INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen
Oettingenstraße 67 D–80538 München

Ludwig
Maximilians–
Universität
München
LMU

Design and Implementation of a New SPEX Version

Fatih Coşkun

Projektarbeit

Beginn der Arbeit:	01.03.2004
Abgabe der Arbeit:	01.06.2006
Betreuer:	Prof. Dr. François Bry
	Dr. Dan Olteanu

Erklärung

Hiermit versichere ich, dass ich diese Diplomarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 01.06.2006 Fatih Coşkun

Acknowledgments

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

Zusammenfassung

XML und *Datenströme*, zwei Schlüsselbegriffe aus zwei verschiedenen Bereichen der Informatik, sind Konzepte, welche aus der Welt der Computer nicht mehr wegzudenken sind. Immer grösserer Beliebtheit und Akzeptanz erfreuend, werden diese beiden Technologien nun in einem gemeinsamen Forschungsgebiet - den *XML-Strömen* - vereint. Eines der wichtigen Probleme in diesem Bereich ist die Anfrageauswertung, im besonderen die Auswertung von XPath-Anfragen gegen XML-Ströme. Die vorliegende Arbeit beschäftigt sich mit genau diesem Problem, den bei einer Auswertung auftretenden theoretischen Schwierigkeiten und deren Lösung, sowie der Implementierung eines Auswerters selbst. Dieser Arbeit zugrundeliegend sind die theoretischen Arbeiten von Dan Olteanu (und anderen) zu SPEX, dem effizienten XPath-Auswerter, welche unter anderem in [SPEX1], [SPEX2] und [SPEX3] zu finden sind. Im Rahmen dieser Arbeit wurde eine neue Version des SPEX-Auswerters entwickelt und implementiert, welche in [SPEX4] beschrieben ist. Die neue Version besitzt folgende Eigenschaften:

- Solide Implementierung in neuester Java 5.0 Technologie
- Unterstützung eines größeren XPath-Fragments (siehe Abschnitt 1.4)
- Opensource-Lizenz und Veröffentlichung auf Sourceforge [SF]
- *Compiletime*-Optimierungen
- Regressions- und Performanztests

Abstract

XML and *Datastreams*, two keywords from two separate sections of computer science, are concepts the world of computers can not live without anymore. Getting more and more important and accepted nowadays, these two technologies get combined in one field of research, *XML-Streams*. One of the more important problems in this field is query evaluation, in particular the evaluation of XPath queries against XML streams. The paper at hand deals with this problem, the theoretical challenge of evaluation and the implementation of the evaluator itself. It is based on theoretical work done by Dan Olteanu (and others) on SPEX, the efficient XPath evaluator, which can be found in [SPEX1], [SPEX2] and [SPEX3]. In the scope of this work a new version of the SPEX evaluator was designed and implemented, as described in [SPEX4]. The new version has the following features:

- solid implementation in up-to-date Java 5.0
- support for larger XPath fragment (see Section 1.4) with polynomial combined complexity
- open source license and release at sourceforge [SF]
- compile-time optimizations
- regression and performance tests

Contents

1 Introduction

Before getting into details about SPEX and the query evaluation, concepts and technologies used in this work are introduced. The idea behind datastreams is explained as well as all parts of the XML technology, including XML itself, XPath queries and also SAX parsers for XML. Also a brief introduction about the theory behind the SPEX evaluator is given.

Following the introduction, the main part of this paper deals with the implementation of SPEX. Each section in that part considers the theoretical issues and ideas as well as the implementation of a particular module of SPEX.

The work finishes with considerations about compile-time optimizations, from which some have been implemented already and others exist in theory only at time being. Performance tests have been done and analyzed for the implemented optimizations and are also shown in this section.

The last section concludes this paper with some words about the *SPEX Project* and its subprojects, which can be found at sourceforge [SF].

1.1 Datastreams

The concept of *datastreams* exists as early as computer technology itself. However in recent time there have been many efforts in a new research field, which is exploring new application areas for datastreams as described in [STREAMS]. These new applications look on streams as a mean to perform the so called *push communication*, which stays in contrast to the conventional *pull communication* heavily used in nowadays Web technology. This means that clients are not requesting any information to be send to them when they are ready to receive (pull), but rather information is send to them at any time (push). Hence clients must make best effort to be able to receive data with possibly unbounded size and use it at any time.

Datastreams can be realized in many different ways. A datastream can be a sequence of signals, bytes or characters, objects and even a sequence of method invocations in a higher programing language. All these different views on datastreams have in common, that large amounts of datasets are sent and received in very short periods of time.

In *XML-Streams* the datastream's datasets are nodes in an XML document. The difference to conventional XML documents is, that XML streams may be unbounded in length, whereas an XML document has a finite size.

1.2 XML

The *Extensible Markup Language* is a semi-structured meta language describing human- and also machine-readable documents. These semi-structured documents are used heavily nowadays for storing large amounts of data. XML documents contain so called XML nodes, which together form a tree structure, beginning with the root node. More information about XML can be found for example in [XML].

There is a large research field and many standards around XML, dealing with parsing, processing and querying XML documents. Some of them are described in

the next sections. Information about other standards and research can be found at [XML].

1.3 Dom & SAX

There are many different approaches for parsing XML. The most important ones are [DOM] and [SAX] parsers. DOM is specified by *W3C* and describes interfaces for parsing and processing XML documents. Using these interfaces, one can navigate easily through an XML document, being able to access randomly any set of nodes. Using DOM parsers has the disadvantage, that the whole document must be hold in memory. This factor makes it unfeasible to use DOM for parsing XML streams, which can be very long or even unbounded in length.

SAX parsers on the other hand make a different attempt. They belong to the event based parsers and as such do not hold any parts of a document in memory. They encapsulate parts of the document in events and forward them to the application. Storing parts of the document is then in responsibility of the application using the parser, which allows for full control of memory usage. SAX specifies also an interface for parsing an XML document or stream. The most important interface in SAX is the *node-handler*, which has to be implemented by the application using the SAX parser. Each time the parser processes a node in the XML stream, the node handler's corresponding method gets invoked. The application can decide what to do with this node, which allows full memory control.

SAX parsers are exactly what is needed for processing unbounded XML streams, and hence this work chose to use SAX parsers. Because SAX only describes interfaces for parsing and processing XML, there are many different implementations for SAX, which are designed for different kinds of applications. The application built in this work is capable of using any SAX parser implementation. If no specific one is specified, *Crimson SAX* is used by default. More information about the use of the SAX interface can be found in Section 2.1.

1.4 XPath

The *XML Path Language*, as a stable W3C recommendation, is a language for addressing parts of an XML document [XPath]. It was initially designed to be used in other querying languages such as *XSLT*, *XForm* and *XQuery*. Nowadays it is used not only in other querying languages, but also as a standalone language for querying XML documents. As such, there are more and more APIs (Application Programming Interface) in modern programing languages such as Java, for evaluating XPath queries against XML documents.

The application implemented in this work, SPEX, is an efficient single pass XPath evaluator, capable of evaluating a subset of XPath, called *ForwardXPath*, which does not contain any reverse axes. This is not a real limitation, as can be seen in theoretical works like [Olt1] and [Olt2], which prove that every XPath query can be rewritten to an equivalent query without any reverse axes. A corresponding rewriter has been implemented in [RexP] and is used by SPEX. The following grammar defines the fragment of XPath 1.0, which can be evaluated by SPEX.

Query	::=	NodeSelection.
NodeSelection	::=	'(' NodeSelection ')' | Union | Path.
Union	::=	NodeSelection '|' NodeSelection.
Path	::=	('/')? Step ('/' Step)*.
Step	::=	Axis '::' NodeTest (Predicate)*.
Axis	::=	'child' | 'descendant' | 'self' | 'following' | 'following-sibling'.
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, and it contains forward axes only (note that queries that contain reverse axes can be rewritten with the rewriter). 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 additions to SPEX.

Figure 1 demonstrates the selection of a node using the XPath expression */descendant::author/parent::book*.

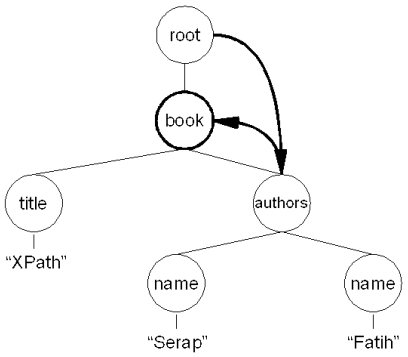

Figure 1: Selection of node by an XPath expression

1.5 SPEX Basics

SPEX, the *Streamed and Progressive Evaluator for XPath*, was initially described in [SPEX3], and has undergone many revisions since then. A proof-of-concept implementation for the original proposal can be found in [SPEX3]. The paper at hand

describes the implementation of an up-to-date revision of SPEX, which fulfills several design principles: this SPEX implementation is efficient, scalable and ready for open-source publishing at sourceforge.net. This section slightly introduces the theory of SPEX evaluation, without going to deep into implementation details.

Querying XML streams with SPEX consists in four steps, as shown in Figure 2. First, the input XPath query is rewritten into a forward XPath query [REXP], i.e., without reverse axes. For the query

```
/desc::process[child::time > 24 or child::memory > 500]
/anc::process[child::priority < 10 and child::state = ”stopped”]
```

the result of this source-to-source transformation is

```
/desc::process[child::priority < 10 and child::state = ”stopped”
and desc::process[child::time > 24 or child::memory > 500]]
```

The forward XPath query is compiled into a logical query plan that abstracts out details of the concrete XPath syntax. Figure 3 gives a logical query plan for the forward query. Then, a physical query plan is generated by extending the logical query plan with operators for determination and collection of answers. Figure 4 shows a physical query plan for the logical query plan of Figure 3. In the last step, the input XML stream is processed continuously with the physical query plan, and the output stream conveying the answers to the original query is generated progressively. All four steps are further detailed below.

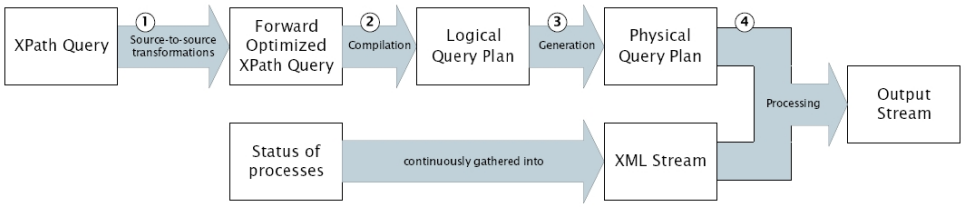

Figure 2: Steps of the SPEX processor

Step 1: Source-to-source query transformations

The forward and reverse XPath axes enable random access to nodes of an XML tree. If queries are to be evaluated against streams conveying XML trees, nodes cannot be accessed randomly, but rather in the stream's sequence. The evaluation of reverse axes, e.g., ancestor and preceding, would demand then the buffering of already processed stream fragments. SPEX proposes a framework [REXP] for rewriting queries with reverse axes into equivalent queries in which only forward axes occur. Further source-to-source transformations that optimize the evaluation of forward XPath queries are also applied in this step. Such optimizations focus on pruning redundant computations. E.g., consider the query $/child :: process/following :: state$ that selects all state-elements following process children of the root. For the set of state-elements that follow the first process child of the root is already the

set of state-elements that follow all process children of the root, this query can be rewritten to $/child :: process[1]/following :: state$, so that only the first process child of the root is considered during evaluation.

Step 2: Compilation into a logical query plan

A forward XPath query is compiled into a logical query plan that consists either in a path, if the query is a sequence of steps, in a tree, if the query has also predicates, or in a directed acyclic graph, if the query has also set operators. Each construct in a forward XPath query, such as an axis or a predicate, induces a corresponding operator in the logical query plan. Figure 3 shows the logical query plan for the query:

```
/desc::process[child::priority < 10 and child::state = ''stopped''
and desc::process[child::time > 24 or child::memory > 500]]
```

Square boxes denote the answers sought for, round boxes correspond to (parts of) predicates. E.g., the and (or) operator of Figure 3 expresses that both (at least one of the) subplans rooted at the subjacent child operator further constrain the answers selected by the first process operator. At this step, further compile-time optimizations can be applied. As shown in Figure 3, both prefixes child of the branches rooted at the and (or) operator are compacted into a single child operator. Note that such a branch compaction is not possible at the level of XPath syntax.

Step 3: Generation of a physical query plan

A physical query plan is a transducer network that computes the answers to the initial query from the XML stream. Such a network is created from a logical query plan in two steps. First, each operator from a logical query plan is realized in a network as a deterministic pushdown transducer. Second, the network is extended at its beginning with a stream delivering transducer *in*, and at its end with an answer collecting *funnel*, i.e., a subnetwork of auxiliary transducers serving to collect the computed potential answers. Figure 4 shows the network constructed from the logical query plan of Figure 3. For each predicate in the query there is a pair (*scope-begin*, *scope-end*) of transducers in the network. The scope-end transducer is preceded by the appropriate boolean transducer, such as an *and* or *or* (corresponding to the *and* or *or* node in the logical query plan). The nesting of such pairs corresponds to the nesting of predicates in the query. The topmost process transducer is the answer transducer, as indicated by the square box (this one is called *head* transducer in the implementation). The last transducer of the funnel is the out transducer that buffers potential answers and delivers the query answers.

Step 4: Processing with a physical query plan

Processing an XML stream corresponds to a depth-first left-to-right preorder traversal of the (implicit) XML tree conveyed by that stream. Exploiting the affinity between preorder traversal and stack management, the transducers use their stacks for remembering the depth of the nodes in the implicit XML tree. This way, forward XPath axes, e.g., child and desc, can be evaluated in a single pass. A physical query plan, i.e., a transducer network, processes the XML stream annotated by its first transducer *in*. The other transducers in the network process stepwise the received

annotated XML stream and send it with changed annotations to their successor transducers. E.g., a transducer child moves the annotation of each node to all children of that node. The answers computed by a transducer network are among the nodes annotated by the answer transducer. These nodes are potential answers, as they may depend on a down-stream satisfaction of predicates. The information on predicate satisfaction is conveyed in network also by annotations. Until the predicate satisfaction is decided, the potential answers are buffered by the *out* transducer. Those optimizations that are specific to stream processing are applied only to the physical query plan. Specialized transducers are employed to minimize the stream fragment processed by transducers in a network. E.g., in the physical query plan of Figure 4, all transducers after the answer transducer require only the stream fragments conveying the subtrees rooted at nodes selected by *desc* :: *process* and the irrelevant stream fragments can be filtered out by an appropriate pushdown transducer placed after the answer transducer. Various structural filters can be added to physical query plans, depending on the kind of transducers existent in a network and on the stream structure. The latter dependencies can be derived, e.g., from schemas of the stream.

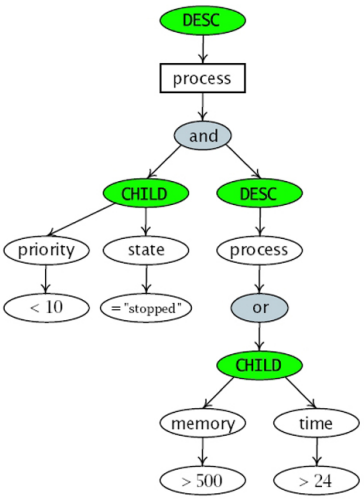

Figure 3: Logical Query Plan

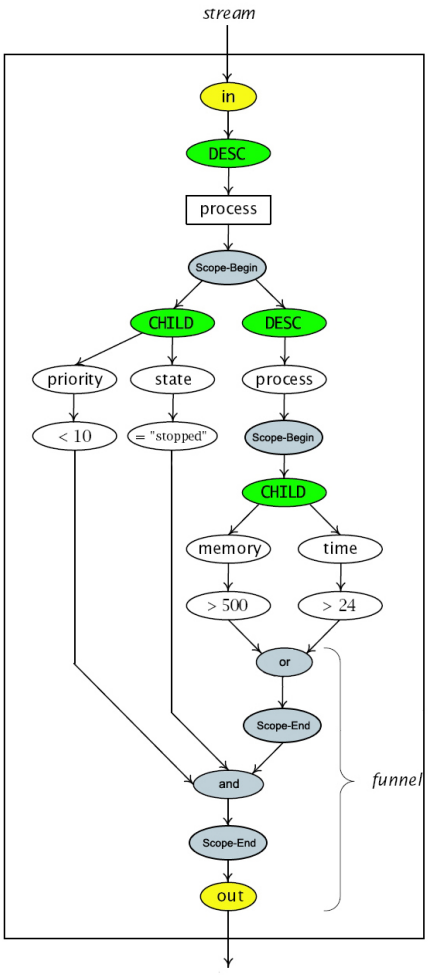

Figure 4: Physical Query Plan

Example evaluation

Figure 5 demonstrates the working of the different transducers in a network of transducers. This figure may imply several passes through the XML stream, but this is not a necessity. Each transducer forwards nodes of the XML stream prior to receiving succeeding nodes, which corresponds in a single pass through the XML stream.

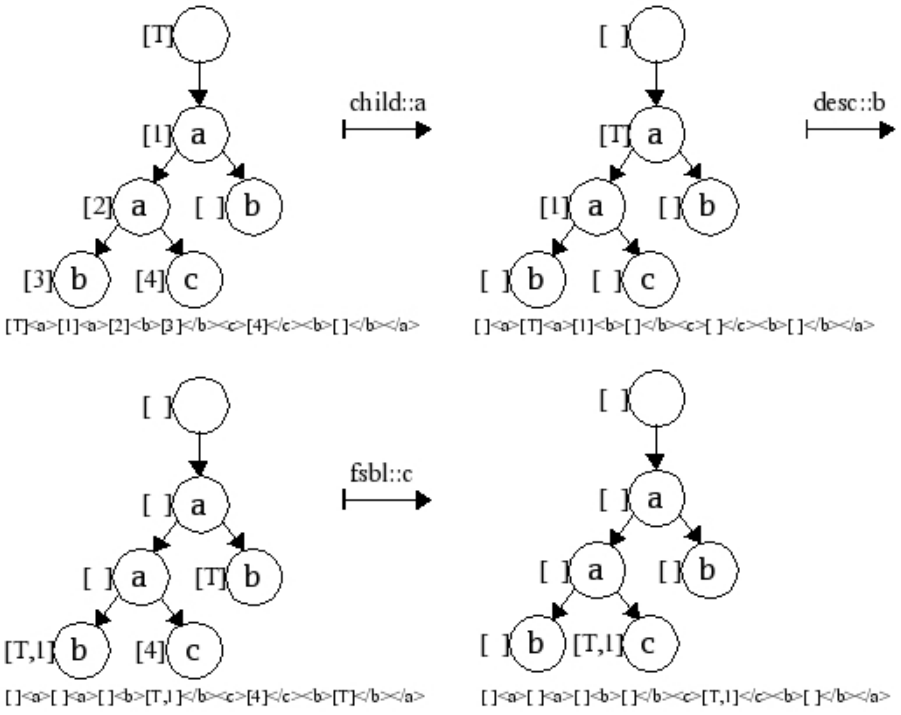

Figure 5: XML tree traversed by several transducers

For this example several nodes in the input XML stream are annotated with different annotations. In a real transducer network only the root node of the input XML stream is annotated with [T] and all other nodes are annotated with the empty annotation. The example demonstrates, how different transducers change the nodes' annotations.

The first transducer *child::a forwards* all annotations of a node X to a node Y if Y is the child of X and Y is labeled with "a". This is the case for the first and the second node in the stream labeled "a". The first node labeled "a" gets annotated with [T], because its parent (the root) was annotated with [T]. The second node labeled "a" gets annotated with [1], because its parent was annotated with [1]. All other nodes get annotated with the empty annotation, because they are not labeled "a".

The second transducer *desc::b forwards* all annotations of a node X to a node Y if Y is a descendant of X and Y is labeled "b". This is the case for two nodes in the stream. The first node labeled "b" gets annotated with [T,1], because there were two preceding nodes, from which one was annotated with [T] and the other was annotated with [1]. Also note that although the root node is a preceding it did

not contain any annotations. The second node labeled "b" gets annotated with [T] only, because it has just one preceding node that was annotated. Other nodes in the stream get annotated with the empty annotation.

The third transducer *fsbl::c* annotates one node labeled "c" with [T,1], because that node is a following-sibling of a node, which was labeled with [T,1] previously. All other nodes get annotated with the empty annotation, because they are not labeled "c".

At the end one node is annotated with [T,1]. This node is a result of the query *child::a/desc::b/fsbl::c* if the context of the query is either the root node (which was annotated with [T] at the beginning) or the first node labeled "a" (which was annotated with [1] at the beginning.

2 Implementation

This section describes all parts of the implementation of SPEX as well as its theoretical background. It is divided in several subsections from which each is describing one module of the SPEX implementation. The parsing of XPath queries is done by the subproject [ReXP] and is not explained in this paper. ReXP is also capable of rewriting all XPath queries to equivalent queries without reverse axes. Hence for the sake of simplicity we are using the word *XPath* throughout this section, although a subset of XPath (*ForwardXPath*) is used actually. This section does not describe all classes of the SPEX implementation, as this would not contribute to the understanding of the implementation. Only the most important ones and common superclasses are explained in detail. Figure 6 shows the modules and the dependencies between each module in the SPEX project. The transducer network is the most important and biggest module.

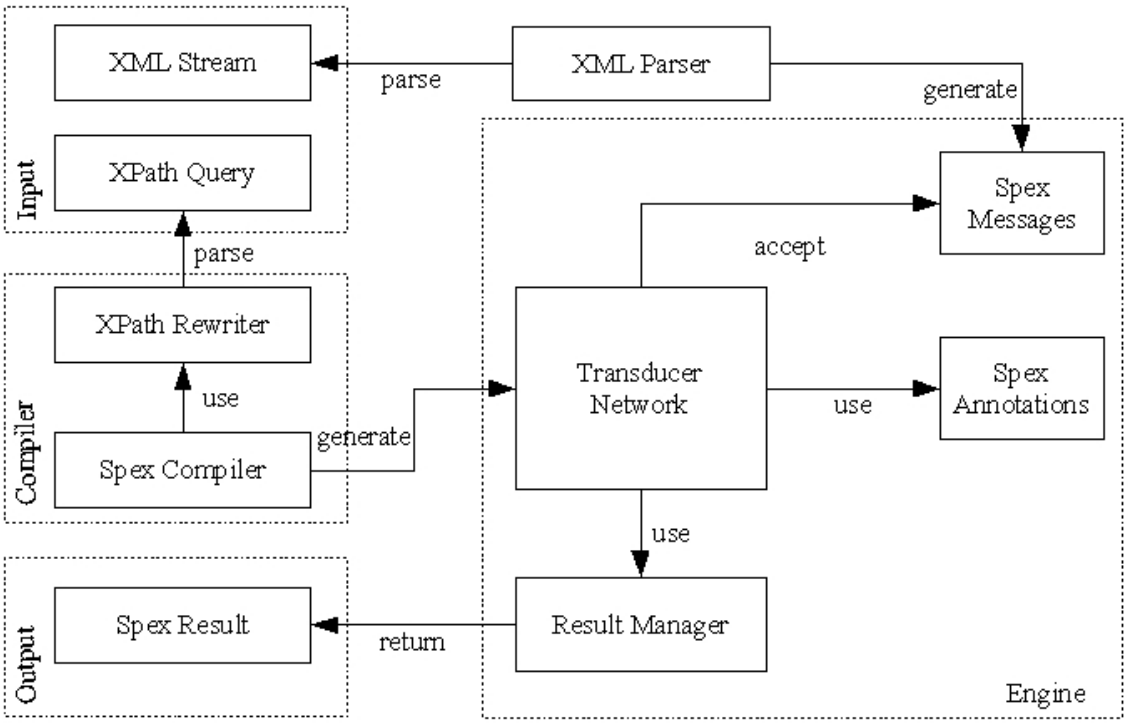

Figure 6: Modules and dependencies of the Spex evaluator

Package Hierarchy

The several modules of the SPEX evaluator are organized in appropriate Java packages. All packages contain the common prefix *de.lmu.ifi.pms.spex*. The following list illustrates these packages and shows links to their javadoc documentation at sourceforge or the section in this paper they are described in.

- **de.lmu.ifi.pms.spex**
  - **main** [JD1]

- **query.xpathcompiler** [JD2]
- **xmlstream** [JD3], Section 2.1
- **engine.messages** [JD4], Section 2.3
- **engine.annotations** [JD5], Section 2.4
- **engine.transducers** [JD6], Section 2.5
  - **filter** [JD7], Section 3
  - **locationstep** [JD8], Section 2.5.3
    - **matcher** [JD9], Section 2.5.4
  - **scope** [JD10], Section 2.5.6
- **engine.results** [JD11], Section 2.6
- **tests** [JD12]
- **utils** [JD13]

2.1 XML Stream Parser

As SPEX is intended to evaluate XPath queries against unbounded XML streams, an adequate XML parser is needed. As argued in Section 1.3, SAX is a candidate for this matter. The SPEX implementation allows the use of any kind of parser simply by implementing one adapter class extending one abstract class of SPEX. That class is called *XMLStreamParser* and one default implementation for SAX parsers is provided in SPEX already. If no other implementation is provided by the user, this default *SAXStreamParser* is used, which once again can be configured to use any of the SAX parsers available. If no SAX parser is specified by the user, then *Crimson SAX* is used.

Class XMLStreamParser

An XMLStreamParser is capable of parsing an XML stream. It is working event based, i.e. when encountering tags in the XML stream it calls appropriate methods of the XMLStreamProcessor, which is registered to this parser. This class is abstract, hence can not be instantiated. However there is one implementation of this class for SAX parsers provided by SPEX already.

Static Method getSystemStreamParser()

This method finds the class specified in the system property *spex.xmlstream.StreamParser* and instantiate it. If not specified a default implementation for SAX is instantiated and returned.

Method parseStream(InputStream)

Abstract method to be implemented by subclasses. This method is called by the public parse(InputStream) method. When a tag or text is encountered in the XML stream, this method must call appropiate methods of the StreamProcessor. This method throws a StreamParserException, if the parsing of the stream can not be continued because of fatal errors.

Method interruptParsing()

Interrupts parsing of XML stream. The parsing is interrupted as soon as one of the registered stream processor's methods returns. The parse(InputStream) method returns immediately then.

Method registerProcessor(XMLStreamProcessor)

Registers the specified stream processor to this parser. The processor is notified about every XML node that has been passed by this parser.

Method getStreamProcessor()

Returns the stream processor, which has been registered to this stream parser.

2.2 SPEX Processor

Every time the stream parser passes a node in the XML stream, an appropriate method in the registered stream processor gets called. There is one default implementation of the stream processor for the SPEX project, which generates messages for each kind of XML node and forwards them to the input transducer of SPEX' transducer network.

This SPEX implementation hence uses the *sequence of method calls* view on datastreams. The stream initially being provided by bits and bytes of characters is translated to a sequence of method calls.

Interface XMLStreamProcessor

Classes implementing this interface are meant to be the connections between the parsing of the XML stream and the processing of the transducers. After registering a stream processor to a stream parser, the parser call the methods of the processor upon passing a corresponding node in the XML stream. There is one default implementation for this interface, which is encapsulated within the *SpexProcessor* class.

Method processDocumentStart()

This method gets called when the start of stream is passed. The default implementation creates a new *DocumentStartMessage* and forward it to the input transducer.

Method processDocumentEnd()

This method gets called when the end of stream is passed. The default implementation creates a new *DocumentEndMessage* and forward it to the input transducer.

Method processElementNode(String, boolean)

This method gets called when a opening or closing tag is passed in the XML stream. The default implementation creates a new *Opening-*

TagMessage or *ClosingTagMessage* and forward it to the input transducer.

Method processTextNode(char[], int, int)

This method gets called when a text node is passed in the XML stream. The default implementation creates a new *TextMessage* and forward it to the input transducer.

Method processAttributeNode(String, String)

This method gets called when a attribute node is passed in the XML stream. The default implementation creates a new *AttributeMessage* and forward it to the input transducer.

Class SpexProcessor

This class provides the default implementation of the StreamProcessor interface and has convenience methods for starting the SPEX processing as simple as possible.

Static Method createProcessor(InputStream,OutputStream,InputTransducer)

Creates a new SpexProcessor instance. The returned processor parse the specified input stream, use the specified transducer network and print the results of the processing to the specified output stream.

Method process()

Starts SPEX processing. This method not return until the SPEX processing finishes (the xml input stream ends) or the interruptProcess() method gets called. The System property *spex.xmlstream.StreamParser* specifies which stream parser class should be used, instead of the default stream parser. If using the default stream parser, the system property *org.xml.sax.driver* specifies the SAX parser class used by the default stream parser. If not specified the *Crimson SAX* parser is used.

Static Method getCurrentProcessor()

This method can be used by any method, which is in the call-hierarchy of the processor to get a reference to the current processor itself.

Method interruptProcess()

Interrupts the SPEX processing. The *process()* method return as soon as the underlying xml stream parser can be interrupted.

2.3 Messages

As described earlier, the datastream, initially being a sequence of bits and bytes, gets translated to a sequence of method calls. For each node in the XML stream a corresponding message type is forwarded to the methods. These message types

basically are all subclasses of a common superclass, the *Message*. For each node type in XML there is a corresponding subclass of Message. For example passing opening tags in the XML stream leads to *OpeningTagMessages*, passing closing tags leads to *ClosingTagMessage*, and so on. Figure 7 shows the classdiagram for the *message* package.

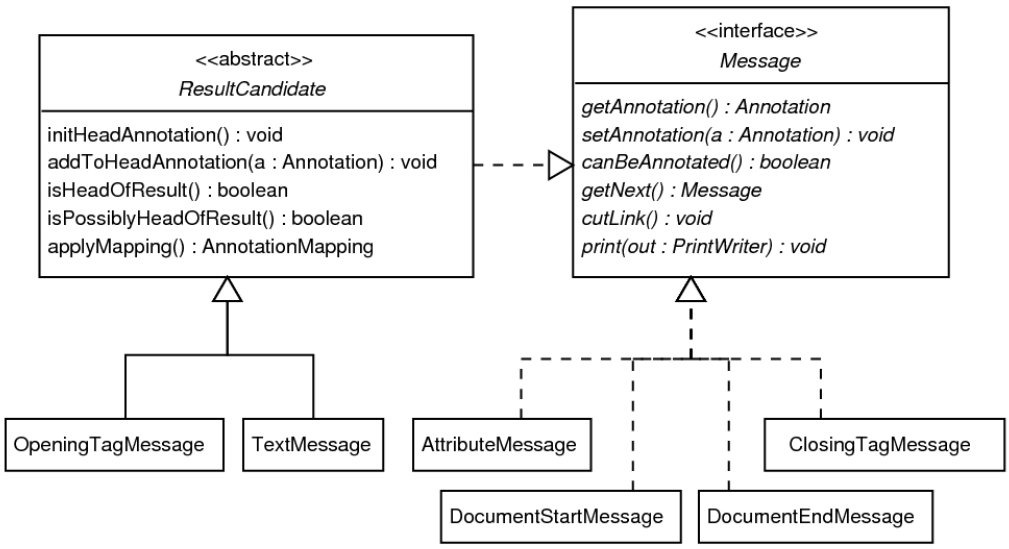

Figure 7: Class diagram for the message package

After getting created a message is forwarded to the input transducer of the SPEX transducer network. The input transducer then forwards it to its succeeding transducers, and this continues until the message reaches the output transducer. The output transducer possibly registers the message at the result buffer, which decides whether a message must be stored or can be ignored, and thus may be deleted by the garbage collector.

Some message types, but not all, can get annotated by transducers using the *Message.setAnnotation(Annotation)* method. Whether a message can be annotated or not is indicated by the result of calling *Message.canBeAnnotated()*. Annotations are an important part of SPEX processing. For example, location step transducers keep track of which annotations have been received by using an stack of annotations. This is explained in detail in later sections of this paper. Further on some messages of this package are extending the *ResultCandidate* class, which makes them capable of being possible (beginnings of) SPEX results. This result candidates can be annotated by some specific transducers with another type of annotation, the head annotation. This is explained in detail in Section 2.6.

Messages get linked together at creation time. Every message has a reference to its direct successor. The successor can be returned by calling *Message.getNext()*, as soon as the stream processor gets notified by the stream parser about this successor. Hence together the messages form a linked list containing all messages, beginning with the start of document and ending with the end of document. The only purpose

of this linked list is simplifying the implementation of result candidate buffering as described in Section 2.6. The linked list has no consequences for the SPEX processing and does not effect how the transducers work. The references to the succeeding messages can be cut off by calling *Message.cutLink()*, which is done by the result manager to discard messages, which either can not be a result or have been printed already, and thus allow the garbage collector to free memory.

Interface Message

Common superclass for all message types. For every node type in XML there is one corresponding class implementing this interface.

Method getAnnotation()

Returns the annotation this message is annotated with.

Method setAnnotation(Annotation)

Annotates this message with the specified annotation.

Method canBeAnnotated()

Returns true, if and only if this message type can be annotated with SPEX annotations. Otherwiese returns false.

Method getNext()

Returns the successor of this message. Returning null, does not necessarily mean, that there not be any succeeding message. It may be, that the stream parser has not yet passed the corresponding node in the XML stream. As soon as the succeeding message gets created this message is automatically linked to it, and calls to this method return it.

Method cutLink()

Cuts the link to the succeeding message. After cutting that link, no reference to the succeeding message is hold in this message and a call to getNext() always return null (exception to this is, when this message has not yet been linked to its succeeding message).

Method print(PrintWriter)

Efficiently prints this message to the specified stream.

2.4 Annotations

SPEX Messages, as described in the previous section, can be annotated with SPEX Annotations. The implementation for SPEX Annotations is one of the most important parts of the project. Most of the computation time of the SPEX processor annotation informations are handled, either reading, analyzing, storing or manipulating the annotations of a SPEX Messages. This section explains what the annotations are good for and what way has been chosen for implementing them. There are

basically two different kinds of annotations: simple annotations and head annotations, which are explained at the end of this section. For the sake of simplification, the word *annotation* is used in all cases when *simple annotation* is meant. Figure 8 show a class diagram of the *annotation* package.

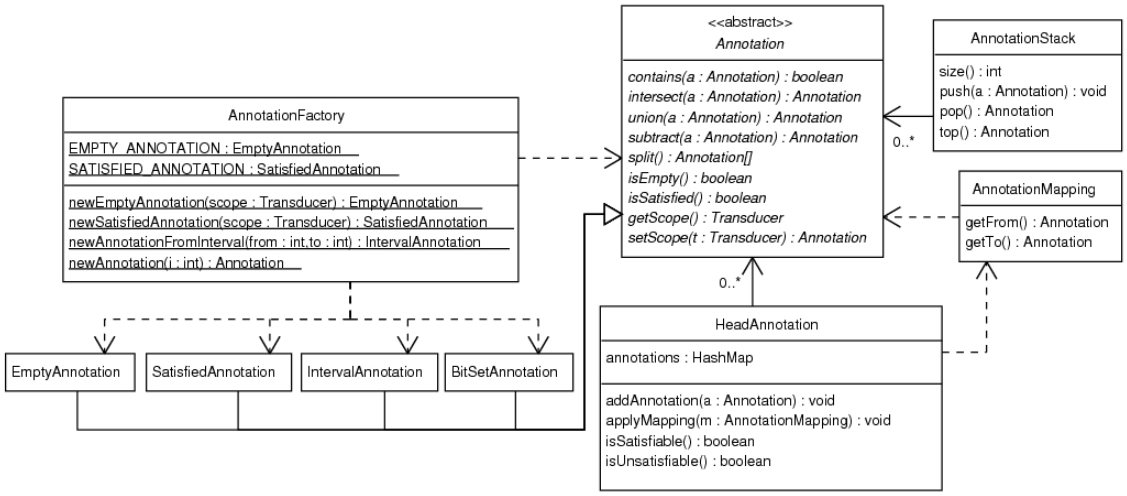

Figure 8: Class diagram for the annotation package

2.4.1 Simple Annotations

An annotation basically is a possibly empty set of integers. One of the integer values (zero) has a special meaning attached, what is be explained soon. At any time, annotations can be in one of two different states, which are *satisfied* and *undetermined*. Annotations containing the special integer (zero) are considered *satisfied*, regardless of which other integer values are contained in that annotation. Annotations neither being empty nor satisfied are called *undetermined* annotations and basically are a finite set of integers. During SPEX processing these annotations get manipulated in many different ways, as for example with simple set operations such as union and intersection and also by mapping some of the values in the annotation to other values.

Some transducers in the SPEX transducer network decide what to do with an incoming message by examining its annotation state (and the state of the topmost annotation on their stack, if they have a stack). For example they may reannotate a message with a new annotation, or just forward it unchanged. Most of the transducers keep track of incoming message annotations with an annotation stack.

Sometimes annotations have a *scope*, which is the *scope begin transducer* (see Section 2.5.6) that created them. In fact, every undetermined annotation always has a scope, because they are created by scope begin transducers only. The only satisfied annotation during SPEX processing is created by the input transducer and only once.

Class Annotation

This is the abstract superclass for all SPEX annotations. There are different implementations for this class, from which some are more efficient than others but are not suitable for all cases (see next Section for more details on this). Use the static *AnnotationFactory* class for creating annotation instances. That factory takes care of which annotation implementation should be used in which cases.

Method isSatisfied()

Returns true if and only if this instance is a satisfied annotation. Otherwise returns false.

Method isEmpty()

Returns true if and only if this annotation is empty. Otherwise returns false.

Method contains(Annotation)

Checks whether this annotation contains all values contained in the specified argument also, and returns true or false for the result. Always returns true if the argument is an empty annotation. Always throws an exception if the argument or this instance is a satisfied annotation (because values contained in that annotation are not specified).

Method intersect(Annotation)

Returns an annotation, which is the intersection of this annotation and the specified argument. The scope of the resulting annotation is the same as of this annotation. Throws an exception if the specified argument or this instance is the satisfied annotation.

Method union(Annotation)

Returns an annotation, which is the union of this annotation and the specified argument. If one of the annotations is the satisfied annotation, the result be the satisfied annotation too. The scope of the resulting annotation is the same as of this annotation. Exception to that is, when this annotation is the empty annotation. Scope of the result is the scope of c.

Method split()

Splits this annotation into an array of annotations, and returns that array. Each cell in the returned array contains an annotation, which contains only one of the values contained in this annotation. If this annotation is empty, the resulting array is empty, too. If this annotation is satisfied, the resulting array only contains this instance. All annotations in the resulting array have the same scope as this annotation.

Method subtract(Annotation)

Returns an annotation, which is the subtraction of this annotation and the specified argument. The resulting annotation includes all values contained in this annotation except the ones, which are contained in the argument also. Returns this instance, if the argument is the empty annotation. The scope of the resulting annotation is the same as of this annotation. Throws an exception if the specified argument or this instance is the satisfied annotation.

Method toString()

Returns the String representation of this annotation. The returned String has the form *"[]"* for empty annotations, *"[$c_1, c_2, ..., c_n$]"* for undetermined annotations where c_i are integers or *"[true]"* for the satisfied annotation.

Method toScopeString()

Same as toString(), but appends the String representation of this annotation's scope also.

Method equals(Object)

Two annotations are equal if and only if they contain the same values, are both the empty annotation, or are both the satisfied annotation. Their scope though may be different.

Method toBitSet()

Returns a BitSet representation of this annotation. All values in this annotation are contained by that bit set, too. If this annotation is the empty annotation, the resulting bit set is empty. If this annotation is the satisfied annotation, the bit set contains the number 0 only. Remember that 0 is a special value and not legal in non-satisfied annotations. Changes to the returned bitset may or may not change this annotation instance, so changes to it are not permitted to maintain immutability.

Method setScope(Transducer)

Returns a new annotation, which is equal to this one except the specified new scope.

Method getScope()

Returns the scope of this annotation.

2.4.2 Implementation Considerations

There are different ways for implementing SPEX annotations. We choose to make all annotation instances immutable. Immutability has both advantages and disadvantages, but in our case the advantages outweigh the disadvantages. During SPEX

processing, an annotation gets manipulated in many different ways, for example by union with another annotation. The immutability prohibits the mutation of the current instance and hence another instance must be returned by these methods. This seems like a big disadvantage but it opens doors for the following design pattern. In many cases an union of annotations (and also other operations) yields either a satisfied or an empty annotation. For example the union of any annotation with the satisfied annotation yields in the satisfied annotation itself. Because annotations are immutable, we can safely use the same instance of the satisfied and empty annotation in these cases. There always be one and only one satisfied (and empty) annotation, and this instance can be used everywhere. But this is not the only advantage provided by the immutability: because operations do not mutate the current instance, but return another instance as the result, we can easily switch between different implementations for annotations. There are several different implementations for the abstract *Annotation* superclass. Some of these implementations are more efficient than others, but can not be used in every case. The immutability permits us to start with the most efficient implementations and switch to another one when necessary. Actually there are the following four implementations for SPEX annotations:

SatisfiedAnnotation This class implements the *Singleton* design pattern. There is always one and only one instance of this class, accessible via a global static variable. This annotation is considered to contain the special integer (zero), and all other contained values are unspecified.

EmptyAnnotation Another singleton class. There is always one and only one instance of this class, accessible via a global static variable. This annotation is considered to contain no integer values.

IntervalAnnotation Most often annotations contain consecutive integer values, hence it is possible to provide an efficient implementation for these cases. An IntervalAnnotation only stores the start and the end value of an interval and represents an annotation containing all values between these two boundaries.

BitSetAnnotation This class has the highest memory consumption, though it is still quite acceptable since the implementation uses a bitset for storing the integers in a very efficient way.

2.4.3 Head Annotations

There is another kind of annotation besides the simple annotation introduced in the previous section, called *Head Annotation*. Basically head annotations are sets containing several simple annotations. The simple annotations contained in a head annotation can have different scopes. In a head annotation several simple annotations may contain the same integer values, if each simple annotation has a different scope.

Some SPEX messages can get annotated with *head annotations* by the so called *head transducers* (see Section 2.5.5). Head transducers succeed the last location step transducer in one branch of the SPEX transducer network and usually (but not necessarily) precede the output transducer. Their job is to decide whether a SPEX

message is a result, a possible result, or no result. This information then is stored in the head annotation of that message. The head annotation provides information about the result state of a message using the following rules:

> If the head annotation of a message contains the simple satisfied annotation, then that message is considered to be a SPEX result.
>
> If the head annotation of a message contains the simple empty annotation only, then that message is considered not to be a SPEX result.
>
> If the head annotation of a message contains simple undetermined annotations only, then that message is a possible result.

Head annotations of possible results change during SPEX processing. When in some time during processing the head annotation of a possible result changes such that it now contains the simple satisfied annotation, then that message finally can be considered to be a result. When the head annotation of possible result changes such that it contains the simple empty annotation only, then that message can be considered to be no result. The result manager (see Section 2.6) takes care of these changes and buffers all possible results as long as necessary.

Class HeadAnnotation

Several different *simple annotation* can be put together into one head annotation. The annotations are stored in a hashmap with their scopes as keys. This means, that for every scope there is at most one annotation in this head annotation. Putting a new annotation with a scope equal to that of an already stored one causes both annotations to be unified and the result to be stored instead of them two. Instances of this class are not immutable like instances of Annotation; usually methods of this class mutate the instance they are invoked on. This class makes best effort to keep time complexity constant: If method contracts are obeyed and no empty annotation is added to this head annotation, one can be sure that time complexity is constant for all operations.

Method isSatisfied()

Returns true if and only if this head annotation contains the simple satisfied annotation. Otherwise returns false. This method computes in constant time. If this method returns true, then *isUnsatisfiable()* always returns false. If *isUnsatisfiable()* returns true, then this method always returns false (transposition).

Method isUnsatisfiable()

Returns true if and only if this head annotation is not satisfiable. This means, that there are no satisfiable annotations in it. This means, that this head annotation contains empty or no annotations at all. This method computes in constant time, if there are no empty annotations stored in the underlying hash map. This is guaranteed if *addAnnotation(Annotation)* is never invoked on the empty annotation. If there are

empty annotations in this hashmap this method computes in linear time (in worst case, that is if all annotations in this head annotation are the empty annotation).

Method addAnnotation(Annotation)

Adds the specified argument to this head annotation. If an annotation with a scope equal to the one of the argument is already stored in this head annotation, then both annotation are unified and the result is stored instead of them. One can ensure that method *isUnsatisfiable()* computes in constant time, if no empty annotations are added via this method.

Method applyMapping(AnnotationMapping)

Applies the specified mapping m to all annotations in this head annotation. There are possibly two cases that can occur:

- If there is no annotation with a scope equal to the scope of *m.getTo()* which also contains *m.getTo()*: Nothing is done and the method returns immediately.
- If an annotation A with scope equal to the scope of *m.getTo()* exists in this head annotation and if A contains *m.getTo()*: First *m.getTo()* is subtracted from A and the result is stored in this head annotation instead of A. If that result is the empty annotation, then it is completely removed from this head annotation. After that, the annotation *m.getFrom()* is added to this head annotation (if it is not an empty annotation). If there already has been an annotation with a scope equal to the scope of *m.getFrom()*, then the both annotations are unified and the result is added to this head annotation instead of them.

Method print(PrintWriter)

Prints the string representation of this head annotation to the specified character stream.

2.4.4 Annotation Mappings

During SPEX processing there is the need for exchanging some annotations by other annotations. For this purpose the AnnotationMapping class has been defined. Annotation mappings are heavily used by several transducers for mutating *head annotations*. Examples for their use can be found in several Sections of this paper.

Class AnnotationMapping

This class represents a mapping from one annotation to another annotation.

Constructor AnnotationMapping(Annotation, Annotation)

Creates a new instance, which maps from the first specified annotation to the second specified annotation.

Method getFrom()

Returns the *from* field of this mapping.

Method getTo()

Returns the *to* field of this mapping.

Method toString()

Returns a string representation of this mapping. The returned string has the form *"[a]->[b]"* for two annotations [a] and [b].

2.5 Transducers

The biggest part of the implementation consists of the transducer network. A transducer basically is a more or less complex pushdown automaton, most often having a stack storing incoming annotations. Every transducer has means for receiving and forwarding SPEX messages. Upon receiving a message, the transducer examines the message's annotation and the annotation on the top of its stack, and so decides how to forward the message to its succeeding transducers. The transducer network is a connection between several such transducers, starting with *InputTransducer* and ending with *OutputTransducer*. An XPath query gets translated into a transducer network by the compiler module not described in further details here. All transducer types are implementing one common interface, the *Transducer*. Figure 9 shows a class diagram of the *transducer* package and its subpackages.

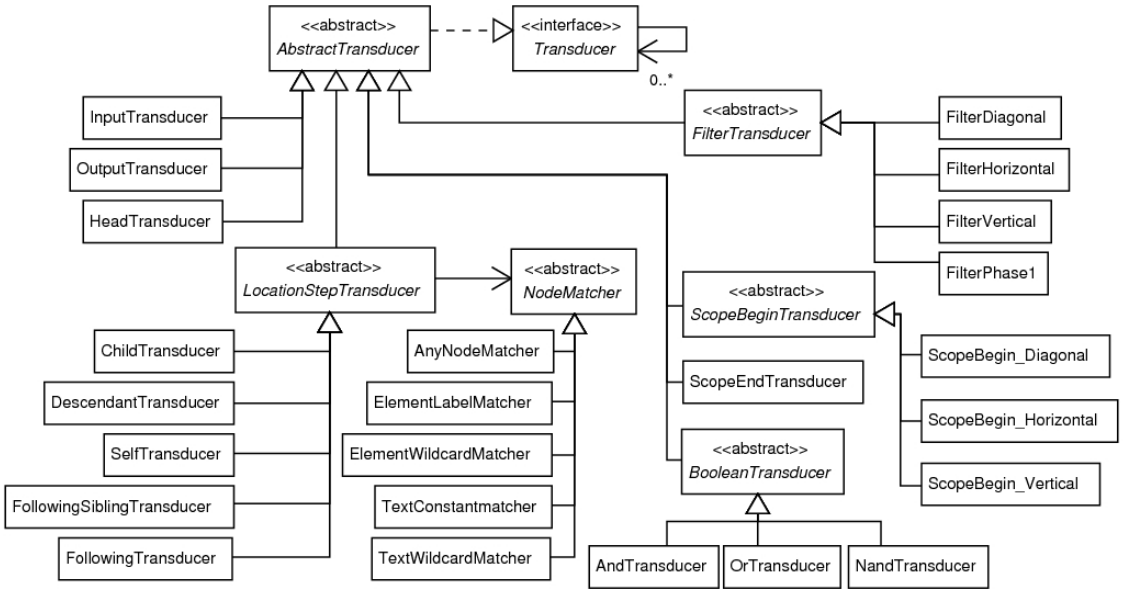

Figure 9: Classdiagram for the transducer package

Interface Transducer

This is the common interface for all transducer types. There are default implementations for some of the methods in this interface, such as the several *out()* methods. These default implementations are provided by the abstract class *AbstractTransducer*, which is extended by most transducer types.

Method equals(Object)

A transducer equals another transducer if they are the same transducer in means of the object identity comparator ==.

Method getLabel()

Returns the name of this transducer. For example if this transducer is a location-step child-transducer then its label is *ChildTransducer*.

Method toString()

Returns a string representing of this transducer. The returned string contains the label of this transducer and several state informations, if there are any.

Method hasNoNextTransducer()

Returns true, if this transducer has no succeeding transducers, and false otherwise. This is the case for the OutputTransducer for example.

Method hasMoreThanOneNextTransducers()

Returns true, if this transducer has more than one succeeding transducer. Returns false otherwise, even if this transducer has no succeeding transducers at all.

Method getNextTransducer()

Returns the succeeding transducer for this transducer. That is the transducer this transducer forwards messages to. If this transducer has more than one succeeding transducer, the method *getNextTransducers()* must be used instead.

Method getNextTransducers()

Returns the array containing the succeeding transducers of this transducer. If this transducer has only one succeeding transducer, the returned array contains the same transducer as returned by the method *getNextTransducer()*. If this transducer has no succeeding transducers, the returned array is empty.

Method setNextTransducer(Transducer)

Sets the succeeding transducer for this transducer. This transducer forwards incoming messages to that transducer.

Method setNextTransducers(Transducer[])

Sets the succeeding transducers of this transducer. This transducer forwards incoming messages to these transducers.

Method insert(Transducer)

Inserts the specified transducer after this transducer. The current succeeding transducers of this transducer becomes the succeeding transducers of the specified argument.

Method insert(Transducer[])

Inserts new succeeding transducers for this transducer.
this.getNextTransducers()[i] becomes the succeeding transducer of the i^{th} transducer in the specified array argument.
this.getNextTransducers().length must be the same as the length of the specified array argument.

Method in(OpeningTagMessage, Transducer)

The transducer handles incoming *opening tag messages* via this method. The transducer argument specifies the originator of that message.

Method in(ClosingTagMessage, Transducer)

The transducer handles incoming *closing tag messages* via this method. The transducer argument specifies the originator of that message.

Method in(AttributeMessage, Transducer)

The transducer handles incoming *attribute messages* via this method. The transducer argument specifies the originator of that message.

Method in(TextMessage, Transducer)

The transducer handles incoming *text messages* via this method. The transducer argument specifies the originator of that message.

Method in(DocumentStartMessage, Transducer)

The transducer handles incoming *document start messages* via this method. The transducer argument specifies the originator of that message.

Method in(DocumentEndMessage, Transducer)

The transducer handles incoming *document end messages* via this method. The transducer argument specifies the originator of that message.

Method out(OpeningTagMessage)

Forwards the specified message to the succeeding transducers of this transducer. That is: the corresponding *in()* method of the succeeding transducers is invoked. There is a default implementation for this method in the abstract class *AbstractTransducer*.

Method out(ClosingTagMessage)

Forwards the specified message to the succeeding transducers of this transducer. That is: the corresponding *in()* method of the succeeding transducers is invoked. There is a default implementation for this method in the abstract class *AbstractTransducer*.

Method out(AttributeMessage)

Forwards the specified message to the succeeding transducers of this transducer. That is: the corresponding *in()* method of the succeeding transducers is invoked. There is a default implementation for this method in the abstract class *AbstractTransducer*.

Method out(TextMessage)

Forwards the specified message to the succeeding transducers of this transducer. That is: the corresponding *in()* method of the succeeding transducers is invoked. There is a default implementation for this method in the abstract class *AbstractTransducer*.

Method out(DocumentStartMessage)

Forwards the specified message to the succeeding transducers of this transducer. That is: the corresponding *in()* method of the succeeding transducers is invoked. There is a default implementation for this method in the abstract class *AbstractTransducer*.

Method out(DocumentEndMessage, Transducer)

Forwards the specified message to the succeeding transducers of this transducer. That is: the corresponding *in()* method of the succeeding transducers is invoked. There is a default implementation for this method in the abstract class *AbstractTransducer*.

Method setObserver(Observer)

This is the connection between *model* and *view*, to be used by graphical user interfaces visualizing SPEX transducer networks. The observer be notified every time this transducer changes its state, by calling the observers *update(Observable,Object)* method.

2.5.1 Transducing Rules Formalism

Different transducer types have different rules regarding annotating and forwarding of messages. A transducing rule basically consists of a left and a right side. When a transducer receives a message and its state matches the left side of a rule, then the right side of that rule declares what the transducer must do. For example the left side may contain the kind of the received message, its annotation and the current state of the transducer's annotation stack. The right side may then cause the transducer to mutate the annotation stack, change the message's annotation and forward the

message to its succeeding transducers. The following formalism is used on the left or right side of any rule:

`in(X Y)` The transducer received a message X with annotation Y. X can be any of the messages below and Y can be any of the annotations below. The annotation may be missing also, because not all messages have an annotation.

`out(X Y)` The transducer forwards the message X with annotation Y to all succeeding messages. X can be any of the messages below and Y can be any of the annotations below. The annotation may be missing also, because not all messages have an annotation.

`<document start>` The document start message.

`<document end>` The document end message.

`<X>` An opening tag message with label X.

`</X>` A closing tag message with label X.

`@X=Y` An attribute message with name X and value Y.

`'Text'` A text message representing a text node 'Text'.

`[ ]` The empty annotation.

`[T]` The satisfied annotation.

`[a]` An undetermined annotation neither being the empty nor the satisfied annotation.

`[_]` Any annotation.

`matches(X)` Indicates whether this location step transducer's node matcher matches the message X (for more information about node matchers see Section 2.5.4)

`stack(X)` Indicates the state of the stack. X is the whole stack.

`stack(X::Y)` Indicates the state of the stack. X is the topmost annotation on the stack and Y is the remainder of the stack.

`[a]|[b]` The union of two annotations [a] and [b].

2.5.2 Input and Output

A SPEX transducer network begins with the *InputTransducer* and ends with the *OutputTransducer*. These transducers are the most simple transducers of all, not having a stack and very simple rules for forwarding messages.

Input Transducer

The input transducer simply forwards most of the incoming messages to its succeeding transducers after annotating them with the *empty annotation*. The only exception to this is the *document start message*, which is annotated with the *satisfied annotation* before it is forwarded to the succeeding transducers. Usually the input transducer has one succeeding transducer only, but this is not a general condition.

$$
\begin{array}{lcl}
in(< documentstart >) & \longrightarrow & out(< documentstart > [T]) \\
in(< documentend >) & \longrightarrow & out(< documentend > []) \\
in(< x >) & \longrightarrow & out(< x > []) \\
in(< /x >) & \longrightarrow & out(< /x >) \\
in('Text') & \longrightarrow & out('Text'[]) \\
in(@att = val) & \longrightarrow & out(@att = val)
\end{array}
$$

Output Transducer

This is the last transducer in a SPEX transducer network, though it does not mark the end of the SPEX processing. This transducer forwards some of the incoming messages, i.e. the result candidates, to the result manager, which potentially buffers, prints or discards them. More information about the result manager can be found in Section 2.6. The following transducing rules use a slightly modified formalism, different to that described in Section 2.5.1. The used formalism is quite self explanatory. The only new formalism that needs explanation is $in([a] \rightarrow [b])$, which means that the output transducer received an annotation mapping. More information about annotation mappings and what they are good for can be found in Sections 2.5.6 and 2.6. One may notice that the annotations of the received and forwarded messages are completely unspecified in the following rules. The reason for this is, that from now on the simple annotation of a SPEX message is irrelevant. The result manager does not examine them, but instead analyzes and mutates the *head annotation* of every message only. One may further notice, that some messages are never forwarded to the result manager, although they surely are parts of possible results. Though this does not limit the printing of results, because all messages are linked together as described in Section 2.3. Only beginnings of possible results are forwarded to the result manager, that is opening tag messages and text messages.

$$
\begin{array}{lcl}
in(< documentstart >) & \longrightarrow & resultmanager.initiate() \\
in(< documentend >) & \longrightarrow & resultmanager.terminate() \\
in(< x > [\_]) & \longrightarrow & resultmanager.register(< x > [\_])\& \\
 & & resultmanager.print() \\
in(< /x >) & \longrightarrow & resultmanager.print() \\
in('text'[\_]) & \longrightarrow & resultmanager.register('text'[\_])\& \\
 & & resultmanager.print() \\
in(@att = val) & \longrightarrow & resultmanager.print() \\
in([a] \rightarrow [b]) & \longrightarrow & resultmanager.apply([a] \rightarrow [b])\& \\
 & & resultmanager.print()
\end{array}
$$

2.5.3 Location Steps

The simplest transducer network is a linear concatenation of an input transducer, several *location step transducers*, one head transducer and one output transducer. This kind of transducer network represents a linear XPath query, i.e. the query does not contain any predicates or unions but contains XPath location steps only. XPath location steps consist of one XPath axis and one XPath nodetest. There are several different transducer implementations for each XPath axis. All these transducer types have one common superclass, the *LocationStepTransducer*, which

centralizes several functionalities needed in almost every location step transducer. For example location step transducers usually are in need of an annotation stack; the transducing rules for these transducers heavily depend on the current state of their annotation stack. The abstract superclass *LocationStepTransducer* is not explained in further detail, as it contains straight forward functionalities only. The remainder of this section explains the five different location step transducer, which actually are *ChildTransducer*, *DescendantTransducer*, *FollowingSiblingTransducer*, *FollowingTransducer* and *SelfTransducer*. Remember that we actually are dealing with *ForwardXPath* queries, which do not contain any reverse axes.

SelfTransducer

This is the simplest location step transducer, representing the XPath axis *self*. It does not need any stack of annotations and its transducing rules are very simple. When the transducer's node matcher matches the incoming message, then the message's annotation is not changed, otherwise the message is annotated with the empty annotation before forwarding it to the succeeding transducers.

$$
\begin{array}{ll}
in(< documentstart > [a])\& & \longrightarrow \quad out(< documentstart > [a]) \\
matches(< documentstart >) & \\
in(< documentstart > [\_])\& & \longrightarrow \quad out(< documentstart > []) \\
notmatches(< documentstart >) & \\
in(< documentend >) & \longrightarrow \quad out(< documentend >) \\
in(< x > [a])\& & \longrightarrow \quad out(< x > [a]) \\
matches(< x >) & \\
in(< x > [\_])\& & \longrightarrow \quad out(< x > []) \\
notmatches(< x >) & \\
in(< /x >) & \longrightarrow \quad out(< /x >) \\
in('Text'[a])\& & \longrightarrow \quad out('Text'[a]) \\
matches('Text') & \\
in('Text'[\_])\& & \longrightarrow \quad out('Text'[]) \\
notmatches('Text') & \\
in(@att = val) & \longrightarrow \quad out(@att = val)
\end{array}
$$

ChildTransducer

This transducer represents the XPath axis *child*. It stores incoming annotations on a stack of annotations, and decides how to forward a message by examining this stack in addition to the message's annotation.

$$
\begin{array}{lcl}
in(< documentstart > [a])\& & \longrightarrow & stack([a] :: s)\& \\
stack(s) & & out(< documentstart > []) \\
in(< documentend >) & \longrightarrow & out(< documentend >) \\
in(< x > [a])\& & \longrightarrow & out(< x > [top])\& \\
matches(< x >)\& & & stack([a] :: [top] :: tail) \\
stack([top] :: tail) & & \\
in(< x > [a])\& & \longrightarrow & out(< x > [])\& \\
notmatches(< x >)\& & & stack([a] :: s) \\
stack(s) & & \\
in(< /x >)\& & \longrightarrow & stack(tail)\& \\
stack([top] :: tail) & & out(< /x >) \\
in('text'[\_])\& & \longrightarrow & out('text'[top]) \\
matches('text')\& & & \\
stack([top] :: tail) & & \\
in('text'[\_])\& & \longrightarrow & out('text'[]) \\
notmatches('text') & & \\
in(@att = val) & \longrightarrow & out(@att = val)
\end{array}
$$

DescendantTransducer

This transducer represents the XPath axis *descendant*. It stores incoming annotations on a stack of annotations, and decides how to forward a message by examining this stack in addition to the message's annotation.

$$
\begin{array}{lcl}
in(< documentstart > [a])\& & \longrightarrow & stack([a] :: s)\& \\
stack(s) & & out(< documentstart > []) \\
in(< documentend >) & \longrightarrow & out(< documentend >) \\
in(< x > [a])\& & \longrightarrow & out(< x > [top])\& \\
matches(< x >)\& & & stack([a]|[top] :: [top] :: tail) \\
stack([top] :: tail) & & \\
in(< x > [a])\& & \longrightarrow & out(< x > [])\& \\
notmatches(< x >)\& & & stack([a]|[top] :: [top] :: tail) \\
stack([top] :: tail) & & \\
in(< /x >)\& & \longrightarrow & stack(tail)\& \\
stack([top] :: tail) & & out(< /x >) \\
in('Text'[\_])\& & \longrightarrow & out('text'[top]) \\
matches('Text')\& & & \\
stack([top] :: tail) & & \\
in('Text'[\_])\& & \longrightarrow & out('Text'[]) \\
notmatches('Text') & & \\
in(@att = val) & \longrightarrow & out(@att = val)
\end{array}
$$

FollowingSiblingTransducer

This transducer represents the XPath axis *following-sibling*. It stores incoming annotations on a stack of annotations, and decides how to forward a message by examining this stack in addition to the message's annotation.

$$
\begin{array}{lcl}
in(< documentstart > [\_])\& & \longrightarrow & stack([] :: s)\& \\
stack(s) & & out(< documentstart > []) \\
in(< documentend >) & \longrightarrow & out(< documentend >) \\
in(< x > [a])\& & \longrightarrow & out(< x > [top])\& \\
matches(< x >)\& & & stack([] :: [a]||[top] :: tail) \\
stack([top] :: tail) & & \\
in(< x > [a])\& & \longrightarrow & out(< x > [])\& \\
notmatches(< x >)\& & & stack([] :: [a]||[top] :: tail) \\
stack([top] :: tail) & & \\
in(< /x >)\& & \longrightarrow & stack(tail)\& \\
stack([top] :: tail) & & out(< /x >) \\
in('Text'[a])\& & \longrightarrow & out('Text'[top])\& \\
matches('Text')\& & & stack([a]||[top] :: tail) \\
stack([top] :: tail) & & \\
in('Text'[a])\& & \longrightarrow & out('Text'[])\& \\
notmatches('Text')\& & & stack([a]||[top] :: tail) \\
stack([top] :: tail) & & \\
in(@att = val) & \longrightarrow & out(@att = val)
\end{array}
$$

FollowingTransducer

This transducer represents the XPath axis *following*. It stores incoming annotations on a stack of annotations, and decides how to forward a message by examining this stack in addition to the message's annotation.

$$
\begin{array}{lcl}
in(< documentstart > [\_])\& & \longrightarrow & stack([] :: s)\& \\
stack(s) & & out(< documentstart > []) \\
in(< documentend >) & \longrightarrow & out(< documentend >) \\
in(< x > [a])\& & \longrightarrow & out(< x > [top])\& \\
matches(< x >)\& & & stack([top] :: [a] :: tail) \\
stack([top] :: tail) & & \\
in(< x > [a])\& & \longrightarrow & out(< x > [])\& \\
notmatches(< x >)\& & & stack([top] :: [a] :: tail) \\
stack([top] :: tail) & & \\
in(< /x >)\& & \longrightarrow & stack([top1]||[top2] :: tail)\& \\
stack([top1] :: [top2] :: tail) & & out(< /x >) \\
in('Text'[a])\& & \longrightarrow & out('Text'[top])\& \\
matches('Text')\& & & stack([top]||[a] :: tail) \\
stack([top] :: tail) & & \\
in('Text'[a])\& & \longrightarrow & out('text'[])\& \\
notmatches('Text')\& & & stack([top]||[a] :: tail) \\
stack([top] :: tail) & & \\
in(@att = val) & \longrightarrow & out(@att = val)
\end{array}
$$

2.5.4 Node Matcher

As described previously, an XPath location step combines an XPath axis and one XPath node test. In the current implementation, these steps are represented by subclasses of the *LocationStepTransducer* class introduced in the previous section. For

each axis there is one transducer implementation, and for each step one transducer instance gets created. These instances encapsulate both, the axis and the node test of that step. However it is theoretically possible to distinguish between axes and node tests. There could have been a separate transducer type for each XPath node test, and because of this the node matchers are explained in this separate section.

Basically a SPEX *NodeMatcher* corresponds to an XPath node test, but also can contain more information than can be expressed with an ordinary XPath node test. *NodeMatcher* is a SPEX interface defining several methods for testing whether that node matcher matches a particular XML node.

Interface NodeMatcher

Method matches(OpeningTagMessage)

Tests whether this node matcher matches the specified opening tag message.

Method matches(ClosingTagMessage)

Tests whether this node matcher matches the specified closing tag message.

Method matches(AttributeMessage)

Tests whether this node matcher matches the specified attribute message.

Method matches(TextMessage)

Tests whether this node matcher matches the specified text message.

Method matches(DocumentStartMessage)

Tests whether this node matcher matches the specified document start message.

Method matches(DocumentEndMessage)

Tests whether this node matcher matches the specified document end message.

Currently there are several different implementations for this interface, some corresponding to ordinary XPath node tests, others corresponding to more complex XPath expressions. Theoretically it is possible to add further arbitrary node matcher implementations.

AnyNodeMatcher Corresponds to the XPath node test *node()*. This matcher matches any node.

ElementWildcardMatcher Corresponds to the XPath node test *. This matcher matches any element node. That is any opening or closing tag message with any label.

ElementLabelMatcher Corresponds to the XPath node test *a* for any label a. This matcher matches any element node with the given label. That is any opening or closing tag message with that label.

TextNodeWildcardMatcher Corresponds to the XPath node test *text()*. This matcher matches any text node. That is any text message.

TextNodeConstantMatcher Corresponds to a more complex XPath expression containing the node test *text()* and a comparison operator. This matcher matches any text node, whose contents match a specific comparison.

2.5.5 Head Transducer

The head transducer always succeeds the last location step transducer in a SPEX network of transducers, i.e. it succeeds the transducer corresponding to the query's last location step outside of predicates. When the network is linear, i.e. there are no predicates in the XPath query, the head transducer precedes the output transducer. Otherwise there may be a number of *scope end* and *boolean* transducers between head and output transducer. A head transducer's job is to initialize a message's *head annotation*, as described in Section 2.4.3. A head transducer determines a received message's result state, which at that time is encoded in its simple annotation, and encodes this information in the message's head annotation. The necessity for the transition from simple annotations to head annotations is explained in Section 2.5.6. If an XPath query always were linear, not having any predicates, this necessity would not exist. In the current implementation the head transducer has a shortening way to the output transducer. If the output transducer is not its succeeding transducer, this shortening way can be used to forward messages directly to the output transducer instead of the succeeding transducer. This is done for all messages, which are not of any importance for the succeeding boolean and scope end transducers.

$$
\begin{array}{lcl}
in(< documentstart >) & \longrightarrow & outputTransducer.in(< documentstart >) \\
in(< documentend >) & \longrightarrow & outputTransducer.in(< documentend >) \\
in(< x > [a]) & \longrightarrow & initHead(< x >, [a])\& \\
 & & outputTransducer.in(< x >)\& \\
 & & out(< x > [a]) \\
in(< /x >) & \longrightarrow & outputTransducer.in(< /x >) \\
in('Text'[a]) & \longrightarrow & initHead('Text', [a])\& \\
 & & outputTransducer.in('Text')\& \\
 & & out('Text'[a]) \\
in(@att = val) & \longrightarrow & outputTransducer.in(@att = val)
\end{array}
$$

2.5.6 Scopes & Boolean Transducers

XPath *predicates* are capable of splitting the XPath query into several subqueries, hence also SPEX must provide means of *splitting* the transducer network. This is done with *scopes*. A scope consists of a pair of (*scope-begin*, *scope-end*) and several *boolean* and location step transducers inside the scope. For each predicate in the query there is a corresponding scope in the network. As predicates in XPath can be nested, also scopes in our transducer network may be nested.

Scope-Begin Transducer

Basically a scope-begin transducer has several succeeding transducers, there is one outgoing path for every subquery in the original XPath query. There are several kinds of scope-begin transducers with slightly different implementations. The most simple one, the *Scope-Begin Diagonal*, is explained first. This is the least efficient one. It must be used, if the XPath query contains any following axes. In the following this transducer is simply referred to as *scope-begin*.

A scope-begin basically maps every non-empty annotation it receives to an annotation containing consecutive numbers. I.e. the first non-empty annotation is mapped to [1], the second non-empty annotation is mapped to [2] and so on. The scope-begin further stores every mapping it does on a mappings-stack, because they are needed later. This is also the only transducer that explicitly sets the *scope* of the outgoing annotations to be itself (hence the name *scope of an annotation*).

$$
\begin{array}{lcl}
in(< documentstart > [])\& & \longrightarrow & stack([] \rightarrow [] :: s)\& \\
stack(s) & & out(< documentstart > []) \\
in(< documentstart > [a])\& & \longrightarrow & stack([a] \rightarrow [n] :: s)\& \\
stack(s)\&n = nextNumber() & & out(< documentstart > [n]) \\
in(< documentend >)\& & \longrightarrow & stack(t)\& \\
stack(h :: t) & & out(< documentend >) \\
in(@att = val) & \longrightarrow & out(@att = val) \\
in(< /x >) & \longrightarrow & out(< /x >) \\
in(< x > []) & \longrightarrow & out(< x > []) \\
in(< x > [a])\& & \longrightarrow & stack([a] \rightarrow [n] :: s)\& \\
stack(s)\&n = nextNumber() & & out(< x > [n]) \\
in('text'[]) & \longrightarrow & out(< x > []) \\
in('text'[a])\& & \longrightarrow & stack([a] \rightarrow [n] :: s)\& \\
stack(s)\&n = nextNumber() & & out('text'[n])
\end{array}
$$

The other two scope-begin transducers are *scope-begin vertical* and *scope-begin horizontal*. They are more efficient than the first one, because they are removing mappings from the stack in situations these mappings are not needed any longer. Besides of this, these transducers are all the same. The *scope-begin horizontal* is used, if the scope-begin transducer is succeeded by a horizontal axis (following-sibling transducer), else the most efficient *scope-begin vertical* can be used. Note that neither of both can be used, if there are any following transducers succeeding this scope transducer.

Boolean Transducers

A scope-begin splits the transducer network into several paths. These paths run together in *boolean transducers*. For every predicate in an XPath query, there is at least one boolean transducer, the *and transducer*, in the network of transducers. When a message receives the first boolean transducer in a network, the message and its annotation get separated. The message gets forwarded directly to the out-transducer. Whether or not the annotations get forwarded depends on the kind of boolean transducer, what is explained below. Note that although the messages lose

their *simple annotations*, they may have been annotated with a *head annotation*, which they are keeping.

The simplest network with scopes corresponds to an XPath query containing one predicate that contains one linear subquery path. The corresponding scope-begin transducer has two succeeding transducers. One path for the query outside of the predicate and one path for the subquery inside of the predicate. For a node to be selected by this XPath query, both paths must fulfill a condition, hence an and-transducer is used. Or-transducers on the other hand are used in situations, in which the XPath or-operator was used inside of a predicate. Both boolean transducers are explained in further detail below.

The different boolean transducers differ in when they are forwarding annotations to their succeeding transducer. An and-transducer forwards an annotation only if it receives this annotation from all its preceding transducers. The or-transducer forwards an annotation as soon as it is received from any preceding transducer. It is possible to define further boolean transducers, which forward annotations depending on different conditions, such as nand-transducer, nor-transducer and so on. Usually an annotation runs through several boolean transducers until it reaches the *scope-end* transducer.

Scope-End Transducer

A scope-end transducer only receives annotations (messages get forwarded to the out-transducer directly by the first boolean transducer). The scope-end transducer is quiet simple. All it has to do is reverse the mapping done by its corresponding scope-begin transducer.

$$\begin{array}{ll} in([a])\&([b] \rightarrow [a]) = & \longrightarrow \quad outputTransducer.in([b] \rightarrow [a])\& \\ scopeBegin.getMapping([a]) & \quad\quad\; out([b]) \end{array}$$

There may be several consecutive scope-end transducers preceding the out-transducer. Every scope-end only receives annotations created by its corresponding scope-begin, because every scope-end is reversing the mapping done by its scope-begin. This way all mappings done by any scope-begins get reversed.

2.6 Query Results

The SPEX processor can be configured to compute results in several different forms. The most common form is to print all XML nodes selected by the XPath query. There is also the possibility to just count the number of selected nodes or just return a boolean value indicating whether the query selected anything. SPEX defines one common interface, the *ResultManager*, which encapsulates commonalities between all of these different result modes. In the following only one of these result manager types are discussed, as the others are very similar or more trivial in theory. The discussed result manager type is the *ResultBuffer*, which is the one printing all selected XML nodes. For the sake of simplicity we just call it the *result manager*.

The result manager is the last module in SPEX processing, and manages analyzing, buffering, printing and discarding of possible results. It is used by the

OutputTransducer, what has been discussed in Section 2.5.2 already. Some messages received by the output transducer get registered to the result manager. These messages are called *result candidates* and are annotated with the *head annotation*. Actually there is an interface *ResultCandidate*, which must be implemented by every message, that shall be capable of being registered to the result manager and being annotated with a head annotation. It is not always possible to determine whether a result candidate is an actual candidate at the time of its registration to the result manager. For example the result candidate may depend on an XPath predicate, which not yet has been evaluated. Because of this, the result manager must have means of buffering result candidates. Furthermore there is the possibility, that for a result candidate B, which has been registered after a result candidate A, it is assured that this candidate B is an actual result even though it is not yet known whether A is an actual result or not. In these cases B can not be printed yet, because XPath specifies that results must be printed in document order. This means for B, that it must wait for A to become an actual result or get discarded. The current implementation of the result manager does not use more memory space than is needed theoretically, but it may not discard result candidates as soon as possible. Though it is assured that the current implementation discard result candidates that can be discarded, before any new candidates get registered to the manager. The following class description describes all methods used by the output transducer.

Class ResultBuffer

This result manager stores every result candidate and all of its succeeding messages that are part of the result in a buffer. The references to the messages are cut off as soon as they are not needed anymore, making them capable of being collected by the garbage collector. Not being needed means, that this messages are not parts of any result candidates.

Printing of results always happens in document order. This means, that later candidates that are assured to be results have to wait for earlier candidates, for which it is not yet known if they are results or not. This is done with a first-in first-out queue for the result candidates. For each result of the SPEX processing this manager prints

```
<spex:result>
  result
</spex:result>
```

where result is the part of the xml input stream, that is the result. At any time, this result manager can be in one of the following two states:

WAITING

For the next result candidate in the queue it is not yet known whether it is a result or not. The result manager has to wait in these cases.

PRINTING_RESULT

For the next result candidate in the queue it is assured that it is a result. It is printed and all succeeding messages, which are part of that result are printed, too. It can happen, that a succeeding message, which is part of the result, has not yet been created (the stream parser has not yet processed that node of the XML stream).

The result manager remain in the *PRINTING_RESULT* state until the remainder of the result has been created and printed. When the entire result has been printed, the result manager returns to the *WAITING* state.

Method initiate(PrintWriter)

Called by the output transducer upon receiving the document start message. This method prints

```
<spex:results xmlns:spex="http://www.pms.ifi.lmu.de/spex/">\n
```

to the specified result stream. This method saves a reference to the specified results stream for further use by other methods.

Method terminate()

Called by the output transducer upon receiving the document end message. This method first prints all remaining results and then prints

```
\n</spex:results>
```

to the results stream.

Method register(ResultCandidate candidate)

Called by the output transducer upon receiving result candidates, i.e. opening tag messages and text messages. This method registers the specified result candidate to this manager. If the candidate's head annotation is satisfiable (see Section 2.4.3) it is stored in a first in first out queue for candidates.

Method getNumberOfCandidates()

Returns the size of the candidates queue.

void printResultCandidates(PrintWriter,int)

Prints a specified number of result candidates currently buffered by this manager, regardless of whether they are actually results or not, to the specified character stream. Only the starting messages of the results are printed, and their annotations are printed also. This method can be used by visualizers or for debugging purposes.

Method printResultsAndClean(PrintWriter,boolean)

Called by the output transducer every time there is the possibility that some result candidates can finally be printed or discarded, i.e. when new candidates have been registered or when new annotation mappings have been applied. This method examines the head annotation of the first result candidate in the candidates queue. There are several cases that can occur:

- The result candidates queue is empty. In this case there is nothing to do and this method returns immediately.
- The head annotation of the first result candidate is satisfied. In this case, the candidate is removed from the queue and as much of the result as possible is printed. If there are parts of the result, that could not be printed (because they have not yet been processed by the stream parser), this result manager goes into the *PRINTING_RESULT* state and this method returns. The remainder of the result then be printed with succeeding calls to this method. If all parts of the result were able to be printed, this method calls itself recursively.
- The head annotation of the first result candidate is unsatisfiable. In this case the candidate is discarded and this method calls itself recursively.
- The head annotation of the first result candidate is neither satisfied nor unsatisfiable. In this case the result manager can not do anything. It remains in the *WAITING* state and this method returns.

Method applyMapping(AnnotationMapping)

Called by the output transducer every time an annotation mapping has been received. The specified annotation mapping is applied to the head annotations of all result candidates in this managers candidates queue. Result candidates with resulting unsatisfiable head annotations are removed from the queue with succeeding calls to *printResultsAndClean()*. Result annotations with resulting satisfied head annotations are printed to the results stream with succeeding calls to *printResultsAndClean()*.

3 Compiletime Optimizations

Optimizations done at compiletime are able to improve the performance of the SPEX processor at runtime. *Compiletime optimization* means, that the transducer network gets improved, before the processing starts. There are several ways of optimizing the transducer network. One way is minimization by factoring out similar transducers, what is demonstrated in Figure 10.

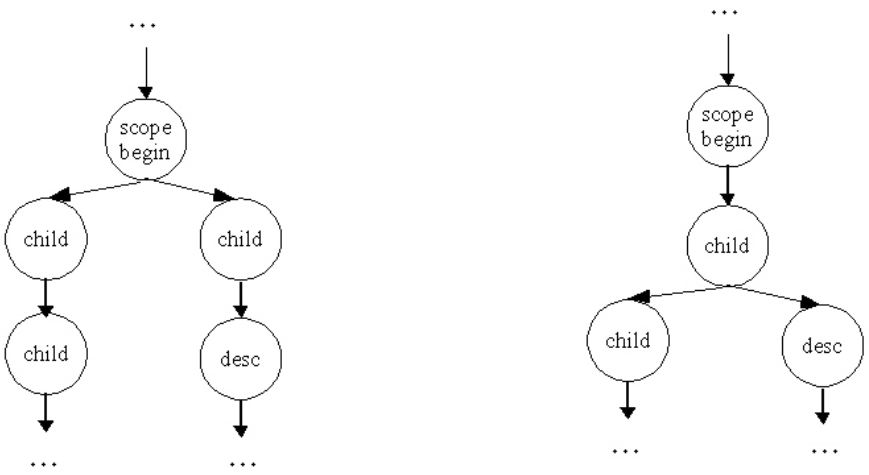

Figure 10: Optimization by minimizing of transducer network

In this figure the original query contains a *scope begin* transducer, which is succeeded by two equal *child* transducers. The optimization factors out these two *child* transducers what yields in an improved network with one less transducer.

3.1 Filtering the Stream

The current implementation realizes another way of optimization: the insertion of several structural filter transducers into the network of transducers, in order to minimize the stream traffic in the network.

This is exemplified on a stream containing information about articles followed by information about books followed by information about papers. Figure 11 demonstrates such a stream. Consider a SPEX network corresponding a query asking about authors of books, as demonstrated by Figure 11. In this conventional transducer network all nodes of the example stream are forwarded to all transducers, regardless of them being a result or an important part of the stream. This way all nodes from the stream reach all transducers from the network.

```
<article>
    <title>An Article</title>
    <author>Some Author</author>
</article>

<-- possibly tons of further articles
    listed here -->

<book>
    <title>A Book</title>
    <price>its price</price>
    <author>The Author</author>
</book>

<-- more books listed here -->

<paper>
    <name>Papername</paper>
    <date>a date</date>
</paper>

<-- more papers listed here -->
```

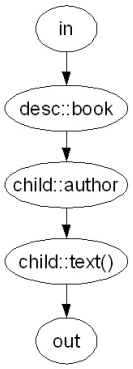

Figure 11: Example stream containing information about articles and books and example query network

However this is not necessary. The current SPEX implementation introduces two ways of filtering the stream traffic between the transducers, called *Phase One* and *Phase Two*. In Phase One the so called *diagonal filters* are used. In our example two of these filters are placed right after the transducers *desc::book* and *child::author*. The first filter sends further only that part of the stream following the first *book* node and the second filter sends further only that part of the stream following the first *author* node it receives. Figure 12 demonstrates the altered transducer network and our example stream. In this stream only the marked parts are forwarded by the filter transducers.

```
<article>
    <title>An Article</title>
    <author>Some Author</author>
</article>

<-- possibly tons of further articles
    listed here -->

<book>                          | DFilter1
    <title>A Book</title>       |
    <price>its price</price>    |
    <author>The Author</author> | | DFilter2
</book>                         | |
                                | |
<-- more books listed here -->  | |
                                | |
<paper>                         | |
    <name>Papername</paper>     | |
    <date>a date</date>         | |
</paper>                        | |
                                | |
<-- more papers listed here --> | |
```

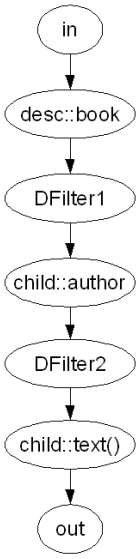

Figure 12: Phase One filtering using Diagonal Filters

In Phase One only diagonal filters are used, which once started to forwarding the stream, forward it until the stream ends. Even more is done in Phase Two with *vertical filters*. Assuming the transducers receiving nodes form the book-transducer look for nodes to be found only inside the fragments corresponding to book-nodes (as is the case in our example with the author node inside of book nodes). Then, a vertical filter can safely send further only such stream fragments corresponding to book-nodes. This is demonstrated in Figure 13.

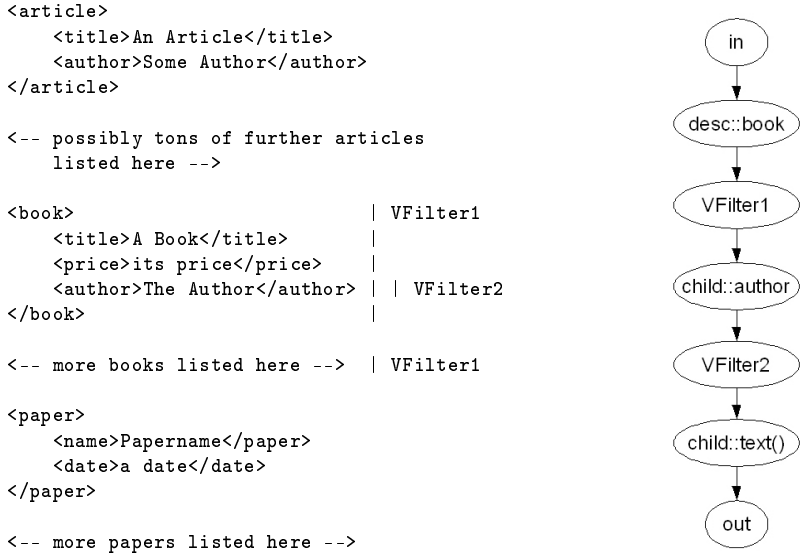

Figure 13: Phase Two filtering using Vertical Filters

Phase Two filtering is more powerful then Phase One filtering, however, unlike the diagonal filters, the vertical filters can not be used in every transducer network. Consider an example query containing the *following* axis. The corresponding transducer depends on receiving the whole stream until the end of the stream, because each node in the stream may be the result of a query containing the *following* axis. For a similar reason, Phase Two filtering can not be used for queries containing the *following-sibling* axis. Though it is possible to introduce a third kind of appropriate filters here, the *horizontal filters*, which are less powerful then *vertical filters* but more powerful than *diagonal filters*.

3.2 Performance Tests

The realized filtering has been tested with several XML documents and thousands of different XPath queries. The following two figures show results of these performance tests. Figure 14 shows a digram of the time needed for SPEX processing in dependence of the length of the XPath query. Thousands of queries have been tested with various path lengths from 5 to 1000 steps. The XML document tested was hundreds of megabytes of length. The diagram demonstrates time results for a usual transducer network, a network with phase-one filters and a network with phase-two filters. As one can see, the processing without any filters is linear in complexity depending on the length of the input query. One can further observe, that the processing of a transducer network with filters included takes much less time.

Figure 15 shows another result. The question must be answered, whether all these filter transducers could have an overhead impact on the processing in cases where they are of minor use. For this purpose the tested XPath queries were generated in such a way, that the filters can not filter out much of the XML stream. The diagram demonstrates that the more descendant axes occur in the input query, the less benefit is gained by the filter transducers. In the worst case (descendant axes 100 %) the processing takes even more time with filters. However the overhead impact is very small and can be disregarded.

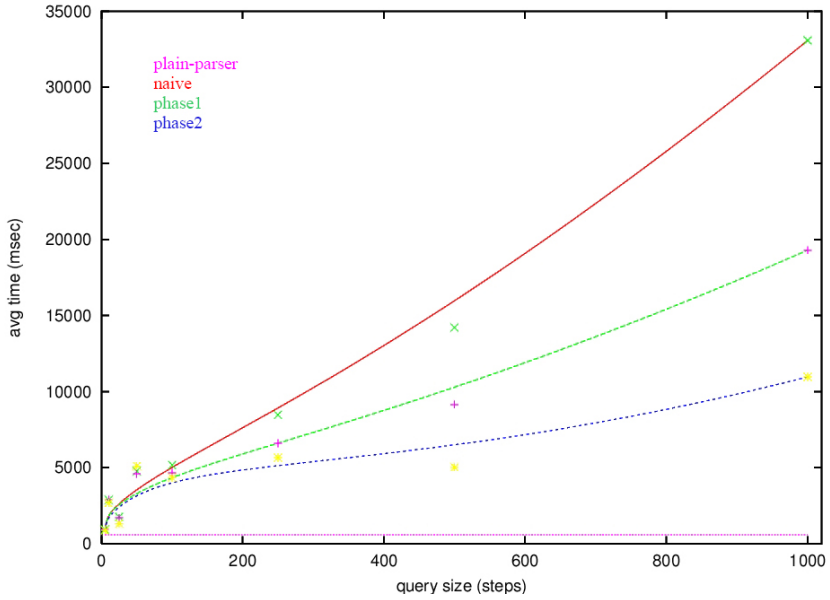

Figure 14: Linear and better time complexity

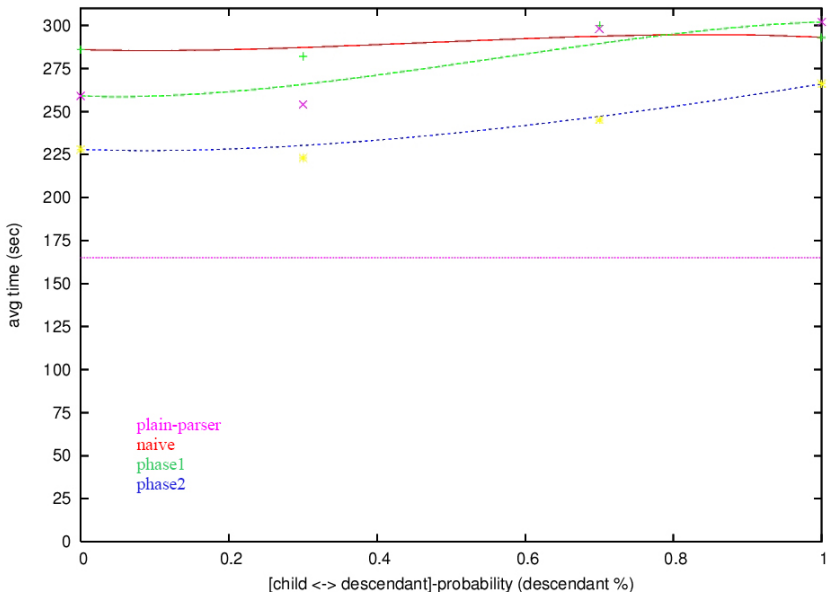

Figure 15: Overhead impact of filters can be disregarded

4 SPEX Project

The SPEX Project at [SF] includes several subprojects, such as [REXP] and several visualizers, called [REXP-GUI] and [SPEX-GUI]. The ReXP subproject is capable of parsing and rewriting XPath queries. The SPEX processor uses the abstract XPath representation produced by the ReXP parser and compiles that representation into the SPEX transducer network. Furthermore the rewriting functionalities of ReXP are used within SPEX: all XPath queries are rewritten to equivalent queries without reverse axes, before the SPEX transducer network is produced. The visualizers provided for SPEX and ReXP can be used for demonstrating the SPEX processing or for debugging purposes. Figures 16,17 show the visualizers in action. The following sections explain installation and usage of SPEX and its subprojects.

4.1 Installation

The installation of SPEX is quite simple. The SPEX project provides executable *jar* files for the evaluator itself and for every subproject. These executables can be downloaded from [SF]. There is no need for an installation routine, downloading the executable to any folder is enough.

SPEX runs on the Java Runtime Environment version 5.0 or higher. An appropriate JRE for different operating systems can be downloaded at [JAVA]. For help on the installation, SUNs documentations may be referred.

4.2 Usage

Every executable file of the SPEX project is launched in the same way. In the command line navigation to the folder containing the executable and typing in the command *java -jar executable.jar* is enough. This launches the downloaded executable. Some of the executables, such as *spex.jar* and *rexp.jar* immediately provide more information and help in the console. These two are command line applications. Others, such as *spex-gui.jar* or *rexp-gui.jar* show a graphical user interface.

In the following the command line application *spex.jar* and its command line arguments are described.

```
Usage: java -jar spex.jar [-options] -xp <xpathexp>

where options include:
        -om <arg>    where <arg> is one of
                         'PRINT_RESULTS' (default),
                         'COUNT_RESULTS' or
                         'MATCH_ONCE'
        -pm <arg>   where <arg> is one of
                         'NAIVE',
                         'PHASE1'
                         'PHASE2' (default) or
                         'PHASE3'
        -i <file>   the xml input document (default is stdin)
```

```
        -o <file>   the output document (default is stdout)
        -rx         rewrite the xpath expression, so that its
                      reverse steps get eliminated
        -v          verbose mode, system messages are printed
                      before and after processing (default)
        -vv         very verbose mode, system messages are
                      printed during processing too
        -V          quiet mode, no system messages are printed
        -smo <file> system messages are print to <file>
                      (default is stdout)
        -d          debug mode, print exception stack traces
        -help       show help

<xpathexp> is an XPath expression generated by the following
EBNF grammar:
   Path              ::= ["/"] Step {"/" Step}.
   Step              ::= <Axis> "::" NodeTest {Predicate}.
   NodeTest          ::= "node()" | "text()" | "*" | <LABEL>.
   Predicate         ::= "[" BooleanExpression "]".
   BooleanExpression ::= BooleanExpression ("and" | "or")
                         BooleanExpression
                       | Path [ComparisonOP Constant]
                       | "contains(" Path "," <LITERAL> ")".
   ComparisonOP      ::= "=" | ">" | ">=" | "<" | "<=" | "!=".
   Constant          ::= <NUMBER> | <LITERAL>.

System Properties:
        spex.xmlstream.StreamParser
                specifies the StreamParser class
                (default is de.lmu.ifi.pms.spex.xmlstream)
        org.xml.sax.driver
                specifies the SAX Parser class
                (default is org.apache.crimson.parser.XMLReaderImpl)
```

4.3 Usage of SPEX API

The file *spex.jar* (and also the other jars) can not only be used as an executable, but can be used as a library for integrating SPEX into other Java applications. When included into the Java *classpath*, this file enables access to the SPEX processor classes. These classes represent a very well-designed API. The Javadoc documentation for this API can be found at [JD].

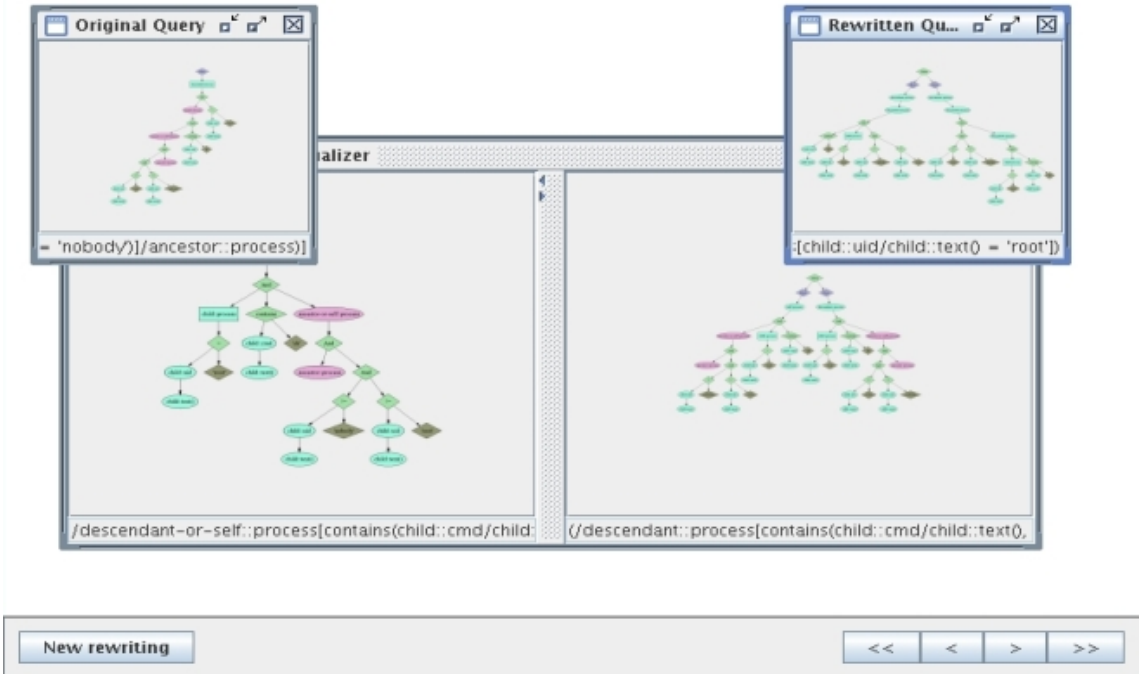

Figure 16: The ReXP visualizing GUI

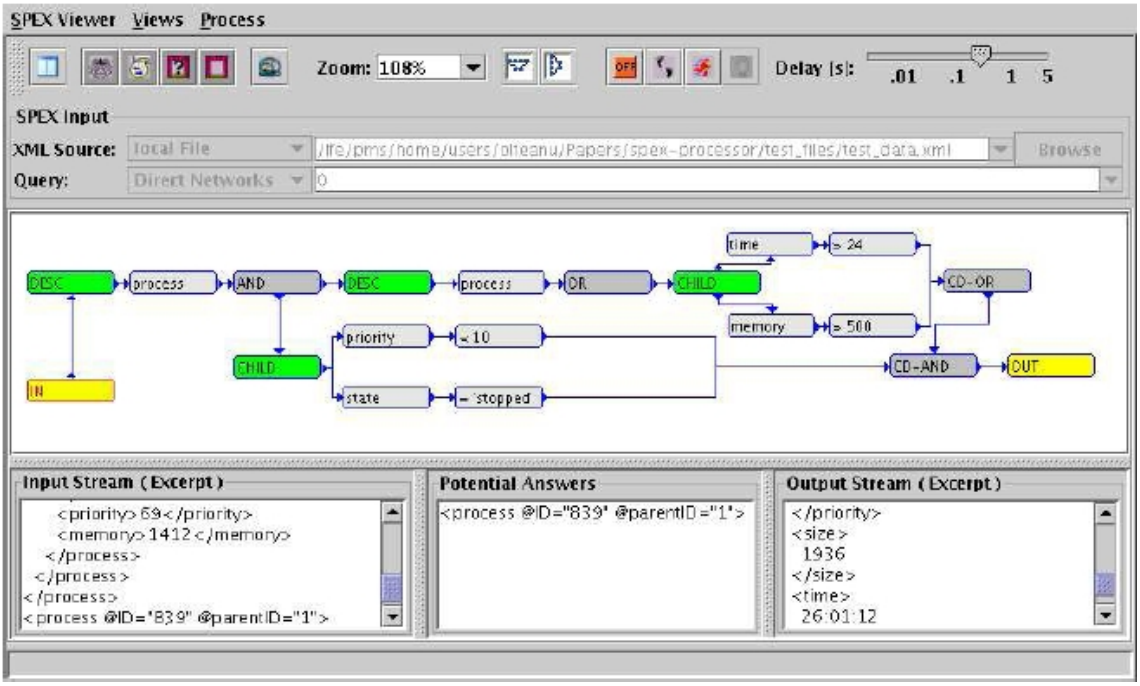

Figure 17: The SPEX visualizing GUI

References

[Spex1] *Evaluating Complex Queries against XML streams with Polynomial Combined Complexity*, Dan Olteanu, Tim Furche, and François Bry
URL: HTTP://WWW.PMS.IFI.LMU.DE/PUBLIKATIONEN/INDEX.HTML#PMS-FB-2003-15

[Spex2] *An Efficient Single-Pass Query Evaluator for XML Data Streams*, Dan Olteanu, Tim Furche, and François Bry
URL: HTTP://WWW.PMS.IFI.LMU.DE/PUBLIKATIONEN#PMS-FB-2004-1

[Spex3] *An Evaluation of Regular Path Expressions with Qualifiers against XML Streams*, Dan Olteanu, Tobias Kiesling, and François Bry.
URL: HTTP://WWW.PMS.IFI.LMU.DE/PUBLIKATIONEN/INDEX.HTML#PMS-FB-2002-12

[Spex4] *The XML Stream Query Processor SPEX (Demonstration)*, François Bry, Fatih Coskun, Serap Durmaz, Tim Furche, Dan Olteanu, Markus Spannagel. Proc. of 21st International Conference on Data Engineering (ICDE), Tokyo, April 2005
URL:
HTTP://WWW.PMS.IFI.LMU.DE/PUBLIKATIONEN/PMS-FB/PMS-FB-2005-1.PDF

[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 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.IFI.LMU.DE/PUBLIKATIONEN/INDEX.HTML#PMS-FB-2001-17

[Streams] *Datenströme*, François Bry, Tim Furche, and Dan Olteanu. Informatik Spektrum 27 (2), April 2004 Springer-Verlag
URL: HTTP://WWW.PMS.IFI.LMU.DE/PUBLIKATIONEN#PMS-FB-2004-2

[SF] The SPEX project at sourceforge.net
URL: HTTP://SPEX.SOURCEFORGE.NET/

[Java] The Java Programming language, runtime environment download.
URL: HTTP://WWW.JAVA.COM/

[XML] Extensible Markup Language (XML)
URL: HTTP://WWW.W3.ORG/XML/

[DOM] Document Object Model (DOM)
URL: HTTP://WWW.W3.ORG/DOM/

[SAX] SAX, the Simple API for XML
URL: HTTP://WWW.SAXPROJECT.ORG/

[XPath] XML Path Language (XPath) Version 1.0
URL: HTTP://WWW.W3.ORG/TR/XPATH/

[ReXP] to be done URL: TOBEDONE

[ReXP-Gui] to be done URL: TOBEDONE

[SPEX-Gui] to be done URL: TOBEDONE

[SUNJava] JavaSE 5.0 Technology, SUN Microsystems
URL: HTTP://JAVA.SUN.COM/

[Ant] The Apache Ant Project, a build-tool for Java
URL: HTTP://ANT.APACHE.ORG/

[JD] JavaDoc documentation for SPEX URL:
HTTP://SPEX.SOURCEFORGE.NET/SPEX/JAVADOC/

[JD1] JavaDoc documentation for package de.lmu.ifi.pms.spex.main URL:
HTTP://SPEX.SOURCEFORGE.NET/SPEX/JAVADOC/DE/LMU/IFI/PMS/SPEX/
MAIN/PACKAGE-SUMMARY.HTML

[JD2] JavaDoc documentation for package de.lmu.ifi.pms.spex.query.xpathcompiler
URL: HTTP://SPEX.SOURCEFORGE.NET/SPEX/JAVADOC/DE/LMU/IFI/PMS/
SPEX/QUERY/XPATHCOMPILER/PACKAGE-SUMMARY.HTML

[JD3] JavaDoc documentation for package de.lmu.ifi.pms.spex.xmlstream URL:
HTTP://SPEX.SOURCEFORGE.NET/SPEX/JAVADOC/DE/LMU/IFI/PMS/SPEX/
XMLSTREAM/PACKAGE-SUMMARY.HTML

[JD4] JavaDoc documentation for package de.lmu.ifi.pms.spex.engine.messages
URL: HTTP://SPEX.SOURCEFORGE.NET/SPEX/JAVADOC/DE/LMU/IFI/PMS/
SPEX/ENGINE/MESSAGES/PACKAGE-SUMMARY.HTML

[JD5] JavaDoc documentation for package de.lmu.ifi.pms.spex.engine.annotations
URL: HTTP://SPEX.SOURCEFORGE.NET/SPEX/JAVADOC/DE/LMU/IFI/PMS/
SPEX/ENGINE/ANNOTATIONS/PACKAGE-SUMMARY.HTML

[JD6] JavaDoc documentation for package de.lmu.ifi.pms.spex.engine.transducers
URL: HTTP://SPEX.SOURCEFORGE.NET/SPEX/JAVADOC/DE/LMU/IFI/PMS/
SPEX/ENGINE/TRANSDUCERS/PACKAGE-SUMMARY.HTML

[JD7] JavaDoc documentation for package
de.lmu.ifi.pms.spex.engine.transducers.filter URL:
HTTP://SPEX.SOURCEFORGE.NET/SPEX/JAVADOC/DE/LMU/IFI/PMS/SPEX/
ENGINE/TRANSDUCERS/FILTER/PACKAGE-SUMMARY.HTML

[JD8] JavaDoc documentation for package de.lmu.ifi.pms.spex.engine.transducers.locationstep URL: HTTP://SPEX.SOURCEFORGE.NET/SPEX/JAVADOC/DE/LMU/IFI/PMS/SPEX/ENGINE/TRANSDUCERS/LOCATIONSTEP/PACKAGE-SUMMARY.HTML

[JD9] JavaDoc documentation for package de.lmu.ifi.pms.spex.engine.transducers.locationstep.matcher URL: HTTP://SPEX.SOURCEFORGE.NET/SPEX/JAVADOC/DE/LMU/IFI/PMS/SPEX/ENGINE/TRANSDUCERS/LOCATIONSTEP/MATCHER/PACKAGE-SUMMARY.HTML

[JD10] JavaDoc documentation for package de.lmu.ifi.pms.spex.engine.transducers.scope URL: HTTP://SPEX.SOURCEFORGE.NET/SPEX/JAVADOC/DE/LMU/IFI/PMS/SPEX/ENGINE/TRANSDUCERS/SCOPE/PACKAGE-SUMMARY.HTML

[JD11] JavaDoc documentation for package de.lmu.ifi.pms.spex.engine.results URL: HTTP://SPEX.SOURCEFORGE.NET/SPEX/JAVADOC/DE/LMU/IFI/PMS/SPEX/ENGINE/RESULTS/PACKAGE-SUMMARY.HTML

[JD12] JavaDoc documentation for package de.lmu.ifi.pms.spex.tests URL: HTTP://SPEX.SOURCEFORGE.NET/SPEX/JAVADOC/DE/LMU/IFI/PMS/SPEX/TESTS/PACKAGE-SUMMARY.HTML

[JD13] JavaDoc documentation for package de.lmu.ifi.pms.spex.utils URL: HTTP://SPEX.SOURCEFORGE.NET/SPEX/JAVADOC/DE/LMU/IFI/PMS/SPEX/UTILS/PACKAGE-SUMMARY.HTML