The algorithm could be modified to allow marking multiple steps of the query as important. The amount of generated documents would grow quickly with each new important step, though. In cases when we need to cover more than one step exhaustively, it might be better to run the application multiple times, each time with a different step marked as important, and gather the results.

Important steps in sub-queries

In the final prototype version only steps of the query itself can be selected as important. When a step with predicates is marked, each step in all its sub-queries is considered important as well. Extending the marking for sub-queries would allow a more fine-grained but also more complex set-up. We could reduce the increase of configuration complexity by choosing smart default values (e.g. by default, the step could be marked as important with all its sub-queries).

Detailed priorities

Currently we can only choose one of two different priority values for a particular step - important (all relevant child nodes are considered) and suppressed (only one of them is considered). These two strategies represent opposite extremes. But we might also want to choose something in between.

Users could be allowed to associate each step with a fraction to denote the percentage of relevant child nodes to consider. Different strategies to choose the target amount of nodes could be employed. Among others:

- **uniform selection** (sort all relevant child nodes by the lengths of the represented sequences and then omit the expected number of them so that the gaps are uniformly distributed),
- **probabilistic selection** (iterate over all relevant child nodes and omit those for which a randomly generated value does not exceed the target threshold).

In case of probabilistic selection the application should accept a fixed random seed as an (optional) argument so that we could get the same results when running the tool multiple times on the same input.

Chapter 7

Conclusion

The goal of this thesis was to design an algorithm to generate synthetic XML documents, which would be reasonably complex yet easy to configure. We have studied and evaluated a wide range of existing approaches. It turned out that, for general-purpose generators, the requirements for a large degree of configurability and for the ease of configuration contradict with each other. We have decided to sacrifice general applicability and focus on generating documents specifically for the purpose of covering the resolution of a given XPath query.

The main part of our effort consisted of research on constraints that can be derived from a given DTD schema instance together with an XPath query. Based on the results we have then designed an algorithm that only required one additional argument (*mc* - the maximal cardinality of repetitive DTD operators, as introduced in Section 4.1.3).

As a next step, a prototype was implemented to demonstrate and validate our conclusions. It turned out that the original algorithm was too heavyweight and we had to apply several optimizations. One more parameter was needed for performance reasons (*stepNo* - the index of an important step of the query, see Section 6.5). Experimental results presented in the last chapter illustrate that, while there still is space for improvement, the time and space demands of the optimized version are promising.

As far as we have found out there are currently no other solutions that would specialize in generating XML documents to cover XPath queries. Paper [1], mentioned in Section 3.1.1, only briefly describes a proposed solution, and no robust implementation is offered. The authors touch some schema and query constructs that we did not deal with, such as data types of elements and attributes and more advanced XPath axes.

Despite all our effort, there still remains space for further research, especially in the following areas.

- **Performance** - we have shown that the proposed solution works well for queries up to a certain amount of complexity. For larger queries, performance issues have to be dealt with. We have outlined some optimization approaches in Chapter 6. Nevertheless, there still is space for further research in this area.
- **DTD constructs** - our solution supports a large part of the DTD specification. We have used conclusions of third-party research regarding current

rates of adoption of individual DTD constructs in real software projects as a base for decisions as to which of them we should focus on and which should be left out. Still, adding support for the rest of them would be beneficial - especially for recursive schemas and the text content of elements.

- **XPath constructs** - in the current version the algorithm only supports elementary XPath features - the child and attribute axes, element/attribute names as node tests and primitive forms of predicates. On these basic constructs, properties of the proposed solution have been demonstrated. We believe that extending support for more advanced XPath features would be a viable task for a future work.

Bibliography

[1] C. DE LA RIVA, J. GARCÍA-FANJUL, J. TUYA. *A partition-based approach for XPath Testing.* In Proceedings of International Conference on Software Engineering Advances, 2006.

[2] T. J. OSTRAND, M. J. BALCER. *The Category-Partition Method for specifying and generating functional tests.* Communications of the ACM, vol. 31. no. 6, 1988.

[3] C. DE LA RIVA, J. TUYA, J. GARCÍA-FANJUL. *Testing XPath Queries using Model Checking.* System Testing and Validation (STV), Potsdam, 2006.

[4] D. S. KIM-PARK, C. DE LA RIVA, J. TUYA, J. GARCÍA-FANJUL. *Generating Input Documents for Testing XML Queries with ToXgene.* Proc. of the 3rd IEEE Testing: Academic and Industrial Conference, Fast Abstract Track, 2008.

[5] A. BERTOLINO, J. GAO, E. MARCHETTI, A. POLINI. *Automatic Test Data Generation for XML Schema-based Partition Testing.* Second International Workshop on Automation of Software Test (AST'07), 2007.

[6] *Oxygen XML editor* [online] Available at http://www.oxygenxml.com/.

[7] *ToXgene - the ToX XML Data Generator* [online] Available at http://www.cs.toronto.edu/tox/toxgene/.

[8] C. DYRESON, H. JIN. *A Synthetic, Trend-Based Benchmark for XPath.* DASFAA Workshops, volume 5667 of Lecture Notes in Computer Science, page 35-48, 2008.

[9] M. FRANCESCHET. *XPathMark - An XPath benchmark for XMark generated data.* International XML Database Symposium (XSYM), 2005.

[10] *XMark - An XML Benchmark Project* [online]. Available at http://www.xml-benchmark.org/.

[11] I. MLÝNKOVÁ, K. TOMAN, J. POKORNÝ. *Statistical Analysis of Real XML Data Collections.* COMAD 2006: Proceedings of the 13th International Conference on Management of Data, 2006.

[12] *World Wide Web Consortium (W3C).* [online] Available at http://www.w3.org/TR/REC-xml.

[13] *Extensible Markup Language (XML) 1.0 (Fifth Edition): W3C Recommendation 26 November 2008.* [online] Available at http://www.w3.org/TR/REC-xml.

[14] *XML Path Language (XPath) Version 1.0: W3C Recommendation 16 November 1999.* [online] Available at http://www.w3.org/TR/xpath/.

[15] *The Eclipse Foundation open source community website.* [online] Available at http://eclipse.org.

[16] *Oracle: Java SE.* [online] Available at http://www.oracle.com/technetwork/java/javase/overview/index.html.

[17] *Maven - Welcome to Apache Maven.* [online] Available at http://maven.apache.org/.

[18] *JUnit - A programmer-oriented testing framework for Java.* [online] Available at http://junit.org.

[19] *Java Compiler Compiler - The Java Parser Generator.* [online] Available at https://javacc.java.net/.

[20] *JDOM.* [online] Available at http://jdom.org.

[21] *Apache Commons - Wellcome to Apache Commons.* [online] Available at http://commons.apache.org/.

[22] *Log4j 2 Guide - Apache Log4j 2* [online] Available at http://logging.apache.org/log4j/.

[23] *Visual VM - All-in-One Java Troubleshooting Tool* [online] Available at http://visualvm.java.net/.

[24] *Git* [online] Available at http://git-scm.com/.

Appendix A - Contents of the CD

The CD attached to this thesis has the following structure:

- **text/**
  - **thesis.pdf** (a PDF version of the thesis),
- **project/** (Maven project of the prototype - contains the POM file, all source files and all test cases),
- **install/** (installation of the prototype)
  - **README.txt** (end user documentation),
  - **xmlgen-1.0.jar** (the application JAR file),
  - ***.jar** (JAR files of 3rd party libraries),
  - **logconfig.xml** (the Log4J configuration file),
  - **example/**
    - **queries/** (batch files to run the application on sample data, together with the corresponding expected outputs),
    - **library-schema.dtd** (a sample DTD schema),
    - **output/** (a directory to store sample output),
- **doc/** (technical java-doc documentation).