Implementation of an XPath-Rewriter and Visualizer Projektarbeit

Hatice Serap Durmaz
Betreuer: Prof. Dr. François Bry
Dr. Dan Alexandru Olteanu

13. March 2006

Zusammenfassung

Jüngste Forschungen versuchen Algorithmen für herkömmliche Datenstrukturen auch für Datenströme zur Verfügung zu stellen. Diese treten in der Datenverarbeitung immer häufiger auf, und bringen somit immer neue Problemstellungen mit sich. Die vorliegende Arbeit beschäftigt sich mit der Lösung eines speziellen Problems, nämlich der Auswertung von XPath-Anfragen auf unbegrenzt lange XML-Ströme. Hierfür ist es zweckmäßig, die XPath-Anfrage so umzuschreiben, dass sie keine Rückwärtsachsen mehr enthält, was eine effiziente Auswertung in einem einzigen Durchlauf ermöglicht, ohne große Datenmengen speichern zu müssen. In [Olt1] und [Olt2] wurde bewiesen, dass dies für jede Anfrage möglich ist. Diese Arbeit implementiert einen solchen XPath-Umschreiber und zugleich einen Visualisierer, der die Umschreibung veranschaulicht.

Abstract

Recent research tries to solve problems that have been solved for usual datastructures, also for datastreams. These are occurring in data processing continously and thus involve more and more problems. This work deals with the solution for a particular problem, the evaluation of XPath queries on unbounded XML streams. For this purpose it is convenient to rewrite the XPath query, so that it has no reverse axes anymore, which allows an efficient single pass evaluation of the query without the necessity of storing large amount of data. [Olt1] and [Olt2] prove that this is possible for every query. In this work such an XPath Rewriter and also an appropriate Visualizer for demonstrating the rewriting are implemented.

Erklärung

Hiermit versichere ich, dass ich diese Projektarbeit selbständig verfasst und dabei keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 13. März 2006

Hatice Serap Durmaz

Acknowledgments

I thank Tim Furche for proof-reading an earlier draft of this paper.

Contents

1 Introduction **4**

2 Preliminaries **5**
2.1 XPath Fragment 5
2.2 Rewriting Rules 6

3 Implementation **10**
3.1 Abstract Representation of XPath 10
3.1.1 XPath Parser 11
3.1.2 Class Structure 11
3.1.3 Design pattern Visitor 19
3.1.4 XPath Properties 20
3.2 The Rewriter Core 21
3.3 Visualization 22
3.3.1 Graph representation of XPath queries 22
3.3.2 Graphical User Interface 25

4 Conclusion **26**
4.1 Integration into the SPEX project 26
4.2 Installation and usage 26

1 Introduction

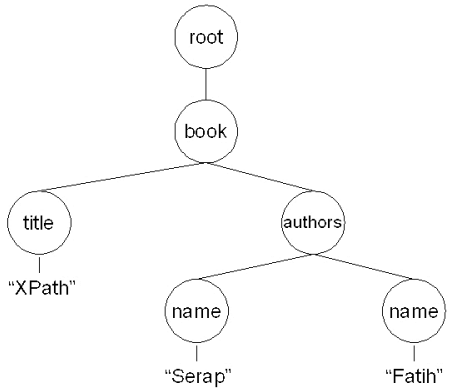
Query languages for XML rely on location paths for selecting nodes in data items. XPath takes a navigational approach for specifying the nodes to be selected. The random access to XML data that is enabled by the various navigational axes of XPath has proven particularly difficult for an efficient stream-based processing of XPath queries. There is a great interest in the identification of a subset of XPath that allows efficient progressive or stream-based processing. Of particular concern for progressive processing are the reverse axes of XPath, i.e. those navigational axes (e.g. parent, preceding) that select nodes occurring before the context node in document order. There are several principal options how to evaluate reverse axes in a stream-based context. One option is storing in memory sufficient information that allows to access past events when evaluating a reverse axis. This approach has the disadvantage that, in worst case, the whole document must be stored, what is not feasible for data-centric applications. Another option is the evaluation of an XPath expression in more than one run, storing additional information to be used in successive runs. Although the information to be stored may be much smaller, also this approach is not feasible for all application scenarios. This paper is dealing with the implementation of a third approach: Replacing XPath expressions by equivalent ones without reverse axes. [Olt2] shows that the third approach is possible. The evaluation of such XPath expressions (without reverse axes) is less time consuming than the second approach and less memory consuming than the first one.

One of the concerns of this paper is the implementation of such an XPath Rewriter. The implemented rewriter application replaces all reverse axes and substitutes many subexpressions by more efficient ones. This rewriter operates on a fragment of the XPath language, which will be described more precisely in Section 2.1. To simplify matters, the word *XPath* is used throughout this paper, although the implementation works for a strict fragment of XPath only. In addition to the rewriter application a visualizer, which illustrates the complete rewriting, is implemented and described in Section 3.3.

2 Preliminaries

The XPath Rewriter application uses rewriting rules introduced in [Olt2] and described in this section also. They apply on a fragment of XPath, which also is described in the following. The mathematical model for paths and equivalences used in this section is adapted from [Olt2], which contains the full formal model as well as the denotational semantics also. Some familiarity with XML and XPath 1.0 is assumed.

Within this section examples will be shown for demonstrating some of the rewriting rules. For this purpose XML documents will be shown as a tree graph representing the XML document. In this tree the root node corresponds to the document node of DOM, hence it is none of the document elements. A leaf is an empty element or a text node. Figure 1 shows an example.




```
<book>
    <title>XPath</title>
    <authors>
        <name>Serap</name>
        <name>Fatih</name>
    </authors>
</book>
```

Figure 1: Tree and XML data it represents

2.1 XPath Fragment

The rewriter is capable of rewriting queries described in a language similar to XPath [XPath]. This language is a fragment of XPath 1.0 and defined by the following grammar.

NodeSelection	::=	'(' NodeSelection ')' | Union | Path | Intersect | Except | $\perp$.
Union	::=	NodeSelection '|' NodeSelection.
Intersect	::=	NodeSelection 'except' NodeSelection.
Except	::=	NodeSelection 'intersect' NodeSelection.
Path	::=	('/')? Step ('/' Step)*.
Step	::=	Axis '::' NodeTest (Predicate)*.
Axis	::=	<ALL_XPATH_AXES>.
NodeTest	::=	'*' | 'node()' | 'text()' | <LABEL>.
Predicate	::=	'[' BooleanExpression ']'.
BooleanExpression	::=	AndExpression | OrExpression | NotExpression | Comparison | Path.
AndExpression	::=	BooleanExpression 'and' BooleanExpression.
OrExpression	::=	BooleanExpression 'or' BooleanExpression.
NotExpression	::=	not '(' BooleanExpression ')'.
Comparison	::=	Path CompareRel Constant | 'contains' (Path ',' Constant ')'.
CompareRel	::=	'==' | '=' | '<' | '<=' | '>' | '>=' | '!='.
Constant	::=	<NUMBER> | <LITERAL>.

This XPath fragment does not contain abbreviated expressions, as XPath 1.0 does. Furthermore the use of functions in this language is restricted to the functions *not* and *contains*. Other functions can be added to this language with simple changes to the rewriter. The only class of functions that needs special treatment are functions for accessing the context position or size of a node.

$\perp$ is the only expression not included in XPath. In fact neither the rewriter does allow input of queries that contain $\perp$. Though $\perp$ is mentioned here because it can be the result of a rewriting. It denotes a canonical equivalent path to the XPath expressions that select no nodes whatever the context node and document are, e.g. /parent::*.

The XPath fragment includes ==, which corresponds to built-in node equality operator (==) in XPath 2.0. As XPath 1.0 has built-in support for equality based on node values only (=), the XPath 1.0 expression count(p1 | p2) $<$ count(p1) $+$ count(p2) can be used for expressing ==.

Figure 2 demonstrates the selection of a node using the XPath expression */descendant::author/parent::book*.

2.2 Rewriting Rules

Two location paths p and q are equivalent (denoted $p \equiv q$) if they select the same set of nodes for every document and every context node in this document. The rewriter application uses rewriting rules introduced in [Olt2] which are described in the following. These rules can be used to rewrite a small part of a query p, which yields in query p'. Successive application of rewriting rules finally results in a query q, which is equivalent to p and does not contain any reverse axes. [Olt2] proves that every XPath query can be rewritten this way until it does not contain any reverse

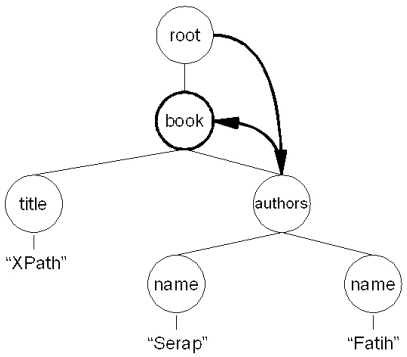


Figure 2: Selection of node by an XPath expression

axes anymore. [Olt2] further proves that this rule system is complete, confluent and equivalence keeping.

Each rule treats the interaction of the reverse axes (ancestor, ancestor-or-self, parent, preceding, and preceding-sibling) with forward axes. A rewriting rule basically is an equivalence that either replaces the reverse location step or rewrites the location path into one, where the reverse step is "pushed leftwise". The equivalences have the structure $p/L_f/L_r \equiv p'$ or $p/L_f[L_r] \equiv p'$, where p is an absolute path (which may be omitted), L_f a forward location step, L_r a reverse location step, and p' the equivalent location path.

Parent

These rules consider rewriting of path expressions, which contain the reverse axis *parent*. In these rules m and n are node tests and p is a path.

descendant::n/parent::m	≡	descendant-or-self::m[child::n]
child::n/parent::m	≡	self::m[child::n]
p/self::n/parent::m	≡	p[self::n]/parent::m
p/following-sibling::n/parent::m	≡	p[following-sibling::n]/parent::m
p/following::n/parent::m	≡	p/following::m[child::n]
		| p/ancestor-or-self::*[following-sibling::n]
		/parent::m
descendant::n [parent::m]	≡	descendant-or-self::m/child::n
child::n[parent::m]	≡	self::m/child::n
p/self::n[parent::m]	≡	p[parent::m]/self::n
p/following-sibling::n[parent::m]	≡	p[parent::m]/following-sibling::n
p/following::n[parent::m]	≡	p/following::m/child::n
		| p/ancestor-or-self::*[parent::m]
		/following-sibling::n

Ancestor

These rules consider rewriting of path expressions, which contain the reverse axis *ancestor*. In these rules m and n are node tests and p is a path.

p/descendant::n/ancestor::m	≡	p[descendant::n]/ancestor::m | p/descendant-or-self::m[descendant::n]
/descendant::n/ancestor::m	≡	/descendant-or-self::m[descendant::n]
p/child::n/ancestor::m	≡	p[child::n]/ancestor-or-self::m
p/self::n/ancestor::m	≡	p[self::n]/ancestor::m
p/following-sibling::n/ancestor::m	≡	p[following-sibling::n]/ancestor::m
p/following::n/ancestor::m	≡	p/following::m[descendant::n] | p/ancestor-or-self::* [following-sibling::*/descendant-or-self::n] /ancestor::m
p/descendant::n[ancestor::m]	≡	p[ancestor::m]/descendant::n | p/descendant-or-self::m/descendant::n
/descendant::n[ancestor::m]	≡	/descendant-or-self::m/descendant::n
p/child::n[ancestor::m]	≡	p[ancestor-or-self::m]/child::n
p/self::n[ancestor::m]	≡	p[ancestor::m]/self::n
p/following-sibling::n[ancestor::m]	≡	p[ancestor::m]/following-sibling::n
p/following::n[ancestor::m]	≡	p/following::m/descendant::n | p/ancestor-or-self::*[ancestor::m] /following-sibling::*/descendant-or-self::n

Preceding-Sibling

These rules consider rewriting of path expressions, which contain the reverse axis *preceding-sibling*. In these rules m and n are node tests and p is a path.

descendant::n/preceding-sibling::m	≡	descendant::m[following-sibling::n]
child::n/preceding-sibling::m	≡	child::m[following-sibling::n]
p/self::n/preceding-sibling::m	≡	p[self::n]/preceding-sibling::m
p/following-sibling::n/preceding-sibling::m	≡	p[self::m/following-sibling::n] | p[following-sibling::n]/preceding-sibling::m | p/following-sibling::m[following-sibling::n]
p/following::n/preceding-sibling::m	≡	p/following::m[following-sibling::n] | p/ancestor-or-self::*[following-sibling::n] /preceding-sibling::m | p/ancestor-or-self::m[following-sibling::n]
descendant::n[preceding-sibling::m]	≡	descendant::m/following-sibling::n
child::n[preceding-sibling::m]	≡	child::m/following-sibling::n
p/self::n[preceding-sibling::m]	≡	p[self::n]/following-sibling::m
p/following-sibling::n[preceding-sibling::m]	≡	p[self::m]/following-sibling::n | p/following-sibling::m/following-sibling::n | p[preceding-sibling::m]/following-sibling::n
p/following::n[preceding-sibling::m]	≡	p/following::m/following-sibling::n | p/ancestor-or-self::*[preceding-sibling::m] /following-sibling::n | p/ancestor-or-self::/following-sibling::n

Preceding

These rules consider rewriting of path expressions, which contain the reverse axis *preceding*. In these rules m and n are node tests and p is a path.

p/descendant::n/preceding::m	≡	p[descendant::n]/preceding::m | p/child::* [following-sibling::*/descendant-or-self::n] /descendant-or-self::m
/descendant::n/preceding::m	≡	/descendant::m[following::n]
p/child::n/preceding::m	≡	p[child::n]/preceding::m | p/child::*[following-sibling::n] /descendant-or-self::m
p/self::n/preceding::m	≡	p[self::n]/preceding::m
p/following-sibling::n/preceding::m	≡	p[following-sibling::n]/preceding::m | p/following-sibling::*[following-sibling::n] /descendant-or-self::m | p[following-sibling::n]/descendant-or-self::m
p/following::n/preceding::m	≡	p[following::n]/preceding::m | p/following::m[following::n] | p[following::n]/descendant-or-self::m
p/descendant::n[preceding::m]	≡	p[preceding::m]/descendant::n | p/child::*[descendant-or-self::m] /following-sibling::*/descendant-or-self::n
/descendant::n[preceding::m]	≡	/descendant::m/following::n
p/child::n[preceding::m]	≡	p[preceding::m]/child::n | p/child::*[descendant-or-self::m] /following-sibling::n
p/self::n[preceding::m]	≡	p[preceding::m]/self::n
p/following-sibling::n[preceding::m]	≡	p[preceding::m]/following-sibling::n | p/following-sibling::*[descendant-or-self::m] /following-sibling::n | p[descendant-or-self::m]/following-sibling::n
p/following::n[preceding::m]	≡	p[preceding::m]/following::n | p/following::m/following::n | p[descendant-or-self::m]/following::n

Example

Our example demonstrates the rewriting of the XPath query

/child::x/parent::y/child::z/parent::u,

which contains reverse axes. The rewriter looks for an appropriate rule and finds

child::n/parent::m ≡ self::m[child::n].

After applying this rule, the rewriter returns the intermediate result

/self::y[child::x]/child::z/parent::u

After another step of rewriting the empty set will be returned as the final result. This means that the original query is not selecting any nodes also. If one looks closer to the original query this is clear. Figure 3 demonstrates the several steps for this rewriting. The figure uses a graph representation of XPath, which is described in Section 3.3.1.

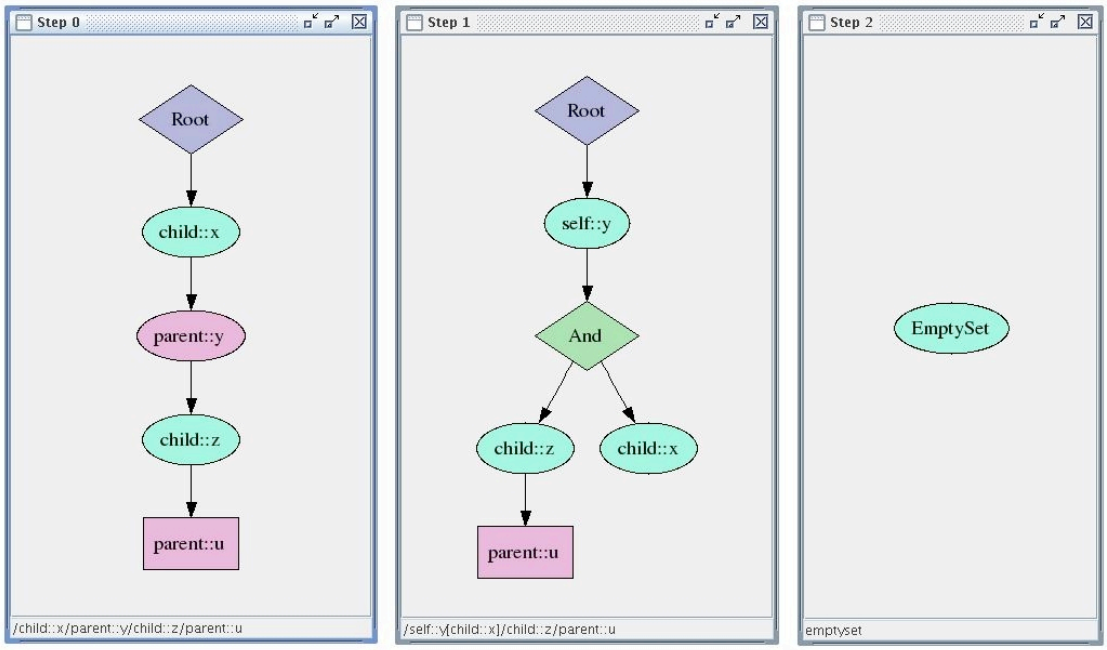


Figure 3: After several steps of rewriting the empty set is the result

3 Implementation

The rewriting application shall be implemented in a modern, platform independent and object oriented programming language. For this purpose Java 5.0 was chosen, which fulfills all these requirements. The *XPath-Rewriter* is a part of the SPEX project at Sourceforge.net [SF], and all its classes are two modules called *ReXP* and *ReXP-GUI*. This section describes all parts of the implementation, beginning with the parser, the abstract class structure, the rewriting itself, and the visualization.

3.1 Abstract Representation of XPath

Each XPath expression is translated into an abstract form. A class structure for this abstract syntax must be developed, which then can be used by an appropriate XPath parser. This parser and the grammar for the used XPath fragment are described in Preliminaries, and after that the class structure for the abstract syntax will be explained.

3.1.1 XPath Parser

The XPath expression has to be translated into abstract syntax before the rewriter can start. This job is done by a parser. There are no special requirements for the parser, but it should work as efficient as the used grammar allows. Because the used XPath fragment can be described by an LL(1) grammar, the parser can be implemented efficiently as an LL(1) parser. The parser will be created using the modern parser generator for Java [JavaCC], which is able to generate an LL(k) parser for any lookahead k. In our case an LL(1) parser is generated by JavaCC.

JavaCC's input is a so called attributed grammar which contains rules for generating the abstract syntax tree in addition to production rules for the cognition of words of the language. The output of the parser generator consists of several Java classes, which will not be specified in more detail in this work. Together all these classes represent the parser that is used to parse the XPath queries and translate them into an abstract form.

LL(1) Grammar

The grammar for the used XPath fragment as shown in Section 2.1 is human readable but not suitable for an LL(1) parser. For this reason another grammar is introduced, which has the property of being an LL(1) grammar and which is equivalent to the first grammar. The input for the JavaCC parser generator is similar to the following grammar.

```
NodeSelection ::= UnionExp.
UnionExp      ::= IntExcExpression ('|' UnionExpression)?.
IntExcExp     ::= ElemNodeSel (('intersect' | 'except') IntExcExp)?.
ElemNodeSel   ::= Path | '(' NodeSelection ')'.
Path          ::= ('/')? Step ('/' Step)*.
Step          ::= Axis '::' NodeTest (Predicate)*.
Axis          ::= <ALL_XPATH_AXES>.
NodeTest      ::= '*' | 'node()' | 'text()' | <LABEL>.
Predicate     ::= '[' BoolExp ']'.
BoolExp       ::= OrExp.
OrExp         ::= AndExp ('or' OrExp)?.
AndExp        ::= NotExp ('and' AndExp)?.
NotExp        ::= ('not' '(' BoolExp ')')
                  | (Path (CompOP Constant)?)
                  | 'contains' '(' Path ',' Value ')'
                  | '(' BoolExp ')'.
CompOP        ::= '==' | '='| '<' | '<=' | '>' | '>=' | '!='.
Constant      ::= <NUMBER> | <LITERAL>.
```

3.1.2 Class Structure

The abstract syntax tree generated by the parser is based on some Java classes developed especially for this project. To describe the XPath queries as exact as pos-

sible, these classes represent a hierarchy similar to the structure of XPath queries. The topmost level is the class *Expression* that is extended by all other classes. Beneath, there are classes like *NodeSelection* or *BooleanExpression*. *NodeSelection* here reflects all XPath queries, which select a nodeset, and the *BooleanExpression* represent all XPath expressions that can occur inside XPath predicates. The class *Path* represents all path expressions within an XPath query. It consists of a list of *Step* objects, which in turn may contain a list of *Predicate* objects. Here *Step* objects represent steps in a path expression. There are also some self-explanatory classes like *Union*, *Intersect*, *AndExpression* and so on. Figure 4 shows the dependencies between classes.

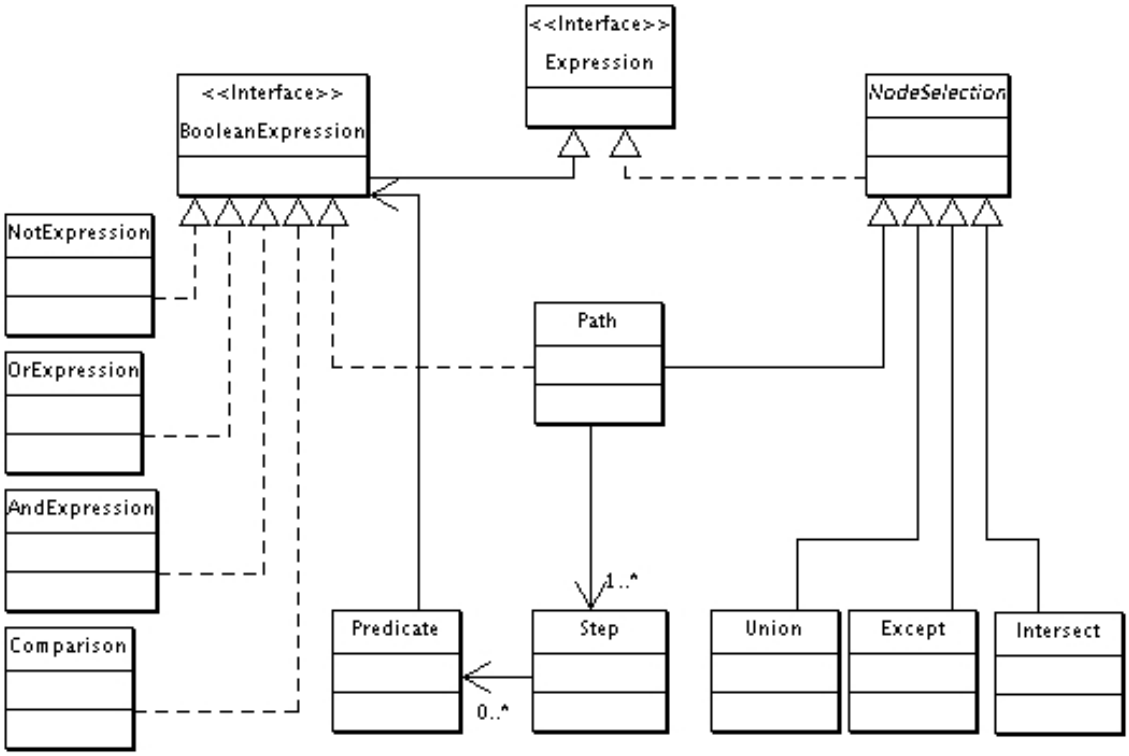


Figure 4: Class diagram for the abstract representation

The figure is far from complete, but it shows the most important information about the dependencies of classes. In the following, the most important of these classes will be described more precisely.

Interface Expression

This interface is implemented by each XPath expression. It specifies methods, which are invokable on every XPath expression.

Method hasBackwardAxis()

Checks whether this XPath expression contains any reverse axes and returns the result as a boolean value.

Method accept(XPathVisitor)

Accepts a so called vistor for this XPath expression. The *Visitor* design pattern allows the separation of algorithms and the traversing of the abstract syntax tree. The vistor is explained more precisely in Section 3.1.3.

Method clone()

The Expression interface specifies that this method has to be implemented in every implementing class in such a way that a deep copy of the object is generated and returned.

Method toString()

The Expression interface specifies that this method has to be implemented in every implementing class in such a way that the string representation of the XPath expression is returned for the corresponding object.

Abstract class NodeSelection

This class represents all XPath expressions whose result is a set of nodes. For example path expressions and unions of path expressions belong to this type of expression.

Method addStep(Step)

Adds a step to this NodeSelection. In path expressions the step is simply added to the path. For other expressions like unions this may not be as trivial. In general the following condition must be satisfied by this method: the new NodeSelection must select all nodes, which would have been selected by going the specified step from all result nodes of the previous NodeSelection.

Method NodeSelection.addSteps(LinkedList)

Similar to the previous method but adds a list of steps.

Method addPredicate(Predicate)

Adds a predicate to this NodeSelection object. The new NodeSelection selects all the nodes, which satisfy the specified predicate and which have been selected by the old NodeSelection.

Class Union

This subclass of NodeSelection represents unions of several other NodeSelection objects. Hence this class represents XPath's union function.

Constructor Union(NodeSelection[])

This constructor creates a new Union instance by unifying the the specified NodeSelection instances.

Method getNodeSelections()

Returns an array of node selections. This array will contain all subsets of this union.

Class Except

Another subclass of NodeSelection. This class represents set operations like $X \backslash Y$ for two sets X and Y. Hence Except is the representation of XPath's except function.

Constructor Except(NodeSelection, NodeSelection)

This constructor creates a new Except instance, using the two specified NodeSelection arguments. This object will represent the set difference of these two arguments.

Method getFirstOperand()

Returns the first operand of this Except instance. That is X in the set operation $X \backslash Y$.

Method getFirstOperand()

Returns the second operand of this Except instance. That is Y in the set operation $X \backslash Y$.

Class Intersect

Similar to the previous NodeSelection also this one represents a set operation, more specifically the intersection of two sets. Hence Intersect is the representation of the intersect function in XPath.

Constructor Intersect(NodeSelection, NodeSelection)

This constructor creates a new Intersect instance, using the two specified NodeSelection arguments. This object will represent the intersection of these two sets.

Method getFirstOperand()

Returns the first operand of this Intersect instance. That is X in the set operation $X \; intersect \; Y$.

Method getFirstOperand()

Returns the second operand of this Intersect instance. That is Y in the set operation $X \; intersect \; Y$.

Interface BooleanExpression

XPath Expressions implementing this interface are capable of being the content of a XPath predicate. Examples for such expressions are paths and logical operations. This interface does not specify any methods. Its only purpose is flagging expressions, which can occur in predicates.

Class NotExpression

This class is a BooleanExpression and hence may occur in predicates. It complements the boolean value of another BooleanExpression, what is not trivial in all cases. For example because path expressions can occur in predicates they are BooleanExpressions also. The negation of such path expressions is allowed in XPath.

Constructorr NotExpression(BooleanExpression)

This constructor creates a new NotExpression object. It represents the complement of the specified BooleanExpression.

Method getOperand()

Returns the operand of this NotExpression, which is complemented by this object. So it returns X in $not(X)$.

Class OrExpression

Another class, which implements BooleanExpression. It connects two more BooleanExpression with a boolean disjunction. It should be recognized that any BooleanExpressions can be connected with this class. For example it is permitted to connect two path expressions with a boolean disjunction.

Constructor OrExpression(BooleanExpression, BooleanExpression)

This constructor creates a new OrExpression object. It represents a boolean disjunction of the specified parameters.

Method getFirstOperand()

Returns the first partial expression in this disjunction. So it returns X in $X\ or\ Y$.

Method getSecondOperand()

Returns the second partial expression in this disjunction. So it returns Y in $X\ or\ Y$.

Class AndExpression

Similar to OrExpression this class also implements BooleanExpression. This class connects two BooleanExpression with a boolean conjunction. It is permitted to connect any BoooleanExpressions, for example path expressions, with a boolean conjunction.

Constructor AndExpression(BooleanExpression, BooleanExpression)

This constructor creates a new AnExpression object. It represents a boolean conjunction of the specified parameters.

Method getFirstOperand()

Returns the first partial expression in this conjunction. So it returns X in $X\ and\ Y$.

Method getSecondOperand()

Returns the second partial expression in this conjunction. So it returns Y in $XandY$.

Class Comparison

Another BooleanExpression. An object of class Comparison corresponds to the function compare in XPath. It is build up by a path object, a comparison and a value to compare with. So this class corresponds to the XPath expression $path\ \lambda\ value$ for any path expression, constant value and comparison λ.

Constructor Comparison(Path, int, String)

This constructor creates a new Comparison object. It represents the comparison of the constant integer value with the result of the evaluation of the path.

Constructor Comparison(Path, String, String)

This constructor creates a new Comparison object. It represents the comparison of the constant string value with the result of the evaluation of the specified path.

Method getOp()

Returns the comparison operation for this comparison.

Method getOpString()

Returns the string representation of the comparison operation of this Comparison object.

Method getPath()

Returns the path object of this comparison.

Method getValue()

Returns the string representation of the constant value to be compared with in this comparison.

Class EmptySet

This class does not correspond to a special XPath expression, but to a possible result of a XPath query that is the empty set. During the rewriting is is possible that some partial expressions of the XPath query may have return no results. Because of this it is useful to include this class into the abstract syntax. An EmptySet is both, a NodeSelection and a BooleanExpression. It does not specify any methods.

Class Path

Objects of this class represent path expressions in an XPath query. Path expressions can occur both, at the outermost level and nested in predicates. That is why this class is both, a NodeSelection and a BooleanExpression. A Path object is build up by a list of Step objects. Path expressions can be relative or absolute, which is indicated in XPath with the existence or non-existence of / at the beginning of a path.

Constructor Path(boolean)

This constructor creates a new Path object. The argument indicates whether this is an absolute path or not.

Constructor Path(boolean, LinkedList<Step>)

This constructor creates a new path object. The argument indicates whether is is an absolute path or not. The path consists of the list of the specified Step objects.

Constructor Path(boolean, Step)

This constructor creates a new Path object. The argument indicates whether this is an absolute path or not. This path consists of one step.

Method isAbsolutePath()

Returns a boolean value, which indicates whether this path is representing an absolute path or not.

Method addFirstStep(Step)

Similar to NodeSelection.addStep(Step) but the new step is added at the beginning of the path expression.

Method getStepList()

Returns the list of steps this path expression consists of.

Method hasPredicates()

Checks whether this path expression contains any predicates. The predicate may occur at any point within the step.

Class Step

Steps are built up from axes (Descendant, Child, Parent, ...) and nodetests (text(), node(), *, <Label>). Additionally each Step may contain a list of predicates of any length. Path expressions in XPath are build up by a list of steps.

Constructor Step(Axis, NodeTest, List<Predicate>)

Creates a new Step object with the specified axis, nodetest and list of predicates.

Method addPredicates(List<Predicate>)

Adds some predicates to this step object.

Method isBackwardAxisStep()

Returns a boolean value, which indicates whether the axis of this step is a forward or a backward axis.

Method hasBackwardAxis()

Checks whether this step object contains any backward axis (this can be its own axis or any axis within its predicates).

Method hasBackwardPredicates()

Checks whether there are backward axes within the predicates of this step.

Method getAxis()

Returns the axis of this Step object.

Method getNodeText()

Returns the nodetest of this Step object.

Method addPredicate(Predicate)

Adds the given predicate to the list of predicates of this step.

Method getForwardPredicates()

Returns the list of predicates of this step, which do not contain any reverse axes.

Method getBackwardPredicates()

Returns the list of predicates of this step, which contain reverse axes.

Method getPredicates()

Returns the list of all predicates of this step object. The list contains exactly the elements, which are returned by getForwardPredicates() and by getBackwardPredicates().

Method setNodeTest(NodeTest)

Sets the nodetest of this Step object.

Method setPredicates(List<Predicate>)

Sets the predicates of this Step object (existing predicates are overwritten).

Class Predicate

Predicates in an XPath expression can occur in a nested way or in a sequence. The class Predicate represents an XPath predicate and simply encapsulates a BooleanExpression.

Constructor Predicate(BooleanExpression)

Creates a new Predicate object, which encapsulates the given BooleanExpression.

Method containsBackward()

Checks whether this predicate object contains any backward axes.

Method flatten()

Converts this predicate so that it does not contain nested predicates anymore. ([p1[p2[p3]]] is equivalent to [p1/p2/p3]).

Method getContent()

Returns the content of this predicate object.

Method setContent(BooleanExpression)

Sets the content of this predicate object. The former content is overwritten.

3.1.3 Design pattern Visitor

In the object oriented programming and software engineering, the *Visitor* design pattern is a way to separate algorithms from data structures. A direct consequence of this is that new algorithms do not imply changes in the data structure.

This design pattern is applied for the class structure, which was introduced in the previous section. For this purpose a new interface *XPathVisitor* is introduced, which can be accepted by any *Expression* by the method *accept(XPathVisitor)*. The various implementations of this interface represent various algorithms and accepting a visitor represents the execution of that algorithm. The *XPathVisitor* interface specifies the following methods:

- visit(AndExpression)
- visit(NotExpression)
- visit(OrExpression)
- visit(Comparison)
- visit(Intersect)
- visit(Except)
- visit(Union)

- visit(Path)
- visit(Step)
- visit(Predicate)
- visit(EmptySet)

An important advantage of the visitor pattern is that the algorithms can be applied on any partial expression of the query. In order to calculate the result of a query, the algorithm applies itself recursively on the partial expressions of the query and combines the intermediate results. An example for the usage of the visitor is shown in Section 3.3.1. More details on this design pattern can be found in [WPVis].

3.1.4 XPath Properties

Some of the properties of XPath queries are useful for the implementation of the rewriter or they indicate how efficiently the corresponding XPath expression can be evaluated.

Within the scope of this work some classes are defined to calculate and to combine properties of an XPath expression. Many visitor classes are defined in order to keep the existing class structure for the abstract syntax. The class *XPathProperty* plays an important role here. Instances of this class can be created for a specific XPath query and they encapsulate properties for this query. More precisely, there are the following properties:

OutermostStepCount The number of steps at the outermost level of this query (not nested in predicates)

StepCount The number of steps in this query (also in nested predicates)

OutermostReverseStepCount The number of steps with reverse axes at the outermost level of this query (not nested in predicates)

ReverseStepCount The number of steps with reverse axes in this query (also in nested predicates)

UnionCount The number of unions in this query

ReversePos Positions of all steps with reverse axes (at the outermost level)

3.2 The Rewriter Core

For the implementation of the rewriter core an interface *RewritingRule* is developed. This interface specifies methods for testing whether a rule may be applied on a specific query and methods for applying that rule. There is a concrete implementation of this interface for about 20 rewriting rules for reverse axes elimination and 10 simplification rules.

The rewriter processes the query from *left to right* what means that the left partial expressions of a query are rewritten first. But the sequence of application of rules is irrelevant. The rewriting is done by matching a fragment of the input query and the *left hand side* of a rewriting rule whereby the variables in that rule get bounded. Hereupon the fragment of the query, which was matched before is substituted by the instance of the *right hand side* under upper binding.

For every rewriting rule in Section 2.2 there exists one concrete implementation of the RewritingRule interface. The following code is a snippet of one of these class, the *ChildParentRule*.

```
public NodeSelection applyRule(Step forward, Step backward,
        boolean isInPredicate, Path prefix)
{
    //... preperations omitted here
    Path result = prefix.clone();

    Step selfStep = new Step(new Self(), backward.getNodeTest(),
            backward.getPredicates());

    if (!isInPredicate) {
        selfStep.addPredicate(
            new Predicate(new Path(false, forward)));
    }

    result.addStep(selfStep);

    if (isInPredicate) {
        result.addStep(forward);
    }

    return result;
}
```

This method implements the following two rules, which have been introduced in Section 2.2 already.

$$\begin{array}{rcl} \text{p/child::n/parent::m} & \equiv & \text{p/self::m[child::n]} \\ \text{p/child::n[parent::m]} & \equiv & \text{p/self::m/child::n} \end{array}$$

Here, the prefix p is the method's *prefix* argument, *child::n* and *parent::m* are the method's *forward* and *backward* arguments and the argument *isInPredicate* specifies which of the two rules above must be applied. The result of this method is the right side of the rule: an XPath expression that is equivalent to the argument and either

does not contain any reverse axes or which has been rewritten such that one of its reverse axes is "pushed leftwise".

Figure 5 shows a class diagram for the rewriter's core module. The class *Rewriter* controls the whole rewriting process. It traverses the abstract syntax tree for the XPath expression (see Section 3.1.2) and chooses rewriting rules to apply until the XPath expression is fully rewritten.

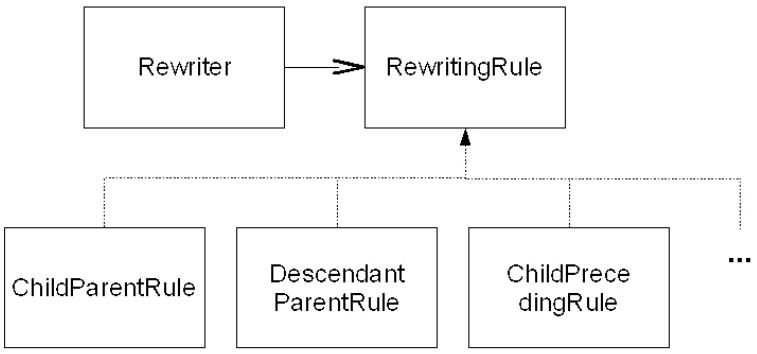


Figure 5: Class diagram for rewriter's core module

3.3 Visualization

Visualization is the last part of this work beside parsing and rewriting of XPath queries. Our solution demonstrates the XPath queries in a graphical structure, which is similar to the structure of the abstract syntax. In addition, a graphical user interface for input, rewriting, and visualization is needed.

3.3.1 Graph representation of XPath queries

XPath queries can be represented as graphs whose structure is similar to the class structure introduced in this work, but has explicit differences also. This graph represents a tree just like the abstract syntax does, but there is not a node for each introduced class in this tree. Many of the introduced classes encapsulate some more values and offer no other information. For example the class *Path* provides many useful methods but actually only encapsulates a list of *Step* objects.

Steps

Most of the nodes in the graph will be counterparts for steps in a path expression. These nodes have an oval form and they are either cyan or magenta colored. The colour of these nodes indicates whether this is a forward (cyan) or backward (magenta) axis, which is also obvious by the label of the nodes. Figure 6 shows some of these step nodes, as they are represented in the graph.

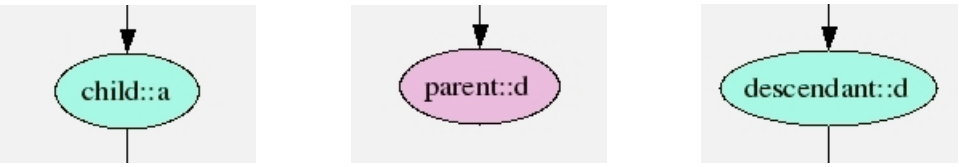


Figure 6: Step nodes with various colours

Paths

The simplest XPath query contains a path only. This is also reflected by the simplicity of the corresponding graph. Paths of the length n are not represented as a root node with n children but as a linear concatenation of n step nodes. Figure 7 shows that each step in the path $desc::x/child::y/parent::z$ matches with a node in that linear graph.

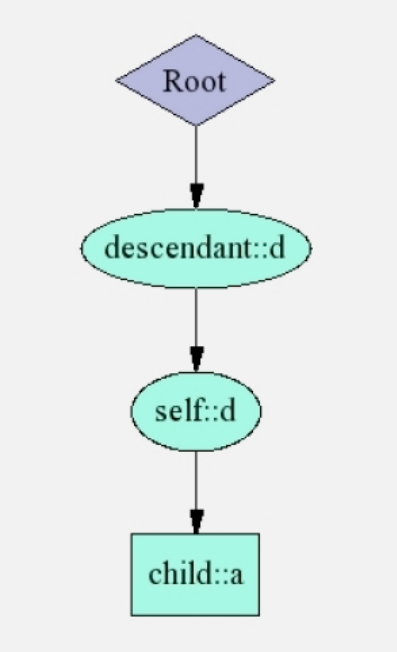


Figure 7: Path query represented as a linear graph

Selecting-Nodes

Furthermore Figure 7 shows that not all step nodes have an oval form. In linear queries there is exactly one step node with rectangular form. These nodes are called selecting-nodes and they are very important for the evaluation of the query. The selecting-node is not always the last node. Figure 8 will demonstrate this on the basis of the queries $/desc::d/child::a[parent::d]$ and $desc::d/child::a[parent::d]/child::b$.

Boolean Operations

Figure 8 already showed an *And* node in the graph, although there was no and conjunction in the corresponding query. This and node was qualified by the predicate in the query. Boolean operations within predicates are described the same way in the graph. Figure 9 will demonstrate this for the following query $desc::d[child::a\ and\ child::b\ or\ foll::z]$.

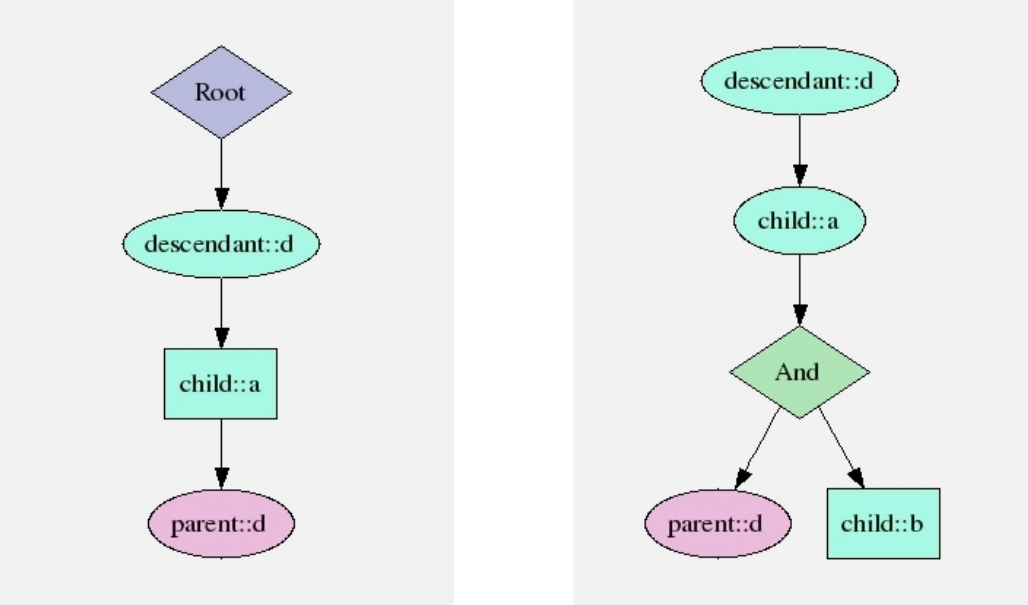


Figure 8: Selecting-nodes in the graph

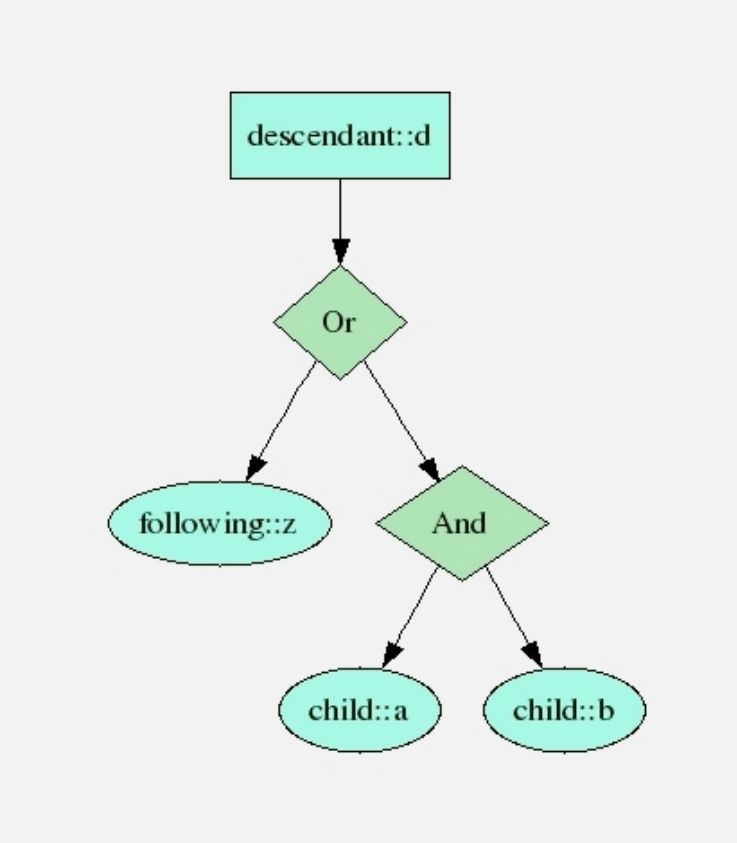


Figure 9: Boolean operations in a query

Set Operations

Since in the used XPath fragment set operations can occur on top level only, they set up the upper part of the graph. Attention should be paid to the fact that here

several selecting nodes are used for the first time. That is shown for the query $desc::a[parent::b] \mid child::x/child::y$ in 10.

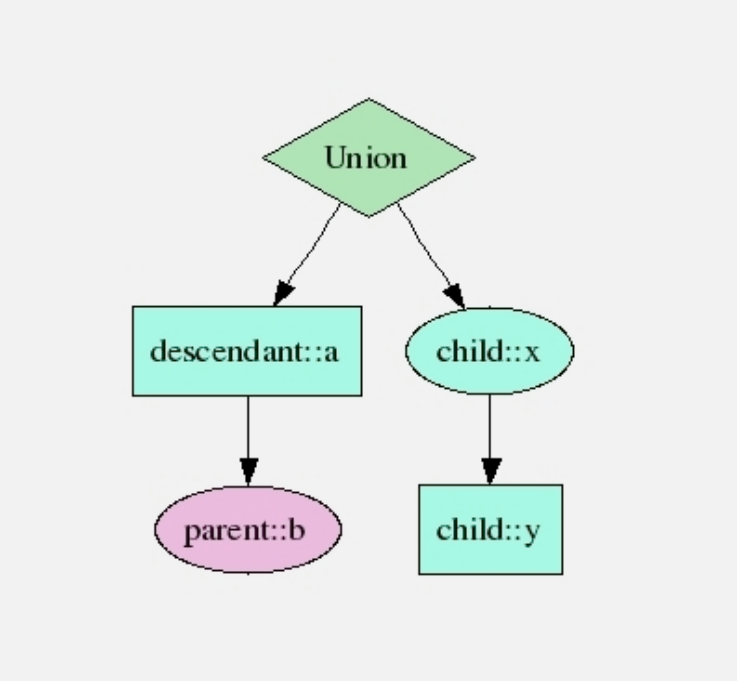


Figure 10: Set operations in a query

Implementation using DOT and Visitor

The graph description language *DOT* of the GraphViz project [Dot] was used for the implementation. DOT is a language for describing graphs as showed in the figures before. The implementation uses a visitor for the generation of the DOT code and then applies the terminal application (also dot) in order to create the graphic. Those generated graphics then can be built into the graphical user interface.

3.3.2 Graphical User Interface

The graphical user interface shows the query and the output of each rewriting step. One can follow the rewriting from the original query to the result query. The user interface offers some additional useful functions, which are easy to use. The last figures in this section demonstrate the user interface. Figure 11 shows the dialog asking the user for the input query. Figure 12 shows a popup menu, which contains some useful functions applyable on each graph. The last figure 13 demonstrates the whole user interface.

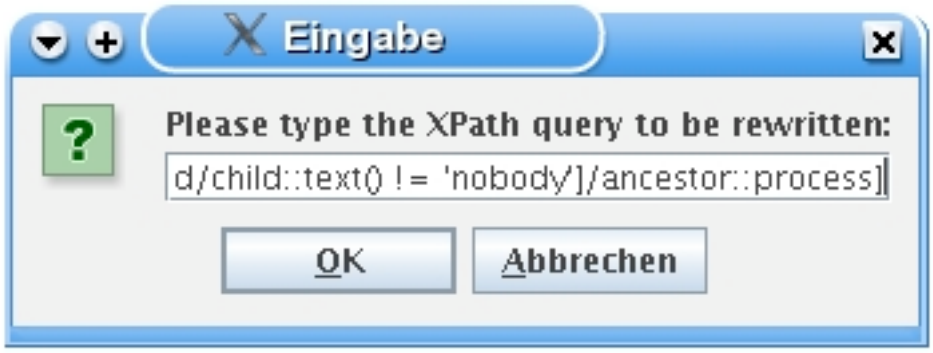


Figure 11: The input of a query

4 Conclusion

4.1 Integration into the SPEX project

The XPath Rewriter is part of the SPEX project, which can be found at *Sourceforge.net* [SF]. The rewriter consists of two modules, *ReXP* and *ReXP-Gui*. Both of these contain an Ant build script [Ant], which can compile the code and generate a distribution package for this project.

4.2 Installation and usage

The installation and usage of the parser, rewriter and visualizer is straight forward, but first one should make sure that necessary software components of third party vendors are installed. Various preconditions have to be fullfilled on the computer depending on whether the rewriter is going to be used as a terminal application only or if it is going to be used with the graphical user interface as well.

As Terminal Application

The XPath Rewriter as terminal application needs a Java runtime environment in the version 5.0 or higher only [SUNJava]. For the usage of the console tool one needs the distribution package of the *ReXP* module, which contains the file *rexp.jar*. The rewriter can be started in the terminal with the command **java -jar rexp.jar**. More information for the usage of the rewriter can be obtained in the console.

With Graphical User Interface

For the graphical version the console tool *dot* of the GraphViz project [Dot] has to be installed on the computer in addition to the Java runtime environment. For the usage of the graphical version the distribution package of the *ReXP-GUI* module is needed, which contains the file *rexp-gui.jar*. The starting is as straight forwards as for the terminal version: the command **java -jar rexp-gui.jar** starts the graphical user interface.

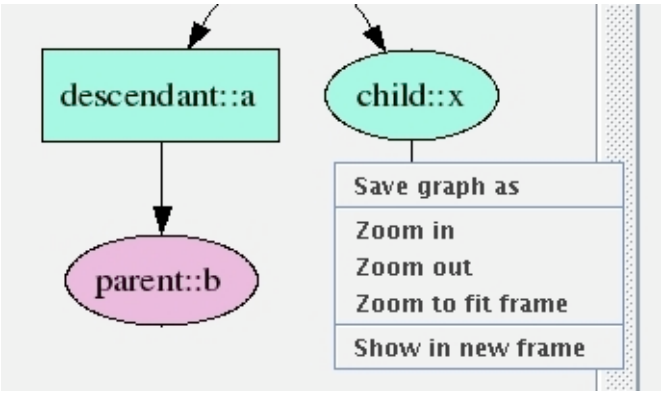


Figure 12: Available functions

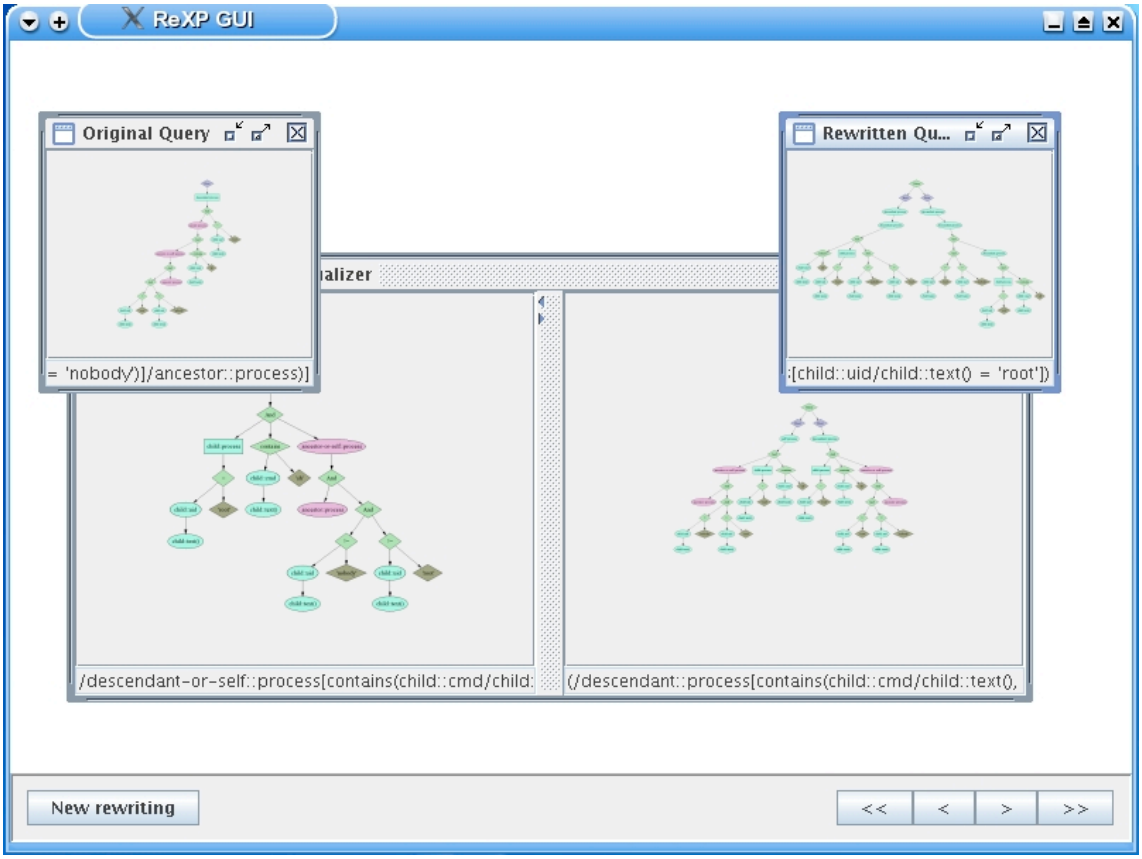


Figure 13: The user interface of the rewriter

References

[Olt1] *Evaluation of XPath Queries against XML streams*, Dan Olteanu. PhD Thesis, Institute for Informatics, University of Munich. December 2004.
URL: `http://edoc.ub.uni-muenchen.de/archive/00003201/01/Olteanu_Dan.pdf`

[Olt2] *XPath: Looking Forward*, Dan A. Olteanu, Holger Meuss, Tim Furche and François Bry, Proc. of Workshop on XML-Based Data Management (XMLDM) at EDBT 2002, Prague, March 2002. ©Springer-Verlag.
URL: `http://www.pms.informatik.uni-muenchen.de/publikationen/index.html#PMS-FB-2001-17`

[SF] The Spex Project on sourcefore.net
URL: `http://spex.sourceforge.net/`

[JavaCC] Java Compiler Compiler [tm] (JavaCC [tm]) - The Java Parser Generator
URL: `https://javacc.dev.java.net/`

[XPath] XML Path Language (XPath) Version 1.0
URL: `http://www.w3.org/TR/xpath/`

[WPVis] *Visitor (Design Pattern)* on de.wikipedia.org
URL: `http://de.wikipedia.org/wiki/Besucher_%28Entwurfsmuster%29`

[SUNJava] JavaSE 5.0 Technology, SUN Microsystems
URL: `http://java.sun.com/`

[Dot] Open source graph (network) visualization project from AT&T Research
URL: `http://www.graphviz.org/`

[Ant] The Apache Ant Project, a build-tool for Java
URL: `http://ant.apache.org/`