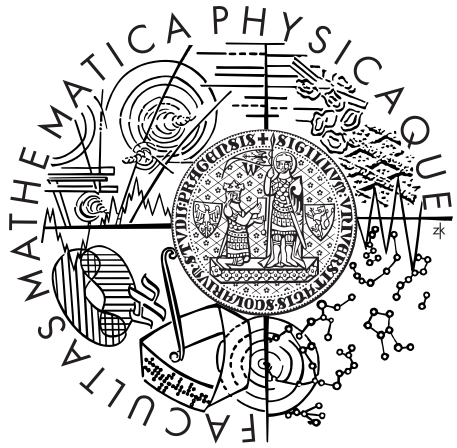
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Marek Polák

XML Query Adaptation

Department of Software Engineering

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

Study programme: Informatics

Specialization: Software systems

Prague 2011

I would like to thank to my supervisor RNDr. Irena Mlýnková, Ph.D. for her helpful suggestions, thorough notes, provided related research material and text corrections.
I would also like to thank to Mgr. Martin Nečaský, Ph.D. for his suggestions and comments.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague on July 1, 2011 Marek Polák

Název práce: *Adaptace XML Dotazů*
Autor: *Marek Polák*
Katedra (ústav): *Karedra softwarového inženýrství*
Vedoucí diplomové práce: *RNDr. Irena Mlýnková, PhD.*
e-mail vedoucího: *irena.mlynkova@ksi.mff.cuni.cz*

Abstrakt: *V předložené práci studujeme evoluci XML schémat, jejích typů a vlivu na dotazy, které jsou na příslušných schématech závislé. Práce obsahuje přehled existujících přístupů tohoto problému. Přístup představený v této práci ukazuje možné řešení jak upravovat dotazy závislé na schématech během jejich evoluce. Práce dále obsahuje popis algoritmu, který upraví dotaz v závislosti na evoluci příslušného schématu. V neposlední řadě práce obsahuje sadu experimentů, které ověřují návrh algoritmů a ukazují jejich výhody a nevýhody.*

Klíčová slova: *XML, XML Schema, XML Path, Evoluce*

Title: *XML Query Adaptation*
Author: *Marek Polák*
Department: *Department of Software Engineering*
Supervisor: *RNDr. Irena Mlýnková, PhD.*
Supervisor's e-mail address: *irena.mlynkova@ksi.mff.cuni.cz*

Abstract: *In the presented work we study XML schema evolution, its types and impact on queries which are related on the particular schema. The thesis contains a review of existing approaches of this problem. The approach presented in this work shows a possible solution how to adapt related queries while schema evolves. The thesis contains a description of an algorithm which modifies queries related to the evolved schema. Finally the work contains a number of experiments that verify proposal of the algorithms and show their advantages and disadvantages.*

Keywords: *XML, XML Schema, XPath, Evolution*

Contents

Chapter 1

Introduction

1.1 Motivation

Since the eXtensible Markup Language (XML) [1] has become a de-facto standard for data representation and manipulation, there exists a huge amount of applications having their data represented in XML. However, since most of applications are dynamic, sooner or later the structure of the data needs to be changed and so have to be changed also all related issues. We speak about so-called evolution and adaptability of XML applications. One of the aspects of this problem is to adapt the respective operations over the evolving XML data, in particular XML queries, expressed, e.g., in XPath [2] or XQuery [3].
The fact that evolution of XML schemes is still an open problem these days causes that XML databases are not used in cases where they will be more suitable than for example relational databases. But the ability of schema evolution in relation databases is the key reason why we still use them [4].

Possible changes of the document schema can be caused by many reasons:

- Changes which are caused by system development.
  If the schema and queries are used as internal, it can be deemed as a development change, which is not a relevant situation. It is a very common situation while system is being developed and designers have to adjust implementation to customer's requirements.

- Adapting the schema to an existing interface.
  If the schema is published for processing by others (for example as a Web Service [5] which provides information about health insurance and which is used by tens of consumers), one change on producer side can cause tens of changes on consumers side. Since this scenario is common in practice, it is solved generally by versioning of the schemes and with a support period during which the consumers must update their queries to the new version compatible with new schema. But it does not solve the problem - the changes

must still be done to ensure the functionality.

The change of the schema can cause that the queries using this schema may return various results when it is applied on the new schema:

- The result is the same as over the original schema.
- It returns an error, for example caused by a non existing element or bad type casting.
- It returns more results than over the original schema.
- It returns less results than over the original schema.

All these presented possibilities are not desired and should be recognized and corrected. Suppose that there exists query *//purchase/item* returning all elements *item* which are children of elements *purchase* in the document. Now, if the name of element *item* is changed (for example to name *items*), all queries where this name is used or which can have impact on the result of the query must be checked by designer and updated alternatively.

1.2 Aim of the Thesis

The aim of this work is a research on possibilities and limitations of XML query adaptation. The thesis analyzes existing solutions and discusses their advantages and disadvantages. The core of the work is a proposal and implementation of own approach dealing with selected disadvantages and open issues. The proposal involves classification of the modification of XML schema and respective queries and adaptation steps as well as discussion of possible/necessary user involvement. The purpose of the thesis is to design a technique and algorithm how to:

- detect changes in XML schema
- analyze these changes and their possible impact on related queries
- if the query needs to be updated, apply correspond update operation on the query to satisfy valid results
- if the correct update is not possible, notify the designer about this situation which have to be solved manually

1.3 Structure of the Thesis

The structure of the thesis is follows. Chapter 2 presents 5-level XML evolution architecture [6] which is used to be able to connect XML schema with queries at platform-specific level. Next are briefly described used background technologies - XML Schema as an XML document schema and XPath language as used query language. In Chapter 3 are presented and discussed related works which are dealing with preserving and identifying changes between XML schemes and queries.

Chapter 4 describes XSEM [7], a conceptual models for modeling XML, especially its platform-specific model XSEM-H which is used in this thesis as a model of XML schema. In Chapter 5 is presented used subset of XPath syntax and its platform-specific visualization model. In the end of the chapter is described a mapping between XSEM-H and XPath models and algorithm how to recognize possible changes in the results of the query if the schema model is changed.

Chapter 6 contains analysis of possible changes in schema model and suggests algorithms how the changes should be propagated to target XPath model to preserve compatibility. Chapter 7 briefly presents experimental implementation of the presented solution and describes experiments with the real-word examples. Finally, Chapter 8 concludes and provides future research directions of this approach.

Chapter 2

XML Evolution Architecture

In this chapter we describe proposal of architecture XML applications [6]. Basically it can be partitioned into five horizontal layers that are interconnected with layer above and bellow. This suggestion of interconnections enables straight propagation of changes between related components of the model. All layers are described bellow and shown in Figure 2.0.1.

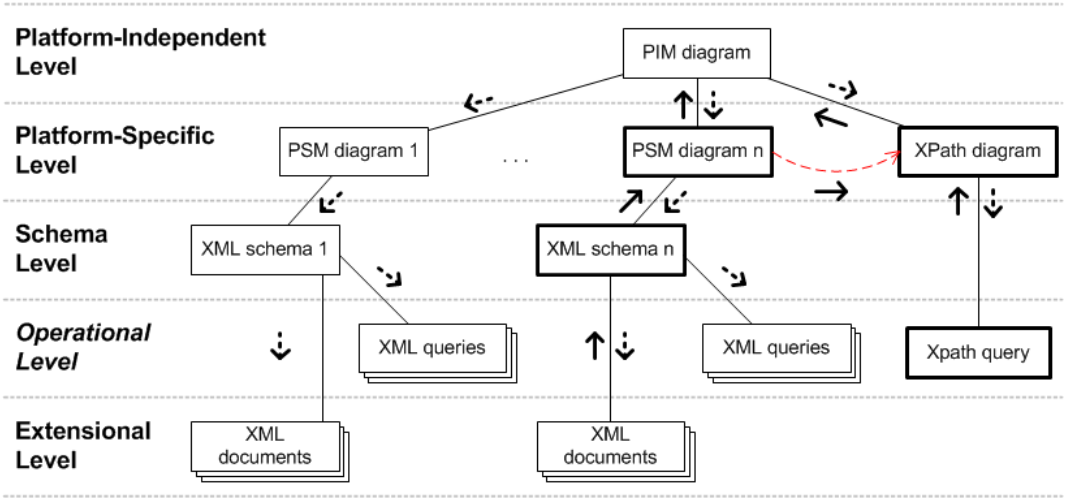


Figure 2.0.1: Five level evolution architecture

Platform-independent Level describes the domain independently of the considered XML formats. The model at this level is called *Platform-Independent Model* (PIM). An example is UML class model. There are many possibilities how can be such model interpreted in the concrete domain, that is presented in next paragraph.

Platform-specific Level describes concrete XML format. The model at this level contains special components and features relevant to its purpose. *Platform-Specific Model* (PSM) is used at this level.

Logical level contains XML schemes which describe XML documents modeled in platform-specific level by PSM. An example of XML schema is XML Schema [8].

Operation level contains queries for appropriate XML formats. The most know examples of query or transformation languages used with XML are XPath (XML Path Language) [9], XQuery [3] and XSLT (eXtensible Style-sheet Language) [10].

Extensional level contains concrete documents respecting the appropriate XML format from schema level.

Besides horizontal division of the architecture, it can be partitioned and interconnected vertically too. That is shown in Figure 2.0.1 with the red line. In this thesis we will focus on models on the platform specific level and evolution process between them. Especially we will use XSEM PSM model (Section 4.2) for XML Schema and XPath model (Section 5.2) for representing XPath queries.

2.1 XML Schema

XML schemes are important when we decide to publish our XML data for others. A schema gives a contract between a producer and a consumer how the data will look like. It provides the structure of an XML document and used data types. The best known and the most used languages these days are DTD [11] and XML Schema [8]. If we have a schema of a particular XML document, we can use it for various reasons:

- If we have an XML document and an XML schema, we can use a parser (or a validator) to check if the document is valid against the schema.
- We can create queries to get data and information from the given document on the basis of its schema in the similar way like in a relation database. The most commonly used query languages today are XPath (Section 2.2) and XQuery languages which are using XML Schema syntax and types to create queries.
- With XSLT standard we can create transformation scripts on the basis of XML schema which for given input document(s) create new documents with absolutely different structure following the transformation script.
- Built-in types which XML Schema definition contains can be used by other standards mentioned above (XPath or XSLT) which makes combination of these technologies easier and complex.

2.2 XML Path Language (XPath)

XML Path Language is a language for addressing parts of an XML document defined by W3C (World Wide Web Consortium) [12], now in version 2.0. It uses the tree structure of the XML document to select particular nodes and to compute values (e.g. boolean, string or numbers) from the content of given XML document as a result.
XPath 2.0 is a super-set of previous version of XPath 1.0. There were made changes in data models and type system. The result of an XPath 2.0 query can be a sequence of nodes from the input document or a sequence of atomic values with cardinality $< 1, n >$.
The XPath 2.0 language is used in XSLT 2.0 and in XQuery 1.0 standards as their subset.

2.2.1 XPath Data Model

Definition 2.1. (Node). Base construct of XPath data model.

Definition 2.2. (XML Document). An *XML document* is a tree-structured (hierarchical) collection of nodes.

The XPath language defines seven types of nodes in its data model (tree). These node types are used in path expressions to select nodes or nodes-sets in XML document. All nodes must satisfy general constraints described closely in [9]. Types of nodes are:

1. Root Node
   It is the topmost node of the tree and encapsulates an XML document.

2. Element Node
   Element nodes encapsulate XML elements.

3. Attribute Node
   It represents an XML attribute.

4. Text Node
   Text Nodes encapsulate XML character content.

5. Comment Node
   Comment Nodes encapsulate XML comments.

6. Processing Instruction Node
   Processing Instruction Nodes encapsulate XML processing instructions.

7. Namespace Node
   A namespace node represents the binding of a namespace URI to a namespace prefix or to the default namespace.

2.2.2 XPath Axes

XPath operates on an XML document as a tree and uses axes which express how the data are related to the current node. Every path consists of steps and every step consists of axes, selection of nodes and predicates (see Section 2.2.3). There are defined thirteen axes in the XPath language. Their graphical representation is shown in Figure 2.2.1.

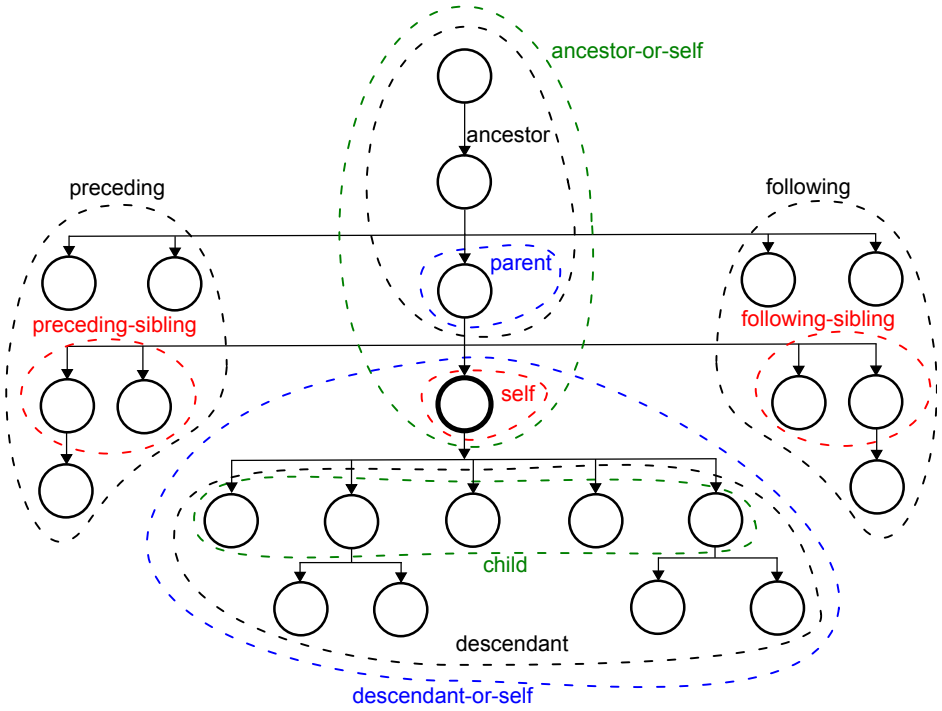


Figure 2.2.1: XPath axes

1. Ancestor
   Selects all ancestors of the current node (parent, grandparent, etc.) to the root.

2. Ancestor or self
   Selects all ancestors of the current node and the current node itself.

3. Attribute
   Selects all attributes of the current node.

4. Child
   Selects all child nodes of the current node.

5. Descendant
   Selects all descendants of the current node (children, grandchildren, etc.).

6. Descendant or self
   Selects all descendants of the current node and the current node itself.

7. Following
   Selects all nodes in the document after the closing tag of the current node.

8. Following sibling
   Selects all siblings after the current node.

9. Parent
   Selects the parent of the current node.

10. Preceding
    Selects all nodes in the document that are before the start tag of the current node.

11. Preceding sibling
    Selects all siblings before the current node.

12. Self
    Selects the current node.

13. Namespace
    Selects all namespace nodes of the current node.

2.2.2.1 Abbreviations

To make the XPath language user-friendly and useable, there are defined some abbreviations for axes, eg. a parent node (..), current node (.) or an attribute node (@*attribute_name*), The full set of them is described in [9].

2.2.3 Location Steps

Location steps are used to navigate to specific node collection. Each location step in the XPath consists of three parts:

1. The axis

   It navigates to the specific nodes. The *axis identifier* and the *node* are separated by double colon $'::'$. It identifies a group of nodes that have specified relationship to the current node.

2. The node test

   A test, which specifies the node type.

3. The set of predicates

   Predicates filter the result set returned by an axis and a node test. There can be zero or more predicates in the location step. They are contained within square brackets $'['$, $']'$.

The syntax for a location step is *axis* :: *node_test*[*predicate*]. A location step defines how to identify a set of nodes relative to a current node.
Location path is the most important construct in XPath. It consists of location steps delimited with forward slash '/' and forms the query.

Definition 2.3. (Absolute Location Path). An *absolute location path* consists of '/' optionally followed by a relative location path. A '/' itself selects the root node of the document containing the context node. If it is followed by a relative location path, then the location path selects the set of nodes that would be selected by the relative location path relative to the root node of the document containing the context node.[1]

Definition 2.4. (Relative Location Path). A *relative location path* consists of a sequence of one or more location steps separated by '/'. The steps in a relative location path are composed together from left to right. Each step in turn selects a set of nodes relative to a context node. An initial sequence of steps is composed together with a following step as follows. The initial sequence of steps selects a set of nodes relative to a context node. Each node in that set is used as a context node for the following step. The sets of nodes identified by that step are unioned together. The set of nodes identified by the composition of the steps is this union.[1]

2.2.4 Other Model Parts

Except described part of XPath syntax it contains another constructs which must be always implemented in any XPath API. An example are *core functions* which offer basic set of operations above the model.

[1] W3C definition [9]

Chapter 3

Related Work

In this chapter existing papers related to XML schema and query evolution are discussed. There exist several approaches dealing with evolution of the XML schemes like CoDEX [13] or with query compatibility. Next, there are approaches analyzing possible changes of the document schema and including recommendations how the schemes should be written to reduce potential changes of the related queries. But we have found no paper discussing how the evolution of the queries should be done automatically or semi-automatically with minimal contribution of the designer.
The chapter contains analysis of three works. The first one deals with preserving valid queries during the schema evolution. The second paper presents a framework for analyzing compatibilities between different versions of the schemes. The last one presents a framework for processing evolution of the XML schemes.

3.1 Preserving XML Queries during Schema Evolution

Paper [4] discusses XML schemes and their evolution and transformation in time and the problems which it brings to administrators. The main aims of the paper are:

- To present taxonomy of possible changes in XML schemes.
- To overview their impact on schema structure.
- To introduce guidelines for managing schema by controlling its changes and writing queries across schema versions.

3.1.1 Taxonomy of XML Schema Changes

The authors examined and divided the changes into two basic groups which are shown in Table 3.1.2 and described below.

Basic Changes	Complex Changes
Refinement	Element composition
Removal	Element decomposition
Extension	Renaming
Reinterpretation	Optionality
Redefinition	Renumbering
	Retyping
	Namespaces
	Default values
	Reordering

Table 3.1.2: Defined changes

1. **Basic changes**

   (a) Refinement
   Adds a new element into the schema.

   (b) Removal
   Deletes an element from the schema.

   (c) Extension
   Adds a new construct into the schema (for example a new complex type definition). This new added construct has no impact on schema of the document - it should not be ever used in the schema definition.

   (d) Reinterpretation
   Changes the semantics of an element and does not change the structure of the document. This change has no impact on the document, only for comprehension of the reinterpreted part.

   (e) Redefinition
   Updates the schema with no impact on the document instances format.

2. **Complex changes**

   (a) Element composition
   Groups elements under a new element.

   (b) Element decomposition
   Divides a group of elements into individual elements (opposite of element composition).

   (c) Renaming
   Changes the name of an element or an attribute.

   (d) Optionality
   Changes the optionality type of an attribute.

(e) Renumbering
Changes the cardinality of an element.

(f) Retyping
Changes the type of an element or an attribute.

(g) Namespaces
Changes the namespace.

(h) Default values
Changes the default value of an element or an attribute.

(i) Reordering
Changes order of sibling elements.

3.1.2 Impact on Queries

Changes of document schema have variable influence on the queries. In this paper possible impacts on queries are divided into the three groups of schema changes.

Definition 3.1. (Original schema). An original version of the schema S.

Definition 3.2. (New schema). A changed version of the original schema S.

1. General queries
   Are all queries which do not use any element affected by the evolution process. The queries work in both schema versions - original and new one in the same way.

2. Basic changes
   From the basic changes defined in Section 3.1.1, only *Removal* and *Refinement* have impact on queries which can cause incompatibility.

3. Complex changes
   From complex changes all possibilities can cause incompatibility of evaluation queries on new and original schemes.

   (a) Element composition
   There are two major problems. First, that the name of the newly created element can be different from the original element. Second, the query may refer to specific path in the document tree. So, although there are the data in the document, the query cannot evaluate them.

   (b) Element decomposition
   It is opposite to element composition.

   (c) Renaming
   The change of an element name causes that the result of the query is different or returns nothing.

(d) Optionality
A change from *required* to *optional* is not a problem, but a change in the opposite direction cause that the original query can return nothing in new schema version.

(e) Renumbering
Changing cardinality can cause problems in both directions. A change from singleton to multiple elements evokes that there can be more results. Changes from multiple elements to a singleton only reduces the result.

(f) Retyping
This case is described as one of the hardest changes. It needs checking of the value type and, if needed, the value must be cast to the needed type.

(g) Default value
This change may cause problems if there are any constraints. Different values can be returned in the new schema.

(h) Namespaces
This change can be a problem if the query uses a specific namespace. A different namespace can give different results.

(i) Reordering
Usage of positions in queries can cause that wrong data will be returned if the position of elements has been changed.

3.1.3 Compatibility of Queries across Schema Versions

The last section of the paper gives advices or patterns how to write queries to prevent the changes in queries while schema evolves and ensure that queries will return correct results.

1. Required elements (or attributes) should not be set into the middle of the sequence of elements. For example all required elements in the sequence should be set at the beginning, then optional. If necessary, queries should have *ancestor*//*descendant* axis.

2. Do not delete required elements from the middle of the sequence of elements. For example to delete the middle element of three sibling elements. A query can contains *exists*() function, which can cause problem in new schema if position predicate is used in the query.

3. If queries depend on the order of elements, do not change it if not necessary.

4. If queries are strongly typed, do not change the atomic type of the used values.

5. Do not change the name of elements used in queries. If it is necessary, use dictionary of synonyms to map them.

6. If queries are sensitive to namespaces which are changed, query returns nothing on new document which has different schema. If the behavior is to return the same results, the query should be written with $'*'$ for namespace.

7. Functions like *exist* should be used carefully. If the required element is removed, the query will always return $false$ in the new XML schema.

3.1.4 Discussion

The paper presents a set of possible changes which can be done through evolution of XML schemes. It classifies them into categories according to their complexity and shows an impact on related queries.
As a conclusion authors give advices how the queries should be written to ensure minimal additional changes in queries while XML schema is being changed.
But, it does not give an optimal solution how to provide XML schema evolution without inspection of related queries in all cases.

3.2 Identifying Query Incompatibilities with Evolving XML Schemes

Paper [14] discusses a system for monitoring the effect of schema evolution on the set of admissible documents and on the results of queries. An implementation of a framework for automatic verification of properties related to XML schema and query evolution is presented. As a query language the framework considers XPath.

The system is based on a set of predicates which allow for an analysis of a wide range of forward and backward compatibility issues. On the other hand, the system can produce counter examples to prove incompatibility of schemes and queries.
The framework was tested with realistic use cases on the real-world data.

Definition 3.3. (Forward Compatibility). Let S be an original schema and S' a new version of S. S' is called *forward-compatible* when all documents valid against S are also valid against S'.

Definition 3.4. (Backward Compatibility). Let S be an original schema and S' a new version of S. S is called *backward-compatible* when all documents valid against S' are also valid against S.

3.2.1 Internal Representation

As an internal representation *regular tree type expressions* [14] are used. This representation can capture and convert a schema expressed, e.g., in DTD, XML Schema and Relax NG [15]. The defined tree type expressions are shown in Table 3.2.2.

$\tau ::=$	tree type expression
$\emptyset$	empty set
$()$	empty sequence
$\tau \mid \tau$	disjunction
$\tau \ , \ \tau$	conjunction
$l(a)[\tau]$	element definition
x	variable
$let\ \overline{x = \tau}\ in\ \tau$	binder

Table 3.2.2: Tree type expressions

3.2.2 Logical Formulas

The core of the framework are *logical formulas*. They operate on binary trees with attributes. The framework translates all *unranked trees* [14], which represent XML documents, into *binary trees*.

The semantics of formulas corresponds to $\mu - calculus$ [16] interpreted over finite trees. All XPath expressions defined by the authors can be translated into these logical formulas. A translated XPath expression operates with a binary tree and uses only *forward axes*. Conversion to $\mu - calculus$ is done, because only with this modification the program can solve both XPath emptiness and other decisions problems such as *containment*.
For this purpose the framework implements a compiler which takes any XPath expression and makes its logical translation.

Definition 3.5. (Containment Problem). The *containment problem* takes as an input XPath expressions E and E', asking whether the output of E is contained in the output of E' on any source document at any node.

3.2.3 Query Representation

The framework is focused on the XPath language which is used in many cases and other standards like XQuery or XSLT. The used semantics of the XPath language is described in [2].

3.2.4 Analysis Predicates

Special predicates and a compiler for them are defined to solve decisions problems at a higher level of abstraction. Users can use the predicates to do basic verification like backward or forward compatibility. Within predefined predicates it is possible to create own custom predicates on the basis of defaults by combining them.

An example of predicate is $backward\_incompatible(\tau, \tau')$ which takes two type expressions as parameters and assumes τ' is an altered version of τ. This predicate is unsatisfiable if all instances of τ' are also valid against τ.

3.2.5 Framework Evaluation Process

The process of evaluation which is done by the presented framework is as follows:

1. A predicate is given to the framework. It contains all information for evaluating and returning a result.

2. The given predicate is parsed. The input schema is converted to regular tree expressions and the input XPath query is converted into a logical formula.

3. A satisfiability test which returns either of two possible results is carried out. If both schema versions are compatible, information about this fact is returned. Otherwise a message with a counter example of the incompatibility is returned.

3.2.6 Framework Real-World Use-case Tests

The paper also gives examples of real usage of the presented framework based on checking backward compatibility between XHTML 1.0 and XHTML 1.1 schema versions. An internal `backward_incompatible` predicate is used in the test. It takes DTDs of XHTML 1.0 and XHTML 1.1 as parameters. As a result it returns a counter example of an HTML document, which is permitted in XHTML 1.1 schema definition, but prohibited in 1.0. The sample result of the framework evaluation is depicted in Figure 3.2.1.

```
predicatebackward_incompatible("xhtml-basic10.dtd",
"xhtml-basic11.dtd","html")
returns

<html>
  <body>
    <head>
      <title/>
      <styletype="other"/>
    </head>
  <body/>
</html>
```

Figure 3.2.1: An example of a framework result

3.2.7 Discussion

The paper describes an implementation of a framework for verifying forward and backward compatibility between XML schemes and queries. The presented solution can be used by XML designers to recognize if the query needs to be updated due to schema evolution.
The tool expects a predicate which should be verified and both versions of the schemes. It returns a result of the predicate, or it can return a counter example, which can help designers with facilitating the queries.
The solution covers most of the frequently used schema languages such as DTD, XML Schema and Relax NG, whose definitions are converted into a common representation of regular-expression tree. The XPath language is translated into logical formulas which allow for validation for the given expression tree.

3.3 CoDEX

CoDEX (Conceptual Design and Evolution of XML schemes) [13] is an approach to schema evolution based on conceptual model. The paper presents the idea and describes its implementation - *CoDEX-tool* application.

The application provides a GUI for modeling XML schemes and their additional changes. There is a possibility to import an existing XML schema and generate a model from it. After the evolution process a new model can be exported to its XML schema representation. Documents which depend on the evolved schema are updated to preserve validity.
All changes which are done by an user with the model are stored - they are recorded and processed in the same time. The evolution process is done after user's confirmation.

3.3.1 Conceptual Model

The conceptual model of CoDEX is a *mixed graph* [17] which can contain four types of *basic components.* The basic components are *element*, *type*, *group* and *module.* Every component can have properties defined as a *key-value* pair.

XML Schema Import CoDEX allows a user to import an existing XML schema which is then translated into the internal model representation of the system.

XML Schema Export As was mentioned in the introduction, the model can be exported into XSD, which always follows the *Venetian blind design style* [18].

3.3.2 Schema Evolution

For each basic component operations *add*, *delete*, *change* and *move* are defined. For an element and a module component there is operation *rename*. All user's design changes are logged and all needed information for evolution process can be generated from the history.

Minimizing and Normalizing Changes After the evolution process is confirmed, the changes are *minimized* and *normalized*, because there can be done redundant operations while changing the schema. For example creating and then deleting the same element is omitted. Or, creating an element and changing its name is converted to only one step - creating an element with the new set name. An example of summarizing is as follows:

$$create\_element(id, name, content) + rename\_element(id, name, new\_name) \rightarrow create\_element(id, new\_name, content)$$

At the end all changes of the model are translated into XML schema evolution steps.

XML Document Update After the changes are done in the XML schema it is needed to revalidate XML documents associated to the schema. All invalid documents must be updated to restore the validity.

3.3.3 Discussion

CoDEX provides a tool for evolution of XML schemes and associated XML documents. There can be done complex changes in a single evolution process. The tool offers changing of existing XML schemes which are translated to their conceptual model too.

As was mentioned in the paper, there are some limits and open problems which are not solved. An example with moving element *address* from element *owner* to element *producer* which can be easily done is presented. Updated document will be valid, but not semantically correct, because the *address* of the *producer* is not the same as the address of the *owner*. A solution which is given by authors is user interaction.

3.4 Comparison of the Related Works

All presented works study a problem of the schema evolution from different points of view and propose solutions how to ensure the compatibility in the system of the schemes and the related documents or queries.

The first paper [4] analyzes possible operations which can be used for an update of the schema, classifies them according to complexity and gives advices to designers how to provide changes to prevent and minimize changes in the related queries.

The second paper [14] presents a complex framework for identifying incompatibilities of the queries and the evolved schemes. They use own internal structures for schemes and predicates for queries. Thanks this solution they can cover the most frequently used query languages such as DTD, XML Schema and Relax NG. But no suggestion is given how to update related queries automatically.

The last analyzed paper [13] presents a tool for providing evolution of the schemes and for updating related XML documents by using these schemes. It offers a complex framework which was tested on the real-word data as well as the second paper.

All mentioned works present various solutions how to preserve compatibility during the schema evolution. In this thesis we will focus on the following problems which were not solved in these papers:

- On the analysis of changes done in a schema.
- On the relation between a schema and a query.
- On the propagation of changes to preserve compatibility of a schema and a query.

Chapter 4

XSEM PSM

In this chapter we describe an approach for conceptual modeling of XML data, whose second layer - XSEM-H model will be used in this thesis as a model for representing XML Schema and which will be described more in detail.

4.1 XSEM

XSEM [7] is an approach for conceptual modeling of XML data. It divides the modeling process into two interconnected layers which use different model types.

First model called *XSEM-ER* is a PIM layer. It is an extension of classical ER model and describes a structure of modeled data.

Second, PSM layer is called *XSEM-H* and it is a conceptual model for modeling hierarchical structure of data modeled by XSEM-ER. XSEM-H schema describes a hierarchical structure of a part (or whole) of the XSEM-ER and this description can be done in different ways for the same XSEM-ER model. Thanks to this layer it is possible to separate and solve different parts of the system by individual components. For example, from one XSEM-ER data payment model we can model schema for producer and another for consumer which look absolutely different, but they are based on the same PIM schema.

Thanks interconnection between models it is possible to propagate changes done in PIM diagram to corresponding components in PSM schema or in opposite direction [19].

An example of XSEM-ER (PIM) and corresponding XSEM-H (PSM) diagram is shown in Figure 4.1.1 and Figure 4.1.2 respectively.

4.2 XSEM PSM

XSEM-H model is an extension of UML Class Diagram [20]. It takes basic components of the language like *UML class*, *UML attribute*, *UML association* and adds extension of own components and profile named *XSEM* [7]. Thanks to the fact

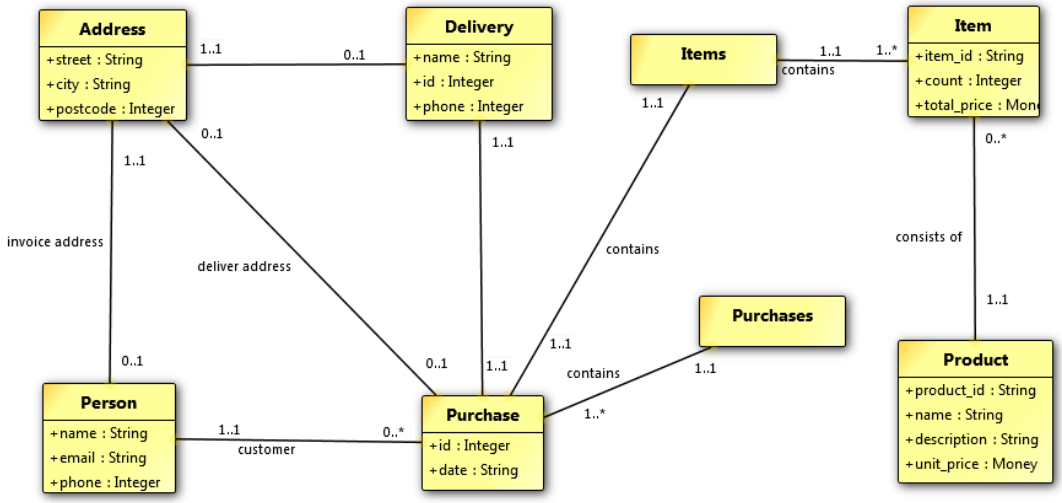


Figure 4.1.1: XSEM-ER (PIM) example

that XSEM PSM is a visual notation of XML document structure, it is equivalent to XML Schema. It is possible to generate XML Schema from existing PSM diagram or create PSM diagram from given XML Schema document too. Both these features were implemented in the XCase tool [19].

A visualization of XSEM-H model was implemented in the DaemonX framework [21] as a modeling plug-in of PSM XML schema. The implementation enables possibility of creating and updating diagrams of this model thanks various operations defined in [22]. The model contains following constructs:

PSM Class is the most important construct of the model. It represents a class from platform independent model and expresses how it is modeled in XML schema. Each PSM class can represent only one class from PIM model. Every class has a name, a label and PSM attributes. Class can be connected with other classes by PSM associations.

PSM Attribute is a construct which can exist only as a part of a PSM class. Each attribute represents zero or one PIM attribute.

PSM Association is a connection between two PSM classes in relation parent–child. Unlike PSM class and PSM attribute it can represent a whole set of PIM associations. There can be only one PSM association leading to class and zero or more associations leading from one class. Child classes are called *content* of the class.

Content Choice is a construct which represents *choice element* of XSD. From the set of child PSM classes only one can be used in an XML document defined

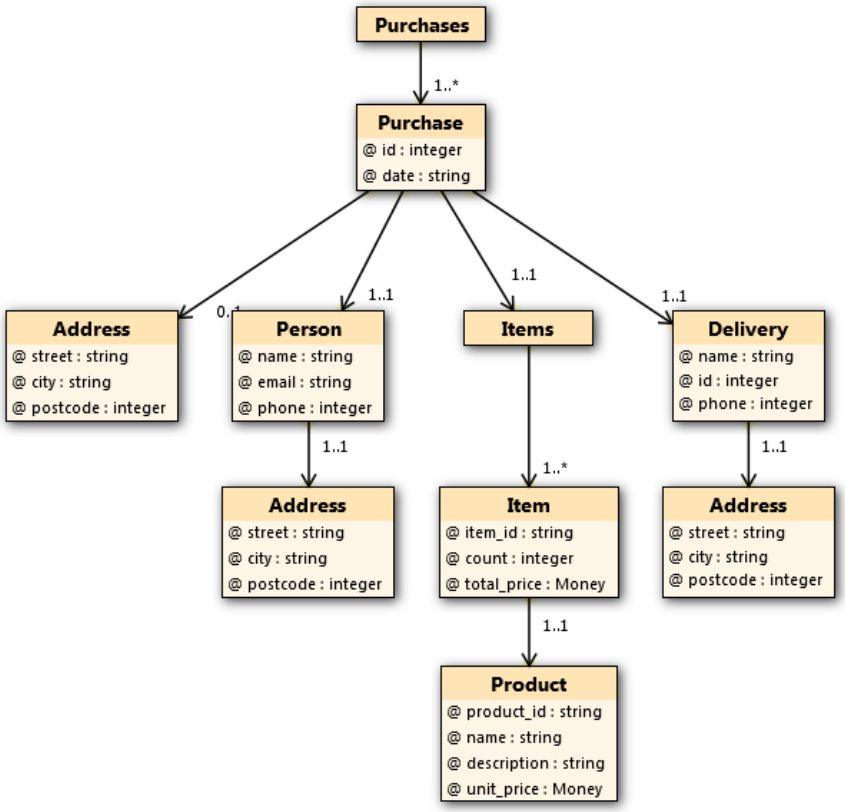


Figure 4.1.2: XSEM-H (PSM) example

by this schema.

Content Sequence is construct represents *sequence element* of XSD. All child PSM classes of this construct must be used in the respective XML document in defined order.

Content Set is a construct which represents *all element* of XSD. All child PSM classes of this construct must be used in XML document in arbitrary order.

An example of a PSM diagram with all constructs which represents a simple purchase schema is shown in Figure 4.2.1.

Definition 4.1. (PSM Schema). A *PSM schema* is a 16-tuple $S = (S_c, S_a, S_r, S_e, S_m, C_S, name, type, class, xform, participant, card, cmtype, attributes, content, repr)$, where the items are:

- S_C
  The set of all PSM classes.

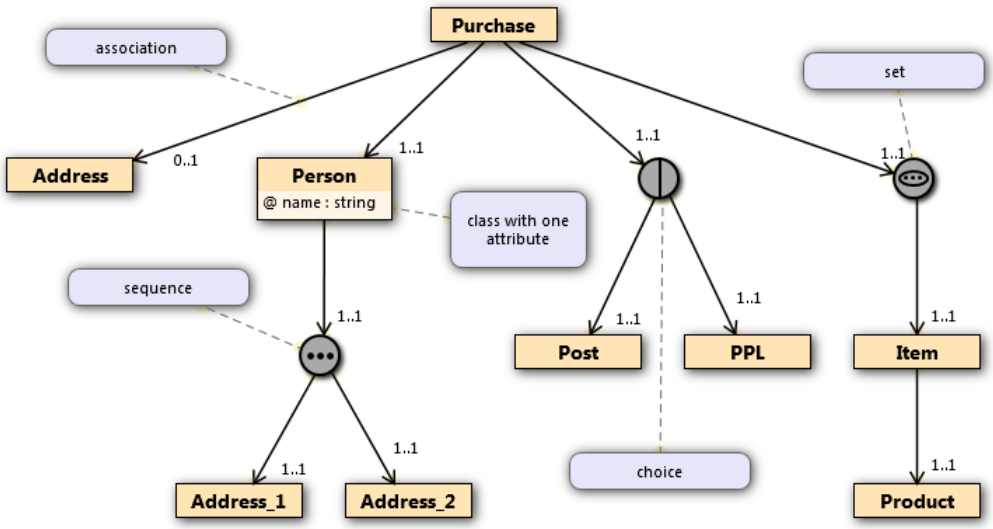


Figure 4.2.1: An example of a PSM diagram

- S_a
  The set of all PSM attributes.

- S_m
  The set of all content models.

- S_r
  The set of directed binary associations in schema (PSM association).

- S_e
  The set of association ends in schema. A directed binary association is a pair $R = (E_1, E_2)$, where E_1, $E_1 \in S_e$ and $E_1 \neq E_2$. For any two associations R_1, $R_2 \in S_r$ it must hold that $R_1 \cap R_2 \neq \emptyset \Rightarrow R_1 = R_2$.

- C_s
  Is a schema class of the schema.

- name
  Assigns a name to each class, attribute and association.

- type
  Assigns a data type to each attribute.

- class
  Assigns a class to each attribute. For $A \in S_a$ we will say that A is an attribute of class C.

- participant
  Assigns a class or content model to each association end.

- xform
  Assigns an XML form to each attribute. Is specifies the XML representation of an attribute using an XML element declaration with a simple content or an XML attribute declaration.

- card
  Assigns a cardinality to each attribute and association end.

- cmtype
  Assigns a content model type to each content model. We distinguish 3 types: sequence, choice and set, respectively.

- attributes
  Assigns an ordered sequence of distinct attributes to each class C.

- content
  Assigns an ordered sequence of distinct associations to each class or content model X.

- repr
  Assigns a class $\overline{C}$ to another class C. $\overline{C}$ is called structural representative of C.

Chapter 5

Mapping XPath to XML Schema

For possibility of evolution of XPath queries related to XML schemes there must exist a mapping between XML schema and XPath query. This mapping helps to conduct the evolution process to evolve the query in relation to the evolved schema.

We will present used XPath syntax, its visualization model and mapping between XSEM PSM and XPath models.

5.1 XPath Syntax

Full XPath syntax is too extensive to be used in this paper, so only a subset was used. There were defined various subsets of XPath syntax, such as Core XPath [23] or Positive Core XPath [24]. Some definitions from these approaches were adopted and updated in this chapter. In particular, we used our subset which is based on Positive Core XPath with some changes. There are no predicates and operator *except* is added to the definition. The abstract syntax is follows:

$$\mathbb{X} \equiv \mathbb{X}|\mathbb{X} \parallel /\mathbb{X} \parallel \mathbb{X}/\mathbb{X} \parallel \mathbb{X}\ except\ \mathbb{X} \parallel \mathbb{A} :: \mathbb{L}$$
$$\mathbb{A} \equiv self \parallel child \parallel descendant \parallel descendant-or-self \parallel parent$$
$$\parallel ancestor \parallel ancestor-or-self \parallel preceding$$
$$\parallel preceding-sibling \parallel following \parallel following-sibling$$

As we can see, the only one node test is possible - name test, denoted $\mathbb{L}$. $\mathbb{X}$ denotes location path and $\mathbb{A}$ represents an axis.

5.1.1 Predicates

Original Positive Core XPath definition contains predicates [24] where can be used only tests for node presence on the path in the schema. But a query using these predicates can be rewritten to the query without them [25] in multiple possible ways returning the same result node set. This solution has only one problem - the

query is transformed to a complex form not transparent for the designer at first sight.

5.1.2 Abbreviations

In the defined syntax it is possible to used all defined abbreviations for axes:

- $'*'$ selects all element children of the context node
- $'para/title'$ is equal to $'para/child :: title'$
- $'.'$ selects the context node
- $'..'$ selects the parent of the context node
- $'//'$ corresponds to $'/descendant - or - self :: node()/'$

5.2 XPath Model Visualization

To be able to map XSEM PSM diagram to XPath query, an XPath model must be defined. We proposed a model that follows ordered tree structure of the XPath query, results from the presented syntax and that visualizes its textual representation. Basically components of the model can be divided to two parts:

- Elements which represent nodes in the location path
- Lines that represent axes

A line and an element together comprise a location step of the XPath query. Visualization of elements and axes is shown in Figure 5.2.1 and in Figure 5.2.2 respectively. The defined model contains the following components:

Figure 5.2.1: XPath model visualization elements

Element An element E represents a node test in the location step. A name test is defined by the name of the element. Every element can have only one input and one output line which represent axes.

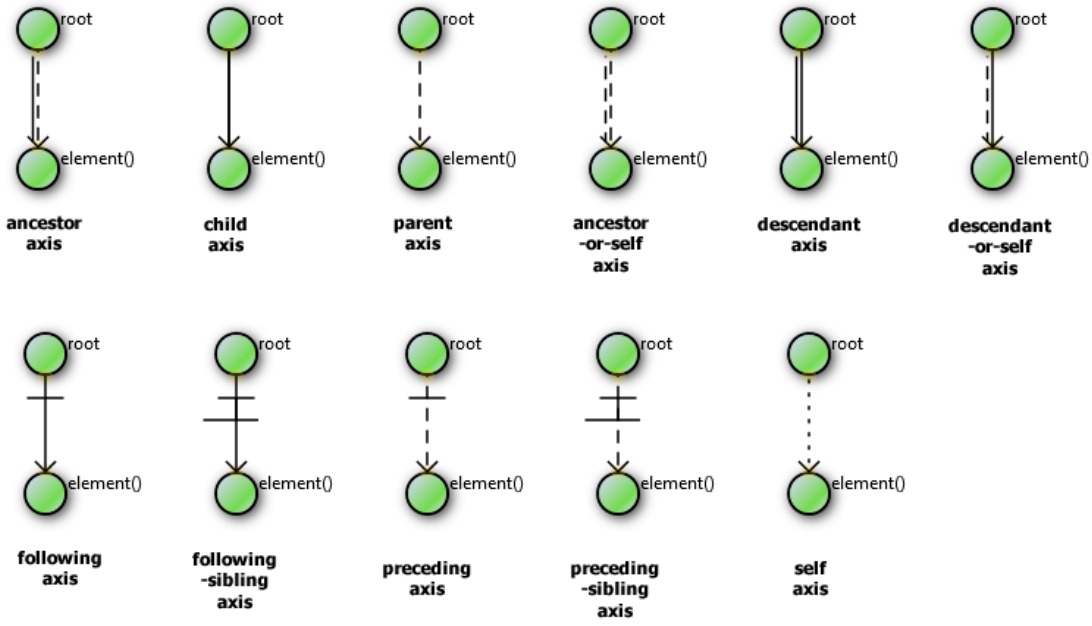


Figure 5.2.2: XPath model visualization axes

Expression Element A special type of element (denoted E_{ex}) that represents *disjunction* and *except* operators. Every expression element can have only one input line and two or three output lines. An input line connected to this element expresses only connection, it has no special sense. The first output line represents first part of the query expression, the second output line represents the second part of the expression. The third line represents following expression part of the query in the sense of $(first\_expression\ operator\ second\_expression)/$ $third\_expression$. An example of a visualization of the query $/purchase/$ $(./descendant :: element()\ except\ ./child :: address)$ is presented in Figure 5.2.3.

Child Axis Line A line with meaning of child axis will be denoted L_{ch}.

Descendant Axis Line A line with meaning of descendant axis will be denoted L_d.

Descendant-or-self Axis Line A line with meaning of descendant-or-self axis will be denoted L_{dos}.

Parent Axis Line A line with meaning of parent axis will be denoted L_{pa}.

Ancestor Axis Line A line with meaning of ancestor axis. will be denoted L_a.

Ancestor-or-self Axis Line A line with meaning of ancestor-or-self axis will be denoted L_{aos}.

Following Axis Line A line with meaning of following axis will be denoted L_f.

Following-sibling Axis Line A line with meaning of following-sibling axis will be denoted L_{fs}.

Preceding Axis Line A line with meaning of preceding axis will be denoted L_{pr}.

Preceding-sibling Axis Line A line with meaning of preceding-sibling axis will be denoted L_{prs}.

Self Axis Line A line with meaning of self axis will be denoted L_s.

Figure 5.2.3: XPath query visualization example

5.2.1 Query of the Model

We have said how to represent an XPath query in a visualization model. But there must be also a way how to get the query of the model. The XPath model is an ordered directed tree. The root of the model (denoted *root*) represents start of the query. The tree model is ordered from left to right and the query can be generated by using a depth-first search algorithm run from the root element. An example is depicted in Figure 5.2.3.

5.3 Location Path Mapping

Since the XSEM PSM schema has a tree structure (especially it is equivalent to XSEM [7]) and the XPath query follows a tree structure, it is straight forward to map XSEM PSM to location path. An example of mapping is shown in Figure 5.3.1. But, if there are any complex axes in the query, it is not possible to map the query in a simple way because an axis can intervene not only one node in the schema tree, but a part of tree. For example *descendant* axis hits whole sub-tree of children of the current node. And all these nodes should be mapped directly to the node in location step, see Figure 5.3.2.

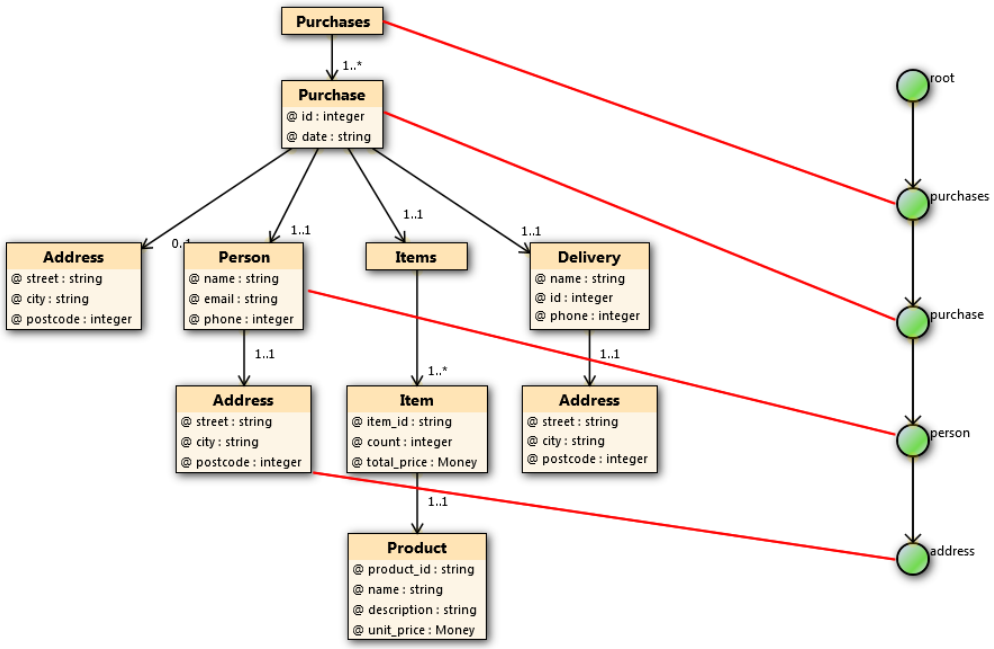


Figure 5.3.1: Simple result mapping

To create a mapping between XSEM PSM and a location path it is needed to define a representation of location step. As was mentioned in Section 2.2.3, a location step consists of three basic parts which can be represented as a graph. These parts correspond with defined XPath visualization model. An axis is represented by one XPath model line L, a node test is represented by XPath element E. A predicate is not considered in defined syntax.

Definition 5.1. (Mapping). A mapping M between XSEM PSM model and XPath model is a pair of XSEM PSM class C and XPath element E. Formally $M = (C, E)$.

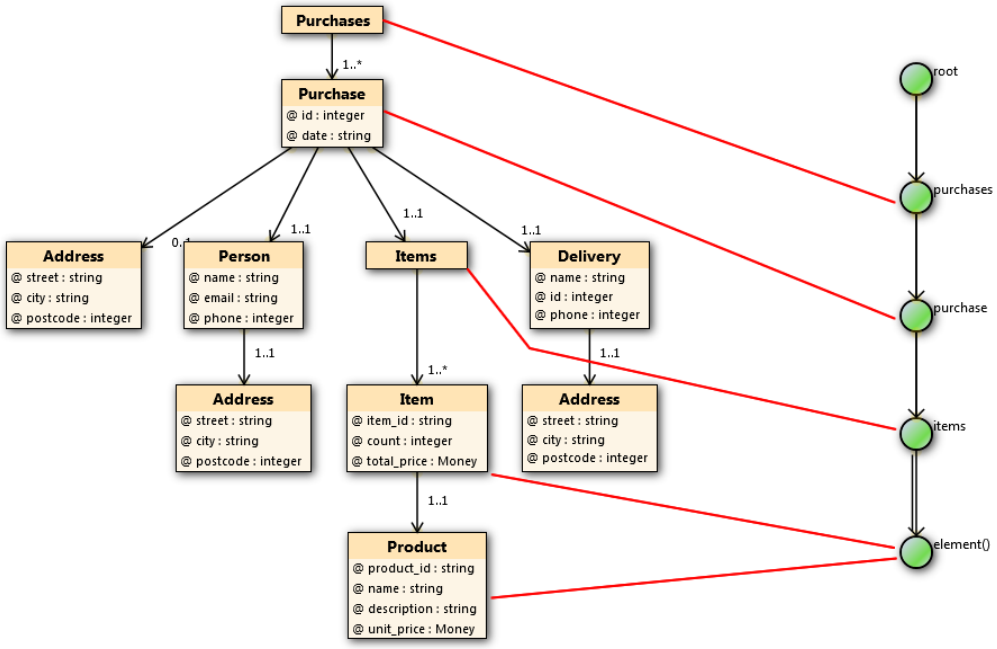


Figure 5.3.2: Multiple result mapping

5.3.1 Mapping with Simple Result

The simplest type of mapping is to map one PSM class C to one XPath element E, formally: $\exists!C, \exists!E, \exists!M : M = (C,\ E)$. This is done in case when location step returns only a single result. For example in case of *parent* axis.
Suppose that for XSEM PSM model we created query $/Purchases/Purchase/Person/Address$ which is an abbreviation for $/child :: Purchases/child :: Purchase/child :: Person/child :: Address$. Mappings created for the query (displayed as red lines) are shown in Figure 5.3.1.

5.3.2 Mapping with Multiple Results

Mapping with multiple results means that one XPath element E is mapped to more XSEM classes C_i, formally: $E,\ C_1, ..., C_n,\ \exists n,\ n > 1,\ \forall i \in \{1,\ ...,\ n\} : M_i = (C_i,\ E)$. This can be done for example by the *descendant* axis, which returns all descendants of an XSEM class.

Now suppose query $/Purchases/Purchase/Items/descendant :: *$. In full notation $/child :: Purchases/child :: Purchase/child :: Items/descendant :: *$. In this case there are multiple mappings from XPath element named *element*() to class *Item* and *Product* thanks descendant axis (denoted as double line arrow), which means all descendants of class *Items*. An example of the mapping is shown in Figure 5.3.2.

5.4 Operations

All presented initial operations which change the source XSEM PSM schema are atomic. This forms a subset of operations defined in [22].

Definition 5.2. (Operation). An *operation* is a function. As input it takes a system of source XSEM PSM, XPath diagrams and additional information about its purpose. Output of the function is a modified system. It is possible that the changed system can be left in inconsistent state and has to be updated manually by designer (e.g. if an XSEM class returned by the XPath model as a result was removed).

Definition 5.3. (Atomic operation). An *atomic operation* is a minimal operation which cannot be divided into smaller operations. It can be used for creating composite operations.

5.5 Atomic Operations

In this section we describe atomic operations of XSEM PSM and XPath models which are used for evolution process.

5.5.1 XSEM PSM Model Operations

All described operations can influence the result of the XPath query model.

Adding PSM Root Class $\alpha(C)$ – Adds a new PSM class C as the root into the XSEM PSM diagram.

Adding PSM Class $\alpha(C,\ C_p)$ – Adds a new PSM class C into XSEM PSM diagram as a child of parent class C_p. This operation creates an association A between these classes.

Removing PSM Association $\rho(A)$ – Removes PSM association from XSEM PSM diagram.

Removing PSM Class $\rho(C)$ – Removes PSM class from XSEM PSM diagram. This operation leads to inconsistent state of the system. When a PSM class C is deleted, all associations connected to the removed class have to be deleted too. Formally: $\forall A, A \in Assocs(C) : \rho(A)$.

Rename PSM Class $\delta(C,\ name)$ – Sets new name to PSM class C.

Change Position of PSM Class $\psi(C,\ direction)$ – Moves PSM class C to the left or to the right in the sequence of its sibling.

Reconnect PSM Class $\mu(C, C_p)$ – Reconnects PSM class C as child of parent PSM class C_p.

Definition 5.4. (Assocs(XSEM class C)). A function which returns all XSEM associations A conntected to the XSEM class C.

5.5.2 XPath Model Operations

Adding XPath Root Element $\alpha(E_{root})$ – Adds a new root element E_{root} into XPath diagram.

Adding XPath Element $\alpha(E,\ E_p)$ – Adds a new element E into XPath diagram as a child of element E_p.

Adding XPath Line $\alpha(L,\ E_p,\ E_{ch})$ – Adds a new axis line L between two elements, child element E_p and parent element E_{ch}.

Removing XPath Line $\delta(L)$ – Removes axis line L from the diagram.

Removing XPath Element $\delta(E)$ – Removes element E and all related axis lines, formally: $\forall L, L \in Lines(E) :: \rho(L)$.

Rename XPath Element $\delta(E,\ name)$ – Sets new name to element E.

Definition 5.5. (Lines(XPath element E)). A function which returns all XPath axis lines L connected to the element E.

5.6 Recognizing of Changes and Propagation

When XSEM PSM diagram and XPath query are mapped correctly, it is possible to propagate operations (changes) done in XSEM PSM diagram to related XPath diagram. We have defined operations which can be done in XSEM PSM and update operations which have to be done in XPath diagram if changes in results were detected to ensure that after propagation will hold $R \equiv R'$ if it is possible. These operations are described in Chapter 6.

Definition 5.6. (Original query). An XPath query applied to original XSEM PSM schema. It will be denoted Q. The result of the query is denoted R.

Definition 5.7. (New query). An updated XPath query applied to changed XSEM PSM schema. It will be denoted Q'. The result of the query is denoted R'.

5.6.1 Recognizing of Changes

Propagation of changes in XSEM PSM is not necessary in all cases. It has to be done only if the mapping between models is changed – the mapping can be added or removed (it has the same meaning as different results of Q and Q'). The algorithm which checks if the mappings have changed goes from the beginning of the query gradually by the location steps and checks original and new results. If a change is recognized, appropriate operations are applied on XPath model by method *Update*. This is done by Algorithm 5.1.

The Algorithm 5.2 compares both original and new result sets returned by the query (location step). If there are some new or missing XSEM classes in the mapping, they are set to appropriate collections of missing and added classes and $false$ is returned. If the results are the same, *true* is returned. If there are any missing or added XSEM classes returned by Algorithm 5.2, mappings between these classes and XPath element must be created or removed respectively by methods $CreateMappings$ and $DeleteMappingsOfXSEMClassesAndXPathElement$.

Algorithm 5.1 RecognizeChanges

Input: original XSEM PSM model $\mathcal{OM}$, new XSEM PSM model $\mathcal{NM}$, XPath query model $\mathcal{QM}$, XSEM PSM operation $\mathcal{O}$

1: $sub\_step \leftarrow$ ""
2: **for** each $location\_step \in \mathcal{QM}$ **do**
3: $sub\_step \leftarrow sub\_step + location\_step$
4: $result' \leftarrow Evaluate(\mathcal{NM}, sub\_step)$
5: $result \leftarrow Evaluate(\mathcal{OM}, sub\_step)$
6: **if** not $CompareResults(result, result', added, missing)$ {see Algorithm 5.2} **then**
7: $DeleteMappingsOfXSEMClassesAndXPathElement(missing, location\_step.targetElement)$ {see Algorithm 5.5}
8: $CreateMappings(added, location\_step.targetElement)$ {see Algorithm 5.4}
9: $Update(location\_step, \mathcal{O}, added, missing)$ {Update query to preserve the same results}
10: **end if**
11: **end for**

Algorithm 5.2 CompareResults

Input: collection of results from the original schema $\mathcal{OC}$, collection of results from the new schema $\mathcal{NC}$, collection for the added classes in the new schema **var** *added*, collection for missing classes in the new schema **var** *missing*

Output: *true* if results are same, otherwise $false$

```
1: for each o ∈ OC do
2:    found ← false
3:    for each n ∈ NC do
4:       if o = n then
5:          found ← true
6:          break
7:       end if
8:    end for
9:    if found = false then
10:      missing.Add(o)
11:   end if
12: end for
13: for each n ∈ NC do
14:   found ← false
15:   for each o ∈ OC do
16:      if n = o then
17:         found ← true
18:         break
19:      end if
20:   end for
21:   if found = false then
22:      added.Add(n)
23:   end if
24: end for
25: if added.Count > 0 or missing.Count > 0 then
26:   return false
27: else
28:   return true
29: end if
```

Algorithm 5.3 Evaluate

Input: XSEM PSM model $\mathcal{M}$, XPath query $\mathcal{Q}$

Output: set of classes hit by the query $\mathcal{Q}$ in $\mathcal{M}$

```
1: return SaxonXPathParser.Parse(M.ToXSD(),Q) {A method which returns
   result of the query applied on the schema generated from XSEM PSM model.}
```

Algorithm 5.4 CreateMappings

Input: collection of XSEM PSM classes *classes*, XPath element e

1: **for** each $c \in classes$ **do**
2: $CreateMapping(C, e)$ {A method which creates mapping between given XSEM PSM class and XPath element.}
3: **end for**

Algorithm 5.5 DeleteMappingsOfXSEMClassesAndXPathElement

Input: collection of XSEM PSM clasess *classes*, XPath element e

1: **for** each $c \in classes$ **do**
2: $DeleteMappingBetweenXSEMClassAndXPathElement(c, e)$ {A method which removes mapping between given XSEM class and XPath element.}
3: **end for**

Definition 5.8. (hit). An XSEM PSM class is *hit* by the XPath element, if it is returned in the result of the location step where the XPath element is used.

A method *Update* in Algorithm 5.1 determines the corresponding method from Chapter 6 by the set location step and applied XSEM PSM operation.

Chapter 6

Evolution Algorithms

This chapter contains an analysis of possible operations (changes) in XSEM PSM schema, their impact on the query and a revalidation of the XPath model if it is possible to preserve that both original and new queries return the same results. Every operation done and analyzed in XSEM PSM is atomic and described in Section 5.5.1. The algorithm of the propagation to XPath model does not have to contain only a single atomic operations, but there can be a set of atomic operations to ensure compatibility.

6.1 Correctness of the Propagation

This section discusses the correctness of the propagation of changes to the target XPath model.

Every change in the source XSEM PSM can cause that there must be changes in multiple parts of the XPath model (in multiple location steps). Furthermore there can be done multiple changes in XSEM PSM model step by step.

Let AO_{XSEM} be an atomic operation done in an XSEM PSM schema which should be propagated. Next, let OS_{XPath} be a sequence of atomic operations in an XPath model which was generated from AO_{XSEM} to preserve the same results of the queries with the original and the new XSEM PSM schema. Formally if Q be the original query of the original schema S, Q' be the query used with the new schema S' (at the beginning $Q = Q'$), $R = Q(S)$ be the result of the original query and $R' = Q'(S')$ be the result of the query applied on the new schema, then holds $R = R' = Q(S) = Q'(S') = OS_{XPath}(Q)(AO_{XSEM}(S))$ if there exists an appropriate propagation algorithm which generates OS_{XPath}.

If the results are the same, the next update of the XSEM PSM schema can be done in the same way: If there exists valid operation sequence OS'_{XPath} for atomic operation AO'_{XSEM} which satisfy $R' = R''$ then holds
$R'' = OS'_{XPath}(OS_{XPath}(Q))(AO'_{XSEM}(AO_{XSEM}(S)))$.

6.2 Analysis of the Changes

In this section we analyze changes done in XSEM PSM schema and their impact on the related XPath model.

Let x be the newly added element, S the original schema and S' new changed schema. If not specified otherwise, all elements are in *collection* element, *choice* and *all* are special cases. In all cases we consider only one location step of the query Q. All described algorithms are used by method *Update* in Algorithm 5.1, line 10.

In all presented situations we suppose that there exist no two sibling elements of the same name in the schema. If this conditions is violated, an update is not possible and user must be asked to repair the query manually.

Every considered operation has preconditions and postconditions of the original schema S and the new schema S'. Described revalidation of the query and algorithms to update an XPath model satisfy that for original result R and new result R' it holds $R = R'$ or it returns that it is not possible.

In preconditions and postconditions of the schema before and after the change, we will use functions that for given element return an element or a set of elements:

- A function *parent(element)* returns a parent element of the given element of the schema in the same way as the XPath parent axis.
- A function *descendant(element)* returns descendant elements of the given element of the schema in the same way as the XPath descendant axis.
- A function *following(element)* returns following elements of appropriate element in the same way as the XPath following axis.
- A function *following-sibling(element)* returns a collection of elements of an appropriate element in the same way as the XPath following-sibling axis.
- A function *preceding(element)* returns a collection of elements of an appropriate element in the same way as the XPath preceding axis.
- A function *preceding-sibling(element)* returns a collection of following elements of an appropriate element in the same way as the XPath preceding-sibling axis.
- A function *pos(element)* returns a position of the element in the collection of the sibling elements of the given element.
- A function *absolute_path_to_element(element)* returns an absolute path from the root to the given element containing only location steps with child axes and a node test for name of the added element in the schema tree. An example of the result of the function applied on the element *color* in the schema in Figure 6.2.1 is a path containing elements *vehicle*, *car* and *color*.

- A function *absolute_path_to_added_element(element)* returns an absolute path from the root to given added element in the save way as the function *absolute_path_to_element*.
- A function *absolute_path_to_next_sibling(element)* returns an absolute path from the root to the next sibling element of the given element in the schema. An example is shown in Figure 6.2.1 on the left. In the blue rectangle there is the source element, in the red rectangle there is the element returned by this function.

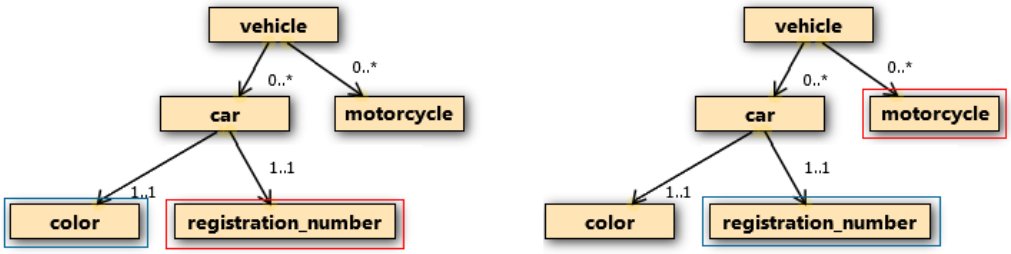


Figure 6.2.1: Next element right

- A function *absolute_path_to_next_element_right(element)* returns an absolute path from the root to the next right element of the given element in the schema. An example is shown in Figure 6.2.1 on the right. In the blue rectangle there is the source element, in the red rectangle there is the element returned by this function.

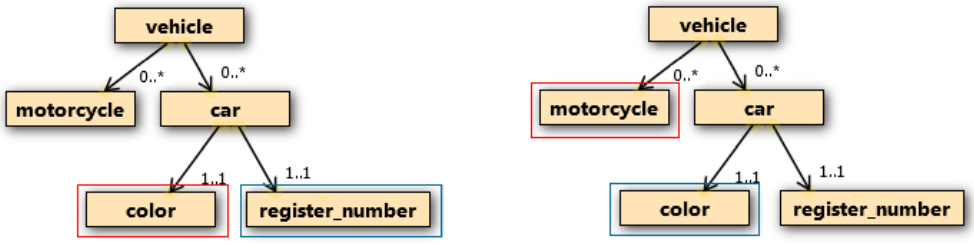


Figure 6.2.2: Previous element left

- A function *absolute_path_to_previous_sibling(element)* returns an absolute path from the root to the previous sibling element of the given element in the schema. An example is shown in Figure 6.2.2 on the left. In the blue rectangle there is the source element, in the red rectangle there is the element returned by this function.

- A function *absolute_path_to_previous_element_left(element)* returns an absolute path from the root to the previous element of the given element in the schema. An example is shown in Figure 6.2.2 on the right. In the blue rectangle there is the source element, in the red rectangle there is the element returned by this function.

- A function absolute_path_to_reconnected_element(element) returns an absolute path from the root to reconnected element in the same way as the function *absolute_path_to_element*.

6.2.1 Refinement

First, suppose that all described insertions add a new element as a child of a *sequence* element. If the type of element is *choice* or the *minOccurs* attribute of the added element is set to 0, no change should be done in the query [4].

Ancestor Axis Suppose we want to get ancestor axis of element p and new element x is added as a child somewhere to the schema tree.

Preconditions: $p \in S \wedge x \notin S$

Postconditions: $x \in S'$

Query revalidation

- The newly added element x cannot be on the path from element p to the root element of the schema tree. It is not possible to set the added element x as a parent. In case when there exists a location step, which will use the element in the way $x/parent :: *$, an occurrence of the x will be discovered by another axis.

Ancestor-or-self Axis This is the same case as the ancestor axis. It is not possible to add new element x to the path from p to the root because x can be set only as a child.

Preconditions: $p \in S \wedge x \notin S$

Postconditions: $x \in S'$

Query revalidation

- Adding a new element does not change the result of the location step, $R = R'$ and no update is needed.

Child Axis Adding a new element x as a child of element p can affect the query result and $R! = R'$. To ensure that the results will be the same with both schemes the query must be updated.

Preconditions: $x \notin S \wedge p \in S$

Postconditions: $x \in S' \wedge p = parent(x)$

Query revalidation
Suppose the query is $p/child :: *$ which returns all child elements of element p.

- New query will be $p/child :: *$ *except* $absolute\_path\_to\_added\_element(x)$.
- If the added element is set to *choice* as a child, no change has to be done in the query because we have no information about the documents valid against the schema.

If the query uses a name test of the child element – $p/child :: name$, no revalidation is needed, because it is not possible to add a child element with the same name as its siblings.

Example 6.1. Let the original schema S be the one in Figure 6.2.3 on the left and original XPath model representing query $/vehicle/child :: */registration\_number$ in Figure 6.2.4 on the left. If new element *motorcycle* is added as a child of element *vehicle* (see Figure 6.2.3 on the right), sub-query $/vehicle/child :: *$ will return all elements including *motorcycle* which is different from the result R. Hence, location step must be updated from $child :: *$ to $(child :: *$ *except* $/vehicle/motorcycle)$ and the query Q' will be $/vehicle/(child :: *$ *except* $/vehicle/motorcycle)/registration\_number$. The model of the new query is shown in Figure 6.2.4 on the right.

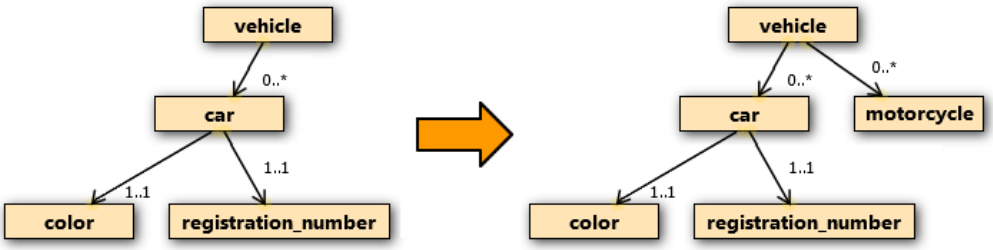


Figure 6.2.3: Schema example

The update algorithm of child axis is provided by the Algorithm 6.2. The method creates a new expression element E_{ex}, reconnects original location step as

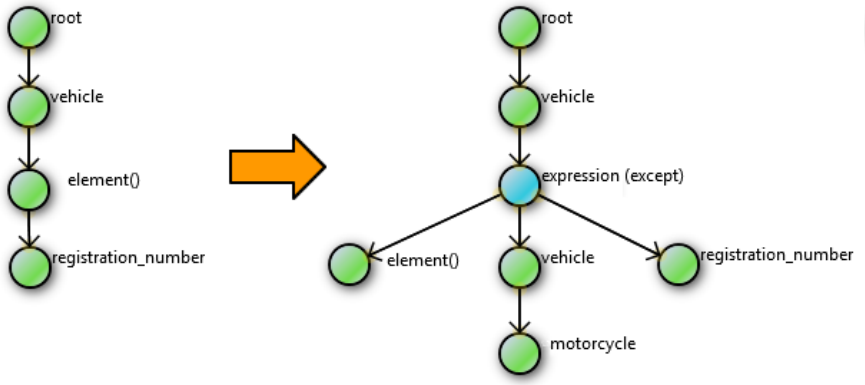


Figure 6.2.4: Query example

the first child and adds new created location step (see Algorithm 6.4) as the second child. If the attached location step is not the last in the query, the following location step is reconnected as the third child of E_{ex}.

A location step ls used in algorithms is a structure which consists of the XPath axis line L and the XPath element E (or expression element E_{ex}). It has references to parent element accessible by property *parentElement* and property to get element of the location step *targetElement*. The element E has properties of its connected axes *parentAxis* and *childAxis* and a property *name*. An expression element E_{ex} has furthermore properties of all possible child axes: $firstAxis$, $secondAxis$ and $resultAxis$.

Algorithm 6.1 GetPathFromRootToClass

Input: XSEM PSM class C^x
Output: collection of XSEM classes from root class to class C^x

1: $path \leftarrow$ **new** collection of XSEM PSM classes {A collection of classes from class C^x to the root class}
2: **while** $C^x != null$ **do**
3: $path.AddFront(C^x)$ {Add class in the front of the collection}
4: $C^x \leftarrow C^x.parent$ {Set parent XSEM class of C^x to C^x}
5: **end while**
6: **return** *path*

The method in Algorithm 6.1 returns a collection of XSEM classes from the root class to the class given as a parameter.

Algorithm 6.2 UpdateWhenClassAddedIntoChildAxis

Input: location step ls, XSEM PSM class C

1: $path \leftarrow GetPathFromRootToElement(C)$ {Algorithm 6.1}
2: $exp\_element \leftarrow$ **new** $XPathExpressionElement(\text{"}except\text{"})$ {Create new expression element}
3: $ReconnectElements(ls, exp\_element)$ {see Algorithm 6.3}
4: $exp\_element.secondAxis \leftarrow$ **new** $XPathAxis(exp\_element, CreateXPathModelPathFromXSEMPath(path, path\_end))$
5: $exp\_element.resultAxis \leftarrow ls.targetElement.childAxis$

Algorithm 6.3 ReconnectElements

Input: location step ls, expression element $exp\_element$

1: $parent \leftarrow ls.parentElement$ {$parentElement$ is an XPath element when the ls starts}
2: $ls.parentElement \leftarrow exp\_element$ {Change $parentElemetn$ of the ls}
3: $exp\_element.firstAxis \leftarrow ls$ {Set the first axis of the expression element}
4: **new** $XPathAxis(parent, exp\_element)$ {Create axis between set elements}

The algorithm 6.3 provides reconnection of the location step as a child of the newly created XPath expression element. The location step is reconnected as the first child of the expression element from the left and the created expression element E_{ex} is set on the previous position of the location step in the query tree in the same way like in Figure 6.2.4.

Algorithm 6.4 CreateXPathModelPathFromXSEMPath

Input: collection of XSEM PSM classes $path$, **var** $path\_end$

Output: path of XPath model elements

1: $path\_end \leftarrow$ **new** $XPathElement(path[0].name)$
2: $element \leftarrow path\_end$
3: $CreateMapping(path[0], path\_end)$ {Create new mapping between XSEM class and XPath element}
4: **for** $i = 1 \rightarrow path.Count$ **do**
5: $\quad next\_element \leftarrow$ **new** $XPathElement(path[i].name)$
6: $\quad axis \leftarrow$ **new** $ChildAxis(path\_end, next\_element)$
7: $\quad CreateMapping(path[i], next\_element)$ {Create new mapping between XSEM class and XPath element}
8: $\quad path\_end \leftarrow next\_element$
9: **end for**
10: **return** $element$

The method described in Algorithm 6.4 creates a path of XPath elements $E_1, ..., E_n$ connected by XPath child axes $L_1, ..., L_{n-1}$ from given XSEM PSM

classes $C_1, ..., C_n$ where axis L_i^{ch} connects elements E_i and E_{i+1}. It returns the first element E_1 of the path and last element E_n as a reference parameter (line 3). Then every PSM class C_i and XPath element E_i are interconnected by mapping M_i (lines 4–9).

Example 6.2. For an example of creating an absolute path in an XPath model and a mapping with classes from an XSEM model we use Example 6.1. The method from Algorithm 6.1 returns classes *vehicle* and *motorcycle* for given class *motorcycle* . Algorithm 6.4 creates from these classes a path of the XPath elements *vehicle* and *motorcycle* connected by child axes L_{ch} and creates a mapping between correspondent classes and elements: $M(C^{vehicle}, E^{vehicle})$ and $M(C^{motorcycle}, E^{motorcycle})$.

Descendant Axis In this case the query result R is changed only if the new element x is added into the sub-tree of p. Other cases must be inspected by different axes. Let elements q and r are in the sub-tree of p.

Preconditions: $p \in S \wedge q = parent(q) \wedge q = parent(r) \wedge x \notin S$

Postconditions: $x \in S' \wedge x \in descendant(p)$

Query revalidation

- Suppose a query $p/descendant :: *$ which returns all nodes in the sub-tree of element p. We add new element into the sub-tree of element p (for example as a child of element q). In this case the query will be updated to the form $p/descendant/ :: * \ except \ absolute\_path\_to\_added\_element(x)$. If there exists a child of the element q which has the same name as x, it is not possible to ensure, that the new result will be the same due to *minOccurs* parameter which preludes to use *position* constructs in the updated query, because it is not possible if no information about data are considered. Duplicate name of sibling elements cause that it is not possible to select precisely the added element.

If the query uses a name test of the child element – $p/descendant :: name$, revalidation is needed only if the node test and the name of the added elements are the same.

The update method for XPath model is the same as for child axis in Algorithm 6.2.

Descendant-or-self Axis This is the same case as a descendant axis. Let element x be added into the sub-tree of element p and let elements q and r be in the sub-tree of element p. Case $R! = R'$ come up only if the element x is added into the sub-tree of element p.

Preconditions: $p \in S \wedge p = parent(q) \wedge q = parent(r) \wedge x \notin S$

Postconditions: $x \in S' \wedge x \in descendant(p)$

Query revalidation

- Suppose a query $p/descendant-or-self :: *$ which returns all nodes in the sub-tree of element p. Let element x is added as a child of element q. The query Q' must be updated to $p/descendant-or-self/ :: *$ *except* $absolute\_path\_to\_added\_element(x)$.

As in previous case, if node test contains a name, revalidation is needed only if the node test and the name of the newly added elements are the same.

The method used for update of descendant-or-self axis is the same as in Algorithm 6.2.

Following Axis When a following axis is used, the query result can change only if the element x is added into the part of the schema which is affected by the following axis. Let element x is added somewhere into this part.

Preconditions: $p \in S \wedge x \notin S$

Postconditions: $x \in S' \wedge x \in following(p)$

Query revalidation
Adding of the element x can cause the same incompatibility like case of descendant axis. A query $p/following :: *$ must be updated to $p/following :: *$ *except* $absolute\_path\_to\_added\_element(x)$ to satisfy $R = R'$. Update method for this axis is the same as Algorithm 6.2.

Following-sibling Axis This axis has the same revalidation as the following axis. The query result can change only if new element x is added into the schema part hit by following-sibling axis applied on element p.

Preconditions: $p \in S \wedge x \notin S$

Postconditions:$x \in S' \wedge x \in following-sibling(p)$

Query revalidation
A query $p/following-sibling :: *$ has to be updated to $p/following-sibling/ :: *$ *except* $absolute\_path\_to\_added\_element(x)$. In case when the name of the

added element is not unique between its siblings, it is not possible to ensure that the update query returns the same result and it must be updated manually by user.
Method which is used for updating has the same algorithm as Algorithm 6.2.

Preceding Axis Preceding axis is a symmetric case to descendant axis. The query must be updated only if the added element x is set into the part of schema returned by the preceding axis which was applied on element p.

Preconditions: $p \in S \wedge x \notin S$

Postconditions: $p \in S \wedge x \in S' \wedge x \in preceding(p)$

Query revalidation
A query $p/preceding :: *$ have to be updated to $p/preceding/ :: *$ *except* $absolute\_path\_to\_added\_element(x)$.
The update method is identical to Algorithm 6.2.

Preceding-sibling Axis As in the case of preceding axis, a change must be propagated only if the added element x is set to the part of the schema which is returned by preceding-sibling axis applied on element p.

Preconditions: $p \in S \wedge x \notin S$

Postconditions: $x \in S' \wedge x \in preceding - sibling(p)$

Query revalidation
The query $Q' = p/preceding - sibling :: *$ must be updated to $p/preceding - sibling :: *$ *except* $absolute\_path\_to\_added\_element(x)$.
The method for updating preceding-sibling axis is the same as in Algorithm 6.2.

Parent Axis If new element x is added into the schema as a child element, the change does not affect the result returned by parent axis applied on element p. The reason is the same as in the case of ancestor axis.

Preconditions: $p \in S \wedge x \notin S$

Postconditions: $x \in S'$

Query revalidation
No revalidation of the query is needed.

Self Axis New added element x cannot cause change in a result of the self axis. If the adding operation cause change in the result and $R! = R'$, it will be detected by another axis.

Preconditions: $x \notin S$

Postconditions: $x \in S'$

Query revalidation
It is not possible.

6.2.2 Removal

This operation removes an element from the schema. As mentioned before, the removed element must be a leaf of the schema tree. If we want to remove whole sub-tree, it can be done by iterating this operation. In all cases we suppose that we remove the element x.

Ancestor Axis Let p be a parent element of removed element x.

Preconditions: $x \in S \wedge p \in S \wedge p = parent(x)$

Postconditions: $x \notin S'$

Query revalidation
If the element x is removed and the original query was $x/ancestor :: *$, the query must be updated to $absolute\_path\_to\_element(p)/ancestor - or - self :: * \mid x/ancestor :: *$ which satisfy $R = R'$. If the removed element is not x, query will be inspected by another axis.

The method shown in Algorithm 6.5 creates new expression element E_{ex} (line 2) and reconnects attached location step as the first child of the element (line 3). Then must be created a path from the root to element E^p (line 4) representing parent class C^p of the removed class C^x and added as the second child of the E_{ex}. Because the class C^x was removed, all mappings of this class have to be removed too (line 7).

Ancestor-or-self Axis Suppose that element p is a parent of element x.

Preconditions: $x \in S \wedge p \in S \wedge p = parent(x)$

Postconditions: $x \notin S'$

Query revalidation
If the element x is removed and the query is $x/ancestor - or - self :: *$, there is no

Algorithm 6.5 UpdateWhenClassRemovedAncestorAxis

Input: location step ls, removed XSEM PSM class C^x

1: $path \leftarrow GetPathFromRootToElement(C^x.parent)$
2: $exp\_element \leftarrow$ **new** $XPathExpressionElement(\text{"}disjunction\text{"})$
3: $ReconnectElements(ls, exp\_element)$
4: $exp\_element.secondAxis \leftarrow$ **new** $XPathAxis(exp\_element,$ $CreateXPathModelPathFromXSEMPath(path, path\_end))$
5: $exp\_element.resultAxis \leftarrow ls.targetElement.childAxis$
6: **new** $XPathAncestorOrSelfAxis(path\_end,$ **new** $XPathElement(ls.targetElement.name))$ {Create L_{aos} between end of the created path and new created XPath element for node test of the ancestor axis}
7: $DeleteMappingsOfXSEMClass(C^x)$

possibility how to preserve compatibility of the query result, because the element x was removed from the schema and $R' \subset R$ and x cannot be in the result of the query. Otherwise no update has to be done.

Child Axis Let p be a parent of element x.

Preconditions: $x \in S \wedge p \in S \wedge p = parent(x)$

Postconditions: $x \notin S'$

Query revalidation
As was mentioned before, we can remove only a leaf element. If the removed element is x, it has no children and the original query will return an empty set. If the removed element is x and the query is $p/child :: *$ where $p = parent(x)$, it is not possible to update the query Q' because the removed element x cannot be hit.

Descendant Axis A descendant axis selects a sub-tree of the element and the removed element x has no descendants, so no update is needed. Next, when the removed element is x and $x \in descendant(p)$, it is not possible to return removed element x in the result and $R' \subset R$ will holds.

Preconditions: $x \in S \wedge p \in S \wedge p = parent(x)$

Postconditions: $x \notin S'$

Descendant-or-self Axis It is the same case as the descendant axis.

Following Axis With the following axis there are different situations depending on the position of the removed element. Suppose that if there are more occurrences of element x (elements with the same names) in the schema and x is in the result of the query, the removed element is the first one from the left to the right in the tree.

Preconditions: $x \in S \wedge p \in S \wedge p = parent(x)$

Postconditions: $x \notin S'$

Query revalidation
Let the element x be only once in the schema.

- Suppose that x is not the only child of p and not the last one. Then the query $x/following :: *$ will be updated to $x/following :: * \mid absolute\_path\_to\_next\_sibling(x)/(descendant-or-self :: * \mid following :: *)$.
- Suppose that x is the only child of p or the last in the sequence, the query $x/following :: *$ will be updated to $x/following :: * \mid absolute\_path\_to\_next\_element\_right(x)/(descendant-or-self :: * \mid following :: *)$.

Algorithm 6.6 has to create expression element E^1_{ex} for disjunction (line 1) of the original location step and the added location path. A path from the root to the next right XSEM class of the removed class C^x is added as the second child of element E^1_{ex} (line 4). The next expression element E^2_{ex} representing an expression in brackets (location steps with axes L_{dos} and L_f) is connected at the end of the path (line 12).

If the element x occurs multiple times in the schema, there are two different situation.

- If the removed element is the first one from the left, there must be done the same update as in previous case.
- If the removed element x is not the first (from left to right) in the schema tree, this element will be not present in the result of the query R' and it is not possible to satisfy that the result R' will be the same as for the original query.

If the removed element is not the element x, but is hit by the following axis, it is not possible to preserve the result of the original query as in previous case.

Example 6.3. Suppose the schema depicted in Figure 6.2.5 and the query $//address/following :: *$ which selects all following elements of the element

Algorithm 6.6 UpdateWhenClassRemovedFromFollowingAxis

Input: location step ls, removed XSEM PSM class C^x

1: $path \leftarrow GetPathFromRootToNextRightClass(C^x)$ {Method which returns path from root to next sibling or next right XSEM class in the schema}
2: $exp\_element \leftarrow$ **new** $XPathExpressionElement(\text{"}disjunction\text{"})$
3: $ReconnectElements(ls, exp\_element)$
4: $exp\_element.secondAxis \leftarrow$ **new** $XPathAxis(exp\_element, CreateXPathModelPathFromXSEMPath(path, path\_end))$
5: $exp\_element.resultAxis \leftarrow ls.targetElement.childAxis$
6: $exp\_element \leftarrow$ **new** $XPathExpressionElement(\text{"}disjunction\text{"})$
7: $new\_element \leftarrow$ **new** $XPathElement(ls.targetElement.name)$
8: $CreateMappings(GetDescendantOrSelfClasses(C^x), new\_element)$ {Create mappings between collection of XSEM classes and XPath element. Method $GetDescendantOrSelfClasses$ returns collection of descendant-or-self classes of the given class in XSEM model}
9: $exp\_element.firstAxis \leftarrow$ **new** $DescendantOrSelfAxis(exp\_element, new\_element)$
10: $new\_element \leftarrow$ **new** $XPathElement(ls.targetElement.name)$
11: $CreateMappings(GetFollowingClasses(C^x), new\_element)$
12: $exp\_element.secondAxis \leftarrow$ **new** $FollowingAxis(exp\_element, new\_element)$
13: **new** $XPathAxis(path\_end,$ **new** $exp\_element)$ {Create axis to connect end of the path with the expression element}
14: $DeleteMappingsOfXSEMClass(C^x)$ {Remove mappings of removed class C^x}

address: *delivery*, *item*, *items*, *address*, *notes*. If the element */purchase/person/address* is removed (Figure 6.2.5 on the right), query must be updated by Algorithm 6.6 to ensure $R = R'$. The original and the new XPath models of the queries Q and Q' are depicted in Figure 6.2.6 on the left and on the right respectively.

Following-sibling Axis Let the removed element be x and p is a parent element of x.

Preconditions: $x \in S \wedge p \in S \wedge p = parent(x)$

Postconditions: $x \notin S'$

Query revalidation

- First, suppose that x is not the only child of element p. If the element x is removed, the original query $x/following - sibling :: *$ must be updated to

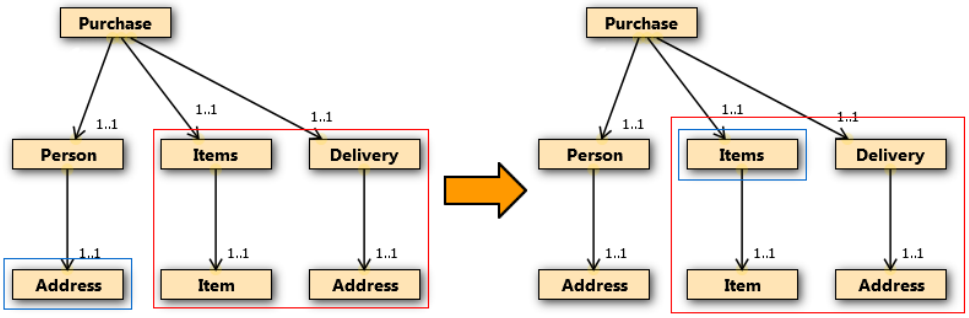


Figure 6.2.5: Removal of the following axis - XSEM PSM

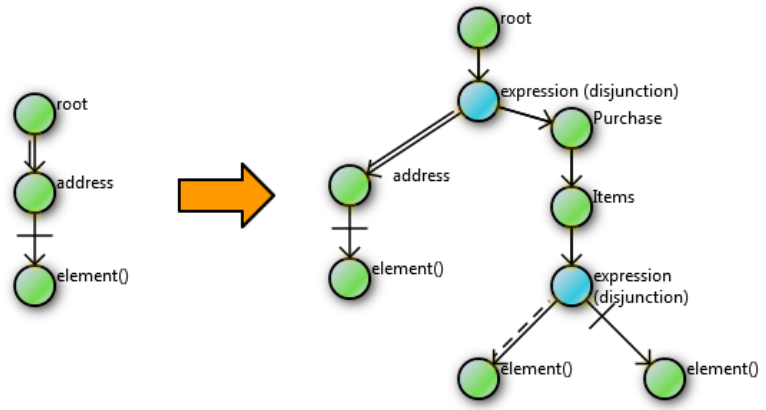


Figure 6.2.6: Removal of the following axis - XPath model

$absolute\_path\_to\_next\_sibling(x)/(self :: * \mid following - sibling :: *) \mid x/following - sibling :: *$ to preserve $R = R'$.

- Suppose that element x is the only child of p, then the original query returns an empty set and it holds $R = R'$.

If the removed element is not x, but is hit by the following-sibling axis, it is not possible to preserve $R = R'$.

Algorithm 6.7 has to create an expression element E^1_{ex} for disjunction of the original location step and the newly added location path. A path from the root to next sibling XSEM class of the removed class C^x is connected as the second child of element E^1_{ex}. At the end of the created path is connected next expression element E^2_{ex} representing expression in brackets - location steps with axes L_s and L_{fs}.

Preceding Axis In case of the preceding axis there must be distinguished different situations depending on position and number of occurrences of the removed

Algorithm 6.7 UpdateWhenClassRemovedFromFollowingSiblingAxis

Input: location step ls, removed XSEM PSM class C^x

1: $path \leftarrow GetPathFromRootToNextSibling(C^x)$
2: $exp\_element \leftarrow$ **new** $XPathExpressionElement(\text{"}disjunction\text{"})$
3: {reconnection of the elements}
4: $ReconnectElements(ls, exp\_element)$
5: $exp\_element.secondAxis \leftarrow$ **new** $XPathAxis(exp\_element,$ $CreateXPathModelPathFromXSEMPath(path, path\_end))$
6: $exp\_element.resultAxis \leftarrow ls.targetElement.childAxis$
7: $exp\_element \leftarrow$ **new** $XPathExpressionElement(\text{"}disjunction\text{"})$
8: $new\_element \leftarrow$ **new** $XPathElement(ls.targetElement.name)$
9: $CreateMapping(C^x, new\_element)$
10: $exp\_element.firstAxis \leftarrow$ **new** $SelfAxis(exp\_element,$ $new\_element)$
11: $new\_element \leftarrow$ **new** $XPathElement(ls.targetElement.name)$
12: $CreateMappings(GetFollowingSiblingClasses(C^x), new\_element)$ {Method $GetFollowingSiblingClasses$ returns collection of following-sibling classes of given class in XSEM model}
13: $exp\_element.secondAxis \leftarrow$ **new** $FollowingSiblingAxis(exp\_element,$ $new\_element)$
14: **new** $XPathAxis(path\_end,$ **new** $exp\_element)$
15: $DeleteMappingsOfXSEMClass(C^x)$

element. Suppose that if there are multiple occurrences of the element x in the schema and $x \in R$, the removed element is the last one from left to right in the tree.

Preconditions: $x \in S \wedge p \in S \wedge p = parent(x)$

Postconditions: $x \notin S'$

Query revalidation
Let element x is only once in the schema.

- Suppose that x is not the only child of element p and not the first one. Then the query $x/preceding :: *$ will be updated to $absolute\_path\_to\_previous\_sibling(x)/(descendant-or-self :: * \mid preceding :: *) \mid x/preceding :: *$.

If the element x occurs multiple times in the schema, there are two different situations.

- If the removed element is the last one from the left, there must be done the same updates like in previous case.

- If the removed element x is not the last one (from left to right) in the schema tree, this element will be not present in the result R' of the query. It is not possible to satisfy that $R = R'$.

Algorithm 6.8 UpdateWhenClassRemovedFromPrecedingAxis

Input: location step ls, removed XSEM PSM class C^x

1: $path \leftarrow GetPathFromRootToPreviousLeftClass(C^x)$
2: $exp\_element \leftarrow$ **new** $XPathExpressionElement(\text{"}disjunction\text{"})$
3: {reconnection of the elements}
4: $ReconnectElements(ls, exp\_element)$
5: $exp\_element.secondAxis \leftarrow$ **new** $XPathAxis(exp\_element,$ $CreateXPathModelPathFromXSEMPath(path, path\_end))$
6: $exp\_element.resultAxis \leftarrow ls.targetElement.childAxis$
7: $exp\_element \leftarrow$ **new** $XPathExpressionElement(\text{"}disjunction\text{"})$
8: $new\_element \leftarrow$ **new** $XPathElement(ls.targetElement.name)$
9: $CreateMappings(GetDescendantOrSelfClasses(C^x), new\_element)$
10: $exp\_element.firstAxis \leftarrow$ **new** $DescendantOrSelfAxis(exp\_element,$ $new\_element)$
11: $new\_element \leftarrow$ **new** $XPathElement(ls.targetElement.name)$
12: $CreateMappings(GetPrecedingClasses(C^x), new\_element)$ {Method $GetPrecedingClasses$ returns collection of the preceding classes of given class in XSEM model}
13: $exp\_element.secondAxis \leftarrow$ **new** $PrecedingAxis(exp\_element,$ $new\_element)$
14: **new** $XPathAxis(path\_element,$ **new** $exp\_element)$
15: $DeleteMappingsOfXSEMClass(C^x)$

The expression element E^1_{ex} for disjunction of original location step and the added location path must be created as in Algorithm 6.8 (line 1). A path from root to previous XSEM class of the removed class C^x is added as the second child of element E^1_{ex}(line 5). At the end of this path is connected next expression element E^2_{ex} representing an expression in brackets - location steps with axes L_{dos} and L_p (line 14).
If the removed element is not x, but is hit by the preceding axis, it is not possible to preserve the result of the original query.

Preceding-sibling Axis Preceding-sibling axis is the opposite of the following-sibling axis. Let the removed element be x and p be the parent element of x.

Preconditions: $x \in S \wedge p \in S \wedge p = parent(x)$

Postconditions: $x \notin S'$

Query revalidation

- Suppose that x is not the only child of element p. If the element x is removed, original query $x/preceding-sibling::*$ must be updated to $absolute\_path\_to\_previous\_sibling(x)/(self::* \mid preceding-sibling::*) \mid x/preceding-sibling::*$.
- Suppose that x is the only child of p, then the original query returns an empty set and no update is needed.

Algorithm 6.9 UpdateWhenClassRemovedFromPrecedingSiblingAxis

Input: location step ls, removed XSEM PSM class C^x

1: $path \leftarrow GetPathFromRootToPreviousSiblingClass(C^x)$
2: $exp\_element \leftarrow$ **new** $XPathExpressionElement(\text{"}disjunction\text{"})$
3: {reconnection of the elements}
4: $ReconnectElements(ls, exp\_element)$
5: $exp\_element.secondAxis \leftarrow$ **new** $XPathAxis(exp\_element, CreateXPathModelPathFromXSEMPath(path, path\_end))$
6: $exp\_element.resultAxis \leftarrow ls.targetElement.childAxis$
7: $exp\_element \leftarrow$ **new** $XPathExpressionElement(\text{"}disjunction\text{"})$
8: $new\_element \leftarrow$ **new** $XPathElement(ls.targetElement.name)$
9: $CreateMapping(C^x, new\_element)$
10: $exp\_element.firstAxis \leftarrow$ **new** $SelfAxis(exp\_element, new\_element)$
11: $new\_element \leftarrow$ **new** $XPathElement(ls.targetElement.name)$
12: $CreateMapping(GetPrecedingSiblingClasses(C^x), new\_element)$ {Method $GetPrecedingSiblingClasses$ returns collection of the preceding-sibling classes of given class in XSEM model}
13: $exp\_element.secondAxis \leftarrow$ **new** $PrecedingSiblingAxis(exp\_element, new\_element)$
14: **new** $XPathAxis(path\_end, exp\_element)$
15: $DeleteMappingsOfXSEMClass(C^x)$

If the removed element is not element x, but is hit by the preceding-sibling axis, there is no way how to preserve $R = R'$.

In Algorithm 6.9 the expression element E^1_{ex} for disjunction is created and the original location step is reconnected as the first child of this element. The path from the root to the previous sibling XSEM class of the removed class C^x is added as the second child of element E^1_{ex}. The expression element E^2_{ex} representing expression in brackets (location steps with axes L_s and L_{ps}) is connected at the end of this path.

Parent Axis Suppose that p is a parent element of x.

Preconditions: $x \in S \wedge p \in S \wedge p = parent(x)$

Postconditions: $x \notin S'$

Query revalidation
In this case the query $x/parent :: *$ have to be updated to form $absolute\_path\_to\_element(p)/self :: * \mid x/parent :: *$ to satisfy that $p \in R'$.

The update method presented in Algorithm 6.10 adds only one expression element E_{ex} of disjunction, reconnects original location step as the first child and creates new path from root to parent class C^p of C^x. At the end of this path is connected location step with self axis L_s.

Algorithm 6.10 UpdateWhenClassRemovedFromParentAxis

Input: location step ls, XSEM PSM class C^x

1: $path \leftarrow GetPathFromRootToElement(C^x.parent)$
2: $exp\_element \leftarrow$ **new** $XPathExpressionElement(\text{"}disjunction\text{"})$
3: $ReconnectElements(ls, exp\_element)$
4: $exp\_element.secondAxis \leftarrow$ **new** $XPathAxis(exp\_element,$ $CreateXPathModelPathFromXSEMPath(path, path\_end))$
5: $exp\_element.resultAxis \leftarrow ls.targetElement.childAxis$
6: $new\_element \leftarrow$ **new** $XPathElement(ls.targetElement.name)$
7: $CreateMapping(C^x, new\_element)$
8: **new** $XPathSelfAxis(path\_end, new\_element)$
9: $DeleteMappingsOfXSEMClass(C^x)$

Self Axis If the element x is removed, a query $x/self :: *$ cannot be updated to preserve $R = R'$. There can be no result for a non-existing element.

6.2.3 Renaming

A renaming operation changes a name of the selected element. Change of the name can cause a change of the result set returned by the query. Let x be an element whose name is changed. Possible situations when the query can return more or less elements in the result R' are similar for all axes.
Update of the query must be done only if the element is hit by the name test. If the location step uses name test with $*$ or $element()$, no change is needed.

Query revalidation
There are two possibilities how the change of the name can affect the result of the

query – it can return more or less elements. Let x be an element whose name is changed to y.

- If more elements are returned ($R \subset R'$), a location step must be updated with *except absolute_path_to_renamed_element*(y), to ensure $R = R'$.
- In case when there are less elements in the result set ($R' \subset R$), a location step must add | *absolute_path_to_renamed_element*(y).

In the algorithm 6.11 has to be distinguished if there were found more or less results. By this fact the expression element E_{ex} is created for disjunction or except operator (line 2 – 6). Path from the root to the renamed class C^y is connected to the element E_{ex} (line 9 – 10). If the renamed class in the mapping with XPath element whose name corresponds with original name of C^x, a mapping must be removed (line 11).

Algorithm 6.11 UpdateWhenClassNameChanged

Input: location step *ls*, renamed XSEM PSM class C^y, the added XSEM classes *added*, missing XSEM classes *missing*

```
1:  exp_element
2:  if added.Count > 0 then
3:     exp_element ← new XPathExpressionElement("except")
4:  else if missing.Count > 0 then
5:     exp_element ← new XPathExpressionElement("disjunction")
6:  end if
7:  path ← GetPathFromRootToClass(C^y)
8:  ReconnectElements(ls, exp_element)
9:  exp_element.secondAxis ← new XPathAxis(exp_element,
    CreateXPathModelPathFromXSEMPath(path, path_end))
10: exp_element.resultAxis ← ls.targetElement.childAxis
11: DeleteMappingBetweenXSEMClassAndXPathElement(C^y,
    ls.targetElement) {References between renamed class and XPath element
    must be removed}
```

Example 6.4. Suppose a schema depicted in Figure 6.2.7 on the left. In the red rectangles are classes which are returned by the query $Q = //address$ whose XPath model is shown in Figure 6.2.8 on the left. If the name of the class */purchase/delivery/address* is renamed to *delivery_address*, then query Q' must be updated to the form *//address* | */purchase/delivery/delivery_address* depicted in Figure 6.2.8 on the right.

6.2.4 Reordering

In all the following cases we suppose, that an element x is in a *sequence* where the order of elements is significant and that element x has at least one sibling. An

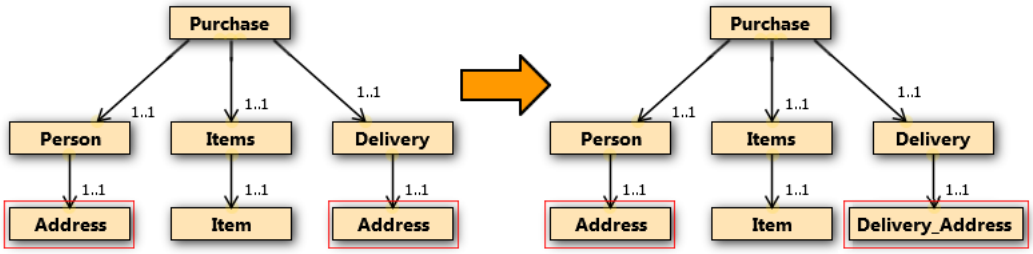


Figure 6.2.7: Renaming example - XSEM PSM

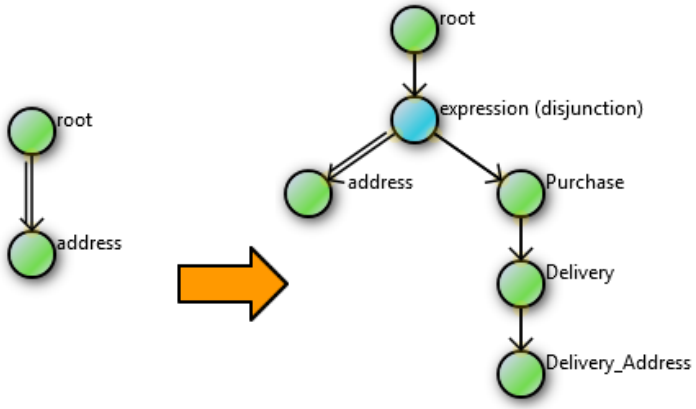


Figure 6.2.8: Renaming example - XPath model

example of reorder is depicted in Figure 6.2.9. An element can be moved only one position left or right in one step. Let $pos(element)$ be a position of an element in the original schema S and $pos'(element)$ a position in the new schema S'.

Ancestor Axis Reordering of the element x which is a child of element p has no impact on a query with ancestor axis.

Preconditions: $x \in S \wedge p \in S \wedge p = parent(x)$

Postconditions: $pos(x)! = pos'(x)$

Query revalidation
If the position of the element x is changed, the result of the query remains the same. A location step starting from the element x does not change the result returned by ancestor axis and it holds $R = R'$.

Ancestor-or-self Axis As the ancestor axis, if the position of element x is changed, it has no impact on the result R'.

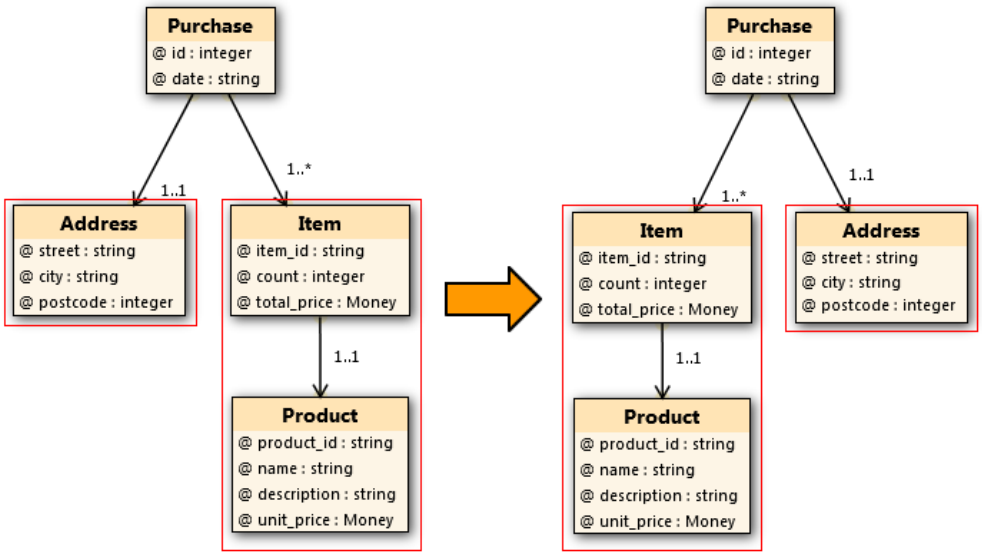


Figure 6.2.9: Reordered PSM schema

Preconditions: $x \in S \wedge p \in S \wedge p = parent(x)$

Postconditions: $pos(x)! = pos'(x)$

Query revalidation
If the position of element x is changed, for results it holds $R = R'$.

Child Axis If the position of the element x is changed, child elements of the element x are still in the same position within x.

Preconditions: $x \in S \wedge p \in S \wedge p = parent(x)$

Postconditions: $pos(x)! = pos'(x)$

Query revalidation
For results it holds $R = R'$ and no change is needed.

Descendant Axis Change of the position of element x has no impact on the result.

Preconditions: $x \in S \wedge p \in S \wedge p = parent(x)$

Postconditions: $pos(x)! = pos'(x)$

Query revalidation
There is no change needed.

Descendant-or-self Axis This is the same case as a descendant axis.

Following Axis Suppose that the element x is a child of p at position $\#pos$, the element y is a child of p at position $\#pos + 1$ and z is a child of the element p at position $\#pos - 1$.
An example of reordering when the following axis is used is shown in Figure 6.2.10.

Preconditions: $x \in S \wedge x \in S' \wedge pos(x) = pos(y) - 1 \wedge pos(x) = pos(z) + 1 \wedge p = parent(x) \wedge p = parent(z) \wedge p = parent(y)$

Postconditions: $pos(x)! = pos'(x)$

Query revalidation

- If the position of x is changed to $\#pos'$, where $\#pos+1 = \#pos'$, the original query $x/following :: *$ must be updated to $x/following :: * \mid absolute\_path\_to\_element(y)/descendant - or - self :: *$.
- If the position of element x is changed to $\#pos'$, where $\#pos - 1 = \#pos'$, then the query $x/following :: *$ must be updated to $x/following :: * \; except \; absolute\_path\_to\_element(z)/descendant - or - self :: *$.

The algorithm 6.12 determines direction of the moved class C^x and creates the appropriate expression element E_{ex} (line 3 – 9) and a path to the class C^z or to the class C^y that are connected as the second child of E_{ex}. The location step with axis L_{dos} is added at the end of this path (line 15).

Example 6.5. In Figure 6.2.10 on the left is the original schema before reordering. In the blue rectangle is the initial element and in the green rectangle is the result returned by the following axis. On the right image is the model after reordering of the element *item* to the left. The element *address* in the red rectangle is additional to the original result and must be eliminated from the query.

Following-sibling Axis Suppose that the element x is a child of p at position $\#pos$, y is a child of p at position $\#pos+1$ and z is a child of element p at position $\#pos - 1$.

Preconditions: $x \in S \wedge x \in S' \wedge pos(x) = pos(y) - 1 \wedge pos(x) = pos(z) + 1 \wedge p = parent(x) \wedge p = parent(z) \wedge p = parent(y)$

Algorithm 6.12 UpdateWhenFollowingAxisReorder

Input: location step ls, XSEM PSM class C^x, direction of the move dir, collection of missing XSEM PSM classes not hit by original query in changed model $missing$

1: $path$
2: $exp\_element$
3: **if** $dir =$ "$right$" **then**
4: $path \leftarrow GetPathFromRootToNextRightClass(C^x)$
5: $exp\_element \leftarrow$ **new** $XPathExpressionElement($"$disjunction$"$)$
6: **else**
7: $path \leftarrow GetPathFromRootToPreviousLeftClass(C^x)$
8: $exp\_element \leftarrow$ **new** $XPathExpressionElement($"$except$"$)$
9: **end if**
10: $ReconnectElements(ls, exp\_element)$
11: $exp\_element.secondAxis \leftarrow$ **new** $XPathAxis(exp\_element,$ $CreateXPathModelPathFromXSEMPath(path, path\_end))$
12: $exp\_element.resultAxis \leftarrow ls.targetElement.childAxis$
13: $new\_element \leftarrow$ **new** $XPathElement(ls.targetElement.name)$
14: $CreateMappings(GetDescendantOrSelfClasses(path.last), new\_element)$
15: **new** $XPathDescendantOrSelfAxis(exp\_element, new\_element)$
16: $DeleteMappingsOfXSEMClassesAndXPathElement(missing,$ $ls.targetElement)$

Postconditions: $pos(x)! = pos'(x)$

Query revalidation

- If the position of the element x is changed to $\#pos'$, where $\#pos + 1 = \#pos'$, the query $Q' = x/following - sibling :: *$ must be updated to the form $x/following - sibling :: * \mid absolute\_path\_to\_element(y)/self :: *$.
- If the position of x is changed to $\#pos'$, where $\#pos - 1 = \#pos'$, the query $Q' = x/following - sibling :: *$ must be updated to $x/following - sibling :: * except\ absolute\_path\_to\_element(z)/self :: *$.

The algorithm 6.13 differs from Algorithm 6.12 only in the axis of the location step connected to the created path – in this case L_s must be added.

Preceding Axis Let the element x is a child of element p at position $\#pos$, y is a child of p at position $\#pos+1$ and z is a child of element p at position $\#pos-1$.

Preconditions: $x \in S \wedge x \in S' \wedge pos(x) = pos(y) - 1 \wedge pos(x) = pos(z) + 1 \wedge p = parent(x) \wedge p = parent(z) \wedge p = parent(y)$

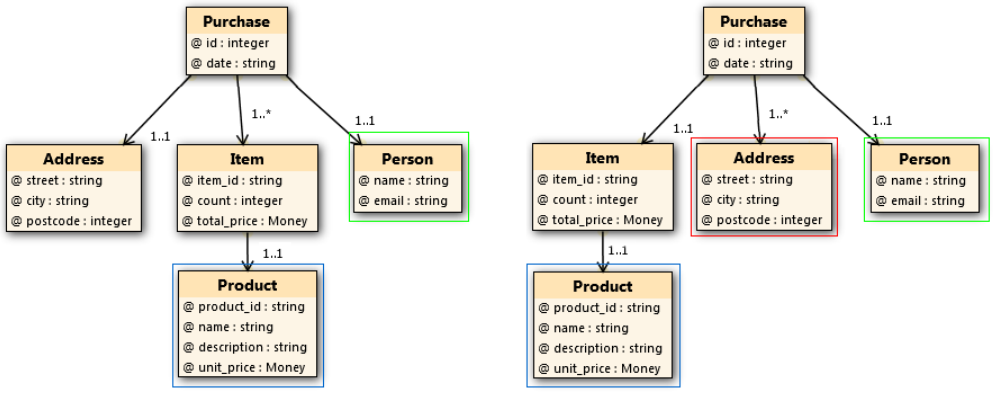


Figure 6.2.10: Reorder with following axis

Postconditions: $pos(x)! = pos'(x)$

Query revalidation

- If the position of x is changed to $\#pos'$, where $\#pos - 1 = \#pos'$, the query $x/preceding :: *$ must be updated to $x/preceding :: * \mid absolute\_path\_to\_element(z)/descendant-or-self :: *$ to satisfy $R = R'$.
- If the position of x is changed to $\#pos'$, where $\#pos + 1 = \#pos'$, then the query $x/preceding :: *$ must be updated to $x/preceding :: *$ *except* $absolute\_path\_to\_element(y)/descendant - or - self :: *$.

A method shown in Algorithm 6.14 is opposite of method for following axis (Algorithm 6.12). It is different only in the condition for direction (line 3).

Preceding-sibling Axis Suppose that element x is a child of p at position $\#pos$.

Preconditions: $x \in S \wedge x \in S' \wedge pos(x) = pos(y) - 1 \wedge pos(x) = pos(z) + 1 \wedge p = parent(x) \wedge p = parent(z) \wedge p = parent(y)$

Postconditions: $pos(x)! = pos'(x)$

Query revalidation

- If the position of x is changed to $\#pos'$, where $\#pos - 1 = \#pos'$, then query $Q = x/preceding-sibling :: *$ must be updated to $x/preceding-sibling :: * \mid absolute\_path\_to\_element(z)/self :: *$.

Algorithm 6.13 UpdateWhenFollowingSiblingAxisReorder

Input: location step ls, XSEM PSM class C^x, direction of the move dir, collection of missing XSEM PSM classes not hit by original query in changed model $missing$

1: $path$
2: $exp\_element$
3: **if** $dir =$ "$right$" **then**
4: $path \leftarrow GetPathFromRootToNextRightClass(C^x)$
5: $exp\_element \leftarrow$ **new** $XPathExpressionElement($"$disjunction$"$)$
6: **else**
7: $path \leftarrow GetPathFromRootToPrevisousLeftClass(C^x)$
8: $exp\_element \leftarrow$ **new** $XPathExpressionElement($"$except$"$)$
9: **end if**
10: $ReconnectElements(ls, exp\_element)$
11: $exp\_element.secondAxis \leftarrow$ **new** $XPathAxis(exp\_element, CreateXPathModelPathFromXSEMPath(path, path\_end))$
12: $exp\_element.resultAxis \leftarrow ls.targetElement.childAxis$
13: $new\_element \leftarrow$ **new** $XPathElement(ls.targetElement.name)$
14: $CreateMapping(path.last, new\_element)$
15: **new** $XPathSelfAxis(exp\_element, new\_element)$
16: $DeleteMappingsOfXSEMClassesAndXPathElement(missing, ls.targetElement)$ {see Algorithm 5.5}

- If the position of x is changed to $\#pos'$, where $\#pos+1 = \#pos'$, then $Q = x/preceding-sibling :: *$ must be updated to $x/preceding-sibling :: * \; except \; absolute\_path\_to\_element(y)/self :: *$.

The algorithm 6.15 determines direction of the moved class C^x and creates appropriate expression element E_{ex} and the path to class C^z or to class C^y by direction of the move (line 3 – 9). Location step with axis L_s is added at the end of path to class C^z or C^y (line 15). Invalid mappings are removed at the end (line 16).

Parent Axis Parent axis is a special case of ancestor axis.

Preconditions: $x \in S \wedge x \in S' \wedge pos(x) = pos(y) - 1 \wedge pos(x) = pos(z) + 1$
$p = parent(x) \wedge p = parent(z) \wedge p = parent(y)$
Postconditions: $pos(x)! = pos'(x)$

Query revalidation
A movement of an element into different position has no impact of the result of parent axis and it holds $R = R'$.

Self Axis Change of position of an element on which is the self axis has no impact on the query.

Algorithm 6.14 UpdateWhenPrecedingAxisReorder

Input: location step ls, XSEM PSM class C^x, direction of the move dir, collection of missing XSEM PSM classes not hit by original query in changed model $missing$

1: $path$
2: $exp\_element$
3: **if** $dir =$ "$right$" **then**
4: $path \leftarrow GetPathFromRootToNextRightClass(C^x)$
5: $exp\_element \leftarrow$ **new** $XPathExpressionElement($"$except$"$)$
6: **else**
7: $path \leftarrow GetPathFromRootToPrevisousLeftClass(C^x)$
8: $exp\_element \leftarrow$ **new** $XPathExpressionElement($"$disjunction$"$)$
9: **end if**
10: $ReconnectElements(ls, exp\_element)$
11: $exp\_element.secondAxis \leftarrow$ **new** $XPathAxis(exp\_element,$ $CreateXPathModelPathFromXSEMPath(path, path\_end))$
12: $exp\_element.resultAxis \leftarrow ls.targetElement.childAxis$
13: $new\_element \leftarrow$ **new** $XPathElement(ls.targetElement.name)$
14: $CreateMappings(GetDescendantOrSelfClasses(path.last), new\_element)$
15: **new** $XPathDescendantOrSelfAxis(exp\_element, new\_element)$
16: $DeleteMappingsOfXSEMClassesAndXPathElement(missing,$ $ls.targetElement)$

6.2.5 Reconnection

Suppose that the element x is moved to another position in the schema tree. It means that we change a parent element p of x to element p'. For simplicity the element x can be reconnected only as a son of one of its siblings or as a sibling of its parent. Multiple iterations of the operation enables movement of the element to any element in the schema. Let $l(x)$ denoted the level of element x and $l(root) = 0$. Then x can move to $l(x) + 1$ or $l(x) - 1$. Next, it is not possible to move an element as a child of an element where its child has the same name. It is the same case as renaming a sibling element to the same name as one of its sibling.

In special cases when an element is moved outside the part of the query hit by an axis, the query must be updated in a specific way. This special situation will be described for respective axes.

An update of the query after a reconnection in the schema can be done correctly with all axes only if the location step is the last one in the query. If not, it is not possible to ensure the same results of the original and the updated queries in a simple way.

A reconnection of an element changes its position in the schema tree. This implies that axes applied on this element can return different result than if they

Algorithm 6.15 UpdateWhenPrecedingSiblingAxisReorder

Input: location step ls, XSEM PSM class C^x, direction of the move dir, collection of missing XSEM PSM classes not hit by original query in changed model $missing$

```
1: path
2: exp_element
3: if dir = "right" then
4:    path ← GetPathFromRootToNextRightClass(C^x)
5:    exp_element ← new XPathExpressionElement("except")
6: else
7:    path ← GetPathFromRootToPrevisousLeftClass(C^x)
8:    exp_element ← new XPathExpressionElement("disjunction")
9: end if
10: ReconnectElements(ls, exp_element)
11: exp_element.secondAxis ← new XPathAxis(exp_element,
    CreateXPathModelPathFromXSEMPath(path, path_end))
12: exp_element.resultAxis ← ls.targetElement.childAxis
13: new_element ← new XPathElement(ls.targetElement.name)
14: CreateMapping(path.last, new_element)
15: new XPathSelfAxis(exp_element, new_element)
16: DeleteMappingsOfXSEMClassesAndXPathElement(missing,
    ls.targetElement)
```

were applied on the element in the original position (see Figure 6.2.11). Let R is a result of an axis applied on element in original schema S and R' is a result of the the axis in new schema S'. Then the location step represented by this axis has to be updated with disjunction of $R \setminus R'$ and exclusion of $R' \setminus R$.

Ancestor Axis Let p be a parent element of x, q be a parent of element p and r a sibling of element x.

Preconditions: $x \in S \wedge x \in S' \wedge p = parent(x) \wedge q = parent(p) \wedge p = parent(r)$

Postconditions: $q = parent(x) \vee r = parent(x)$

Query revalidation
Suppose that we reconnect element x whose parent is element p. To preserve $R = R'$, query Q' must be updated.

- First, element x is moved up, reconnected to the element q as a child. The query $x/ancestor :: *$ must be update to $x/ancestor :: * \mid absolute\_path\_to\_element(p)$ to ensure that $R = R'$.

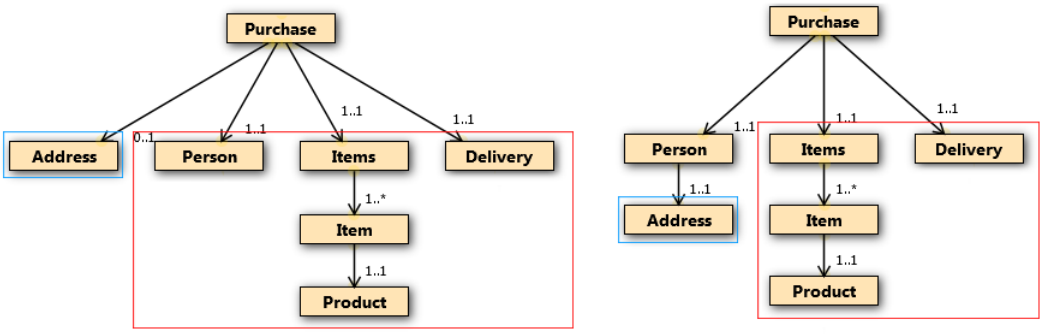


Figure 6.2.11: Example of reconnection problem

- Second, element x is moved down, reconnected to the element r as a child. The query $x/ancestor :: *$ must be update to $x/ancestor :: * \ except \ absolute\_path\_to\_element(r)$.

Algorithm 6.16 UpdateWhenReconnectAncestorAxis

Input: location step ls, reconnected XSEM PSM class C^x, sibling XSEM PSM class C^r where was class C^x reconnected, bool value if XSEM classes in new result are added or missing *added*

1: $path$
2: $exp\_element$
3: **if** $added = true$ **then**
4: $\quad path \leftarrow GetPathFromRootToClass(C^r)$
5: $\quad exp\_element \leftarrow$ **new** $XPathExpressionElement(\text{"}except\text{"})$
6: **else**
7: $\quad path \leftarrow GetPathFromRootToClass(C^x.parent)$
8: $\quad exp\_element \leftarrow$ **new** $XPathExpressionElement(\text{"}disjunction\text{"})$
9: **end if**
10: $ReconnectElements(ls, exp\_element)$
11: $exp\_element.secondAxis \leftarrow$ **new** $XPathAxis(exp\_element, CreateXPathModelPathFromXSEMPath(path, path\_end))$
12: $exp\_element.resultAxis \leftarrow ls.targetElement.childAxis$

A method for updating ancestor axis (Algorithm 6.16) differs by the results R and R'. If the result set increased ($R \subset R'$), the expression element E_{ex} for operator *except* has to be created (line 5), otherwise operator *disjunction* is set to preserve $R = R'$ (line 8). Then the attached location step is reconnected. The path to the parent class C^p is connected as the second child of the element E_{ex} if the result increased or class C^r if elements are missing.

Ancestor-or-self Axis Let element p be a parent of element x, element q a parent of element p and r a sibling of element x.

Preconditions: $x \in S \wedge x \in S' \wedge p = parent(x) \wedge q = parent(p)$
$\wedge\ p = parent(r)$

Postconditions: $q = parent(x) \vee r = parent(x)$

Query revalidation
Suppose that we reconnect element x whose parent is the element p. As in case of ancestor axis, there are two possible situations.

- First, the element x is moved up, reconnected to the element q as a child. The query $x/ancestor-or-self :: *$ must be update to $x/ancestor-or-self :: * \mid absolute\_path\_to\_element(p)$.
- Second, the element x is moved down, reconnected to the element r as a child. The query $x/ancestor-or-self :: *$ must be update to $x/ancestor-or-self :: *\ except\ absolute\_path\_to\_element(r)$.

The update method for ancestor-or-self axis has the same algorithm as Algorithm 6.16.

Child Axis Let element p be a parent of element x which is being reconnected. And let element q be a parent of element p and r a sibling of x. In this case there are various possibilities how the result R' can be changed.

Preconditions: $x \in S \wedge x \in S' \wedge p = parent(x) \wedge q = parent(p)$
$\wedge\ p = parent(r)$

Postconditions: $q = parent(x) \vee r = parent(x)$

Query revalidation

- First, the element x is moved up as a child of the element q. The query $p/child :: *$ must be updated to $p/child :: * \mid$ $absolute\_path\_to\_moved\_element(x)$.
- Secondly, the element x is moved down as a child of element r. The query $p/child :: *$ must be updated to $p/child :: * \mid$ $absolute\_path\_to\_moved\_element(x)$.
- If the element x is moved down as a child of element r. The query $r/child :: *$ must be updated to $r/child :: *\ except$ $absolute\_path\_to\_moved\_element(x)$.
- If the element x is moved up as a child of element q. The query $q/child :: *$ must be updated to $q/child :: * \mid$ $absolute\_path\_to\_moved\_element(x)$.

Algorithm 6.17 UpdateWhenReconnectChildAxis

Input: location step ls, reconnected XSEM PSM class C^x, bool value if XSEM classes in new result are added or missing *added*

1: $path \leftarrow GetPathFromRootToClass(C^x)$
2: $exp\_element$
3: **if** $added = true$ **then**
4: $\quad exp\_element \leftarrow$ **new** $XPathExpressionElement(\text{"}except\text{"})$
5: **else**
6: $\quad exp\_element \leftarrow$ **new** $XPathExpressionElement(\text{"}disjunction\text{"})$
7: **end if**
8: $ReconnectElements(ls, exp\_element)$
9: $exp\_element.secondAxis \leftarrow$ **new** $XPathAxis(exp\_element, CreateXPathModelPathFromXSEMPath(path, path\_end))$
10: $exp\_element.resultAxis \leftarrow ls.targetElement.childAxis$

The algorithm 6.17 is the same in all cases of reconnection of the class C^x. A type of the created expression element E_{ex} depends of the parameter *added* (line 3–7). After the reconnection of the original location step to the element E_{ex} (line 8), a path to reconnected class C^x must be created and connected (line 9–10).

Descendant Axis In comparison with a child axis there are different situations with validation of the queries. Suppose that element p is the parent of element x that is reconnected. Element q is the parent of element p and r is a sibling of x.

Preconditions: $x \in S \wedge x \in S' \wedge p = parent(x) \wedge q = parent(p) \wedge p = parent(r)$

Postconditions: $q = parent(x) \vee r = parent(x)$

Query revalidation

- First, the element x is moved up as a child of element q. The query $p/descendant :: *$ must be updated to $p/descendant :: * \mid absolute\_path\_to\_reconnected\_element(x)/\ descandant - or - self :: *$ to satisfy $R = R'$.
- The element x is moved down as a child of element r. The original query $p/descendant :: *$ does not need any update. The moved element x is still in the sub-tree of p.
- Now the element x is moved as a child of element r. In this case the query $r/descendant :: *$ must not return the reconnected element and the query must be updated to the form $r/descendant :: *\ except\ absolute\_path\_to\_reconnected\_element(x)/descendant - or - self :: *$.

Algorithm 6.18 UpdateWhenReconnectDescendantAxis

Input: location step ls, reconnected XSEM PSM class C^x, bool value if XSEM classes in new result are added or missing *added*

```
1:  path
2:  change ← false
3:  exp_element
4:  if added = false then
5:     path ← GetPathFromRootToClass(C^x)
6:     exp_element ← new XPathExpressionElement("disjunction")
7:     change ← true
8:  else
9:     path ← GetPathFromRootToClass(C^x)
10:    exp_element ← new XPathExpressionElement("except")
11:    change ← true
12: end if
13: if change = true then
14:    ReconnectElements(ls, exp_element)
15:    exp_element.secondAxis ← new XPathAxis(exp_element,
       CreateXPathModelPathFromXSEMPath(path, path_end))
16:    exp_element.resultAxis ← ls.targetElement.childAxis
17:    new_element ← new XPathElement(ls.targetElement.name)
18:    CreateMappings(GetDescendantOrSelfClasses(C^x), new_element)
19:    new XPathDescendantOrSelfAxis(path_end,
       new_element)
20: end if
```

The algorithm 6.18 depends on the direction of the reconnection (up or down) and on the class where the location step starts – these conditions influence result R' of the query with the schema S'. If it holds $R' \subset R$, element E_{ex} with operator *disjunction* must be added (line 6). Else if there were found new possible mappings ($R \subset R'$ and parameter *added* is *true*), E_{ex} with operator *except* is created (line 10). Otherwise, no change of location step is needed.

Descendant-or-self Axis This is the same case as in descendant axis. Reconnection of the whole sub-tree does not change this tree and $R = R'$. A reconnection of elements which adds or removes elements from the sub-tree has the same problems as descendant axis when holds $R! = R'$ and query Q' must be updated.

Preconditions: $x \in S \wedge x \in S' \wedge p = parent(x) \wedge q = parent(p) \wedge p = parent(r)$

Postconditions: $q = parent(x) \vee r = parent(x)$

Query revalidation

- First, the element x is moved up as a child of the element q. The query $p/descendant-or-self :: *$ must be updated to the form $p/descendant-or-self :: * \mid$ $absolute\_path\_to\_reconnected\_element(x)/descandant-or-self :: *$.
- If the element x is moved down as a child of element r. The original query $p/descendant-or-self :: *$ does not need any update. The moved element is still in the sub-tree of p.
- Now the element x is moved as a child of element r. In this case the query $r/descendant-or-self :: *$ must not return reconnected element and the query Q' must be updated to $r/descendant-or-self :: *$ *except* $absolute\_path\_to\_reconnected\_element(x)/descendant-or-self :: *$.

A method for updating is the same as in Algorithm 6.18.

Following Axis In case of the following axis a reconnection of an element causes, that the result given by this axis can return less or more elements which depends on the place in schema where the element was moved to. Let following axis is applied on element x ($x/following :: *$) and the element y is another element in the schema.

Preconditions: $x \in S \wedge x \in S' \wedge p = parent(x) \wedge q = parent(p)$
$\wedge\, p = parent(r)$

Postconditions: $q = parent(x) \vee r = parent(x)$

Query revalidation

- If the reconnected element is not the element x and the reconnection is done in the part of the tree hit by the following axis, no update is needed.
- If the reconnection of the element y cause, that the element is added into a part of tree hit by the axis ($R \subset R'$), query $x/following :: *$ must be updated to $x/following :: *$ *except* $absolute\_path\_to\_reconnected\_element(y)/descendant-or-self :: *$.
- On the other hand, if the element y is moved out from the hit part of the tree ($R' \subset R$), the query $x/following :: *$ must be updated to the form $x/following :: * \mid$ $absolute\_path\_to\_reconnected\_element(y)/descendant-or-self :: *$.

- If the reconnected element is x, a revalidation depends on the final position of the element x in the tree and between its siblings (see Figure 6.2.12) when holds $R! = R'$. There must be added location paths to the missing elements and locations paths to exclude redundant elements. In Figure 6.2.12 in the blue rectangle is an element on which is applied a following axis. In the top-left schema are the results shown in the red rectangle. In the top-right image is shown schema after the removing of the element *Address*. Additional elements which must not be returned are in the green rectangles. On the other hand the missing elements are in the green rectangles in the bottom-right schema.

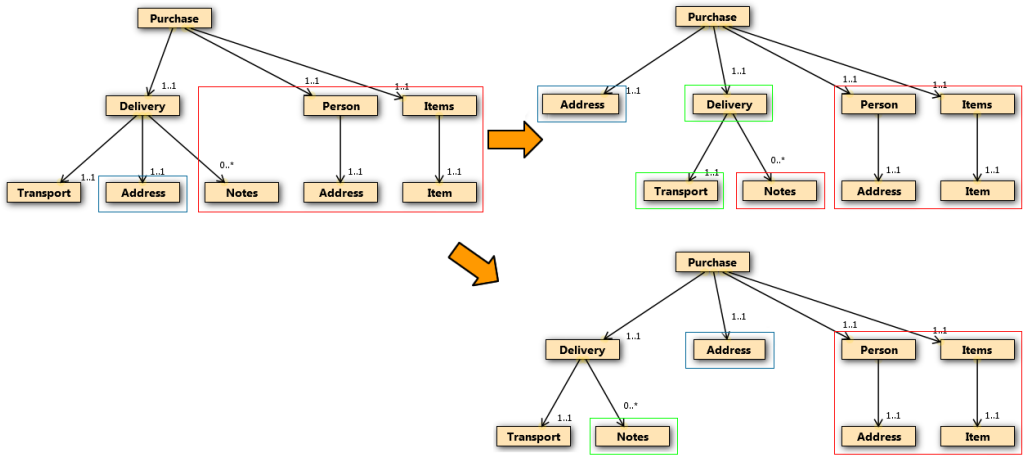


Figure 6.2.12: Reconnecting of the element with the following axis

The method for updating following axis case presented in Algorithm 6.19 depends on the reconnection of the class C^y. If it is moved out from the hit part of the tree, this class must be added to the query: a new expression element E_{ex} with *disjunction* operator is added and new a path to the moved class C^y is added (line 5). Otherwise this class must be eliminated and E_{ex} with operator *except* is added to the query (line 9). Then the reconnection of the element E_{ex} is done (line 12) and appropriate axis is created (line 16).

Following-sibling Axis In this case suppose that the axis is applied on element x. Let element p be the parent of x, r be one of the siblings element of x and y is the reconnected element.

Preconditions: $x \in S \wedge x \in S' \wedge p = parent(x)$
$\wedge (y = sibling(x) \vee y = sibling(p) \vee y = child(r))$

Algorithm 6.19 UpdateWhenReconnectFollowingAxis

Input: location step ls, reconnected XSEM PSM class C^y, bool value if XSEM classes in new result are added or missing *added*

```
1: path
2: exp_element
3: if added = true then
4:    path ← GetPathFromRootToClass(C^y)
5:    exp_element ← new XPathExpressionElement("except")
6:    change ← true
7: else
8:    path ← GetPathFromRootToClass(C^y)
9:    exp_element ← new XPathExpressionElement("disjunction")
10:   change ← true
11: end if
12: ReconnectElements(ls, exp_element)
13: exp_element.secondAxis ← new XPathAxis(exp_element,
    CreateXPathModelPathFromXSEMPath(path, path_end))
14: new_element ← new XPathElement("element()")
15: CreateMappings(GetDescendantOrSelfClasses(C^y), new_element)
    {Method GetDescendantOrSelfClasses returns collection of descendant
    classes of set class in XSEM model}
16: new XPathDescendantOrSelfAxis(path_end, new_element)
17: exp_element.resultAxis ← ls.targetElement.childAxis
```

Postconditions: $y = sibling(p) \ \vee \ r = parent(y)$
$\vee \ (y = sibling(x) \ \wedge \ pos(x) < pos(y))$

Query revalidation

- First, let y be a sibling of the element x where $pos(x) < pos(y)$.

  If the element y is moved up as a sibling of p or y is moved down as a child of one of its siblings, query $x/following - sibling :: *$ must be updated to form
  $x/following - sibling :: * \mid absolute\_path\_to\_reconnected\_element(y)$.

- Second, let y be a sibling of p or a child of one of siblings of element x and be reconnected as sibling of x where holds $pos(x) < pos(y)$. Then the query $x/following - sibling :: *$ must be updated to $x/following - sibling :: * \ except \ absolute\_path\_to\_reconnected\_element(y)$.

- When we reconnected element x, the same situation as in the case of a following axis came out.

A method for updating query in the first and the second case is shown in Algorithm 6.20.

Algorithm 6.20 UpdateWhenReconnectFollowingSiblingAxis

Input: location step ls, reconnected XSEM PSM class C^y, bool value if XSEM classes in new result are added or missing *added*

```
1: path ← GetPathFromRootToClass(C^y)
2: exp_element
3: if added = true then
4:    exp_element ← new XPathExpressionElement("except")
5:    change ← true
6: else
7:    exp_element ← new XPathExpressionElement("disjunction")
8:    change ← true
9: end if
10: ReconnectElements(ls, exp_element)
11: exp_element.secondAxis ← new XPathAxis(exp_element,
    CreateXPathModelPathFromXSEMPath(path, path_end))
12: exp_element.resultAxis ← ls.targetElement.Childaxis
```

Preceding Axis Preceding axis has the same problem with updating as the following axis which depends on the position of the reconnected element. Let x be an element on what is the axis applied. Let R be the set of elements returned by query $x/preceding :: *$ on the original schema and R' the set of elements returned by the query on the changed schema.

Preconditions: $x \in S \wedge x \in S' \wedge p = parent(x) \wedge q = parent(p) \wedge p = parent(r)$

Postconditions: $q = parent(x) \vee r = parent(x)$

Query revalidation

- If the reconnected element is not the element x and the reconnection is done in the part of the tree returned by the following axis, no update is needed.
- If the reconnection of the element y cause, that the element is added into the part of the tree hit by axis when holds $R \subset R'$, the query $x/preceding :: *$ must be update to $x/preceding :: * \; except \; absolute\_path\_to\_reconnected\_element(y)/descendant-or-self :: *$.
- If the element is moved out from the hit part of the tree ($R' \subset R$), the query $x/following :: *$ must be updated to $x/preceding :: * \mid absolute\_path\_to\_reconnected\_element(y)/descendant-or-self :: *$ to preserve $R = R'$.
- If the reconnected element is x, a revalidation depends on the final position of the element x in the tree and among its siblings. Location paths to add

missing elements and locations paths to exclude redundant elements must be added.

The algorithm for update case of preceding axis is the same as Algorithm 6.19.

Preceding-sibling Axis In this case suppose that the axis is applied on element x. Element p be a parent of x and y the reconnected element.

Preconditions: $x \in S \wedge x \in S' \wedge p = parent(x)$
$\wedge (y = sibling(x) \vee y = sibling(p) \vee y = child(r))$

Postconditions: $y = sibling(p) \vee r = parent(y)$
$\vee (y = sibling(x) \wedge pos(x) < pos(y))$

Query revalidation

- First, let y be a sibling of x where $pos(y) < pos(x)$.

  If the element y is moved up as a sibling of p or y is moved down as a child of one of its siblings, the query $x/preceding - sibling :: *$ must be updated to $x/preceding - sibling :: *$
  $| \ absolute\_path\_to\_reconnected\_element(y)$.

- Second, let y be a sibling of p or a child of one of siblings of element x and is reconnected as a sibling of element x where holds $pos(y) < pos(x)$. The query $x/preceding-sibling :: *$ must be updated to $x/preceding-sibling :: * \ except \ absolute\_path\_to\_reconnected\_element(y)$.

A reconnection of element x has the same problem as in case of the preceding axis.

The method for updating preceding-sibling axis is the same as in Algorithm 6.20.

Parent Axis Let p be a parent element of x , q be a parent of element p and r a sibling of element x.

Preconditions: $x \in S \wedge x \in S' \wedge p = parent(x) \wedge q = parent(p)$
$\wedge p = parent(r)$

Postconditions: $q = parent(x) \vee r = parent(x)$

Query revalidation
Reconnection of the element x in both cases (up or down) causes that the query $Q' = x/parent :: *$ will be updated to the form $absolute\_path\_to\_element(p)/self :: * \; except \; absolute\_path\_to\_reconnected\_element(x)/parent :: *$ | $x/parent :: *$ which satisfies $R = R'$.

An update of the XPath model of the query $x/parent :: *$ is shown in Figure 6.2.13. On the left is shown a model of the original query Q, on the right is the updated model of the new query Q'.

The method for updating a location step with parent axis (Algorithm 6.21) is

Algorithm 6.21 UpdateWhenReconnectParentAxis

Input: location step ls, parent XSEM PSM class C^p, reconnected XSEM PSM class C^x

1: $path$ {Varible initialization}
2: $exp\_element$ {Varible initialization}
3: $path \leftarrow GetPathFromRootToClass(C^x)$
4: $exp\_element \leftarrow$ **new** $XPathExpressionelement(\text{"}except\text{"})$
5: $parent \leftarrow ls.parentElement$
6: **new** $XPathAxis(parent, exp\_element)$
7: $exp\_element\_next \leftarrow$ **new** $XPathExpressionElement(\text{"}disjunction\text{"})$
8: $exp\_element.firstAxis \leftarrow$ **new**$XPathAxis(exp\_element, exp\_element\_next)$
9: $exp\_element.secondAxis \leftarrow$ **new** $XPathAxis(exp\_element, CreateXPathModelPathFromXSEMPath(path, path\_end))$
10: $exp\_element.resultAxis \leftarrow ls.targetElement.childAxis$
11: $new\_element \leftarrow$ **new** $XPathElement(ls.targetElement.name)$
12: $CreateMapping(C^x.parent, new\_element)$
13: **new** $XPathParentAxis(path\_end, new\_element)$
14: $exp\_element\_next.firstAxis \leftarrow ls$
15: $path \leftarrow GetPathFromRootToClass(C^p)$
16: $exp\_element.secondAxis \leftarrow$ **new** $XPathAxis(exp\_element\_next, CreateXPathModelPathFromXSEMPath(path, path\_end))$
17: $new\_element \leftarrow$ **new** $XPathElement(ls.targetElement.name)$ {Create new axis between elements}
18: $CreateMappings(C^p, new\_element)$
19: **new** $XPathSelfAxis(path\_element, new\_element)$

the same in both cases of possible move directions. First, element E^1_{ex} of *except* operator is created (line 4). Then an element E^2_{ex} (with operator *disjunction*) is created and connected as the first child of E^1_{ex} (line 7–8). The original location step is set as the first child of element E^2_{ex} (line 14). The second child of E^2_{ex} is a path to the class C^p (line 17). Finally, the second child of E^1_{ex} is a path to the

parent of the reconnected class C^x.

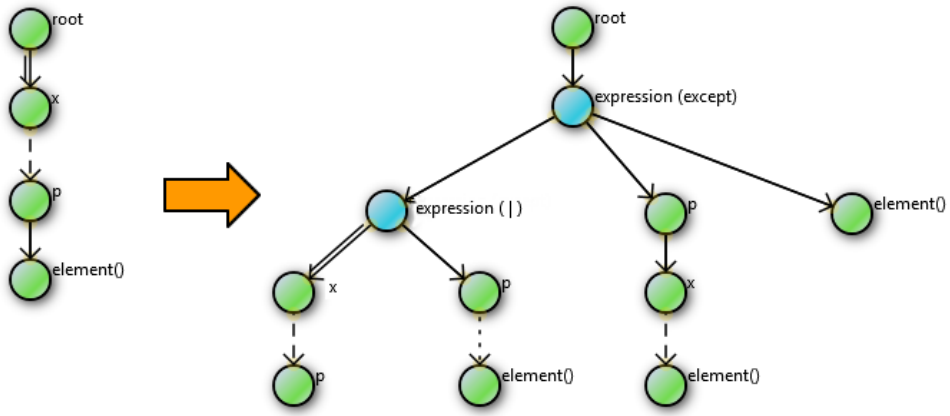


Figure 6.2.13: Reconnection of the parent axis

Self Axis A change of the position in the tree does not change the result of the query of the self axis. The original result R and the new result R' are still the same.

Chapter 7

Implementation and Experiments

In this chapter we will describe a prototype implementation of the algorithms described in Chapter 6. It was implemented as an extension of DaemonX, a framework for modeling and processing evolution [21].

7.1 DaemonX

DaemonX is a framework developed as a software project on the Faculty of Mathematics and Physics, of the Charles University in Prague [21]. It is a pluginable tool developed for data and/or process modeling. The main aim of the framework is to provide functionality for processing evolution between models defined by a developer. The functionality is provided via various plug-ins, which use services provided by the framework. All what have to be done by developer of additional plug-ins is to define models and set rules how the atomic operations created in a source model have to be propagated to operations in a target model. A detailed description of the framework can be found on project web pages [21].

7.2 Implementation

A prototype implementation uses existing XSEM PSM modeling plug-in developed as a part of the first release of the project and adds two new plug-ins. The first modeling plug-in for XPath query and the second plug-in for evolution processing from XSEM PSM model to XPath model. The experimental implementation has the following features:

- Modeling plug-in for visualization of XPath query
  - It implements all components necessary to create and visualize XPath query - nodes and axes. It is described in Section 5.2.2 and user documentation of the plug-in is available on the attached CD.

  - it provides generating query from the model.
  - It provides generating model from the query.

- Evolution plug-in from XSEM PSM to XPath modeling plug-ins

  - It enables creation of mapping between XSEM PSM and XPath models.
  - It provides ability of propagation of operations from the source to the target model.
  - It implements parsing of XPath query (thanks to the Saxon XQuery Processor, HE [26]), generating XPath model from the query and creating mapping with XSEM PSM model.
  - It implements algorithms described in Chapter 6.

The prototype implementation generates a model from given XPath query and creates mapping between given XSEM PSM model and the generated model. Then, changes done in XSEM PSM model by a user are propagated to the XPath model and new query is generated from the original one.

7.3 Experiments

Because there are no existing real-world project that provides similar abilities, it is not possible to compare our solution and results with others. Therefore queries from XPathMark XPath-TF [27] were used to provide the proof of the concept. Because these test sets are quite simple and do not use more locations steps together, we created own more complicated queries using various axes to test abilities of the solution.

7.3.1 XPathMark XPath-TF

From the presented test set we took tests corresponding with our XPath syntax:

1. A1 - A11
2. P1 - P11 which were rewritten to queries without predicate
3. O1, O3, O4

All these queries were applied on test document schema and then were applied all possible changes described in Chapter 6. All updates were executed correctly in all cases.

Examples of the Queries

- $//l/ancestor :: *$ (A5)
- $//l/following :: *$ (A9)
- $//l/descendant :: *$ (P5)
- $//l/preceding-sibling :: *$ (P7)
- $//q/following :: */parent :: * \ except \ //g/ancestor :: *$ (O1)

7.3.2 Sophisticated Queries

To test the approach on more complicated queries from the real word, we took XML schema of an order from Amazon AWS [28] used for communication with customers by Web Services. XSEM PSM model was created from this schema and a set of XPath queries utilizing all available axes in various combinations were defined. These queries were automatically mapped to the schema by the DaemonX framework.
Next, we made various changes in the schema to simulate a designer who has to change the schema for different reason like to add new information, remove unused parts, rename parts to make better sense or change the structure of the schema tree. After propagation the results of both original and new queries were checked if they are the same if there were found no limitations in the evolution process. All queries and applied changes of the schema can be found on the attached CD.

Examples of the Queries

- $//RegionDefinition/parent :: ExcludedRegions/parent :: *$
- $/Order/ParameterizedUrls/*/*$
- $//AmazonUpsellPreferences/child :: *$
- $//ShippingRate/following-sibling :: */descendant :: *$
- $//MerchantUpsellItem/Images/preceding-sibling :: *$
- $//ShippingMethods/following :: *$
- $//RegionDefinition/ancestor :: *$
- $//Taxamount/following :: Shipping/child :: *$
- $//Images/parent :: */ItemCustomDate/ancestor :: Cart$

Chapter 8

Conclusion

In this thesis we presented an approach to XML schema evolution and query adaptation. According to changes performed in the schema it should determine the impact on the queries and update them in order to return the same results like the original queries with the original schema.
We began with introducing of problem with updating schemes and possible incompatibility of these schemes with related queries (see Chapter 1). In Chapter 2 we introduced an XML evolution architecture. We focused on the platform-specific level which is utilized in this thesis.
Chapter 3 contains analysis of various existing approaches which are concerned to queries incompatibility while evolving XML schema changes.
In Chapter 4 is presented XSEM, a conceptual model for XML, especially its platform-specific model XSEM-H utilized in this thesis.
Chapter 5 describes used XPath syntax based on Positive Core XPath and its visualization model - a platform specific model which represents the query. Next operations which can be applied on the model are described. Then mapping between components of XSEM PSM and XPath models used in evolution process is described. The chapter ends with description of the algorithm for recognizing of changes that were done in original XSEM PSM model and a decision if it should be propagated to related XPath model.
In Chapter 6 the main contribution of this thesis is described – the analysis of the possible changes which can be done in XSEM PSM model and their impact on the XPath model mapped to this model. We proposed algorithms performing needed changes which have to be done with the queries to preserve compatibility. The analysis is divided into the sections by the possible changes. Then every section is divided by axis type where can be different algorithms how the query (especially XPath model of the query) should be updated if it is possible.
The evolution process between the models can be summarized into these steps:

1. Analyze the change done in the XSEM PSM model.

2. Evaluate results of the queries with the original and the new XSEM PSM model.

3. If there are any differences (different mapping), perform evolution operation on the XPath model if possible.

An experimental implementation of the algorithms was implemented as an extension of DaemonX framework and is available on the attached CD. The implementation and experiments are described in Chapter 7.

Main Contributions

The main contribution of our approach is the ability to recognize and to analyze changes done in XML schema and the ability to update related queries to preserve the compatibility. It enables to design XML schemes without an inspection of all related queries and looking for incompatibility of the queries after every change manually. If the revalidation of the query is not possible, this situation is reported to the designer to update the query manually or to restore the changes in the schema. Changes in the schema are propagated immediately thanks to the interconnection between the schema and the queries.

8.1 Open Problems

Even though the approach is complex and robust, there exists some problems and issues that were found during the survey and are not covered by this thesis.

8.1.1 Suggestion When Propagation Is Impossible

There are some cases when the propagation of the change could not be proceeded. For example in Section 6.2.5 if the changed location step is not the last in the query. In this situation the designer must be informed about this situation and the query and mapping have to be updated manually. A similar problem can arise with elements with the same name - the results depend on the semantics. In this case an analyzer could provide some clue to the designer to make the query update more simple or help with choosing the appropriate one.

8.1.2 Query Optimization

After changes are done in the XSEM PSM model the related XPath model can be in an unoptimized form. For example repeating of inverse operations like adding and removing the same XSEM PSM class can cause redundant location steps in the XPath model. This problem can be solved by using an additional query optimization that can be performed after the propagation process. This optimization will create new optimized query (and XPath model) which will be mapped to the XSEM PSM model.

8.2 Future Work

8.2.1 Richer XPath Syntax

The XPath syntax based on presented Positive Core XPath is quite simple. It does not cover predicates, attributes or built-in XPath functions. It seems that enlargement of the syntax only expands the analysis of the changes done in the XSEM PSM model and theirs propagation to the XPath model and makes the propagation more complicated.

8.2.2 Semantic Relations

As mentioned in Chapter 6, we do not consider semantics of the XSEM PSM model and the XPath model. This brings some limitation in an evaluation process and a propagation of the changes. For example two sibling XSEM PSM classes of the same name are not permitted in the presented suggestion and the evaluation process could not be proceed in this situation.
For example if there are two classes *Address* representing *home address* and *delivery address*. There is no possibility how to distinguish between these two XSEM PSM classes with the used XPath syntax. A possible solution of this problem would be a special annotation of the XSEM PSM classes and the XPath model when the mapping between these classes and elements from the XPath model is being created.

Appendix A

CD Contents

The attached CD contains:

- PDF version of the thesis - *thesis.pdf.*
- Installer of DaemonX framework with appropriate plug-ins in the folder *implementation.*
- Examples of XSEM PSM and XPath models for evolution process in the folder *examples.*

Appendix B

Used XSD Schemes and XPath Queries

B.1 Purchase Schema

File with this schema whose XSEM PSM model is shown in Figure 4.1.2 is saved on the attached CD in the file *examples/purchase/purchase.xsd* and a project with this schema in the file *examples/purchase/purchase.dx*.

B.2 XPathMark

A test schema of the XPathMark is stored in the file *examples/xpathmark/schema.xsd* and a project used for testing in the file *examples/xpathmark/xpathmark.dx*. All used queries are in the file *examples/xpathmark/queries.pdf*.

B.3 Order Schema

The schema of Amazon order is in the folder *examples/order/order.xsd* on the attached CD. Next, queries applied on this schema are strored in the file *examples/order/queries*.pdf. In the folder *examples/order* is the file *order.dx* with the saved project for the DaemonX too. This project contains an XSEM PSM model of the schema.

Bibliography

[1] W3C. Extensible markup language (xml). http://www.w3.org/XML/, 09 2010.

[2] W3C. Xml path language (xpath) version 1.00. http://www.w3.org/TR/1999/REC-xpath-19991116/, 11 1999.

[3] W3C. Xquery 1.0: An xml query language (second edition). http://www.w3.org/TR/xquery/, 12 2010.

[4] Mirella M. Moro, Susan Malaika, and Lipyeow Lim. Preserving xml queries during schema evolution. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 1341–1342, New York, NY, USA, 2007. ACM.

[5] W3C. Semantic annotations for wsdl and xml schema. http://www.w3.org/TR/sawsdl/, 08 2007.

[6] Martin Necasky and Irena Mlynkova. Five-level multi-application schema evolution. *DATESO*, 1:90–104, 2009.

[7] Martin Necasky. Xsem: a conceptual model for xml. In *Proceedings of the fourth Asia-Pacific conference on Comceptual modelling - Volume 67*, APCCM '07, pages 37–48, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc.

[8] W3C. Xml schema part 1: Structures second edition. http://www.w3.org/TR/xmlschema-1/, 10 2010.

[9] W3C. Xml path language (xpath) 2.0 (second edition). http://www.w3.org/TR/xpath20/, 12 2010.

[10] W3C. Xsl transformations (xslt) version 1.0. http://www.w3.org/TR/xslt, 11 1999.

[11] W3C. Xhtl 1.0 the extensible hypertext markup language (second edition). http://www.w3.org/TR/xhtml1/dtds.html, 08 2001.

[12] World Wide Web Consortium. http://www.w3.org/.

[13] Meike Klettke. Conceptual xml schema evolution - the codex approach for design and redesign. In *BTW Workshops*, pages 53–63, 2007.

[14] Pierre Genevès, Nabil Layaïda, and Vincent Quint. Identifying query incompatibilities with evolving xml schemas. *SIGPLAN Not.*, 44:221–230, August 2009.

[15] W3C. Relax ng. `http://relaxng.org/`, 01 2011.

[16] Pierre Genevès, Nabil Layaïda, and Alan Schmitt. Efficient static analysis of xml paths and types. *SIGPLAN Not.*, 42:342–351, June 2007.

[17] Egon Wanke and Rolf Kotter. Oriented paths in mixed graphs. In Rudolf Fleischer and Gerhard Trippen, editors, *Algorithms and Computation*, volume 3341 of *Lecture Notes in Computer Science*, pages 1467–1490. Springer Berlin / Heidelberg, 2005. 10.1007.

[18] Roger L. Costello. Global versus local. `http://www.xfront.com/GlobalVersusLocal.html`.

[19] XCase team. Xcase - tool for xml data modeling. `http://xcase.codeplex.com/`.

[20] OMG. Documents associated with uml version 2.2. `http://www.omg.org/spec/UML/2.2/`, 02 2009.

[21] DaemonX Team. Daemonx. `http://daemonx.codeplex.com/`, 6 2011.

[22] Jakub Klimek, M. Necasky, and Irena Mlynkova. Evolution and change management of xml applications. *none*, 1:0–0, 2011.

[23] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Xpath processing in a nutshell. *SIGMOD Rec.*, 32:21–27, June 2003.

[24] Pieter H. Hartel. A trace semantics for positive core xpath. In *Proceedings of the 12th International Symposium on Temporal Representation and Reasoning*, pages 103–112, Washington, DC, USA, 2005. IEEE Computer Society.

[25] Balder ten Cate and Maarten Marx. Axiomatizing the logical core of xpath 2.0. *Theor. Comp. Sys.*, 44:561–589, April 2009.

[26] Michael Kay. Saxon the xslt and xquery processor. `http://saxon.sourceforge.net/`, 5 2011.

[27] Massimo Franceschet. `http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark/FT.html`.

[28] Amazon. Amazon web services. `http://amazonpayments.s3.amazonaws.com/documents/order.xsd`.