Generating Schema Information for Views over Semistructured Data

Sebastian Boßung

(Includes post-submission fixes of spelling and punctuation.)

Submitted in partial fulfillment of the requirements for the degree
Master of Science in Information and Media Technologies.

supervised by
Prof. Dr. Joachim W. Schmidt (STS)
Prof. Dr. Helmut Weberpals (TI6)
Dr. Hans-Werner Sehring (STS)

TU Hamburg-Harburg
Department Softwaresysteme

Abstract

The concept of views is well understood and widespread in the world of relational databases. As databases for semistructured data became evermore popular, views were carried over to these. Due to the richer structure of the semistructured views, schema generation is more complicated than in the relational case.

This thesis presents some theoretical background on schema generation for views over semistructured data as well as an algorithm and its prototype implementation. A hybrid approach is taken: Ideally, all schema information is generated from the query expression (XQuery) itself. However, for different reasons this is not always possible; schema information is then induced from the result data.

Declaration

I declare that:
this work has been prepared by myself,
all literal or content based quotations are clearly pointed out,
and no other sources or aids than the declared ones have been used.

Hamburg, 11 July 2004
Sebastian Boßung

I would like to thank Professor Joachim W. Schmidt of STS for supervising this thesis and being very helpful with finding a topic. Thanks also go to Professor Helmut Weberpals of TI6 for being the co-corrector.

Dr. Hans-Werner Sehring of STS was very patient in providing advice during the project as well as on the thesis itself. Discussions with him guided the project into the shape it has now.

Thanks also go to Verena Pannecke and Oliver Witt for answering all sorts of questions about English.

Contents

1 Introduction **6**
1.1 Motivation . . . 6
1.2 Related Work . . . 7
1.2.1 Views . . . 7
1.2.2 Query Languages for Semistructured Data . . . 8
1.2.3 Schema Generation . . . 9
1.3 Structure of this Thesis . . . 10

2 The Problem **11**
2.1 Problem Description . . . 11
2.2 Definition . . . 11
2.3 Example . . . 12
2.4 Differences to Schema Induction from Instances . . . 12

3 Background **15**
3.1 XML Query . . . 15
3.2 XML Schema . . . 18
3.3 Induction Algorithms . . . 20
3.4 The eXist Database . . . 21
3.4.1 Overview . . . 21
3.4.2 The Query Processor . . . 22

4 Design **25**
4.1 Choice of Query Language . . . 25
4.2 Schema Generation Options . . . 27
4.3 Interaction with the Query Processor . . . 28
4.4 System Structure . . . 29

5 Implementation of Necessary Infrastructure **31**
5.1 Views . . . 31
5.2 Schema Store . . . 32
5.3 Extensions of the Interactive Client . . . 32

6 Implementation of Schema Generation **35**
6.1 Datastructures . . . 36
6.2 Generation of Fixed Parts . . . 37
6.2.1 External References . . . 38
6.2.2 Constructors . . . 38
6.2.3 Finding Simple Types . . . 40
6.2.4 Selections . . . 41
6.2.5 Functions . . . 42

6.2.6 Conditions and Iteration . . . 44
6.2.7 Variable References . . . 46
6.3 Cardinalities in Fixed Parts . . . 47
6.3.1 Literal Value Sequence . . . 48
6.3.2 Single-valued Sequences . . . 48
6.3.3 XPath: Location Steps . . . 48
6.3.4 User defined Functions . . . 49
6.4 Generation of Induced Parts . . . 49
6.4.1 Induction Assumptions . . . 49
6.4.2 Algorithm . . . 50
6.4.3 Implementation . . . 50
6.5 Schema Simplification . . . 53

7 Evaluation **56**
7.1 Test Cases . . . 56
7.1.1 XQuery Usecases . . . 56
7.1.2 Own Views . . . 61
7.2 Results . . . 64
7.2.1 Covered Parts of XQuery . . . 64
7.2.2 Quality of the Generated Schemas . . . 64

8 Conclusions **66**
8.1 Usefullness of the Prototype . . . 66
8.2 Observations . . . 66

9 Future Work **68**
9.1 Enhancements to Schema Generation . . . 68
9.2 Applications of Schema Generation . . . 69
9.3 System Context . . . 69

A Glossary **74**

B Details on Testcases **76**
B.1 XQuery Usecases . . . 76
B.1.1 Primitive Restructure . . . 76
B.1.2 Nesting of variable bindings . . . 78
B.2 Own Cases . . . 80
B.2.1 Summary of bills by day-of-week . . . 80

C The CD **84**

List of Figures

1.1 XMAS example 9
1.2 Induction example 9

2.1 Illustration of view definition 12
2.2 View example 13

3.1 XQuery example 18
3.2 Comparison of global and local definitions in XML Schema 19
3.3 Induction automata example 20
3.4 Query execution 23
3.5 Core of the query processor 24

4.1 Schema generation contexts 29
4.2 System structure 30

5.1 Class diagram of the ViewService 31
5.2 SchemaService class 33
5.3 New query dialog of the interactive client 33

6.1 General workflow of the generation process 36
6.2 Schema representation 37
6.3 Classes involved in XPath selections 41
6.4 Full for-loop syntax 46
6.5 Modelling of sequence values. 48
6.6 Cardinalities for return types of user defined functions 49
6.7 Class diagram of schema induction 51
6.8 Collaboration diagram of schema induction 51
6.9 Unsimplified schema. 54
6.10 UML diagram of schema simplification classes 54

Chapter 1

Introduction

1.1 Motivation

It is often desirable to have a schema definition for an XML document. This has many benefits for the user who works with the document. It can easily be handled by generic editors that still only let the user input valid data. More knowledge of the document's structure also makes smarter ways of displaying possible. Another important factor is that many existing tools further down the process chain rely on a schema definition to work. Prominent examples of these are the generation of a set of classes that can represent a document instance adherent to the schema (e.g. JAXB in Java) and various data exchange interfaces at system boundaries (for example Web Services to name a buzz-word). Finally, having schema information available makes the formulation of queries much easier.

A concept that is well known and understood in the field of relational databases is the concept of views. To create a view, a query expression over the existing relations is used to create a new relation on the fly, who's contents is the query result. The data in this relation is not a copy of the original data but a reference to it that changes as the original data is updated. Views have a variety of uses, access restrictions and restructuring of data for easier retrieval are two examples. A view can be treated like an ordinary relation[1] making the concept of views orthogonal to other concepts in the database system.

Just as with relational databases the user might desire to restrict access or restructure data in XML databases. This need leads to the concept of views being introduced into XML databases. An introduction to views for semistructured data can be found in "Views for Semistructured Data" [Ab97].

In the relational world obtaining schema information for the view is not very difficult. Just like a relation the view is flat and fully structured allowing relatively simple generation of schema information. Speaking in SQL terms, you can get most of the schema for a view by looking at projection and table clause. In particular, there is no need to consider the data. If the underlying database is not relational but allows semistructured data, views inherit properties from the documents they are based on. A little thought reveals that one also needs to consider the data when generating the view's schema. As the structure of the documents the view gets its data from can be anything (this is one of the key points of a semistructured database), it is impossible to predict which nodes will be in the view's result. One cannot obtain

[1] Not quite. There are some restrictions that are ignored here for simplicity of argument.

sufficient structural guarantees by investigating the view's definition. More details on this can be found in chapters 2 and 4.

There are plenty of algorithms available to induce a schema from semistructured data. As the value of a semistructured view is again semistructured data, it is tempting to use these algorithms. Generating the schema from the query expression of the view gives some benefits. Most prominently it allows correct typing on simple type level[2] (i.e. not having to guess simple types such as decimal numbers based on instance data) and allows realistic specification of cardinalities. One can for example determine in many cases that a list will have a certain number of elements or even notice that an element is really optional even though it appears in all instances presented.

1.2 Related Work

1.2.1 Views

Views are present in many systems including relational and semistructured databases. Relational views are a much reasearched topic, related functionality and problems are very well understood. There is a wide range of literature about them, a user-centric introduction can be found in "Views (Virtual Tables) in SQL" [EN00]. As the concept of views is already carried over to the semistructured domain in other literature, the discussion of further sources on relational views and their influence on semistructured ones is omitted here.

One of the key purposes of a semistructured database is to allow storing of semantically related data with diverse structure. To make use of this collection of data, any application would need to understand all the different structures in use. This clearly leads to lots of duplicate code (at best) when more than one application is using the database. It also moves concerns under the maintenance of application developers which they should not have to care about, namely the unification of data. Instead this unification should be done at database level. An obvious choice here is to create a view to perform this unification, as is argued in "On Views and XML" [Ab99]. In 1999 when this article was written, the world of query languages for XML was still very diverse. The author discusses the then already emerging XSLT, but goes on to mention that his own language should be based on the OQL.

The authors of "Views for Semistructured Data" [Ab97] also deal with the issues involved with views in semistructured databases. They present a view specification language. Note that this is different from a query language. Query languages at that time were mainly selection languages that retrieved nodes from the database. These nodes did not have any semantics outside the context of this database. The authors note, that this is insufficient for a view because views will need to create what they call "small databases" (in XML terms: documents that make sense independently of the orignal database). The authors also note that a key difficulty of introducing a view mechanism into a semistructured database is the lack of schema (in contrast to relational databases). This complicates the definition and evaluation of the view, and also makes use of the view more difficult. Finally the article gives strategies for evaluations of the view, namely materialization and query rewriting. The former approach stores of the view's result (allowing the use of a standard query processor), while the latter changes the query defining the view to also incorprate the query posed to the view.

[2]Referential typing on complex structure level is more easily achieved. The implementation chapter will clarify why this is the case.

The two approaches of query rewriting and view materialization are discussed in [Ab97]. For each option the problems specific to semistructured data are discussed. There also is much additional literature on issues around materialization of semistructured views, two articles on the topic are [Gl97] and [AMV98]. A major advantage of materialized views is the possibility to use a standard query processor. It will be able to run queries on the view's nodes just like on the other ones in the database because it simply sees no difference. Materialized views also offer performance benefits if a view is queried a lot more than updated.

A central point of the implementation of views is the choice of query language. Both [Ab97] and [Ab99] discuss requirements for such a language. It mainly has to support creation of new structures of nodes for the view (i.e. nodes that do not appear in the database, for example as a wrapper around a query result) as well as the import of nodes from the database into the view. The two articles discuss their own query language. Today there are quite a few languages available to choose from, seeming to make the creation of yet another language unnecessary, see the next section.

1.2.2 Query Languages for Semistructured Data

A variety of query languages has been proposed to deal with semistructured data and many of them would be suitable for defining a view. "Features and Requirements for an XML View Definition Language" [BLP98] lists key requirements to such a language: It should produce semistructured output, should be closed under composition (i.e. the result to queries to views should be completely describable in the query language), and should be able to handle order (preserving it or changing it). Of course it should also be efficiently implementable in a semistructured database, otherwise the defined views would be of little use. Different languages conform to these criteria to varying degrees.

It seems important to use a well-known language to avoid a mismatch between view definition and ordinary query formulation (that might be done in another language otherwise), as is argued in [BLP98]. This brings to mind the W3C languages XPath [W3C99], XML Query [W3C03], and XSLT [W3C01]. These three languages are path based, meaning that they use XPath expression for the selection of nodes. Views defined in these languages are often called views by extents. The query expression gives a set of nodes (e.g. by selection via XPath) and a transformation that is applied to these objects. The important point is the explicit selection of the nodes. See [AMV98] for more details.

There are also pattern based languages, for example Xcerpt [BS02]. Pattern based languages select nodes by giving a pattern that nodes (and their children) have to match. Xcerpt is conceptually similar to XMAS, an example of which is shown in figure 1.1. Views defined by pattern based languages are also called views by intent. These languages essentially give a transformation to apply to the set of nodes. There is no explicit select (i.e. no navigation through a hierarchy) to select the nodes. Rather the nodes are obtained from the database by specifying properties they should match in patterns. Section 4.1 gives some more details on pattern based languages.

When generating schema information for a view, one wants to create a schema that the view's result document will conform to. Thus one might consider creating the schema from the output directly, without looking at the view. This induction of schema information has indeed be subject of much research, for example in "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases" [GW97], "XTRACT: A System for Extracting Document Type Descriptors from XML Documents"[Gra00] and "On Structural Inference for XML Data" [Wo03]. These articles discuss algorithms that create some kind of schema informa-

```
CONSTRUCT
    <answers>
        [ <addison_wesley_book> $T
            [ $A ]
        </addison_wesley_book> ]
    </answers>
WHERE
    <book>
        $T: <title/>
        $A: <author/>
    </book>                                         (1.1)
```

Figure 1.1: XMAS example

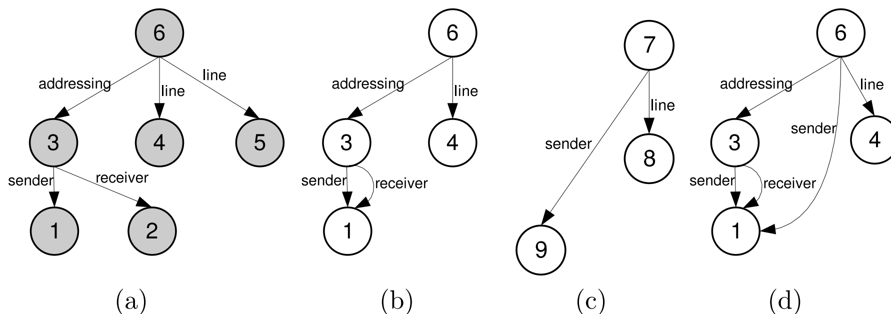


Figure 1.2: Example of an induction automaton. (a) A sample document (in reality several); (b) and (c) two generated automata (instance data for second one not shown); (d) the final automaton.

tion for various types of semistructured data, [Wo03] compares and combines several. As will be explained in section 2.4, the problem of generating a schema for a view is somewhat different (and neither a sub- nor a super-problem) from that of inducing from instances. Nevertheless, inductive algorithms are important as the presented prototype can sometimes not create schema information for the whole view directly. The induction algorithms usually work by first learing automatons from example data sets. They are then simplified to some minimal form and converted into a schema description language (process illustrated in figure 1.2). Induction algorithms are discussed in detail in section 3.3, the induction approach of the prototype is explained in section 6.4.

1.2.3 Schema Generation

There also is existing work on generating schema information from the view's definition. “View Definition and DTD Inference for XML” [LPV99] proposes an algorithm to infer DTDs for views over XML data. The prototype presented there allows the definition of views over XML documents and generates DTDs for the views. The view definition language used, XMAS, is pattern based language. Please see section 4.1 on what such a language is. The output of the view is structured by using templates that give parts of XML documents which are repeated for every binding of a variable. The generated schema information is in DTDs, which does not allow much detail in both types (on simple as well as complex type level) and cardinalities. Nodes from the

input document are selected via a sample structure they have to match. This will not allow for documents being of diverse structure, which is common in semistructured databases. The XMAS language has the advantage of simplifying schema generation, because the user specifies a structure in the CONSTRUCT clause that is very similar to the schema.

1.3 Structure of this Thesis

This thesis first briefly describes the motives behind tackling the problem presented and also gives an initial fuzzy problem statement in the next section. Also in this chapter, related work with its similarities and differences is discussed. In chapter 2 the problem is described in greater detail and defined more formally. An additional background chapter is devoted to the underlying technologies, namely the used query language, generic induction algorithms and the database into which the prototype is implemented. Chapter 4 gives a general overview of the implemented prototype in its surroundings (i.e. the eXist database). It also breaks down the overall problem into smaller ones that have to be solved in order to implement schema generation from the query. The implementation is discussed in greater detail in chapters 5 and 6, the latter talks about the implementation of schema generation in case it can work directly from the query expression and also when it cannot. The evaluation chapter details how the capabilities of the prototype were assessed, while the following chapter draws some conclusions both from what was found during the evaluation as well as from general observations made during the whole project. This thesis concludes with a chapter on future work that proposes enhancements to the schema generation algorithm as well as applications for it.

Chapter 2

The Problem

Summary. *This thesis deals with the generation of schema information for a view over XML data. This chapter tries to further clarify both the problem at hand and what is hoped to be achieved as a solution.*

2.1 Problem Description

In an XML database documents are usually stored in collections. These are similar to directories in filesystems, as they can contain documents or other collections. A query is usually not executed over one document, normally not even over a named set of documents, but in the context of a collection. This is very different to the approach in relational database systems where the relations in the query are explicitly named.

As a collection can contain arbitrary documents (they are stored without reference to their schema, schema information might not exist at all), a query is generally not executed over a fixed set of schemas. The query can be written in such a way that its result is again a well-formed XML document (see section 2.3 below for an example). To enable this result to be treated just like an ordinary physical document in the database, the notion of views (see section 1.2.1) is introduced. A view is thus defined by three things: The query that is used to determine its contents, the context of the collection in which the query is executed, and the place in the database where the result appears as a document.

It is important to point out that the generated schema should describe the view. That is, the generated schema should be invariant if generation is repeated (i.e. data does not change but two generations happen a while appart). It would be preferable if the schema was also invariant to changes to the data, but this will not always be realizable, as documents with completely new structure can be introduced.

Generating schema information for the result document of the view is the problem dealt with in this thesis. To increase the quality of the generated schema, it is desirable that there is schema information available for all the underlying documents (see section 8.2). Otherwise computing the schema involves much guess-work.

2.2 Definition

A **collection** C is a set that contains XML documents or other collections. An XQuery expression χ can be executed with input C to yield a result $r := \chi(C)$. If χ is constructed such that r is again a well-formed XML document β, a **view** V is

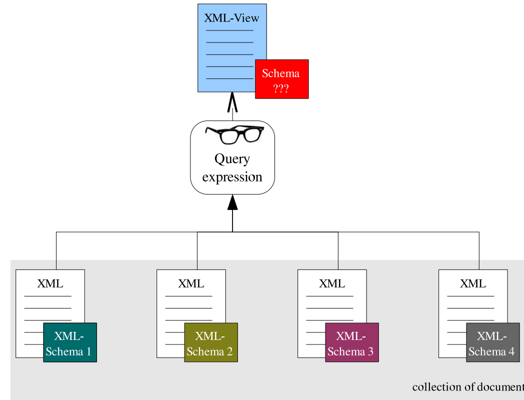


Figure 2.1: A view's definition. The document 'XML View' will again live in a collection, but this is not depicted here.

defined by $\beta := r = V(\chi, C, P)$, where C is the input data for the query (also called the context) and P is the collection that ontains the document β.

Let $S = S(\gamma)$ denote the **XML Schema** of document γ as defined in [W3C02]. Then the schema S_β of above result β can be expressed as $S_\beta = S(\beta) = S(V(\chi, C, P))$. The problem is now to compute the **generated schema** S_β given χ, C, and P. The generated schema must be stable for an invariant database. That is it must have the property that for two schemas $S^*_\beta = S(V(\chi^*, C^*, P^*))$ and $S'_\beta = S(V(\chi', C', P'))$, which are generated at different times but over identical views, $S^* = S'$ holds. Identical views means that $\chi^* = \chi'$, $P^* = P'$ and in particular the data does not change: $C^* = C'$.

Note that P must not be a subset of C to avoid infinite recursion. Infinite recursion does not necessarily have to occur. Whether it will is however undetectable as XQuery is Turing-complete (see [Kes]).

2.3 Example

Figure 2.2 shows an example of the use of a view. The view operates on XML instance documents—only one of which is shown—by running the XQuery given over them. This query expression again creates XML output that is returned as the view's contents.

2.4 Differences to Schema Induction from Instances

The problem of generating a schema for a view appears to be exactly the same as generating a schema from instance documents. After all, you could just take the query result and run a schema induction algorithm. Indeed the two problems are related and the presented prototype resorts to schema induction for parts of the schema when all else fails, but there are important differences.

The schema that is created by induction (S_I) is neither a super-schema (i.e. less specific) nor a sub-schema of that created from the query (S_C). Here is why:

- Induction algorithms "learn" the schema by looking at instance documents. They usually generate some kind of automaton that accepts exactly the pre-

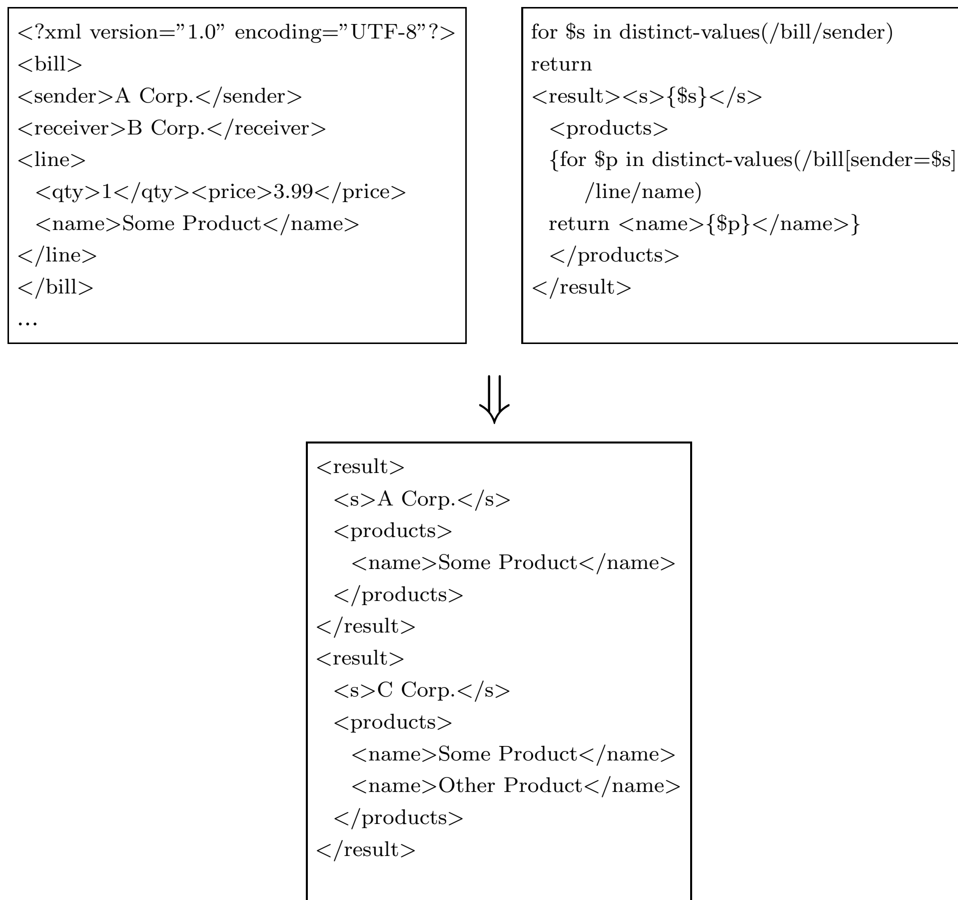


Figure 2.2: Sample document (left) and XQuery (right) to define a view that lists products per sender. Result (bottom). The point is to find the schema for the result.

sented instances. As the corresponding induced language is usually too strict, the automaton is then simplified by heuristics to reflect what the user would expect for a schema (e.g. cardinalities of 0..1783 are weakened to 0..unbounded). Unforunately, this process requires a few[1] input documents, but there is only one (the query result) available here. This would then make it necessary to use rather coarse heuristics[2] which discard much of the precise information.

- Consider a query that is run over multiple documents and returns a sequence of different elements. As the order in which the documents are processed is not specified, it is not difficult to construct cases where the order of the elements in the sequence will change. However, the schema must certainly be stable in this case, but as S_I violates this property, further heuristics would have to be introduced, blurring S_I even more. This is especially bad, because during induction the information about which elements might be subject to such a change in order is already lost. Thus technically one must not induce any XML schema <sequence> particles[3] at all, but use <choice> instead. Also see section 3.2 for more details on XML Schema.

- Much of the information in the query result is just a copy of information in the base-documents. This implies that usually at least some of the elements will also be copies, sharing schema information. The induced schema S_I looses this information as it generates new types for elements that are in reality of known type. This significaly weakens the relation between base-documents and view, as it discards schema information without any need to do so.

- A similar argument applies to attributes. Here not only attributes that are copies of existing ones can be typed correctly in S_C. As XQuery is a typed language, even newly introduced attributes with computed value can be declared with correct type. The algorithm that induces S_I might be able to guess a type based on the contents, but this choice is not certain. It would also be based on a too weak input set as there is only one instance document available.

The differences outlined above show, that it is very much worthwhile to make an effort and compute the greatest possible portion of the schema from the query.

[1] Few is usually from 10 to about 100.

[2] This is actually what this prototype does if it is unable to compute parts of the schema from the query.

[3] Particles are used to define the contents of a complex type in XML Schema. Also see the glossary in appendix A, the full specifications can be found in [W3C02].

Chapter 3

Background

Summary. *The prototype presented with this thesis was not implemented from scratch, but builds on a variety of previously existent techologies. These are namely: The W3C recommendations XML Schema and XML Query, schema induction algorithms in other literature and the open-source XML database eXist. All four are introduced in this chapter.*

3.1 XML Query

The language XML Query is—as the name suggests—a language to pose queries against XML data. As the reader will already have guessed, it is commonly abbreviated XQuery. It is currently a W3C working draft that is nearing completion[1]. This section will give a very brief introduction to the parts of XQuery that are used in this thesis. A more complete description can be found either at the W3C [W3C03] or in an introduction by Philip Wadler [Wad02].

XQuery is a functional language used to query XML data. It attempts to fulfill the needs of two comunities: Those seeing XML as documents (largely coming from the HTML or plain SGML worlds) and those treating XML as data (mainly in semi-structured databases and data exchange between systems). XQuery uses XPath as a technique for selecting nodes. Note that the XPath specification used in XQuery is version 2.0, the important difference being that version 1.0 worked on node sets[2] whereas version 2.0—and thus XQuery—works on node sequences[3]. Also note that the return type of any expression in XQuery is a sequence (even though many of these only contain zero or one value).

Some important XQuery constructs are described below. You can find a more complex query that uses many of the syntax elements from this section in figure 3.1.

- *Constructors:* XQuery allows the construction of new nodes (elements, attributes, text, comments and processing instructions). The syntax for this is the same that would appear in an XML document which contains the respective node. The following code will for example construct an element node called "result" with an attribute called "name" (and value "first result"). The element contains the text "123":

[1] Current version of 12 November 2003, last call working draft.
[2] Node sets are sets of nodes, they are unordered and do not contain duplicates.
[3] Sequences have order and may contain duplicates.

```
<result name="first result">123</result>                    (3.1)
```

- *Enclosed expressions:* These are closely related to constructors. Curly braces "{" and "}" are use to disambiguate expressions nested inside constructors from literal text. Consider this example:

```
<result>{ $a }</result>                    (3.2)
```

  The contents of the "result" element will be the value of variable a. By contrast, without the curly braces the contents of the "result" element would be the text literal "$a".

- *Path expressions:* XQuery uses XPath for selecting nodes. XPath expressions are made up of several steps (called location steps) that start from a context node, typically from the root of the document. The different steps are delimi-nated by forward slashes "/".

```
/bill/sender[@name="Peter"]                    (3.3)
```

  This will start from the document root and select the element bill (which is the first element) and then descend to its child sender. All sender elements that are children of a bill element are returned if they have a name attribute (denoted by the "@") with value "Peter"'. XPath is not limited to navigating from an element to its children. It allows different navigation paths called axes. Important axes besides the "child" axis are "descendent-or-self" (the node or any of its descendents), "parent" or "attribute" (abbreviated by "@" in the example). XPath also allows wildcards in node and attribute names. For example

```
    /descendent-or-self::*                    (3.4)
```

  simply selects all nodes in the document. Results of path expressions are—of course— sequences of nodes.

- *FLWOR expression:* Iterating over a sequence of nodes is very important in XQuery. First selecting a sequence of nodes via a path expression and then iterating over it to perform some kind of transformation or computation is common practice. The syntax of the FLWOR (for-let-where-order by-return) expression is illustrated in this example:

```
for $bill in /bill
let $name := $bill/sender/@name
where $bill/@total-sum > 100
order by $bill/@total-sum
return
    <sender>{ $name }</sender>                    (3.5)
```

  This looks very complicated at first glance. Here is a breakdown: The first line is an iteration over all nodes in "/bill" (an XPath expression returning all bills), where in each iteration the current node is bound to the variable $bill. The next line then binds an additional variable $name to the name of the sender of the bill. The where clause filters the sequence over which to iterate

to only contain bills for which the total-sum (an attribute of the bill element) is over 100. Order by sorts the bills by their total-sum attribute. Finally return gives the expression that is evaluated for every iteration with the new variable bindings (here returning a new sender element with the name of each sender).

The output could look like this:

```
<sender>Peter</sender>
<sender>Texaco</sender>
<sender>Real Stationary</sender>
<sender>Infinite Computers Ltd.</sender>        (3.6)
```

Note that this is not a well-formed XML document!

- *Functions:* XQuery has many built-in functions but also allows users to specify their own. Calling functions is quite simple:

```
<overall-sum>{ sum(/bill/@total-sum) }</overall-sum>        (3.7)
```

This example sums the totals of all bills and puts the result as the contents of the overall-sum element. Note that the sum function works on a sequence of values. Available functions include: sum, count, avg, max, min and many more. User defined functions also are possible:

```
declare function local:totalValue($bill as element) as xs:double {
  sum(for $l in $bill/line
        return ($l/qty * $l/itemPrice))
};        (3.8)
```

The function is declared in two parts: The signature and the body expression. The signature gives the namespace (local in this case), the function's name ("totalValue"), the arguments along with their types ($bill of type element), and the return type (xs:double[4]). The body can be any XQuery expression. The example makes use of the compositional feature of XQuery by nesting a FLWOR expression (with the let, where and order-by parts omitted) inside a sum function. It iterates over all line elements under the element passed to the function and multiplies the values of their qty and itemPrice elements. This creates a sequence of doubles where each value is the product of a quantity and a price. The sum function then computes the total of all the values in the sequence, this value is returned as the value of the function.

Calling a user-defined function works just like calling a built-in function, but you also have to supply the namespace (which is implicit in the case of a built-in function).

- *Conditional expressions:* Evaluating one of two branches based on the outcome of a test expression is handled by the if-expression in XQuery. The syntax is similar to other languages. Here is an example that uses it:

```
if ($bill/@total-sum > 100) then <gold-customer/>
  else <normal-customer/>        (3.9)
```

[4]xs is the namespace prefix for XML Schema.

```
declare namespace bill="http://localhost/testbill/bill.xsd";
declare namespace n="http://localhost/bill_view";

declare function local:totalValue($bill as element) as xs:double {
    sum(for $l in $bill/bill:line return ($l/bill:qty * $l/bill:itemPrice))
};

<n:result>
{
    for $day in ('Mon', 'Tue', 'Wed', 'Thu', 'Fri')
    return
    <n:day name="{$day}"> {
        for $bill in /bill:bill[contains(bill:due-date, $day)]
        return
            <n:bill-summary>
                <n:short-sender>
                    {$bill/bill:addressing/bill:sender/bill:company/text()}
                </n:short-sender>
                <n:short-receiver>
                    {$bill/bill:addressing/bill:receiver/bill:company/text()}
                </n:short-receiver>
                <n:item-count>{count($bill/bill:line)}</n:item-count>
                <n:total-value>{local:totalValue($bill)}</n:total-value>
            </n:bill-summary>
        }
    </n:day>
}
</n:result>                                                         (3.10)
```

Figure 3.1: Example of a more complex XQuery that uses many of the features described in section 3.1. The query creates a well-formed XML document with bill summaries sorted by day of week.

This inserts a gold-customer or normal-customer element depending on the total value of the bill.

3.2 XML Schema

XML Schema is an XML language that can be used to describe the structure of XML instance documents. It is itself represented in an XML syntax, which makes it easy to process but hard to read. XML Schema is defined in the W3C recommendations available at [W3C02].

The key concept of XML Schema is the description of an element's contents by its type. That is XML Schema associates types with elements. There are two kinds of types: complex and simple ones. Complex ones are roughly types who's contents is composed of other elements, while the contents of simple types is atomic (e.g. a number or a character string). Each schema has a target namespace into which it defines elements, attributes and types. Thus a namespace is a naming context, in

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:t1="http://localhost/trivial1.xsd"
    targetNamespace="http://localhost/trivial1.xsd"
    elementFormDefault="qualified">
    <xs:element name="box">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="width" type="xs:integer"/>
                <xs:element name="height" type="xs:integer"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>                                                    (3.11)
```

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:t2="http://localhost/trivial2.xsd"
    targetNamespace="http://localhost/trivial2.xsd"
    elementFormDefault="qualified">
    <xs:element name="width" type="xs:integer"/>
    <xs:element name="height" type="xs:integer"/>
    <xs:complexType name="rectType">
        <xs:sequence>
            <xs:element ref="t2:width"/>
            <xs:element ref="t2:height"/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="rectangle" type="t2:rectType"/>
</xs:schema>                                                    (3.12)
```

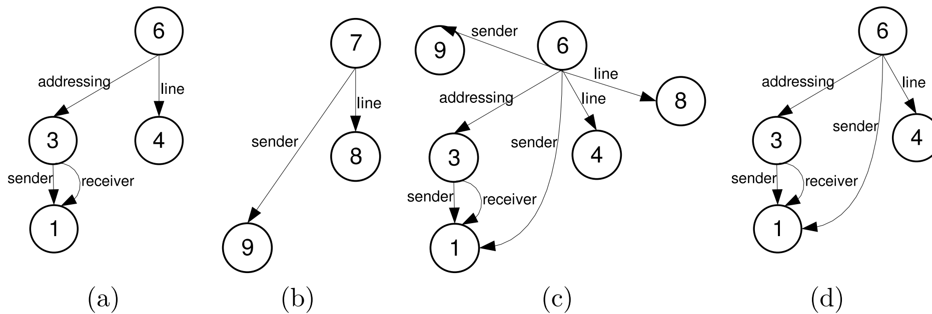
Figure 3.2: Two schemas that use local (top) and global (bottom) element and type definitions. Note that the top schema has one global element into which the element's complex type and other elements are nested, while the other schema first defines two elements of simple type that are later used in the (also global) definition of a complex type, which in turn is used to define the rectangle element.

which the names of the schema parts are unique[5]. The concept of namespaces makes it possible to define an element in one schema that is of a type which is defined in another schema (i.e. the type is referenced).

Not all definitions in an XML Schema are referencable from outside. To be referencable a definition must be global (i.e. directly in the xs:schema element). The schemas in figure 3.2 show examples of referencing a type in a different namespace (xs:integer is in the http://www.w3.org/2001/XMLSchema namespace). The bottom schema also makes much use of global definitions.

XML Schema defines the contents of complex types by means of particles. In figure 3.2 the bottom schema defines a complex type called “rectType”. Its contents is described by a sequence that contains two elements. This means that inside each element of this type, there must be exactly two elements and they must occur in

[5]See http://www.w3.org/TR/1999/REC-xml-names-19990114/ for a proper definition of namespaces.



(a) (b) (c) (d)

Figure 3.3: Example of induction automata. (a) and (b) two seperate automata, each having learned one document, (c) the combined NFA when learning several documents and (d) the simplified automaton.

this order. Besides the two particles sequence and element used in the figure, there is another important structuring particle: choice. Its use is similar to that of sequence, but it means that any of the elements inside can occur in the respective place.

3.3 Induction Algorithms

This section gives a brief overview of existing algorithms that induce structure information from instances. There is quite a bit of previous research on the topic: “DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases” [GW97], “XTRACT: A System for Extracting Document Type Descriptors from XML Documents” [Gra00] and “On Structural Inference for XML Data”[Wo03] to name a few. Short descriptions of each of the articles can be found in the related work section. The presented algorithms are similar in structure to the one used in the prototype, differences are outlined in section 6.4.1 where the prototype’s algorithm is described in detail.

Very roughly, induction of structure is usually done in these steps:

1. Interpret the semi-structured instance as a word of a language. This is convenient as it allows application of existing knowledge and a concise representation.

2. Find a non-deterministic finite automaton (NFA) that describes the language, see figure 3.3 (c) for an example. This automaton is an NFA because no effort is made in this step to exploit common structural features of the input documents. The NFA will typically be rather large and contain a lot of redundancies, depending on the algorithm used to create it. This algorithm is one important property of the overall induction algorithm.

3. Apply a set of rules that simplifies this NFA. These typically take out redundancies by merging states and sometimes relax structural requirements according to predefined heuristics. This set of rules is the second important property of the induction algorithm.

4. If needed, convert the automaton into a structure description language of your choice.

[GW97] constructs DataGuides as structure information for a semi-structured database. In power these are about equivalent to DTDs. The main difference is that a DTD describes the structure of one XML document, whereas the DataGuide

describes the structure of the whole semi-structured database. The notion of documents simply does not exist in the database used. [Gra00] presents an algorithm to generated DTDs for a repository of XML documents. At first glance, the problems dealt with in the two works seem to be identical save the choices of data and structure representation. However, there are two key differences:

- [GW97] deals with the whole database at once, whereas [Gra00] uses multiple instance documents to induce a common DTD.
- [GW97] constructs a DataGuide that represents the given database exactly. [Gra00] notes that this approach can lead to very large and complex DataGuides or DTDs respectively. In order to simplify the generated DTDs the authors of [Gra00] propose several techniques working together to achieve both simplification and generalization.

An important notion stressed in both [Gra00] and [Wo03] is that of Minimum Description Length (MDL). Both use the MDL to rank candidate DTDs accoring to the size of their encoding. Resulting in the DTD sought being the one that is general enough to capture the structure of all presented instances, but does so with a fair amount of detail.

In order to provide candidate DTDs [Gra00] implement their own set of rules for step 3. (as described above). [Wo03] compare multiple rule sets.

The discussed algorithms are either completely unaware of XML or still use DTDs as a schema description.

3.4 The eXist Database

3.4.1 Overview

The eXist database is an implementation of an XML database. It is developed as an open source project (licence is LGPL[6]), making it easy to extend. It stores documents natively (without mapping to an underlying relational database) in hierarchical collections. The user can run queries on collections or individual documents using XQuery [W3C03]. The storage structure is index-based allowing efficient implementation of various XPath axes. eXist implements different methods giving clients access to the database. The one used in the presented prototype is based on the XML:DB API for Java [XDB]. This choice was made because eXist's interactive client software also uses it. All parts of eXist are implemented in Java.

There are two kinds of resouces (documents) in the XML:DB API and thus in eXist: binary and XML resources. Binary resources are stored as raw bytes and eXist does not implement any functionality beyond storing and retrieving for them. XML resources are XML documents for which eXist has a wide range of functions. You can of course store and retrieve them, but you also get means of querying, indexing and updating.

Non-core functionality is implemented in services as specified by the XML:DB API found in [XDB]. A service is associated with a collection of documents, i.e. works in the context of this collection. Important services used in the prototype handle collection management and query execution. Services are obtained from the context

[6] Lesser General Public License. Opensource licence that allows the user to freely distribute the software and make any changes desired. Any version of the software (changed or unchanged) must be distributed in source (and binary) if distributed at all. The LGPL also allows linking of libraries coverd by it to non-open software. Also see http://www.fsf.org.

collection. Along with a service name, a version is also given when retrieving the service. This allows consistent functionality across system evolution.

The prototype implements services for schema and view handling. The schema service allows storing and retrieving of schema documents, retrieving of schema parts (elments, types and attributes) by qualified name, and the validation of documents for which schemas have been stored. The view service is responsible for creating, testing and evaluating views.

Further information on eXist in general can be found on its website [eX] and in an article by the primary author [Mei02]. More details on the query processor are given in the next section.

3.4.2 The Query Processor

eXist implements XQuery on top of its native XML database layer. After a user poses a query, the following steps are executed to find the result:

1. The query code is parsed into expression nodes. This expression tree is called a compiled query C, which is cached for future use. The expression nodes mostly correspond directly to the syntax terms of XQuery. A noteworthy exception to this rule is PathExpr that is used to warp XPath expressions as well as expressions in general (such as the whole query).

2. C is initialized by calling reset() on the topmost expression node N (a PathExpr). This is necessary as there might be old context settings from previous executions of the query.

3. To execute the whole query, N.eval() is called. This recursively executes the query:

    (a) Each node has an eval() method that computes the part of the query result that corresponds to its expression. This might involve calling the eval() methods of potential subnodes.

    (b) When calling eval() on a subnode, the proper sequence of context nodes is set. What exactly this is very much depends on the type of node that is calling the subnode.

As an example of how the context sequence works, consider this expression:

```
/chapter[@name="Introduction"]/section                                    (3.13)
```

The expression will be rendered to a PathExpr containing a list of LocationSteps (three in this case, even though you only see two in the query). "(" and ")" denote the PathExpr, LocationSteps are given as in the query but with verbose axes names:

```
(
  ROOT
    /child::chapter[(attribute::name = Introduction)]
      /child::section
)                                                                         (3.14)
```

When the compiled query is executed, the eval() of the PathExpr p bracketing the whole expression is called. From here on figure 3.4 also depicts the flow of control. p then iterates over its steps and calls eval() on each. Calling eval() on the RootNode r causes it to gather all document roots from the documents visible to the query. It returns this sequence of nodes to p, which sets it as the context sequence for

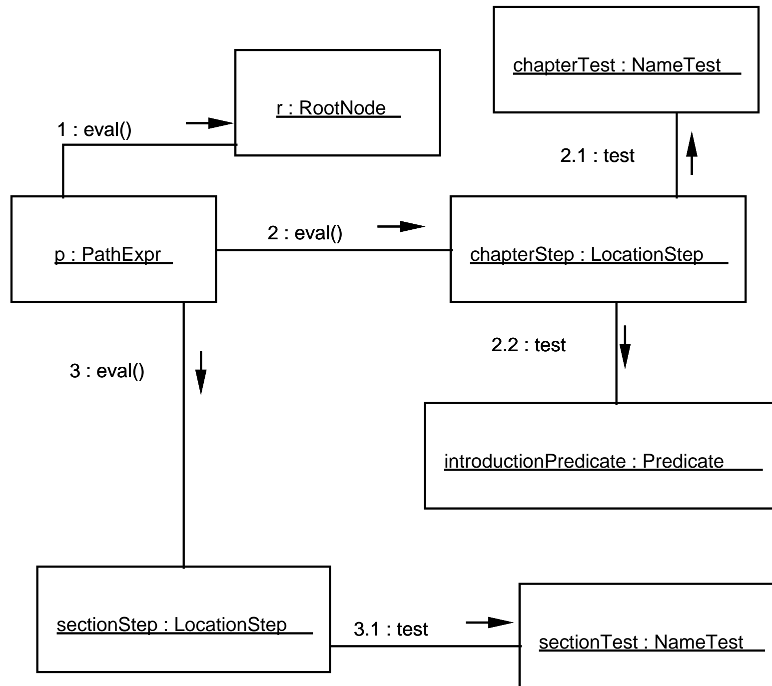


Figure 3.4: Collaboration diagram of a query execution. Starts at p:PathExpr.

the next step chapterStep by passing it to the eval() method of this LocationStep. chapterStep then acts according to its axis (child) and finds all the children of the nodes in the context sequence who's name is "chapter". This sequence is further filtered by the predicate to only contain those nodes that have a name attribute with value "Introduction". The node list now contains all chapters called "Introduction" from all the documents initially visible to the query. The section step finally receives this list as its context sequence and gets all children with name "section". This sequence of nodes is passed back to the PathExpr and thus the final result of the query.

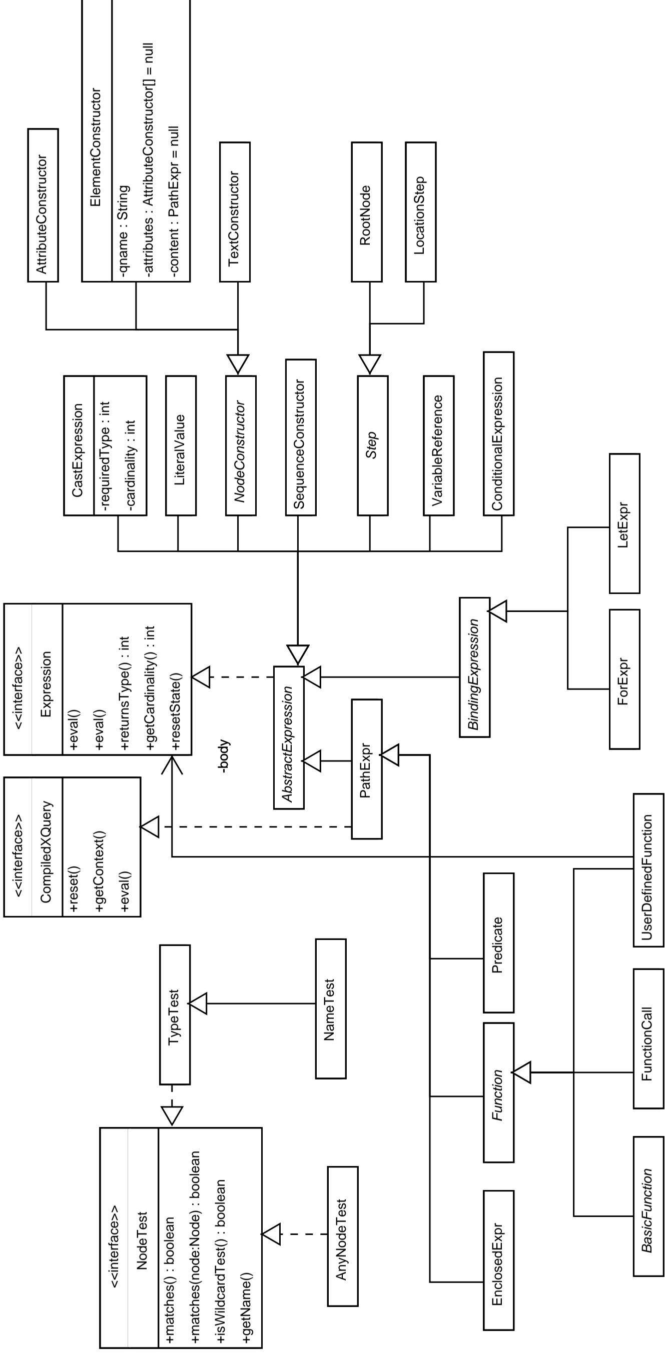


Figure 3.5: UML class diagram of some important syntax nodes of the query processor. This is only a small subset of the complete implementation.

Chapter 4

Design

Summary. *The language used to describe a view is of course a central property of the system. This chapter first deals with selecting a language, XQuery is found to be suited best.*
Generating schema information from the query expression of the view directly seems to be the purest way of obtaining such information. Unfortunately, one quickly finds out that it is also not generally applicable. In particular, it is not possible to determine which elements will be part of the query result without at least partially executing the query. This chapter details why this is the case.
The second part of this chapter is devoted to the interaction of query processor and schema generation algorithm. In cases where schema information can only be generated by executing the query (as is argued here) these two system pieces are closely related.
Finally this chapter gives a brief overview of the system structure. This is done last, because the decisions made on schema generation as well as on interaction with the query processor influence it very much.

4.1 Choice of Query Language

The query language used for the definition of a view is of course a central part. Many differences in functionality arise from it alone. Here is an overview of the languages that were considered, as well as a short discussion of each:

- *XPath:* XPath [W3C99] is a selection path oriented language. In fact it entirely consists of selection paths. To retrieve nodes from instance documents, XPath uses different "axes" along which a selection can occur starting from the root node. There are several axes, the most common ones being "descendant-or-self"[1] and "child"[2]. The constriction to path selections makes XPath somewhat limited as a query language. While some functionality is achievable in non-obvious ways, this is certainly not very readable. XPath is implemented in the opensource XML database xindice found at [Xin].

- *XQuery:* XQuery is a functional language that includes most of the usual concepts of such languages. It has a syntax that is generally more readable than

[1] The current node or any of this (recursive) children.
[2] All direct children of the current node.

that of XPath, as it is possible to break things down into several steps. XQuery incorporates XPath as a selection languages for nodes. On top of this, it offers additional constructs such as functions, iteration, and conditions. The language is fully typed on simple type level. There are many XQuery implementations available, an opensource one can be found in the database eXist [eX].

- *XSLT:* XSLT is the third XML transformation language proposed by the W3C. Contrary to XPath and XQuery it is a template based transformation language, but iterative approaches are supported as well. Its syntax is XML which makes it somewhat hard to read. Like XQuery, XSLT uses XPath for the selection of input nodes. XSLT was mainly developed with the use in conjuction with the extensible stylesheet language (XSL) in mind. This point of focus can make things that are not related to presentation a bit harder to accomplish. Unfortunately, XSLT is not implemented directly in any opensource XML database.

- *Xcerpt:* Xcerpt [BS02] is quite different from the other languages presented. It is a pattern based language, that uses patterns for selection of nodes from the input documents. These patterns can contain variables that are bound to the corresponding parts of input data. Think of this as putting a mask over the input, where the variables are holes in the mask. The variables are bound to the pieces of data visible through the holes. The output is generated via templates, that can again contain the variables, this time they are dereferenced. Lists of nodes are created by iterating over all hits of the pattern (the hits generate different variable bindings). The notation lends the user to a two-step process of thinking: First defining the pattern part to select nodes and bind the variables, then writing the template part to create output for the variable bindings. This means that you have to keep track of all variables at once, which is probably quite difficult for more complicated views.
  Making the output generation template based gives Xcerpt the very nice property that the user already specifies half the schema of the output. The structure (on complex type level) is there, one mainly has to worry about simple types and cardinalities (also see the XMAS language discussed in section 1.2.3 on this).

XQuery is best suited as a query language for the presented prototype. Here is why:

- XPath is unsuited because it has two crucial disadvantages: It is difficult to implement (and thus to read and understand[3]) because of the "everything is a selection"-paradigm. One also has to take extra care to make the output well-formed, it usually is not. To make the result of an XPath expression a well-formed document, you could take two approaches: One is to write selection expressions that will result in only one root node, this severly limits what you can select from the database. Another would be to let the view mechanism create an additional element around the result of the XPath expression. This would complicate view definitions, and also violates the requirement of closedness (see above) as there is an intermediate step (the creation of the new element), which cannot be expressed in XPath, when running queries on views.

- XQuery is relatively easy to write and understand. It is powerful enough to implement any view (in fact it is Turing-complete, see [Kes]). There actually is an opensource implementation which will allow extension of the query processor

[3] A small program called 'fortune' occasionally gives the following quote which is attributed to Brian W. Kernighan: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

for the task of schema generation. Making the output well-formed is a simple task.

- XSLT also is not the preferred choice because it is clumsy to read due to its XML syntax. On top of this, some common problems are hard to implement because it was designed to be a presentation centric language.
- Xcerpt has a compelling approach, especially since it very much simplifies schema generation. However, there are a few critical problems: (1) It is a research project and the prototype is implemented in Haskell (as there is no XML database implemented in Haskell a reimplementation would have been necessary), (2) it lacks some important features for views, such as simple ways of computing sums or averages over a number of elements, and (3) few people know it which would lead to a mismatch between view definition and query formulation (because the latter would be done in a different language).

4.2 Schema Generation Options

When generating schema information for a view, it would be preferrable to do the generation from the view's definition alone, without actually runnging the query. This approach provides a clean separation between meta- and instance-data. Unfortunately a closer look at XQuery reveals that this approach will not work. The main cause for this is that the XML data, which is structured in documents and collections, is queried using—among other things—XPath expressions. Without loss of generality it is impossible to determine the context documents of and nodes returned by the path expression without looking at the data.

As an example, consider this query:

//*:bill (4.1)

Upon evaluation of the view that uses this query, the query itself is evaluated in the collection context set in the view's definition. The query selects all elements with local name "bill" and arbitrary namespace URI. Finding out in which documents below the collection context these occur, is of course not possible without taking a look at the actual documents. From the location step "*:bill" by itself, the schema generation algorithm cannot find out, which bill elements are actually affected. This information however is crucial to generate the schema.

A little investigation of real life queries shows, that almost all contain at least one XPath expression. An XPath expression that contains fully qualified elements only can be used in schema generation without reference to the instance docuements, more complicated ones cannot. In this simple case the query gives enough informaiton to retrieve the appropriate schema definitions from the schema store.

Another problem is that XQuery expressions can be too complicated to interpret manually (without an actual execution), as XQuery is Turing-complete. This would be equivalent to the task of a compiler trying to establish all possible execution paths.

For the reasons outlined above, the prototype does not attempt to interpret the whole query to generate schema pieces directly from it. Instead common parts of XQuery are handled directly, less common or very complex pieces are executed and schema information is induced from the resulting data. This approach preserves the benefits for direct generation for the common parts while still being able to fully cover XQuery[4].

[4]Some XQuery extensions special to eXist—namely the Java und XUpdate bindings—excluded as they allow side effects on the underlying data.

4.3 Interaction with the Query Processor

Having decided on the hybrid approach to implement some parts directly and to induce others, one is left with putting each XQuery expression in one of the two groups. A non-representative look at existing queries, for example at [W3CUc], gives an impression of the most popular syntax expression. Amongst these, those that promised to benfit the most of direct handling were identified:

- *Explicit constructors* (for elements, attributes and text): The occurance of the node in the respective place is certain and the cardinalities are also easily established in case of element constructors.
- *Selections:* By looking directly at an XPath expression it is often possible to exactly identify all possible nodes that it can return (e.g. by looking at the last location step) and often also their cardinalities (by multiplying those of all location steps).
- *Functions:* In case the function is of simple return type you can determine this type for certain and do not have to guess from the string representation of the returned data. With user defined functions, you can at least specify a sequence-bracket around the function. Function definitions also give useful information to construct appropriate cardinalities. More on this in the implementation chapter 6.
- *Conditions:* An if-expression in the query corresponds to a choice particle with two nested sequences in the schema.
- *For expressions:* A FLWOR expression corresponds to a schema sequence who's cardinalities are mainly determined by the in-expresssion. Thus the precision of the cardinalities for the schema sequence depends on how well the cardinalities of the in-expression can be established.

The schema parts for other XQuery expressions can be induced from the query result these parts give. To determine this result it is of course necessary to execute the corresponding expression. Doing this quickly results in executing the whole query, as one does not want to determine the result of the isolated expression but first has to establish the proper context (for example variable and namespace bindings) as well. Instead of reimplementing the query processor in the schema generator to determine this context, the schema generator will be embedded into the query processor.

Each expression node, i.e. all classes that implement Expression[5], are responsible for generating the part of the schema that corresponds to the respective expression. This means that in addition to the quey execution context, each node also has a schema generation contex. This schema context usually takes the form of a schema particle and is specific to the node instance. Unlike the execution context, the schema context is held in a local variable to enable the node to reidentify its schema particle during multiple executions of the node. Have a look at figure 4.1 for an example of this setup. Associating the schema generation context permanently with each syntax node is necessary because many nodes are evaluated more than once during the query. However, despite these multiple evaluations schema information should not be duplicated.

The local information in each node needs to be combined into a full schema. To achieve this, nodes are passed a SchemaGenerationInfo (see section 6.1 for details) when executed, from which they can get the current type and element as well as

[5] An interface implemented by all classes that are XQuery expression nodes.

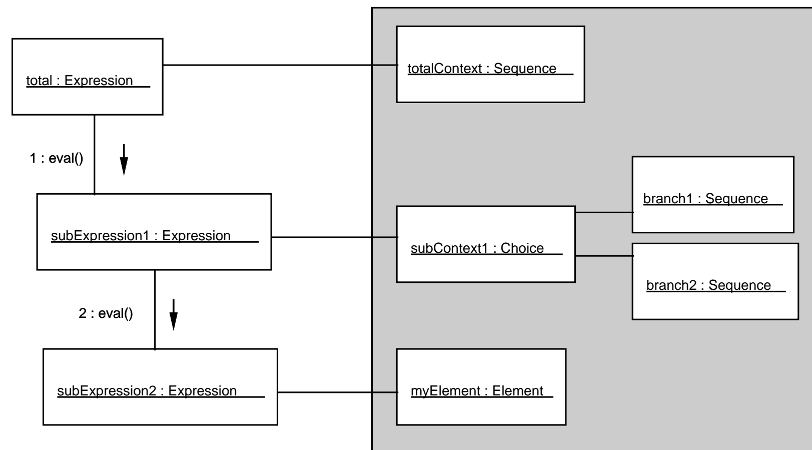


Figure 4.1: Each syntax node of XPath expression on the left has a schema generation context of its own (box on the right). The XPath expression is evaluated by recursively calling the eval() methods of all syntax nodes. The nodes create their schema generation contexts and also make sure these are connected in the proper places into a full schema (not shown here).

the particle inside the current type that is being worked on (i.e. the place where to insert a new schema piece). Using this information, the node can now put its local knowledge in the global schema context.

4.4 System Structure

The setup of the system structure is depcited in figure 4.2. Views are managed by the View Service which is described in chapter 5. Clients can use it to create views and evaluate present ones. Upon receiving an evaluation request from a client, the View Service uses the Query Processor to run the query and obtain the view result. Schema generation is done during processing of this query, thus the Query Processor will use the Schema Store to access schema information about the existing documents. This link between Query Processor and Schema Store is very important as it ensures the referencing of existing schemas. References in the generated schema show how the view is related to data in other documents. This makes it easier to understand the semantics of the view.

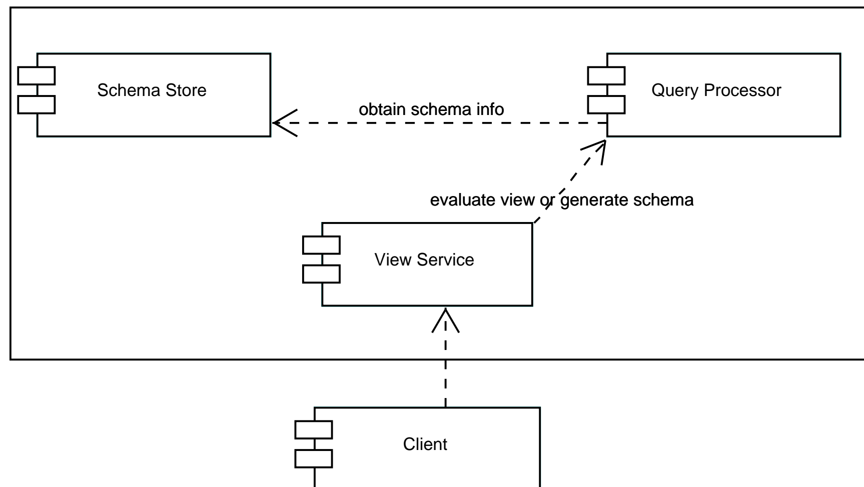


Figure 4.2: System structure. View generation and schema storage happen on the server.

Chapter 5

Implementation of Necessary Infrastructure

Summary. *This chapter is about the infrastructure which was added to eXist in order to be able to implement schema generation for views. eXist did not have views, thus a simple view mechanism was added. It neither had any means for storing and retrieving XML Schema information, this was added as well. To make the workflow of testing the prototype more efficient, eXist's interactive client was also extended. These changes are not central to this thesis, thus relatively little regard is given to them.*

5.1 Views

Views are a necessary prerequisite if you want to implement schema generation for them. Unfortunately, eXist did not have any view implementation. This made the addition of a very simple one necessary. It mainly consists of two parts. Firstly a view service that allows creation, explicit evaluation, and schema generation. Secondly some changes to resource retrieval to return the result of the view's evaluation instead of the view's metadata. View service and metadata are shown in figure 5.1.

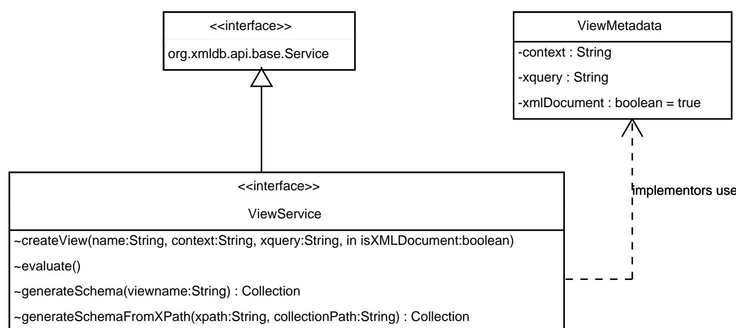


Figure 5.1: Class diagram depicting the ViewService interface and ViewMetadata class, which is stored in a binary resource.

The view implementation of the prototype has some shortcomings that make it ill-suited for real life use. It is however still sufficient for the purposes of this thesis. The defficiencies are:

- The name of a view has to start with ".view". This is just a means of keeping the implementation simple. Without going into too much detail here, this workaround avoided changes to almost all layers of eXist, but instead confined those changes to two layers (view service and resource retrieval).
- Inefficient implementation. To evaluate a view, its query is simply executed every time. There are much better albeit more complex algorithms available, see for example [Gl97] and [AMV98].
- Views must be non-nested, i.e. views found in the query scope of another view are ignored in query processing. This is a side-effect of implementing views in just two layers. In fact, as much of eXists query functionality is index based (i.e. many queries can be answered without even retrieving the document), a proper nested implementation would be quite complex.

5.2 Schema Store

Handling XML Schemas is not implemented in eXist. However, as XML Schemas are just XML documents, there is at least functionality to store and retrieve them. This is not enough to implement for example validation of documents. In order to work with schemas you typically need access to them by their target namespace (i.e. the namespace into which they define their elements and types). Furthermore there will often be need to access only parts of a schema (e.g. a type definition) as well. To support all this, a service was implemented. Its class diagram is shown in figure 5.2.

The actual storing of the schema documents is done by using eXist's functionality of storing XML. All schemas are stored in a collection and have a unique filename. The collection also contains an additional document mapping target namespaces to schema documents. This document is called ".index".

To parse the schema documents (in order to retrieve parts) the schema functionality of the Castor project [Cas] is used. It converts the schema into a tree of Java objects. The native Java representation makes traversing the schema much easier than using the DOM tree (that could be obtained by standard means in eXist).

Thus when the schema service is asked to produce an element definition from namespace N, it goes to the schema collection to consult the index file for the filename corresponding to N. Having obtained the file, it uses Castor to parse the contents. From the Castor respresentation it retrieves the element definition and returns it.

5.3 Extensions of the Interactive Client

eXist comes with an interactive client. The primary features of this program are browsing the database, displaying and editing documents, and posing queries. All features are accessible on a command line as well, making it easy to extend the client by adding new commands. This was done for the functionality of the schema and view services.

However, the workflow of testing schema generation is quite complicated on the command line: You first create a new query file ("putquery" command), then you open the file to edit (or paste) the actual query in the document display window. Next you create a view with this query ("mkview" command) and generate schema

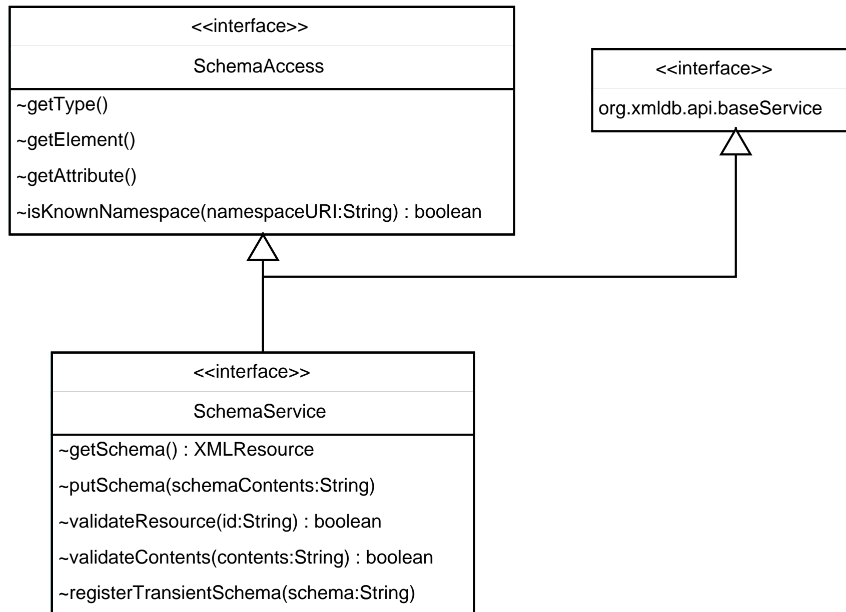


Figure 5.2: The SchemaService interface and its super interfaces.

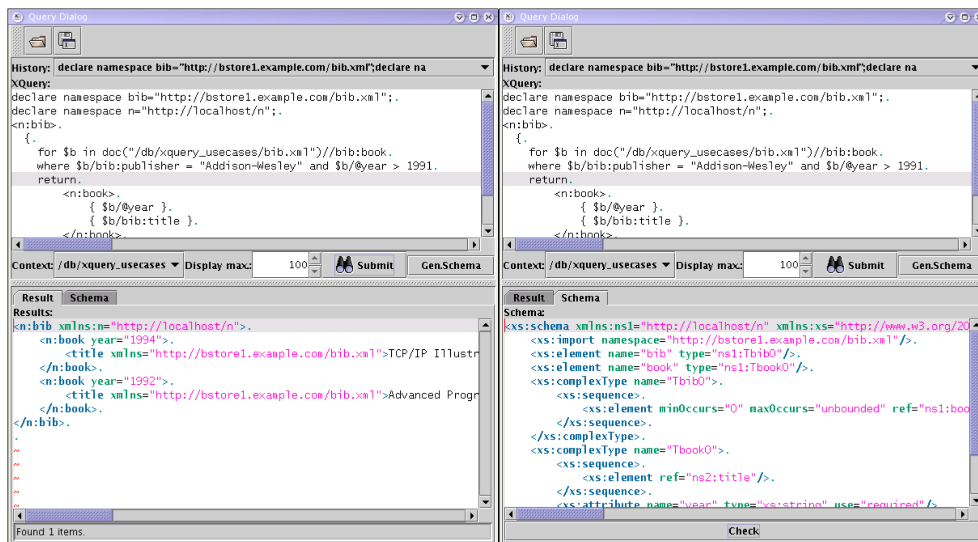


Figure 5.3: The extended query dialog of the interactive client. Left side shows the results, right side the generated schema. There also is an additional “Check” button that allows quick validation of the result against the schema (thus verifying whether the generated schema is correct).

information for it ("generateschema"). You now get schema output to the command line of the interactive client. You save each schema individually ("putschema"). To test whether schema generation actually worked correctly, you can now validate the view ("validate").

Repeating this process many times a day was found to be quite awkward. Instead a simpler approach was implemented. It does not actually create a view, which does not matter, the view would only be evaluated once anyhow. The simple method uses the query dialog of the interactive client. Its extended version is shown in figure 5.3. To execute a query, you write a query expression in the top part, choose a context collection in the middle and click "Submit". Note that all the information to create a view is there (there is no place given to save the view to, but this does not matter, as there is no intention of saving the view). Thus one can simply pretend to be creating a view and add an extra panel on the bottom to show the generated schema. The schema panel also contains a "Check" button that you can click to verify the schema against the query result.

Chapter 6

Implementation of Schema Generation

Summary. *This chapter details how the generation of schema information is implemented. It first gives an overview of the datastructures used, then goes on to discuss modifications to each supported syntax node individually. The computation of proper cardinalities is presented in a separate section as this is generic to the type of syntax node that takes advantage of it. Schema information for some parts of the output document has to be induced from the contents, this induction is described in section 6.4. The chapter concludes with the schema simplification mechanisms implemented, these are necessary to (a) create a fully XML Schema compliant schema and (b) remove some unnecessary particles that are created in the schema generation process.*

As an overview, here is a brief sketch of the overall algorithm used for schema generation (see figure 6.1 for an illustration):

1. Retrieve the view's **metadata** to determine context and query expression used to get the actual values.
2. **Compile** the query expression. This step is purely based on previous functionality of eXist.
3. **Initialize** the query execution and schema generation contexts. On the schema generation side this is done by creating a new container for the schema information to be generated. A boolean flag in the query's context is also set, indicating that the query is to generate schema information.
4. **Run** the query and in the process also **generate** schema information for each node. This step is detailed in this chapter.
5. After the query is done, **simplify** the internal schema representation.
6. Finally convert the internal schema representation into the well known **XML representation**.

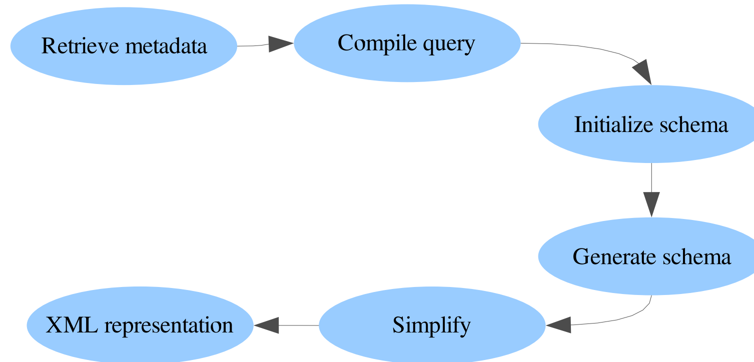


Figure 6.1: The schema generation process happens in six steps. Steps 4 and 5 are subject of this chapter.

6.1 Datastructures

Schema generation uses a number of datastructures to achieve its aim. An introduction to these is given in this section, but most are rather obvious anyway.

To generate a schema you first of all need a way to represent its parts. As the current status of the implementation by far does not make extensive use of all the possibilities in XML Schema, the internal schema representation is a simplified view of XML Schema. Having an own internal implementation of schema parts also has the benefit that you can handle some special cases that would be difficult to represent in XML Schema directly. Why this is important will become clear in the course of this chapter.

There are internal representations for XML Schema elements, types (simple and complex, but this is handled in one class) and attributes in the Element, XMLType and Attribute classes respectively. On top of this, XML Schema also provides grouping, which is done by choice and sequence particles[1], both of which are also represented in the Choice or Sequence classes. There is one special particle, the SimpleParticle, that is an addition. Its purpose will be clarified later in this chapter (see section 6.2.3 on simple types if you are curious). There is a UML diagram in figure 6.2.

To hold all the information needed by the generation algorithm during execution, a class called SchemaGenerationInfo was created. The information stored in SchemaGenerationInfo could also have been put in the XQueryContext class, that previously was present in eXist. In fact, there is a one-to-one relationship between the two. However, this would have led to even further mixture of schema generation and query execution and was therefore decided against.

SchemaGenerationInfo holds the element and type that are currently processed. In addition to the type there also is a reference to the schema particle into which further schema information is to be put. This gives sub-expression the means necessary to connect their generated schema pieces to the whole schema. As most expression nodes will set at least a new particle (their own) before executing sub-expression and will also restore the previous particle (their parent's) before returning from their own execution, SchemaGenerationInfo also supports these stack operations. There are methods for pushing and popping both current type and current particle.

[1] A particle in XML Schema is a sequence, choice or element. See chapter 3.2 and [W3C02].

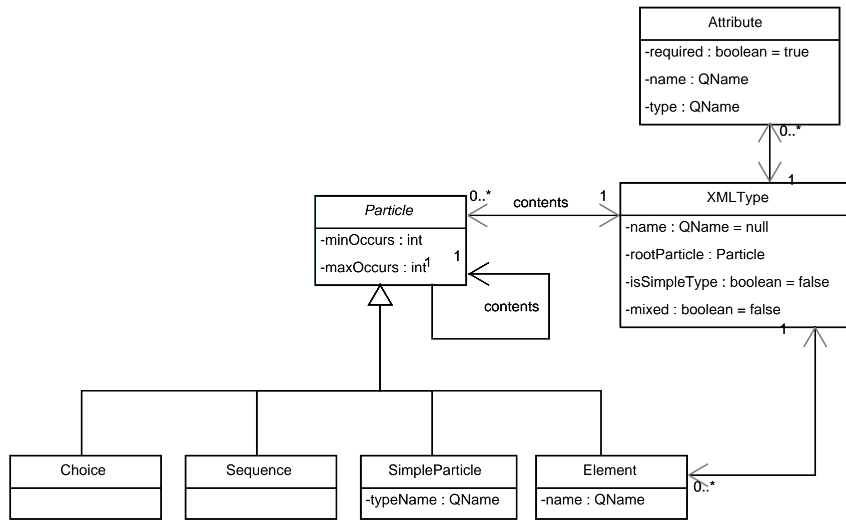


Figure 6.2: UML class diagram with all the elements of the internal XML Schema representation. All classes except SimpleParticle have counterparts in XML Schema.

Some expression nodes might not be able to generate a schema from the query alone. For this case, there is a reference to the so called manager. The manager is a wrapper around the induction algorithm, that allows induction of the contents of whole types as well as fragments thereof. There is need for the manager as the induction algorithm cannot be run alone but must be able to work in the context of the already generate schema pieces. Finally, all generated elements and attributes are collected in the SchemaGenerationInfo. From here they are later extracted and passed on to the schema simplification and XML generation algorithms.

6.2 Generation of Fixed Parts

A note on terminology: the elements and types generated directly from the query without induction from parts of instance documents are certain to be correct in that way. Furthermore, they cannot change during the generation (unlike the induced parts that might need upgrading as more instance examples are presented) and are therefore called fixed types and fixed elements. In contrast the induced parts are called induced types and induced elements.

The evaluation of the query happens by recursively calling the eval() method on the syntax nodes. Each of the nodes is responsible for generating part of the schema during this evaluation. More specifically, each node generates the schema information that corresponds to the data it returns. This is usually not a full complex type. To enable the nodes to put the generated schema information in the right place, they are passed the schema context of their parent node. This context is encapsulated in the class SchemaGenerationInfo and described in section 6.1. Most nodes create their own context (mostly an XML Schema sequence) to keep track of the part of the schema they are responsible for. This leads to many unnecessary particles in the final schema (i.e. nested sequences), making the schema simplification step described in section 6.5 important.

6.2.1 External References

XPath selection expressions can select and return any node of a document the expression is executed over. If there is schema information available for the returned node, it is necessary to reference it (elements must not be declared twice). During the generation of the schema for the view, a reference is therefore inserted in the appropriate places. This reference points to the element definition of the node in the external schema. Obviously this only works if the element declaration that the reference points to is a global declaration.

If the cut-point is at a non-global element, generation of a schema is not possible. The generator could chose to reference the element, which would lead to an invalid schema, because the reference points to an element that is not accessible[2]. Another approach would be to generate new schema information for the referenced element and making this information global. However, this (a) breaks the reference and (b) can cause name-clashes, as there might already be an element by that name. The second argument becomes even stronger if you consider that this problem could occur multiple times: If two local elements by the same name were made global there would be a collision even without any previously existing global element by that name. For these reasons, the prototype refuses to generate a schema if any of the cut-points is at a local element. It may be possible to resolve the situation in some special cases (for example in case you have control over the referenced schema).

6.2.2 Constructors

XQuery allows for explicit construction of elements, attributes and text[3] The constructor syntax is identical to what appears in the XML document later. For example following XQuery expression:

```
<chapter name="Introduction">Some text</chapter>        (6.1)
```

constructs an element called “chapter” with a name attribute containing some text. The three types of constructors will be described seperately here, because they are dealt with in different ways.

Element Constructors

Element constructors have the following properties:

1. They always create a fixed element.
2. They also create a fixed type. Note that this is a simplification in the prototype, the constructed element might be of a known type, but this is only detected with simple types.
3. Their contents will determine the structure of the fixed type. This structure does not have to be fixed, if the contents expression is complicated, it could be induced.

Element constructors make use of these properties to generate their parts of the schema when invoked. Keep in mind that an element constructor is possibly invoked multiple times (for example when it is contained in a FLWOR expression).

[2] A closer look reveals that the situation is even slightly worse: A non-global element is not identified uniquely by the usual (namespace, local-name) pair, as there may well be multiple local declarations of elements by the same (local) name in different compley types.

[3] And also of processing instructions and comments, but these two are ignored there. Comments do not affect the schema anyway.

Upon invocation an element constructor executes the following steps:

1. If it has not been executed before, it creates a new fixed type T.
2. It sets its fixed type T as the current type, T's contents (the outermost sequence) is set as the current type context.
3. The eval() method of the contents is called. This will result in the generation of all necessary schema information for the contents. This information will be added to T as it and its contents are set as the current type and the current type context.
4. Upon first execution T is given a name ("T" + *constructed element's name* + *unique number*).
5. Also upon first execution a new fixed element with the proper name of type T is created.

Attribute Constructors

The functionality in the AttributeConstructor class is quite simple. The only thing it has to do is add an attribute of appropriate name and type to the current type. This is quite easily achieved:

1. Create a new Attribute.
2. Set the attribute's name.
3. Turn off schema generation. The expression that is evaluated for the attribute's contents does not create any schema information. Even if it is of complex type, it will be converted to its string representation.
4. Call the eval() method of the attribute's contents to find the actual value.
5. Turn on schema generation.
6. Find the attribute's type:
   (a) If the value is a sequence, this sequence will be converted into a string as attributes have to be of simple type in XML Schema. Thus in this case, the proper type for the attribute is xs:string;
   (b) otherwise the proper type is that of the first item in the value sequence.
7. Set the previously determined type.
8. Store the attribute in the current type. This type is obtained from SchemaGenerationInfo (see 6.1).

Text Constructors

Text constructors insert a string literal into the query result (hence their name). Inserting text can have two effects: (1) the enclosing type is a simple type that only contains text or (2) the enclosing type also contains other elements in which case it is a complex type, albeit a mixed[4] one.

[4] See the XML Schema specifications at http://www.w3c.org/TR/xmlschema-1/#Complex_Type_Definitions on what a mixed complex type is.

Unfortunately, the text constructor cannot determine whether it alone generates the contents of the enclosing type. It can only mark, that simple contents occurred. It does so by inserting a SimpleParticle (this is not an XML Schema particle, see section 6.1). During schema simplification the SimpleParticles are found and the surrounding types changed accordingly.

6.2.3 Finding Simple Types

One of the main advantages of generating schema information from the query is that it is possible to know the correct type of simple types and elements. Finding out where these simple types actually occur can be quite difficult. It is trivial in the case of attributes: these are always of simple type. It is more involved in the general context of syntax nodes. A node might know that the contents it generates is of simple type, but how does it embed this information into the whole schema? Atomic functions for example will generate contents of simple type. However, this does not justify making the current type found in the SchemaGenerationInfo a simple one, even if it is still empty when eval() is invoked on the function node. On the other hand, even if it is not empty, the final contents of the type might still be simple (two functions could consecutively insert pieces of a string, e.g. first and last name).

This problem is solved by introducing a marker particle into the generated schema. This particle is called SimpleParticle as it is used to denote that simple contents occurs here. It is inserted into types (which are not yet complex or simple) just like the real XML Schema particles sequence, choice and element. Specifically the use of SimpleParticle is similar to that of the element particle.

After the query has been fully executed and all the generated schema information has been collected, each type is made complex or simple. This happens during the simplification phase by employing a special simplification algorithm that removes the SimpleParticles from each type's contents. Depending on the other information found in the contents, the type is:

- made simple if it does not contain any complex particles (sequence, choice or element). If there is exactly one SimpleParticle inside, the now simple type is set to be of the type found in the SimpleParticle (e.g. of type xs:integer if the SimpleParticle stemmed from a count function). If there are more SimpleParticles xs:string is used[5].
- made complex if it contains complex particles.
- marked as a mixed type if along with the complex particles a SimpleParticle of any type is found.

As types are now simple or complex, the final conversion to the usual representation of XML Schema now generates simple types (or references to built-in simple types).

While XQuery is a fully typed language on primitive type level, there seems to be one flaw: There is no useful common supertype to all numeric types. The type hierarchy makes decimal (which has many numeric children), float and double (the latter two have no children) direct subtypes of anyAtomicType. Unfortunately anyAtomicType has many children as well (for example boolean, string, QName) which would make the use of anyAtomicType as a common supertype rather unspecific while of course technically correct.

[5]There is room for improvement here, it might be possible to find an concatination type more specific than xs:string, but this is ignored in this prototype implementation.

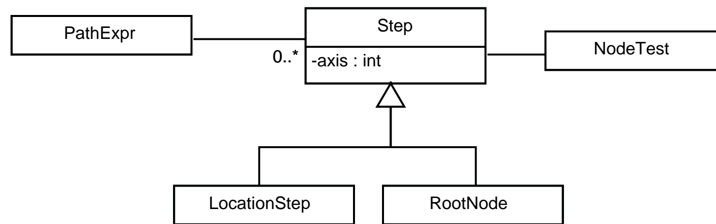


Figure 6.3: Classes involved in XPath selections. Steps are modeled by Step and its subclasses and held in a PathExpr.

To get around this problem, the generation algorithm uses xs:double as a common supertype. In the XQuery implementation in eXist it is sometimes possible to find out that an expression will be of type numeric (which is a simple type eXist defines outside XML Schema). In these cases xs:double is used as a type in order to avoid discarding information.

6.2.4 Selections

XPath selection expressions are composed of a number of steps (mainly LocationStep instances) wrapped in a PathExpr node (see figure 6.3). This implementation has two inconvenient side effects on schema generation: (1) LocationSteps don't know each other even if they belong the same XPath expression making important information inaccessible and (2) PathExpr is also used for wrapping various other things, making ugly case distinctions necessary. A LocationStep needs to know whether it is the last one in a XPath expression, because if it is, it determines the return type of the whole expression. However, because of (1) this information is not available and things need to be handled one level higher in PathExpr. Unfortunately due to problem (2) this does not lead to nice maintainable code.

As indicated above, schema generation for selection expression is split in two parts:

- In PathExpr selection expression are detected. The whole expression is evaluated by evaluating the individual step expressions from left to right. During this process each expression is evaluated with the result of the previous as input. This process is comparable to the pipes and filters pattern.

  The only step that has direct effect on the schema is the last one. It determines the result type of the whole selection expression.

- In LocationStep the proper element references are inserted into the generated schema. In the simplest case, the last step gives a fully qualified element name in which case this element can be directly referenced.

To make sure that more than the simpelest case is handled, LocationStep takes a different approach:

1. If the location step is along the child-axis and the node-test is not a wildcard test (i.e. there is a fully qualified element name given in the node-test):

(a) If this is the first execution, take the node name from the node test and create a new element in the current schema generation context.

2. else:

(a) Upon first execution create a new sequence s that serves as the LocationStep's schema context. This sequence is inserted into the surrounding type.

(b) Do the evaluation to find which nodes have actually been selected.

(c) Create a new list for elements L. Iterate over the result list obtained from the evaluation of nodes and for each node N:

i. If N is an element node, add N to L.

ii. If N is an attribute node, add the attribute to the surrounding type.

(d) If L is non-empty:

i. Use the manager to induce a XML Schema sequence $s\prime$ describing the list L.

ii. Employing a TypeMerger, as used in the induction algorithm and described in section 6.4.1, to merge $s\prime$ with what is already stored in the schema context sequence s.

6.2.5 Functions

XQuery allows the declaration of functions in much the same way most functional languages do. The XQuery implementation in eXist has a variety of built-in functions and of course lets users declare their own. The functions mainly fall in three categories: built-in functions with atomic return type, built-in functions with non-atmoic return type, and user defined functions. The main difference between the first two categories is an implementational one: functions with an atomic return type can all be handled in a generic way, non-atomic return types call for special code depending on return type and function in question. The prototype does not implement schema generation for built-in functions with non-atomic return type. Finally schema generation for user defined functions is also implemented. This is not very hard as these functions are composed of further XQuery expression.

There are two kinds of function related syntax nodes: function declarations and function calls. Of course the declaration of a function as such does not generate any schema information. Only when the function is called, the necessity to generated something might arise. Exactly what is to be generated in this case is however up to the function's declaration. Thus, if schema generation for functions is to be implemented, it has to happen in two places: the declaration and the call. Calling is the same for any type of function (as by the above classification) and is therefore described here once. Different types of function declarations are treated differently and are described below in further detail.

A function call does not do very much (schema-wise):

1. Turn off schema generation.

2. Iterativly call eval() of the argument expressions.

3. Turn schema generation back on.

4. Call eval() of the function's declaration and pass the arguments.

Built-in Atomic Functions

All functions with simple return type are covered. This allows all attributes and simple types who's values are computed by a simple-typed function to have the most accurate type possible. Induction from instance documents would lead to all of them being typed as xs:string (or another *induced* simple type).

The atomic functions can roughly be subdivided by their return type into three categories: Those returing xs:string, numbers or boolean. Below is a list of the covered functions to give an impression of which attributes and simple types will be typed correctly. The generation of schema parts for all atomic functions is handled in a common superclass. This is possible as the only difference is, of which simple type the result actually is (but it will always be simple). The algorithm for this is not very complicated: Insert a SimpleParticle and set its type according to the return type.

In many cases the function's return type is declared as xdt:anyAtomicType[6]. For some functions this seems very strange at first, as they obviously return numbers (e.g. the avg funtion that computes an average). However, a look at the hierarchy of XPath's type system[7] reveals, that the most specific common supertype of all numbers is xdt:anyAtomicType. For this reason, the actual return type is determined and used for schema generation.

Numeric return type:

- *abs:* Returns the absolute value of a number.
- *avg:* Returns the average of the elements of a sequence.
- *ceiling:* Next integer that is greater than the argument.
- *count:* Takes a sequence and counts the number of elements in the sequence.
- *floor:* Next integer that is smaller than the argument.
- *get-(day|month|year)-from-date:* Returns part of a date.
- *get-(days|hours|minutes|seconds)-from-dayTimeDuration:* Returns part of a duration.
- *max, min:* Maximum/minimum of a sequence.
- *number:* Converts the argument into a number (more specifically: into an xs:double).
- *position:* Returns the current position of the context sequence, i.e. the index of the currently processed element in that sequence.
- *round:* Rounds the argument to the nearest integer.
- *str-length:* Returns the length of the argument string.
- *sum:* The sum of the elements of the argument sequence.

Boolean return type, here determining the return type is not a problem as it is always xs:boolean:

- *boolean:* Computes the boolean value[8] of the argument.

[6] xdt is a prefix for the namespace of XPath types http://www.w3.org/2003/05/xpath-datatypes

[7] http://www.w3.org/TR/2003/WD-xpath-functions-20030502/#datatypes

[8] This is the boolean value as defined in XPath. It is defined as that cast to xs:boolean for atomic values, ($s.length > 0$) for sequences or strings s. More detailed information can be found in [W3COp] or directly at http://www.w3.org/TR/2003/WD-xpath-functions-20030502/#func-boolean.

- *empty:* Test whether the argument is an empty sequence.
- *starts-with, ends-with:* Test whether a string begins/ends with another one.
- *not:* Negates a boolean argument.

String return type, again the return type is not a problem:

- *document-uri:* Takes a node as an argument and returns the URI of the document the node came from.
- *namespace-uri, local-name:* The namespace URI or local part of a node's name respectively.
- *lower-case, upper-case:* Converts a string to all (non-)capitals.
- *name:* The full name of a node, depending on the type of the node.
- *normalize-space:* Normalizes all space characters to a single ' '.
- *string:* Converts the argument into a string.
- *string-pad:* Concatenates a string multiple times.
- *substring(|-after|-before):* Return part of a string.
- *translate:* Replaces occurrences of a string fragment with another.

User Defined Functions

A user defined function is just a query expression given a convenient name. Thus it is not very hard to handle when generating the schema. It happens in three steps:

1. Upon first evaluation, create a new schema sequence and add it to the current type.
2. Set the sequence as the current type context.
3. Evaluated the body expression. This will generate further schema information.

The function signature makes it possible to make specific guarantees about the cardinalities of the returned data sequence (see section 6.3).

6.2.6 Conditions and Iteration

If-Statements

In XQuery conditional statements of the form

$$\textbf{if}\ (\textit{condition})\ \textbf{then}\ \textit{expression}_1\ \textbf{else}\ \textit{expression}_2 \tag{6.2}$$

are possible. It is not very difficult to find out that the corresponding XML Schema part will approximatly look like this:

```
<choice>
  <sequence>
    <!– schema for expression1 –>
  </sequence>
  <sequence>
    <!–schema for expression2–>
```

</**sequence**>
</**choice**> (6.3)

Note that no schema part appears for *condition*. It can however be any XQuery expression that evaluates to a boolean value. During the evaluation of this expression, generation of schema information must be turned off. Thus the overall eval() method of ConditionalExpression works like this:

1. Upon first execution create a choice particle c as a braket and two sequence particles s_1 and s_2 to hold the schema parts for *expression*$_1$ and *expression*$_2$ respectively. Insert s_1 and s_2 into c.
2. Turn off schema generation in the query execution context. This will prevent any schema information from being generated during the evaluation of *condition*.
3. Evaluate *condition* to find out which branch to take.
4. Turn schema generation back on.
5. Set the appropriate sequence s_1 or s_2 as the current type context.
6. Call eval() on the expression. This will fill the corresponding sequence.

It is interesting to observe, that ConditionalExpression creates no schema parts which represent contents in an instance document. Everything generated by the if-statement is pure structure that is present in the schema alone. This is a general property of control-flow structures and can also be observed with the next expression, FLWOR.

FLWOR Expression

XQuery provides the FLWOR expression to create output for all items of a sequence. In its simplest form it looks like this:

for *variable* **in** *expression*$_{in}$
return *expression*$_{return}$ (6.4)

The expression *expression*$_{in}$ evaluates to a sequence[9] s. The total F(LWO)R expression is executed by iterating over the elements of s, binding the current element to *variable* and executing *expression*$_{return}$ with this binding. This means that the number of executions of *expression*$_{return}$ depends on the number of elements in s. The corresponding XML Schema piece takes this form:

<**sequence** minOccurs="α" maxOccurs="β">
<!– *contents for expression*$_{return}$ –>
</**sequence**> (6.5)

Where α and β are cardinalities of the body and thus are related to the number of items n in s. As this number is known for certain, it is tempting to simply choose $\alpha = \beta = n$. However, one should keep in mind that the problem at hand is not to generate a schema that describes *one*, i.e. this single, XML instance, but the view as a whole. Thus a special effort is made to find practical cardinalities for the sequence. How this is done is specific to the expression returning the sequence and described in section 6.3 in more detail.

[9] Remember that any expression in XQuery evaluates to a sequence.

for *variable* **in** $\mathit{expression}_{in}$
let *variable′* := $\mathit{expression}_{let}$
where $\mathit{expression}_{filter}$
order by $\mathit{expression}_{sort}$
return $\mathit{expression}_{return}$ (6.6)

Figure 6.4: A more complete syntax of a for-loop.

In addition to the syntax shown above, FLWOR expressions can also have let, where and order by clauses. Let clauses bind further variables and have not direct effect on the schema. It is therefore not necessary to handle them explicitly. Where clauses filter the elements in s and reduce the actual number of iterations. This generally leads to overestimated cardinalities, which is not nice but also potentially harmful as it can result in a too high minOccurs. To avoid this problem minOccurs is set to zero whenever a where clause is encountered in a FLWOR expression. Finally order by clauses change the order in which the iterations occur. This may influence the structure of the output. As it is difficult to detect, when this is the case, order by clauses are disallowed in the prototype for the time being. An exception is thrown when a FLWOR expression contains order by clause.

6.2.7 Variable References

Dereferencing variables can create the need to generate appropriate schema information. As always, the decision whether this is necessary is left to the parent node. A common use of a variable reference is placing a copy of the value in the output document:

<**result**>
{ $var }
</**result**> (6.7)

This will create an element of name “result” who’s contents is determined by the value of $var.

The generation of schema information of the variable’s contents works like this[10]:

1. Upon first execution create a new sequence s that serves as the variable’s schema context. This sequence is inserted into the surrounding type.
2. Do the evaluation to find which nodes have actually been selected.
3. Create a new list for elements L. Iterate over the result list obtained from the evaluation of nodes and for each node N:
   (a) If N is an element node, add N to L.
   (b) If N is an attribute node, add the attribute to the surrounding type.
4. If L is non-empty:
   (a) Use the manager to induce a XML Schema sequence s' describing the list L.

[10] the algorithm is similar to the one used in LocationStep, but as there is no information on the complex type of a variable, there is no analogy to the handling of child-axis steps with fully qualified element names.

(b) Employing a TypeMerger as used in the induction algorithm and described in section 6.4.1 to merge $s\prime$ with what is already stored in the schema context s.

6.3 Cardinalities in Fixed Parts

Generating schema information directly from the query can result in more realistic—as compared to induction—cardinalities of particles in many cases. Note that "realistic" does not necessarily mean tighter or smaller, in the sense that they are closer to the number of elements actually occurring in the instance. When inducing a schema, you are presented with instance examples which also contain examples of the cardinalities of various particles. However, there is no way of telling what the declared cardinality of the respective particle would be if a user with knowledge of the particle's semantics were to declare the schema manually. When generating the schema from the query, it is often possible to get this information from properties of the query. As an illustration consider this query:

```
<week-days>{
  for $day in ("Mon", "Tue", "Wed", "Thu", "Fri")
  return
    <count day="$day">
      {count(/bill[day=$day]/line)}
    </count>
}</week-days>                                         (6.8)
```

The result of this query will be similar to this:

```
<week-days>
  <count day="Mon">1</count>
  <count day="Tue">5</count>
  <count day="Wed">3</count>
  <count day="Thu">0</count>
  <count day="Fri">8</count>
</week-days>                                          (6.9)
```

From the query one can tell that there will always be exactly five <count> elements. However, a induction algorithm that looks at the instance document only sees five <count> elements but has only one document as an example. This certainly does not justify generating a schema claiming that there are always exactly five elements, as there is high danger that this schema will only match the single instance document presented. Plausible cardinalities for <count> element from the point of view of the induction algorithm are: 0..5, 5..5, 5..*unbounded*, and 0..*unbounded*. In fact it could also employ a heuristic that would slightly broaden the counted cardinalities.

In order to preserve information about cardinalities *guaranteed cardinalities* of a sequence of nodes are introduced. *Guarateed cardinalities* are cardinalities which will hold regardless of the data the query is run against. Before each expression node returns the result of its evaluation to the parent node, it tries to attach guaranteed cardinalities as precise as possible to the sequence. The default guaranteed cardinalities of a new sequence are 0..*unbounded*, which hold for any sequence contents but are also as imprecise as possible. A node does not modify the guaranteed cardinalities of a sequence if they are not the default cardinalities (0..*unbounded*). The rationale behind this is that the innermost nodes have the most information available to deter-

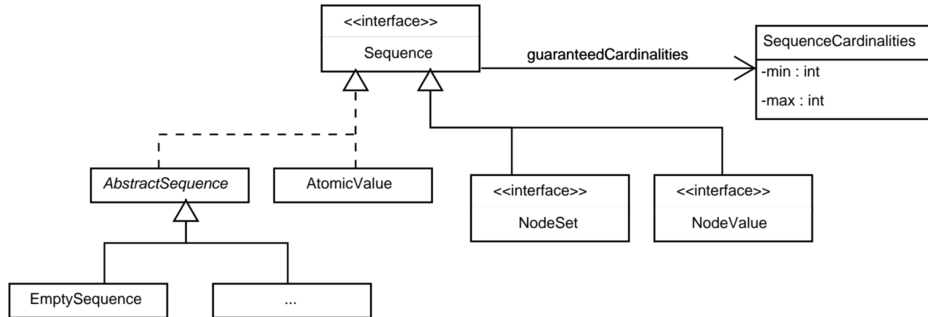


Figure 6.5: The Sequence class and its descendents model values in eXist's query processor.

mine the guaranteed cardnalities. Therefore the estimates made by the child nodes will generally be better then what the parent node can establish.

There are quite a few opportunities in XQuery to determine better guaranteed cardinalities, four of them are implemented in the prototype: sequences of literal values, sequences with only one item, XPath expressions and user defined functions. The cardinalities are stored in the classes representing the sequences. These classes all implement the Sequence interface as depicted in the class diagram in figure 6.5.

6.3.1 Literal Value Sequence

If the expression node that constructs a sequence can determine that the sequence is entirely constructed from literals, i.e. is a sequence literal, it can attach the precise guaranteed cardinalities to the sequence before passing it on. In the above example these are 5..5.

6.3.2 Single-valued Sequences

The XQuery implementation in eXist has a class AtomicValue that implements the Sequence interface. Its children and also the classes representing node values, have fixed guaranteed cardinalities of 1..1. In addition, there also is an implementation EmptySequence, which of course has guaranteed cardinalities of 0..0.

6.3.3 XPath: Location Steps

Consider a series of XPath location steps in the XPath expression E

$$E := /l_1/l_2/\dots/l_n \tag{6.10}$$

in which all l_i are location steps along the child-axis and give a fully qualified element name.

Let c_l^{min} be the minimum declared cardinality of l and c_l^{max} the maximum one. Then the guaranteed cardinalities of the whole expression E can be computed as:

$$c_E^{min} = \prod_{i=1}^{n} c_{l_i}^{min} \tag{6.11}$$

declared (XQuery notation)	declared (meaning)	guaranteed
	exactly one	1..1
?	zero or one	0..1
*	at least zero	0..*unbounded*
+	at least one	1..*unbounded*

Figure 6.6: Cardinalities for return types of user defined functions and their mapping to guaranteed cardinalities for the data list returned by the function.

$$c_E^{max} = \begin{cases} unbounded & \text{, if } \exists i \in \mathbb{N} \text{ with } c_{l_i}^{max} = unbounded \\ \prod_{i=1}^{n} c_{l_i}^{max} & \text{, else} \end{cases} \tag{6.12}$$

Where of course c_E^{max} is *umbounded* as soon as at least one $c_{l_i}^{max}$ is *unbounded*. If E starts from the root node as in 6.10, there still is the problem that E will be evaluated against an unknown number of documents. This leads to $c_E^{min} = 0$ and $c_E^{max} = unbounded$, because the guaranteed cardinalities of the root node are 0..*unbounded* as the number of documents is unknown.

If on the other hand E starts from the context of a variable reference or explicit document() node, for example:

document("/db/examples/april_bills.xml")/s:bill/s:line (6.13)

you can use 6.11 and 6.12 to compute the guaranteed cardinalities of the whole expression.

6.3.4 User defined Functions

User defined functions in XQuery have to declare their return type. This declaration also states how many instances of a primitive type are returned, figure 6.6 lists the possibilities. This extra information given by the user can be turned into useful guaranteed cardinalities. Again figure 6.6 shows which declaration is mapped to which cardinality.

6.4 Generation of Induced Parts

6.4.1 Induction Assumptions

Inducing schema information from (parts of) query results is a different problem setting than inducing from multiple complete static instance documents. The key differences are the lack of multiple instance documents and the absence of stability of node ordering. Some details were already outlined in section 2.4.

Because there is only one instance example, the induction algorithm implemented in the prototype creates relatively simple schema information and does not make use of many XML Schema constructs. More precisely, the contents of a complex type is assumed to always be flat, i.e. not employing any nesting of sequence or choice particles. The algorithm also does not attempt to determine a type more specific than xs:string for the attributes. While these assumptions seem rather simplistic, it also is difficult do justify the creation of both more complicated (with a structure that matches exactly the given instance) complex types and more specific simple types.

6.4.2 Algorithm

The algorithm recursively iterates over all elements in the instance. For each element e encountered it does this:

1. Determine whether it sees an element by this qualified name for the first time.
2. Construct a complex type n from the contents of the element. This complex type will exactly match the contents of this element.
3. If the element is not new:
   (a) Merge the created complex type n with type t of the existing element. Merging is done by moving along the contents of both complex types and generalizing t such that an instance complying to the old type t will also comply to n and the new t.

   else:

   (a) Store the new element and new type.
4. For each sub-element of e start at 1.

6.4.3 Implementation

The induction of schema information from instance data was implemented for different granularities of input data. The possibilities are: whole documents, single nodes or fragements of nodes. Induction for whole documents starts at the root node and recursively works its way all along the document. Likewise induction for single nodes happens by first handling the node and then recursively all its descendents. Finally induction can also be performed on a collection of siblings (i.e. a list of nodes that appears on the same level but has no common parent node associated).

As induction can happen multiple times during the generation of a schema for a view, the induction algorithm does not start anew every time, but needs to be aware of its own elements and types as well as elements or types generated directly from the query. The algorithm's main class (you can find a class diagram in figure 6.7) is Manager. It executes the outermost steps by moving along the node tree. To allow handling of the three types of input data discussed above, the Manager class has methods handleNode, handleDocument and handleFragment, which are responsibe for single nodes, whole documents and collection of siblings respectively.

To make the induction algorithm flexible, core pieces are encapsulated in strategies. Manager works with a strategy, that implements the fine points of the induction in a specific way. In the currenty implementation two strategies are supplied: SimpleStrategy (mainly of illustation and as a superclass for other strategies) and TypemergeStrategy (which is much superior to SimpleStrategy). Manager calls the appropriate methods on its strategy as needed. Figure 6.8 shows the four basic steps of the induction algorithm.

The partial results are stored in a ResultContainer that is accessible to both Manager and Strategy.

Manager

The core functionality of the Manager class is the traversal of the node tree. This is implemented in slightly different ways in the three entry-points (see above). All methods are rather similar, here handleNode is shown in more detail. The general

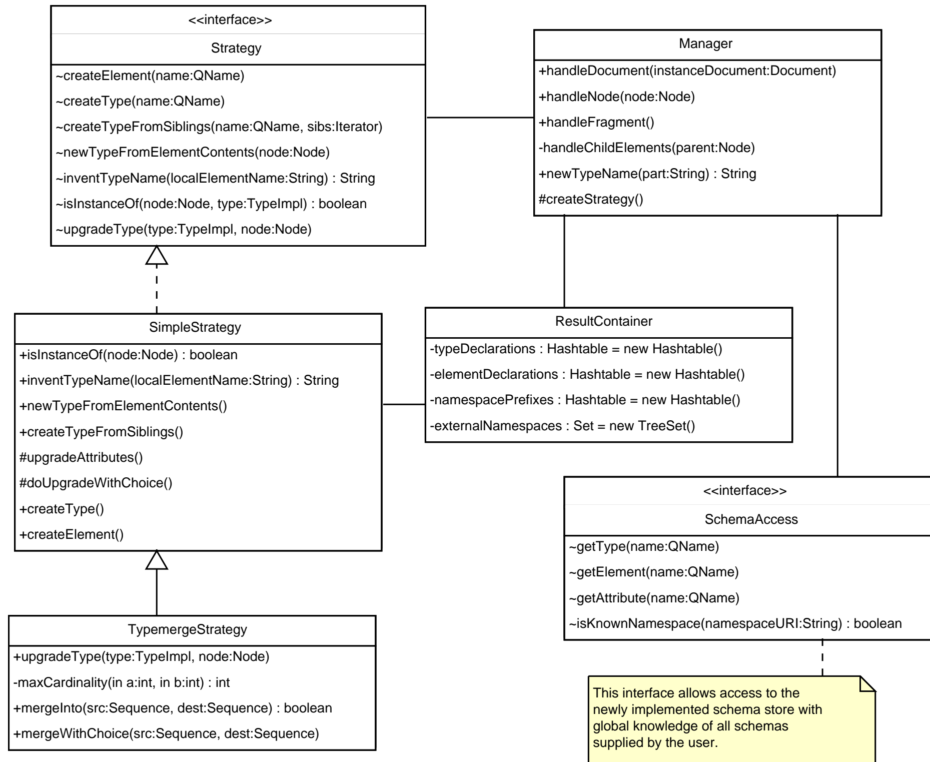


Figure 6.7: The most important classes related to schema induction.

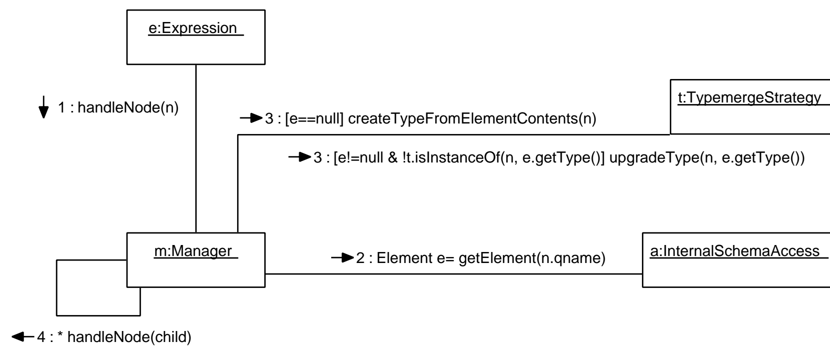


Figure 6.8: Collaboration of the classes involved in schema induction. The Manager recursively calls handleNode on the children of its current node. However, only one level of recursion is shown.

approach is to first find out, whether there already is an element declaration for the supplied node in the external schema store (i.e. created by the schema generation of an explicit element constructor). If this is the case, this declaration is also referenced in the ResultContainer instance internal to the induction algorithm. This is mainly done to simplify further handling in the strategies. If no declaration can be found externally, the method checks whether there is an internal one. If so, the associated type might need upgrading (i.e. the instance node presented now might be richer in structure than what was previously analyzed). If not, a new type is created.

```
method handleNode(Name node)
    QName elementName := node.name;
    ElementDeclaration externalElement := schemaAccess.getElement(elementName);
    if externalElement = null then
        // not externally known ...
        ElementDecl ownElment = results.getElement(elementName);
        if ownElement = null then
            // ... also not internally, create new type.
            ownElement = new ElementDecl;
            Type type = strategy.newTypeFromElementContents(node);
            ownElement.type = type;
            results.storeElement(ownElement);
        else
            // ... but internally, upgrading the type might be necessary.
            strategy.upgradeType(ownElement.type, node);
        end;
        handleChildElements(node);
    else
        // externally known, also make internally known.
        ElementDecl ownElement = new ElementDecl;
        element.type = externalElement.type;
        results.storeElement(ownElement);
    end;
end;                                                                    (6.14)
```

TypemergeStrategy

Figure 6.7 shows the two implemented strategies and the Strategy interface. The pseudo code for handleNode above shows that there are two important methods in strategies, namely newTypeFromElementContents and upgradeType. The former creates a new XML Schema type that will match the presented instance node. It is used when no previous schema information was found for the element. The latter extends the type to match the presented node while still matching any previously presented ones.

Here is how upgradeType from TypemergeStrategy works:

```
method upgradeType(Type type, Node node)
    Type tempType = createTypeFromContents(node);
    if not mergeInto(tempType, type) then
        doUpgradeWithChoice(type, node);
    upgradeAttributes(type, node);
end;                                                                    (6.15)
```

This does not seem too complicated, but as always the devil is in the details. First a new type called tempType is created to match the contents of the presented node. Keep in mind that the induction algorithm keeps the type contents flat, i.e. uses one sequence or choice particle to structure it. First merging tempType into the passed type is tried. This keeps the sequence particle and does not introduce any further nesting. Obviously it is not very hard to construct an example where this approach will not work. In this case, false is returned and the upgrade of the type is done using a choice element. Finally the attributes are upgraded. The last step is not difficult, as an attribute cannot appear more than once and the ordering also does not matter.

Merging tempType with the passed type is a bit tricky though. It can be subdivided into two steps, which—ignoring cardinalities greater than one—work like this:

1. Iterate over the elements a_i in the contents of type. Try to match them with the elements b_j from the contents of tempType. For each a_i:
   (a) If a_i and b_j match, all is well, increment i and j.
   (b) else if a_i matches b_{j+1} (i.e. there is an additional element in tempType), insert b_j into type but set b_j.minOccurs=0. There are two things to keep in mind with this step: (a) you might have to repreat it, as there can be multiple additional elements and (b) if you are inserting an element that has previously occured, the process fails. The last condition is important as it will prevent the creation of degenerated structures for types that are not really representable by a simple sequence structure.
   (c) else if a_i does not occur in tempType, set a_i.minOccurs=0;
2. When you reach the end of either the contents of type or the contents of tempType and there are elements left in the other type:
   (a) If there are elements left in type, set all their minOccurs to zero.
   (b) If there are elements left in tempType, append them to type. Also set their minOccurs to zero to preserve matching with previously presented nodes.

6.5 Schema Simplification

After the query is executed, complete schema information has been generated or induced. However, the representation is not quite standard. SchemaGenerationInfo holds one list of types and one of elements even though types and elements might be in different namespaces. Types also still contain SimpleParticles that need to be mapped to XML Schema. On top of this, these intermediate type representation are usually very bloated as each syntax node typically opens its own schema context. This leads to many unnecessary sequences calling for simplification.

Thus in this step all types are treated by simplifiers and converted to standard XML Schema representations. Different aspects of the simplification are encapsulated in individual strategies, which leads to (also see the UML diagramm in figure 6.10): EmptyParticleRemover, TrivialParticleRemover and SimpleTypeDiscoverer.

The simplification algorithm implemented in Simplifier iterates over all strategies and applies each to all types. The order in which the strategies are applied is important. Formally the simplification works like this:

for s in (EmptyParticleRemover, TrivialParticleRemover, SimpleTypeDiscoverer)
    for t in SchemaGenerationInfo.types
        (**new** s).accept(t) (6.17)

```
<xs:schema xmlns:ns1="..." xmlns:xs="..." xmlns:ns2="..."
      xmlns:xdt="..." targetNamespace="...">
   <xs:import namespace="http://localhost/l2/source.xsd"/>
   <xs:element name="list" type="ns1:Tlist0"/>
   <xs:complexType name="Tlist0">
      <xs:sequence>
         <xs:sequence>
            <xs:sequence>
               <xs:sequence minOccurs="0" maxOccurs="unbounded">
                  <xs:sequence>
                     <xs:element ref="ns2:bill"/>
                  </xs:sequence>
               </xs:sequence>
            </xs:sequence>
         </xs:sequence>
      </xs:sequence>
   </xs:complexType>
</xs:schema>                                                      (6.16)
```

Figure 6.9: Example of an unsimplified schema. The schema contains a chain of sequences only one of which is necessary. The simplified version can be found in section 7.1.2

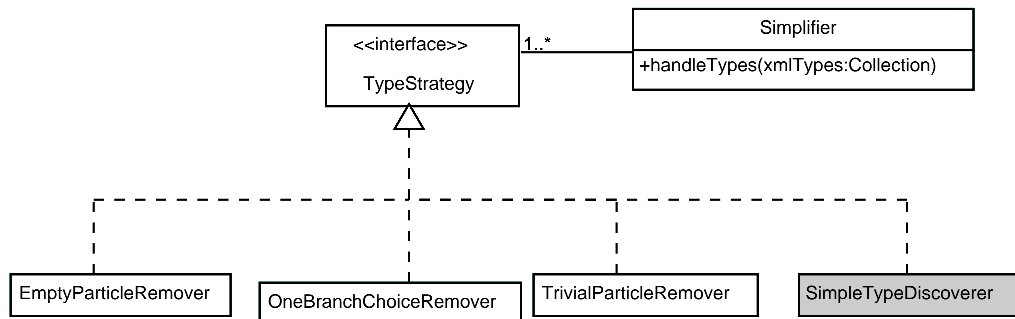


Figure 6.10: The classes concerned with schema simplification. The Simplifier runs a number of TypeStrategies that carry out different simplification tasks. SimpleParticleDiscoverer is not a simplifier in the strict sense but handles the SimpleParticles that are special in the schema representation used here.

In more detail, each strategy implements a task that can be separated from the other strategies. In the following “<s>” denotes a XML Schema sequence, “<c>” a choice, and “<e>” an element.

- *EmptyParticleRemover:* Removes empty sequence or choice particles. As a general rule, empty particles have no effect. For example:

  <s><e/><c/></s> ⇒ <s><e/></s> (6.18)

  There is however one exception: if a choice particle contains two particles, of which one is empty and the other is not, the empty particle must not be removed. If the minimum cardinality of the non-empty particle is 1, the whole choice is equivalent to a simpler form:

  <c><e/><s/></c> ≡ <e minOccurs='0'/> (6.19)

  The last optimization is not implemented in the EmptyParticleRemover.

- *TrivialParticleRemover:* A trivial particle contains only one particle and is thus obsolete. Choices and sequences can be trivial particles (element obviously cannot as they do not contain anything in the first place). From all combinations of sequence and choice, one gets four cases:

  <s><s><e/></s></s> ⇒ <s><e/></s>
  <s><c><e/></c></s> ⇒ <c><e/></c>
  <c><s><e/></s></c> ⇒ <s><e/></s>
  <c><c><e/></c></c> ⇒ <c><e/></c> (6.20)

  Note that when replacing the outer particle, the new particle is always of the type of the inner particle, regardless of the ordering of sequence or choice. The cardinalities of the new particle are computed by multiplying those of inner and outer particle.

- *SimpleTypeDiscoverer:* This strategy is used to find SimpleParticles and modify the containing type accordingly. The possible cases were already outlined in section 6.2.3.

Without simplification the generated schemas can be rather ugly. Figure 6.9 gives an example of an unsimplified schema. It contains a chain of useless sequence particles that was caused by different syntax nodes inserting their sequence as the current type context. The query expression of the view is that of the second testcase in section 7.1.2. The TrivialParticleRemover removes all but one sequence while preserving the proper cardinalities. The simplified schema is given along with the testcase.

Chapter 7

Evaluation

Summary. *The prototype was tested using testcases selected from the XQuery usecases found in [W3CUc] as well as own ones. This chapter briefly introduces the testcases and the generated schemas. It also discusses the results. Full input and output data can be found in appendix B.*

7.1 Test Cases

7.1.1 XQuery Usecases

A number of XML Query usecases are available at [W3CUc] to demonstrate applications of the language. Out of the 83 usecases presented, four were selected for a test of the prototype. The choice was made based on three criteria: (a) the usecase can of course only use XQuery syntax that the prototype can process, otherwise the test would make little sense, (b) the query has to produce a wellformed XML document (one common failure was to not have a root element) and (c) out of the remaining usecases some were picked that keep the effort to write XML schemas for the underlying documents low. The usecases are based on DTDs (which are given on the website) and there are no XML Schemas available. Thus schemas had to be written for every example document used. Using DTDs means that (among other things) the possible cardinalities are limited. Element cardinalities in DTDs can only be 0..1, 1..1, 0..*unbounded* and 1..*unbounded*. All usecases work on a single instance document.

Views were defined using each query. Note that the view context does not really matter as all queries explicitly name the document they work on. The queries were prefixed with namespace declarations and all element constructors were given fully qualified names to give the schema generator a target namespace for the schemas. The URL of the input document also had to be changed as it pointed to a document on a remote server.

The usecases are:

1. *Primitive Restructure:* This usecase creates a structure that is very similar to the original document. It iterates over a sequence of input nodes and outputs one new node for each. Inside this node, information from the originating document is copied.

```
declare namespace bib="http://bstore1.example.com/bib.xml";
```

```
declare namespace n="http://localhost/n";
<n:bib>
  {
    for $b in doc("/db/xquery_usecases/bib.xml")/bib:bib/bib:book
    where $b/bib:publisher = "Addison-Wesley" and $b/year > 1991
    return
      <n:book year="{ $b/year }">
        { $b/bib:title }
      </n:book>
  }
</n:bib>                                                        (7.1)
```

Contained syntax nodes that are relevant to schema creation:

- ElementConstructor (element constructors)
- PathExpr (containing the whole query; also around the XPath expressions)
- LocationStep (steps along the child and attribute axes)

Generated Schema:

```
<xs:schema
    xmlns:ns1="http://bstore1.example.com/bib.xml"
    xmlns:xs="..." xmlns:ns2="http://localhost/n" xmlns:xdt="..."
    targetNamespace="http://localhost/n">
  <xs:import namespace="http://bstore1.example.com/bib.xml"/>
  <xs:element name="bib" type="ns2:Tbib1"/>
  <xs:element name="book" type="ns2:Tbook1"/>
  <xs:complexType name="Tbib1">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="ns2:book"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Tbook1">
    <xs:sequence>
      <xs:element ref="ns1:title"/>
    </xs:sequence>
    <xs:attribute name="year" type="xs:string" use="required"/>
  </xs:complexType>
</xs:schema>                                                    (7.2)
```

2. *Nesting of variable bindings in FLWOR loop:* This query creates a result element for every title-author pair in the originating document. Pairing the titles with the respective authors is done in the for-clause with a nested binding structure. Note that the bindings of $t as well as $a both reference $b, which is bound first. What this actually means can be best understood by looking at the structure created by the query processor. It is converted to: for $b in ... return (for $t in ... return (for $a in ... return ...))). Thus this usecase also is a test of whether the schema generation for for-expressions is truely orthogonal.

```
declare namespace bib="http://bstore1.example.com/bib.xml";
```

```
declare namespace n="http://localhost/n";
<n:results>
  {
    for $b in doc("/db/xquery_usecases/bib.xml")/bib:bib/bib:book,
      $t in $b/bib:title,
      $a in $b/bib:author
    return
      <n:result>
        { $t }
        { $a }
      </n:result>
  }
</n:results>
```
(7.3)

Contained syntax nodes that are relevant to schema creation:

- ElementConstructor
- ForExpr
- EnclosedExpr (containing the variable references)
- VariableReference (copying the title and authors bound the for-clause)

Generated Schema:

```
<xs:schema
    xmlns:ns1="http://bstore1.example.com/bib.xml" xmlns:xs="..."
    xmlns:ns2="http://localhost/n" xmlns:xdt="..."
    targetNamespace="http://localhost/n">
  <xs:import namespace="http://bstore1.example.com/bib.xml"/>
  <xs:element name="result" type="ns2:Tresult2"/>
  <xs:element name="results" type="ns2:Tresults2"/>
  <xs:complexType name="Tresult2">
    <xs:sequence>
      <xs:element ref="ns1:title"/>
      <xs:element ref="ns1:author"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="Tresults2">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="ns2:result"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```
(7.4)

3. *Joining two documents:* The result of this usecase is somewhat similar to the joining of two relations in SQL. Analogously, it works on two sequences of nodes. These nodes are books from two documents, but are represented by elements of different complex type. However, both complex types contain the title of the book, which is used for joining, and the price. Note that the actual joining is performed with a FLWOR expression that contains a where-clause. This

expression has—similar to the previous usecase—nested variable bindings, that create a cartesian product of the two book sequences. The where-clause then selects those pairs that are of interest[1]. Finally, the body creates a new node for every pair that contains title and prices from both files.

```
declare namespace bib="http://bstore1.example.com/bib.xml";
declare namespace rev="http://bstore1.example.com/reviews.xml";
declare namespace n="http://localhost/n";
<n:books-with-prices>
  {
    for $b in doc("/db/xquery_usecases/bib.xml")//bib:book,
      $a in doc("/db/xquery_usecases/reviews.xml")//rev:entry
    where $b/bib:title = $a/rev:title
    return
      <n:book-with-prices>
        { $b/bib:title }
        <n:price-bstore2>{ $a/rev:price/text() }</n:price-bstore2>
        <n:price-bstore1>{ $b/bib:price/text() }</n:price-bstore1>
      </n:book-with-prices>
  }
</n:books-with-prices>                                          (7.5)
```

Contained syntax nodes that are relevant to schema creation:

- ElementConstrutors
- FLWOR loop
- EnclosedExpr
- PathExpr (for the XPath originating from the variable references)
- VariableReference

Generated Schema:

```
<xs:schema
    xmlns:ns1="http://localhost/n" xmlns:xs="..."
    xmlns:ns2="http://bstore1.example.com/bib.xml" xmlns:xdt="..."
    targetNamespace="http://localhost/n">
  <xs:import namespace="http://bstore1.example.com/bib.xml"/>
  <xs:element name="book-with-prices" type="ns1:Tbook-with-prices0"/>
  <xs:element name="books-with-prices" type="ns1:Tbooks-with-prices0"/>
  <xs:element name="price-bstore1" type="xs:string"/>
  <xs:element name="price-bstore2" type="xs:string"/>
  <xs:complexType name="Tbook-with-prices0">
    <xs:sequence>
      <xs:element ref="ns2:title"/>
      <xs:element ref="ns1:price-bstore2"/>
      <xs:element ref="ns1:price-bstore1"/>
```

[1] It is quite obvious that there is some room for optimization for the query processor here: It is unnecessary to compute the full cartesian product only to later filter it. Due to the manner in which for-loops with nested binding are implemented in the query processor, eXist does not do this yet.

```
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Tbooks-with-prices0">
        <xs:sequence>
            <xs:element minOccurs="0" maxOccurs="unbounded"
                ref="ns1:book-with-prices"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```
(7.6)

4. *Restructure with attributes:* This usecase again transforms the base document into a simpler version. Of the original book data only year and title are kept. The usecase also employs a where-clause to filter the sequence of books. The main difference to previous usecases lies in the handling of attributes. The query copies an attribute from the original node. It does so not by specifying an attribute constructor and copying the value, but by giving an XPath expression inside the element construtor that references the attribute ($b/@year).

```
declare namespace bib="http://bstore1.example.com/bib.xml";
declare namespace n="http://localhost/n";
<n:bib>
    {
        for $b in doc("/db/xquery_usecases/bib.xml")//bib:book
        where $b/bib:publisher = 'Addison-Wesley' and $b/year > 1991
        return
            <n:book>
                { $b/@year }
                { $b/bib:title }
            </n:book>
    }
</n:bib>
```
(7.7)

Contained syntax nodes that are relevant to schema creation:
• PathExpr (containing the whole query) • ElementConstructor • FLWOR expression • EnclosedExpr • LocationSteps (this time also for attributes, note that the attribute is automatically inserted into the surrounding element)

Generated Schema:

```
<xs:schema
        xmlns:ns1="http://localhost/n"
        xmlns:xs="..." xmlns:ns2="http://bstore1.example.com/bib.xml"
        xmlns:xdt="..." targetNamespace="http://localhost/n">
    <xs:import namespace="http://bstore1.example.com/bib.xml"/>
```

```
<xs:element name="bib" type="ns1:Tbib0"/>
<xs:element name="book" type="ns1:Tbook0"/>
<xs:complexType name="Tbib0">
    <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="unbounded" ref="ns1:book"/>
    </xs:sequence>
</xs:complexType>
<xs:complexType name="Tbook0">
    <xs:sequence>
        <xs:element ref="ns2:title"/>
    </xs:sequence>
    <xs:attribute name="year" type="xs:string" use="required"/>
</xs:complexType>
</xs:schema>
```

(7.8)

7.1.2 Own Views

The XQuery usecases have a few disadvantages. Firstly they only work on one document, which somewhat limits their practical relevance as most views in real databases will be defined over whole collections of documents. Secondly they originally were designed to work on documents described by DTDs. For puroposes of testing the DTDs were converted into XML Schemas here, but these schemas of course do not come close to fully utilizing the capabilities of XML Schema. An example of this are the cardinalities, which are—compared to XML Schema—rather limited in a DTD. Thirdly the chosen usecases do not actually use all the XQuery functionality the prototype can generate schema information for. To address these issues, the prototype was also tested using some self-invented view definitions.

1. *Summary of bills by day-of-week:* This view works over a collection of bills each stored in an individual document. The bills conform to the schema bill.xsd which can be found in appendix B.2.1 alongside with an example of the input documents and the output of the query.

```
declare namespace bill="http://localhost/testbill/bill.xsd";
declare namespace n="http://localhost/bill_view";

declare function local:totalValue($bill as element) as xs:double {
    sum(for $l in $bill/bill:line return ($l/bill:qty * $l/bill:itemPrice))
};

<n:result>
    {
    for $day in ('Mon', 'Tue', 'Wed', 'Thu', 'Fri')
    return
    <n:day name="$day"> {
        for $bill in /bill:bill[contains(bill:due-date, $day)]
        return
        <n:bill-summary>
            <n:short-sender>
                {$bill/bill:addressing/bill:sender/bill:company/text()}
            </n:short-sender>
            <n:short-receiver>
```

```
            {$bill/bill:addressing/bill:receiver/bill:company/text()}
         </n:short-receiver>
         <n:item-count>{count($bill/bill:line)}</n:item-count>
         <n:total-value>{local:totalValue($bill)}</n:total-value>
      </n:bill-summary>
      }
   </n:day>
   }
</n:result>
```
(7.9)

Generated Schema:

```
<xs:schema
      xmlns:ns1="http://localhost/bill_view" xmlns:xs="..." xmlns:xdt="..."
      targetNamespace="http://localhost/bill_view">
   <xs:element name="bill-summary" type="ns1:Tbill-summary0"/>
   <xs:element name="day" type="ns1:Tday0"/>
   <xs:element name="item-count" type="xs:integer"/>
   <xs:element name="result" type="ns1:Tresult0"/>
   <xs:element name="short-receiver" type="xs:string"/>
   <xs:element name="short-sender" type="xs:string"/>
   <xs:element name="total-value" type="xs:double"/>
   <xs:complexType name="Tbill-summary0">
      <xs:sequence>
         <xs:element ref="ns1:short-sender"/>
         <xs:element ref="ns1:short-receiver"/>
         <xs:element ref="ns1:item-count"/>
         <xs:element ref="ns1:total-value"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="Tday0">
      <xs:sequence>
         <xs:element minOccurs="0" maxOccurs="unbounded" ref="ns1:bill-summary"/>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" use="required"/>
   </xs:complexType>
   <xs:complexType name="Tresult0">
      <xs:sequence>
         <xs:element minOccurs="5" maxOccurs="5" ref="ns1:day"/>
      </xs:sequence>
   </xs:complexType>
</xs:schema>
```
(7.10)

Contained syntax nodes that are relevant to schema creation:

- ElementConstructor
- FLWOR
- Sequence of literal values
- Enclosed expressions
- Path expressions

2. *Structural Unification:* This view transforms bill documents with a simple structure (namespace "http://localhost/2") into bills with the richer structure used in the previous example. The new bills are given as a list and surrounded by a new list element. Transformations to a different schema like the one presented here usually have a rather complex query. This is due to the input data having to be unwrapped from its previous structure and put into the newly created target structure. To do this, it is generally necessary that one copies pieces of atomic data. It might sometimes also be possible to copy whole node structures if the source and target schema overlap.

```
declare namespace a="http://localhost/2";
declare namespace s="http://localhost/l2/source.xsd";
declare namespace n="http://localhost/full-bill-list";
<n:list>
{
for $b in /a:bill
return <s:bill>
        <s:addressing>
            <s:sender>
                <s:company>{$b/a:sender/text()}</s:company>
                <s:street/><s:city/><s:country/>
            </s:sender>
            <s:receiver>
                <s:company>{$b/a:receiver/text()}</s:company>
                <s:street/><s:city/><s:country/>
            </s:receiver>
        </s:addressing>
        { for $l in $b/a:line
        return
            <s:line>
                <s:qty>{$l/a:qty/text()}</s:qty>
                <s:name>{$l/a:name/text()}</s:name>
                <s:itemPrice>{$l/a:price/text()}</s:itemPrice>
            </s:line> }
    </s:bill>
}</n:list>                                                    (7.11)
```

The generated schema is of course very simple, this was the whole point of doing the unification:

```
<xs:schema xmlns:ns1="http://localhost/full-bill-list" xmlns:xs="..."
        xmlns:ns2="http://localhost/l2/source.xsd" xmlns:xdt="..."
```

```
targetNamespace="http://localhost/full-bill-list">
    <xs:import namespace="http://localhost/l2/source.xsd"/>
    <xs:element name="list" type="ns1:Tlist0"/>
    <xs:complexType name="Tlist0">
        <xs:sequence>
            <xs:element minOccurs="0" maxOccurs="unbounded" ref="ns2:bill"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```
(7.12)

Contained syntax nodes that are relevant to schema creation:

- Element constructors
- FLWOR
- Enclosed expressions

7.2 Results

7.2.1 Covered Parts of XQuery

The prototype does not cover the whole of XQuery, but it can handle the most common syntax nodes: Constructors for text, elements and attributes, enclosed expressions, XPath expressions, atomic and user defined functions, FLWOR, conditional expressions, variables and cast expression. This allows a wide variety of view definitions.

One does have to be careful though not to employ any unsupported syntax, as this is likely to yield wrong schemas. The prototype does not throw an exception, if schema generation for a node is not handled. It ignores the node and pretends all is well. This is a major drawback, but fixing it would have involved modifying about 200 classes to throw the exception.

7.2.2 Quality of the Generated Schemas

The overall quality of the schemas generated in the above testcases is fairly good. Particularly it is an improvement over what could have been achieved by induction (see section 2.4 for some limitations of induction). All schemas describe the view well, the double-check against the result data worked in all cases. This section discusses some aspects in greater detail.

Cardinalities

The cardinalities that are generated (i.e. not induce from result data) are guanateeable for further evaluations of the view. This was highly desireable as a schema for the view and not some result at a particular point in time was sought. Cardinalities of induced parts have the usual defficiency of not being certain. The prototype induces rather coarse cardinalities, because it only has one instance document to induce from. However, these might still be wrong (e.g. optional elements not detected).

Grouping

The grouping (structuring by sequence and choice particles) works well and reflects the actual structure of the view, because it is created directly from the query. Inducing

rich structure is tricky, especially if there is insufficient exmaple data as is the case here. Therefore the prototype keeps induced grouping flat. It uses a single sequence with appropriate cardinalities of elements if possible, otherwise the whole contents of the complex type is converted into a choice. This will certainly not result in the best possible schema, but it is difficult to find arguments that would justify a more complex grouping structure.

One problem that is related to grouping might occur when generating schema information for conditional (if) expressions. As described in section 6.2.6 two sequences are nested inside a choice particle. Each sequence corresponds to one of the possible execution paths. If however one of the paths is never taken no schema is generated for it. This is a problem, as the generated schema is to describe the view and not a particular result. Unfortunately fixing this is hard: While negating the test-expression in order to take the other branch is not difficult, setting up a proper execution context for this branch is. The algorithm would have to find out appropriate bindings for all variables used in the branch. There is hope for the special case where the branch does not dereference any variables. This could be implemented in a future version.

Simple Types

Though the most accurate simple type is found in many cases, there is one case where it is not: When the contents of simple type is created by an XPath expression. This is due to the fact that the LocationSteps do not yet know their schema type and can thus not give any information about the simple types of their attributes or elements. At first glance it seems quite easy to add this, however there are a few issues: (1) LocationSteps can be a lot more complex than just simple child-axis steps with no wildcards[2]. Generally as soon as there is a wildcard test, the XPath expression will create a whole tree of possible types. On top of this, type information is not easily accessible if it is defined locally in the schema. Finally there are other XQuery structures that exhibit similar problems. Thus the problem of complex typing should be addressed for the whole language. This was beyond the scope of the prototype.

There are non-standard XQuery extensions in some implementations that give the language a more powerful complex type system (such as casting for complex types and defining complex types). However, there is no such extension in the XQuery implementation in eXist.

Unnecessarily Generated Schema Parts

As is illustrated in the testcases above, the prototype generates some unnecessary schema pieces. Firstly namespaces are sometimes declared that are not referenced in the schema. This is simply a weakness of the XMLGenerator class that transforms the internal schema representation into the final XML structure. The prefix-namespace mappings are stored in a global datastructure. When the DOM tree of a schema is created, XMLGenerator does not check which other schemas are actually referenced. Secondly attributes that are non-optional are marked as “required”, which is not wrong but also the default.

[2]Example: /child::*[local-name(.)="bill" or local-name(.)="invoice")/@*[contains(., "123")], figuring out the type of those attributes it certainly not trivial.

Chapter 8

Conclusions

Summary. *As discussed in the previous chapter, the generation of schema information for views works. There are a few points that have influence on the quality of the generated schema and are discussed here.*

8.1 Usefullness of the Prototype

The presented prototype shows that generation of schema information from the view's query is very much worthwhile. The two main reasons for this are: (1) the schema represents the view itself instead of a single result instance and (2) more precise information on various parts is available from the query that makes the schema more stable.

By extending the prototype further, it will be possible to create schema information for most parts of the view directly. The full support of all XPath expression is the major issue here. Solving it will be difficult and only possible with full XML Schema awareness of XQuery. It is not feasible for the generation algorithms to partially implement this, as there are many places where this can be of use (e.g. also with variable references). Schema awareness must be increased for the whole language.

8.2 Observations

Some points worth mentioning were noticed in the course of this thesis:

- Schema generation works much better if base documents of the query have schemas associated. This seems quite obvious, there are two strong reasons for it:

  1. Element and type declarations can be referenced. This leads to the connection of input and output data being reflected in the view's schema. Preserving this information can be very important to allow users to grasp the semantics of the view.
  2. If there is no schema available for the input documents, it has to be inferred. This is not as catastrophic as inferrence for the view's output, as there are normally more sample documents available and orderings are less likely to change. Still the inferred schema will be less precise than a given one.

- The Prototype fails to type attributes and elements of simple type more specifically than xs:string if their contents is copied into a new attribute/element (e.g. with .../text()). This is the case because the result of text() will certainly be of type xs:string (which the prototype can of course figure out) but more specific types are lost in the process of evaluating the LocationStep. To avoid this problem, it would be necessary to process schema types along with the data while doing the LocationSteps. Since this is a bit tricky[1], it was left out in the prototype. This problem would also be solved by raising the XQuery implementation's awareness of XML schema.

- One has to be careful when using XPath expressions in queries for views. It is quite easy to reference elements this way who's element and/or type declarations are not accessible from the outside. This impedance mismatch between XML Schema and XQuery has already been discussed in detail in section 6.2.1. There are two measures one can take to avoid it: (1) be careful to only select elements that lead to cut-points (i.e. references from one schema to another) at global elements and (2) make all element declarations global, but one often does not have control over the referenced schema[2].

[1] In practice LocationSteps aren't always child-axis steps without any wildcards and no predicates. The simple use of the decendant-of-self axis complicates things very much.

[2] To really solve this problem, two approaches are possible. On the one hand, one could introduce changes to XML Schema to allow referencing of local elements. This could for example be done by giving a "path" of parent elements starting from the next higher global element. If on the other hand "local" is to be taken to mean "private" XPath selections that retrieve local elements should be disallowed, at least if they are carried out in the context of schema generation. After all, a private element is an element that usually does not make sense by itself.

Chapter 9

Future Work

Summary. *This chapter first gives ideas on further work to enhance the qualitiy of the generated schema. The second part shows two applications for the generated schema that the author believes to be useful. Finally some possible future developments in the database system as well as interconnections to outside systems are discussed.*

9.1 Enhancements to Schema Generation

Several possible fields for enhancements were discovered during development of the prototype. Generally these could help to allow a more complete coverage by schema generation (as opposed to induction) and a better quality of the resulting schema.

An important point is the use of existing schema information to improve schema generation. In this context the question of meta-queries over the schema documents arises. Does using meta-queries over the schemas stored in the database allow any improvements? If so how do you figure out what the relevant schema pieces are? This might also have benefits for query execution itself as it may be possible to limit the search scope by an appropriate schema context. Relevant queries over the schema store will need to be identified. To make meta queries practicable it is very important that they run very efficiently as they will by far outnumber the user queries.

A problem that highly restricts the possible queries is that of non-global cut-points (discussed in section 6.2.1). If any advances can be made in further invesigation of this problem of mismatch between XQuery and XML Schema, they would be very helpful. This problem is a fairly common one and there may be special cases where it is possible to find solutions for cut-points at non-global elements. Implementing these can relieve the user of some constraints.

The raw schema representation obtained from the generation during query execution is an inefficient encoding. It contains numerous redundancies which are taken out by the simplification step. This multi-step process is similar to what is done in compilers that first generated a non-optimal intermediate language that is optimized in the next step.

An interesting topic would be to try to increase the scope of optimization in the simplifiers of the prototype. Currently a simplifier usually works on just one type, some look at the direct contents of this type. There is a lot of research in the compiler

community about optimization scopes and their benefits. There might be room for much improvement here.

Generic algorithms for induction of schema information from instance examples are a well-researched subject. Adapting such an algorithm to the needs of schema generation from queries (i.e. using it for the inference parts) can help to increase the quality of induced parts. One has to be careful to take special properties of schema generation for views into account. The achieved advantages would slowly fade away as the prototype is made capable of generating schema for more and more parts directly.

As the schema generation algorithms in the prototype become evermore complete, the question of which queries can fully treated by the generation algorithms is of rising interest. It will thus be important to find criteria for a classification of queries. Ideally this classification would allow an a priori decision whether induction will be necessary for a specific query.

9.2 Applications of Schema Generation

The generated schema can be used to aid the formulation of queries over the view. Taking this idea further requires a system-wide schema store and its integration with the user frontend in the query dialog. The DataGuides in [GW97] were developed with this application in mind.

The query dialog could be enhanced to allow a form of "code completion" similar to that of integrated development environments today. This can decrease the error rate as well as speed up query formulation. The user no longer has to consult the individual view definitions for information on possible structures. Instead the system could gather this information for the user.

Another benefit of the availablility of schemas for views are the new opportunities in change control. One could use the schemas to check for structural changes and thus give applications a guaranteeable datastructure for their input. This can be achieved by formulating a special view for the application (that contains the data needed, possibly transformed from various input documents). The schema can be regenerated every time the application uses the view. To inform the application of structural changes, an execption is thrown when the schema of the view changes. This allows applications to work on stable representations of data. They can thus make the assumptions necessary for efficient processing. Note that this requires a firm notion of "schema equivalence". Apart from being a bit difficult to establish in general, different applications might also have different ideas about its details.

9.3 System Context

In future developments it will generally be interesting to make the whole database system aware of schema information wherever possible. Having schema information integrated and not as an external add-on would give the opportunity to perform various tasks and optimizations. There are many places where this integration has to happen, for example in making the query language fully schema aware, but also during importing of data. Being able to give type information on elements can enable semistructured databases to implement concepts like type-aware indexing known from relational databases. As these concepts are well researched much might be achievable with relatively little effort. A key challenge here will be to exploit the new possibilities

when schema information is available, but to still function when it is not. After all an important task of semistructured databases is the unification of diverse structure.

Flexible mechanisms to store and use schema will be of great interest because of the diverse nature of the stored data. When new data with an associated schema arrives, the schema information needs to be integrated at once. This should not cause any disruptions to database operations. Implementing this will require the investigation of various problems related with schema version evolutions.

Finally as one has evermore information on structure available, the evaluation of semantic awareness of the database will be interesting. One could use the schema information to make associations with an external ontology. In a primitive first attempt concepts from the ontology could be associated with element of a particular name, the main use of this approach being in the domain of XML-as-data. If XML is used as markup, element names have usually little semantic meaning (e.g. the fact that something is a paragraph of a book usually does not tell you which concepts from the ontology are used in the paragraph).

Bibliography

[Ab97] Serge Abiteboul et al. "Views for Semistructured Data". *Proceedings of SIGMOD Workshop on Management of Semistructured Data (with SIGMOD'97)*, Tuscon, Arizona, May 1997.

[Ab99] Serge Abiteboul. "On Views and XML". *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, Philadelphia, 1999.

[AMV98] S. Abiteboul, J. McHugh, M. Rys, V. Vassalos, J. L. Wiener. "Incremental Maintenance for Materialized Views over Semistructured Data". VLDB 1998 submission 653.

[Cas] Castor Project. Internet at *http://www.castor.org/*. Accessed 1 July 2004.

[BLP98] C. Baru, B. Ludscher, Y. Papakonstantinou, P. Velikhov, V. Vianu. "Features and Requirements for an XML View Definition Language: Lessons from XML Information Mediation". In *Proceedings of QL98 - The Query Languages Workshop* (1998). Or internet at *http://www.db.ucsd.edu/publications/xmas.html* or *http://www.w3.org/TandS/QL/QL98/pp/xmas.html*, accessed 24 June 2004

[BS02] F. Bry and S. Schaffert. "A Gentle Introduction into Xcerpt, a Rule-based Query and Transformation Language for XML". Internet at *http://www.pms.informatik.uni-muenchen.de/publikationen/PMS-FB/PMS-FB-2002-11.pdf*. Accessed June 2004.

[EN00] Ramez Elmasri, Shamkant B. Navathe. "Views (Virtual Tables) in SQL". In *Fundamentals of Database Systems*. Third Edition, Addison-Wesely, 2000.

[eX] Open Source XML Database. Internet at *http://exist-db.org*. Accessed on 24 June 2004.

[Gl97] Dieter Gluche et al. "Incremental Updates for Materialized OQL Views". In *Proceedings of International Conference on Deductive and Object-Oriented Databases (DOOD'97)*, 1997, pp. 52-66.

[Gra00] Minos Graofalakis et al. "XTRACT: A System for Extracting Document Type Descriptors from XML Documents". In *Proceedings of ACM SIGMOD'2000*, Dallas, Texas, May 2000, pp. 165-176.

[GST04] Torsten Grust et al. "XQuery on SQL Hosts". *Proceedings of the 30th Int'l Conference on Very Large Databases (VLDB 2004)*, Toronto, Canada, August/September 2004.

[GW97] Roy Goldman, Jennifer Widom. "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases". In *Proceedings of 23rd International Conference on Very Large Data Bases*, 1997

[Jag02] H.V. Jagadish et al. "TIMBER: A native XML database". In *The VLDB Journal (2002) 11*, pp. 274-291, 2002.

[Kat01] Howard Katz. "An Introduction to XQuery". IBM developerworks at *http://www-106.ibm.com/developerworks/xml/library/x-xquery.html* (2001). Accessed 24 June 2004.

[Kes] Stephan Kesper, "A proof of the Turing-completeness of XSLT and XQuery". Technical Report. University of Tübingen, 2002. Or Internet at *http://tcl.sfs.uni-tuebingen.de/~kepser/papers/xsltxq.pdf*, accessed on 1 July 2004.

[KKN03] Rajasekar Krishnamurthy, Raghav Kaushik, Jeffrey F. Naughton, "XML-to-SQL Query Translation Literature: The State of the Art and Open Problems". *XML Symposium (XSym) 2003*, 2003

[Mei02] Wolfgang Meier. "eXist: An Open Source Native XML Database". In *Akmal B. Chaudri, Mario Jeckle, Erhard Rahm, Rainer Unland (Eds.): Web, Web-Services, and Database Systems.* NODe 2002 Web- and Database-Related Workshops, Erfurt, Germany, October 2002. Springer LNCS Series, 2593.

[W3C99] World Wide Web Consortium. "XML Path Language (XPath) Version 1.0", Internet at *http://www.w3c.org/TR/xpath.* Accessed on 24 June 2004.

[W3C01] World Wide Web Consortium, "The Extensible Stylesheet Language Family (XSL)", Internet at *http://www.w3c.org/Style/XSL/.* Accessed on 24 June 2004.

[W3C03] World Wide Web Consortium, "XML Query (XQuery)", Internet at *http://www.w3.org/XML/Query.html.* Accessed on 24 June 2004.

[W3COp] World Wide Web Consortium. "XQuery 1.0 and XPath 2.0 Functions and Operators". At *http://www.w3.org/TR/2003/WD-xpath-functions-20030502/* accessed 14 June 2004.

[W3CUc] World Wide Web Consortium. "XQuery Use Cases", Internet at *http://www.w3.org/TR/xquery-use-cases/.* Accessed on 24 June 2004.

[Wo03] Raymond K. Wong, Jason Sankey, "On Structural Inference for XML Data". Technical Report, University of New South Wales, Australia, June 2003.

[XDB] XML:DB Initiative, "XML Database API Draft". Internet at *http://www.smb-tec.com/xmldb/xapi/xapi-draft.html.* Accessed on 1 July 2004.

[Xin] xindice XML database. "Apache xindice". At *http://xml.apache.org/xindice/* accessed on 1 July 2004.

[LPV99] B. Ludäscher, Y. Papakonstantinou, P. Velikhov, V. Vianu. "View Definition and DTD Inference for XML". Post-ICDT Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats (SSD99). Jerusalem, January 1999.

[Wad02] Philip Wadler. "XQuery: A Typed Functional Language for Querying XML". In *Advanced Functional Programming: 4th International School*. 2002 Oxford, UK. Internet at http://www.cs.uu.nl/~johanj/afp/afp4/xquery-afp.pdf.

[W3C02] World Wird Web Consortium. "XML Schema". At http://www.w3.org/XML/Schema, as accessed 24 June 2004.

Appendix A

Glossary

Axis (XPath axis) Each step of an XPath expression happens along an axis. This axis denotes the relation between originating node(s) and selected node(s). For example if a selection along the parent axis is made on a sequence of nodes N, the result of this selection will be a sequence containing all nodes that are parent to a node in N.

Choice (XML Schema choice particle) Choice particles are used to indicate that one item from the contents can occur.

Complex type (in XML Schema). Any type that is not a simple type. See [W3C02] for a detailed definition.

DOM (XML DOM) Document Object Model. Datamodel for XML documents.

DTD Document Type Definition. Used to describe the structure of XML documents. Due to some severe limitations of DTDs (such as lack of powerful typing) new description languages for XML structures were introduced, including XML Schema.

eXist XML database written in Java. The code base for the presented prototype as it is implemented into eXist's query processor. Further information can be found in [eX] and [Mei02].

FLWOR (XQuery expression) For-Let-Where-Order by-Return expression that allows iteration over nodes in a sequence given in the for clause. Let allows binding of additional variables, where filters, order by manages sorting and return constructs the return value of the expression.

MDL Minimum Message Length.

LGPL Lesser Gnu General Public License. Opensource license that allows the user to freely distribute the software and make any changes desired. Any version of the software (changed or unchanged) must be distributed in source (and binary) if distributed at all. The LGPL also allows linking of libraries covered by it to non-open software.

Particle (XML Schema Particle) Particles in XML Schema are used to represent the contents of a complex type. Available particles are choice and sequence

particles (see other glossary entries), which are used to structure the contents, as well as element particles, which represent XML elements.

Sequence (XML Schema sequence particle) Sequence particles are use to indicate that the all sequence's contents must occur in the instance document and that it must be in the order given.

Simple type (in XML Schema). A simple type is an atomic type (e.g. numbers, dates, but also character strings). Type are divided into simple and complex in XML Schema. See [W3C02] for a more detailed definition.

Xcerpt Pattern based selection language for XML. Discussed in section 4.1 and [BS02].

xindice XML database hosted at the Apache Software Foundation. Uses XPath as a selection language for queries. See [Xin] for more details.

XML Query Path based Query language for XML. Discussed in section 4.1 and [W3C03].

XML Schema Language to describe the structure of XML instance documents. Successor technology of DTDs, major enhancements lie the areas of types (especially simple ones) and cardinalities.

XPath Path based selection language that is used in various other techologies (e.g. XML Query and XSLT) to select nodes. Selection is done by specifying a path through the document tree. Note that this path does not have to be from top to bottom, thanks to various selection axes (see there).

XQuery abbreviation of XML Query (see there).

XSLT eXtensible Stylesheet Language Transformations. XML transformation language mainly for use in conjunction with the Extensible Stylesheet Language. Can also be use independently as a query language for XML. See section 4.1 and [W3C01].

Appendix B

Details on Testcases

Summary. *This appendix shows two of the XQuery usecase based testcases and one of the own ones in full detail for ease of reference.*

B.1 XQuery Usecases

B.1.1 Primitive Restructure

Input Document

The input document for this testcase is show here in an abbreviated form:

```
<?xml version="1.0" encoding="UTF-8"?>
<bib xmlns="http://bstore1.example.com/bib.xml">
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author>
    <last>Stevens</last>
      <first>W.</first>
    </author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <!– many more books –>
</bib>                                                    (2.1)
```

Input Schema

The input schema is used in many of the XQuery usecases, it is only given here with the first one:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:r="http://bstore1.example.com/bib.xml"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://bstore1.example.com/bib.xml">
  <xs:element name="affiliation" type="xs:string"/>
```

```
    <xs:element name="author" type="r:author_t"/>
    <xs:element name="bib" type="r:bib_t"/>
    <xs:element name="book" type="r:book_t"/>
    <xs:element name="editor" type="r:editor_t"/>
    <xs:element name="first" type="xs:string"/>
    <xs:element name="last" type="xs:string"/>
    <xs:element name="price" type="xs:string"/>
    <xs:element name="publisher" type="xs:string"/>
    <xs:element name="title" type="xs:string"/>
    <xs:complexType name="author_t">
        <xs:sequence>
            <xs:element ref="r:last"/>
            <xs:element ref="r:first"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="bib_t">
        <xs:sequence>
            <xs:element maxOccurs="unbounded" minOccurs="0" ref="r:book"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="book_t">
        <xs:sequence>
            <xs:element ref="r:title"/>
            <xs:choice>
                <xs:element ref="r:editor"/>
                <xs:element maxOccurs="unbounded" minOccurs="1" ref="r:author"/>
            </xs:choice>
            <xs:element ref="r:publisher"/>
            <xs:element ref="r:price"/>
        </xs:sequence>
        <xs:attribute name="year" type="xs:integer" use="required"/>
    </xs:complexType>
    <xs:complexType name="editor_t">
        <xs:sequence>
            <xs:element ref="r:last"/>
            <xs:element ref="r:first"/>
            <xs:element ref="r:affiliation"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

(2.2)

Query

```
declare namespace bib="http://bstore1.example.com/bib.xml";
declare namespace n="http://localhost/n";
<n:bib>
    {
        for $b in doc("/db/xquery_usecases/bib.xml")/bib:bib/bib:book
        where $b/bib:publisher = "Addison-Wesley" and $b/year > 1991
        return
            <n:book year="{ $b/year }">
```

```
            { $b/bib:title }
         </n:book>
   }
</n:bib>                                                        (2.3)
```

View Result

```
<n:bib xmlns:n="http://localhost/n">
   <n:book year="1994">
      <title xmlns="...">TCP/IP Illustrated</title>
   </n:book>
   <n:book year="1992">
      <title xmlns="...">Advanced Programming in the Unix environment</title>
   </n:book>
</n:bib>                                                        (2.4)
```

Generated Schemas

```
<xs:schema
      xmlns:ns1="http://bstore1.example.com/bib.xml"
      xmlns:xs="http://www.w3.org/2001/XMLSchema"
      xmlns:ns2="http://localhost/n"
      xmlns:xdt="http://www.w3.org/2003/05/xpath-datatypes"
      targetNamespace="http://localhost/n">
   <xs:import namespace="http://bstore1.example.com/bib.xml"/>
   <xs:element name="bib" type="ns2:Tbib1"/>
   <xs:element name="book" type="ns2:Tbook1"/>
   <xs:complexType name="Tbib1">
      <xs:sequence>
         <xs:element minOccurs="0" maxOccurs="unbounded" ref="ns2:book"/>
      </xs:sequence>
   </xs:complexType>
   <xs:complexType name="Tbook1">
      <xs:sequence>
         <xs:element ref="ns1:title"/>
      </xs:sequence>
      <xs:attribute name="year" type="xs:string" use="required"/>
   </xs:complexType>
</xs:schema>                                                    (2.5)
```

B.1.2 Nesting of variable bindings

Input Documents

Same as testcase "Primitive Restructure".

Input Schema

Same as testcase "Primitive Restructure".

Query

```
declare namespace bib="http://bstore1.example.com/bib.xml";
declare namespace n="http://localhost/n";
<n:results>
    {
        for $b in doc("/db/xquery_usecases/bib.xml")/bib:bib/bib:book,
            $t in $b/bib:title,
            $a in $b/bib:author
        return
            <n:result>
                { $t }
                { $a }
            </n:result>
    }
</n:results>                                                    (2.6)
```

View Result

The actual result contains more books, like the source document:

```
<n:results xmlns:n="http://localhost/n">
    <n:result>
        <title xmlns="http://bstore1.example.com/bib.xml">TCP/IP Illustrated</title>
        <author xmlns="http://bstore1.example.com/bib.xml">
            <last>Stevens</last>
            <first>W.</first>
        </author>
    </n:result>
    <!-- more books -->
</n:results>                                                    (2.7)
```

Generated Schema

```
<xs:schema
        xmlns:ns1="http://bstore1.example.com/bib.xml"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:ns2="http://localhost/n"
        xmlns:xdt="http://www.w3.org/2003/05/xpath-datatypes"
        targetNamespace="http://localhost/n">
    <xs:import namespace="http://bstore1.example.com/bib.xml"/>
    <xs:element name="result" type="ns2:Tresult2"/>
    <xs:element name="results" type="ns2:Tresults2"/>
    <xs:complexType name="Tresult2">
        <xs:sequence>
            <xs:element ref="ns1:title"/>
            <xs:element ref="ns1:author"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Tresults2">
```

```
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="ns2:result"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

(2.8)

B.2 Own Cases

B.2.1 Summary of bills by day-of-week

Input Documents

Only one input document is shown here, the others only differ in the values:

```
<?xml version="1.0" encoding="UTF-8"?>
<b:bill xmlns:b="http://localhost/testbill/bill.xsd">
  <b:addressing>
    <b:sender>
      <b:company>A Corp.</b:company>
      <b:street>1st Street</b:street>
      <b:city>Somewhere</b:city>
      <b:country>Cuba</b:country>
    </b:sender>
    <b:receiver>
      <b:company>B Corp.</b:company>
      <b:street>2st Street</b:street>
      <b:city>Somewhere</b:city>
      <b:country>Cuba</b:country>
    </b:receiver>
  </b:addressing>
  <b:due-date>Mon 5 Jan 2004</b:due-date>
  <b:line>
    <b:qty>1</b:qty>
    <b:name>Some Product</b:name>
    <b:itemPrice>3.99</b:itemPrice>
  </b:line>
  <b:line>
    <b:qty>2</b:qty>
    <b:name>Some Product II</b:name>
    <b:itemPrice>15.43</b:itemPrice>
  </b:line>
</b:bill>
```

(2.9)

Input Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:s="http://localhost/l2/source.xsd"
    elementFormDefault="qualified"
    targetNamespace="http://localhost/l2/source.xsd">
  <xs:element name="company" type="xs:string"/>
```

```
    <xs:element name="street" type="xs:string"/>
    <xs:element name="city" type="xs:string"/>
    <xs:element name="country" type="xs:string"/>
    <xs:element name="qty" type="xs:integer"/>
    <xs:complexType name="address">
        <xs:sequence>
            <xs:element ref="s:company"/>
            <xs:element ref="s:street"/>
            <xs:element ref="s:city"/>
            <xs:element ref="s:country"/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="sender" type="s:address"/>
    <xs:element name="receipient" type="s:address"/>
    <xs:complexType name="addresses">
        <xs:sequence>
            <xs:element ref="s:sender"/>
            <xs:element ref="s:receipient"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="lineType">
        <xs:sequence>
            <xs:element ref="s:qty"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="itemPrice" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="billType">
        <xs:sequence>
            <xs:element name="addressing" type="s:addresses"/>
            <xs:element maxOccurs="100" minOccurs="0" name="line" type="s:lineType"/>
        </xs:sequence>
    </xs:complexType>
    <xs:element name="bill" type="s:billType"/>
</xs:schema>                                                                    (2.10)
```

Query

```
declare namespace bill="http://localhost/testbill/bill.xsd";
declare namespace n="http://localhost/bill_view";

declare function local:totalValue($bill as element) as xs:double+ {
    sum(for $l in $bill/bill:line
        return ($l/bill:qty * $l/bill:itemPrice))
};

<n:result>
{
    for $day in ('Mon', 'Tue', 'Wed', 'Thu', 'Fri')
    return
    <n:day name="{$day}"> {
```

```
        for $bill in /bill:bill[contains(bill:due-date, $day)]
        return
            <n:bill-summary>
                <n:short-sender>
                    {$bill/bill:addressing/bill:sender/bill:company/text()}
                </n:short-sender>
                <n:short-receiver>
                    {$bill/bill:addressing/bill:receiver/bill:company/text()}
                </n:short-receiver>
                <n:item-count>{count($bill/bill:line)}</n:item-count>
                <n:total-value>{local:totalValue($bill)}</n:total-value>
            </n:bill-summary>
        }
    </n:day>
}
</n:result>
```

(2.11)

View Result

```
<n:result xmlns:n="http://localhost/bill_view">
    <n:day name="Mon">
        <n:bill-summary>
            <n:short-sender>A Corp.</n:short-sender>
            <n:short-receiver>B Corp.</n:short-receiver>
            <n:item-count>2</n:item-count>
            <n:total-value>34.85</n:total-value>
        </n:bill-summary>
        <n:bill-summary>
            <n:short-sender>Fruits Online</n:short-sender>
            <n:short-receiver>B Corp.</n:short-receiver>
            <n:item-count>3</n:item-count>
            <n:total-value>136.8</n:total-value>
        </n:bill-summary>
    </n:day>
    <n:day name="Tue">
        <n:bill-summary>
            <n:short-sender>Real Stationary</n:short-sender>
            <n:short-receiver>B Corp.</n:short-receiver>
            <n:item-count>2</n:item-count>
            <n:total-value>403.7</n:total-value>
        </n:bill-summary>
    </n:day>
    <n:day name="Wed"/>
    <n:day name="Thu"/>
    <n:day name="Fri"/>
</n:result>
```

(2.12)

Generated Schema


```
<xs:schema
        xmlns:ns1="http://localhost/bill_view"
        xmlns:xs="http://www.w3.org/2001/XMLSchema"
        xmlns:xdt="http://www.w3.org/2003/05/xpath-datatypes"
        targetNamespace="http://localhost/bill_view">
    <xs:element name="bill-summary" type="ns1:Tbill-summary0"/>
    <xs:element name="day" type="ns1:Tday0"/>
    <xs:element name="item-count" type="xs:integer"/>
    <xs:element name="result" type="ns1:Tresult0"/>
    <xs:element name="short-receiver" type="xs:string"/>
    <xs:element name="short-sender" type="xs:string"/>
    <xs:element name="total-value" type="xs:double"/>
    <xs:complexType name="Tbill-summary0">
        <xs:sequence>
            <xs:element ref="ns1:short-sender"/>
            <xs:element ref="ns1:short-receiver"/>
            <xs:element ref="ns1:item-count"/>
            <xs:element ref="ns1:total-value"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="Tday0">
        <xs:sequence>
            <xs:element minOccurs="0" maxOccurs="unbounded" ref="ns1:bill-summary"/>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" use="required"/>
    </xs:complexType>
    <xs:complexType name="Tresult0">
        <xs:sequence>
            <xs:element minOccurs="5" maxOccurs="5" ref="ns1:day"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>                                                          (2.13)
```

Appendix C

The CD

Attached you will find a data CD that contains files related to this thesis.

Literature - The referenced literature in PDF format, as far as it is available publically.

Presentation - The final presentation of this thesis.

Software - A runnable version of eXist that includes the presented prototype. You additionally need a recent Java Runtime Environment.

Source - The whole sourcecode of the modified version of eXist.

Thesis - This thesis in PDF and Postscript format.