INSTITUT FÜR INFORMATIK
Lehr- und Forschungseinheit für
Programmier- und Modellierungssprachen

Oettingenstraße 67 D–80538 München

Towards a Streamed XPath Evaluation

Tobias Kiesling

Diplomarbeit

Beginn der Arbeit:	21.01.2002
Abgabe der Arbeit:	08.07.2002
Betreuer:	Prof. Dr. François Bry
	Dipl.-Inf. Dan Olteanu

Erklärung

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

München, den 06.07.2002 Tobias Kiesling

Zusammenfassung

XPath ist eine Sprache zur Adressierung von XML-Dokumenten die in Anfrage- und Transformationssprachen wie XQuery und XSLT benützt wird. Für viele Anwendungen ist es wünschenswert, einen Eingabestrom im Vorrübergehen und progressiv mit XPath auszuwerten. Diese Diplomarbeit beschäftigt sich mit progressiven Anfragen auf XML Datenströmen.

Progressive XPath-Auswertung auf einem Eingabestrom verringert erheblich den Speicherbedarf. Dies ist essentiell für eine XPath-Auswertung von sehr großen Dokumenten. Außerdem ermöglicht sie die Auswahl von Informationen von *continuous services* wie Börsen- oder Wetterdaten. Der vorgeschlagene Ansatz hat einiges für sich. Erstens sind beide Übersetzungen, die von allgemeinem XPath in *forward XPath*, und die von *forward XPath* in Transducer-Netze in linearer Zeit durchführbar, mit einem Speicherbedarf der linear in der Größe des Eingabedokumentes ist. Zweitens weisen die mit einem einfachen Prototyp durchgeführten Experimente auf eine bemerkenswerte Effizienz des Ansatzes hin. Drittens kann die Auswertung von XPath-Ausdrücken in einem einfachen Durchgang über den Eingabestrom durchgeführt werden und hat einen Speicherbedarf der quadratisch in der Tiefe des Eingabedokumentes ist.

Abstract

XPath is a language for addressing fragments of XML documents, used in query and transformation languages such as XQuery and XSLT. For many applications it is desirable to process XPath *on the fly* and *progressively* against data streams. This diploma thesis is devoted to streamed and progressive evaluation of XPath.

A streamed and progressive XPath evaluation considerably reduces the needed memory space. This is essential for XPath evaluation against very large documents. Furthermore, it enables the selection of informations from *continuous services*, such as stock market or meteorology data. The proposed approach enjoys attractive features. First, both translations, of general XPath into forward XPath, and of forward XPath into transducer networks generate output in linear time, with a size linear in the input size. Second, experiments with a straightforward prototype implementation suggests a remarkable efficiency of the approach. Third, the evaluation of an XPath expression can be performed in one pass over an XML stream and uses space at most quadratic in the document depth.

Danksagung

Für die sehr gute Betreuung und eine angenehme Atmospäre bei meiner Diplomarbeit möchte ich mich bei meinen Betreuern François Bry und Dan Olteanu bedanken. Zusätzlicher Dank geht an Tim Furche für seine Unterstützung bei der Entwicklung des Modells, aber auch für seine Hilfestellungen zu LaTeX.

Contents

1 Introduction

Querying data streams is a field of growing importance, motivated by real-time measurement applications, e.g. monitoring the number of passing vehicles on a motorway for traffic routing, and nowadays *continuous services*, which select informations from a continuous flow of data, such as stock exchange or meteorology data. For selective dissemination of information (SDI), streams have to be filtered according to complex requirements, specified as queries, before being distributed to the subscribers [14, 11]. Also, to integrate data over the Internet, particularly from slow sources, it is desirable to progressively process the input before the full data is retrieved [19, 18].

The data streams considered in such applications can be infinite (or, more precisely, unbound) and often they consist of structured messages. Such messages are conveniently modeled with XML and message selection can be expressed using XPath [29], a declarative language for addressing fragments of XML documents, used as a part of query and transformation languages such as XQuery [31] and XSLT [32]. In contrast to other proposals [18, 14] the approach described here accommodates XPath qualifiers. An evaluation model for XPath on streams, as proposed in this thesis, is focused on and considerably influenced by the migration from simple regular path expressions to path expressions with qualifiers, as XPath proposes.

This thesis describes an approach to a streamed and progressive evaluation of XPath expressions. A *streamed* evaluation considers that an XML data stream is read and not completely buffered, *progressive* processing delivers streamed result on the fly. This approach is based upon a rewriting of general XPath expressions into forward XPath expressions in which no reverse axes such as the `ancestor` and `preceding` occur.

The rewriting of general XPath expressions into forward XPath expressions is described in detail in [27]. Therefore, it is only briefly recalled in this thesis.

Some approaches to streamed query processing consider *window operations* of a fixed size that restricts query answering to parts of the input data [13]. This way, input streams with unbound sizes can be processed without storing more data than specified by the window. However, this is at the cost of returning incorrect or incomplete answers. Such an approach is *not* considered here, although it could be combined with the method described here, which performs an exact evaluation of XPath expressions. As a consequence, parts of the data stream may be stored in memory. However, the memory usage is kept as low as possible and is defined in terms of low-level pushdown stores.

The approach proposed in this thesis enjoys attractive features. First, both translations, of a general XPath expression into a forward XPath expression, and of a forward XPath expression into a transducer network generate output in linear time, with the size linear in the size of the input expression. Second, experiments with a rather straightforward prototype implementation show a remarkable efficiency of the approach. Third, the evaluation of an XPath expression can be performed in one pass over the data stream using space at most quadratic in the document depth. There are unavoidable cases where the whole document (or data stream) needs to be stored in memory. E.g. if only after the whole data stream is processed it can be determined whether the entire document is an answer, the memory needed is linear in the size of the stream.

1.1 Use Cases

This section discusses in more detail several of the above mentioned environments where streamed processing is essential.

Very Large Data Environments The query processing of very large databases ($\geq$ 100MB) poses a problem if the data is not stored persistently with the help of a database management system that ensures efficient access to the data. This might be the case if the data is highly dynamic or highly structured.

For very large data it is often not possible to keep an in-memory representation of it, or it is too expensive to build one. E.g. with a size of 10MB of an input document, the amount of main memory used up by the Saxon XSLT processor [8] is about 70MB. Here, an approach is needed that ensures a constant (or at least sub-linear) space complexity.

Examples for very large data environments are biological [12], astronomical [1], or directory [3] data.

Data Integration Data integration targets the integration of data provided by data sources that are distributed over a network [19, 18]. In this environment, data is transfered over (relatively) slow network links as streams. Therefore, it is natural to process such data in a streamed way. Consider an application that has to wait until the whole input data has been received over a slow network. Only when all of the data has been received, such an application starts processing. During the time of data transfer the target machine is idle waiting for the data, wasting precious CPU cycles.

In contrast, in an application that processes incoming data streams in a progressive way[†], the target machine is busy during the time of data transfer, hence leading to a higher processor utilization. Additionally, answer times are much lower in a streamed approach.

Selective Dissemination of Information Selective dissemination of information (SDI) is used to route various information to users that subscribed certain parts of this information [14, 11]. Examples for SDI systems are stock and sports tickers, traffic information systems, electronic personalized newspapers, and entertainment delivery. Data is provided to subscribers according to user profiles that describe the data the user is interested in. Such a user profile can be interpreted as a query that has to be satisfied by the piece of information in question.

The characteristics of an SDI system are a large number of incoming data items (e.g. XML documents) and a large number of queries to be performed on every item. Therefore, quick decisions on information distribution are neccessary, whereas it is generally not possible to create an in-memory representation of a data item and perform all queries on this item before the next item arrives.

Pipelined Processing In the last time there was an increasing use of application frameworks like Cocoon [2] or XPipe [22]. In these frameworks various processing components (e.g. XQuery engines, XSLT processors) can be plugged together to provide a highly customizable service. The gain is flexibility while still using XML as a standard interface between components. Earlier versions of these frameworks passed XML data between components as an in-memory representation (e.g. DOM [30]). However, because of the advantages of the progressive nature and low memory consumption of streams, developers decided to switch to the usage of XML streams for the communication between components (e.g. using SAX [23]).

With this approach it is very important to ensure a streamed processing of all the components in the pipeline. If only one components breaks the streamed processing by buffering the whole

[†]A progressive application processes incoming data items as soon as they are received, possibly leading to some result.

input stream, the whole system does not work in a streamed way. Momentarily, this is the case with some important components (e.g XSLT processors).

1.2 Overview of the Thesis

The thesis is organized as follows. Section 2 shortly overviews the XML stream data model and the XPath language used for querying and gives an introduction to the problems of querying XML streams. The SPEX evaluation model is informally introduced in Section 3, a more formal description follows in Section 4. At the end of Section 4 the translation from forward XPath to SPEX transducer networks is specified. Complexity results are given in Section 5. Section 6 describes the implementation of a simple SPEX prototype. Section 7 compares this prototype with standard XPath implementations, which mainly construct an in-memory representation of the XML stream. Related work is summarized in Section 8 and we conclude with Section 9.

2 Preliminaries

2.1 XPath

The XML path language XPath has been specified by the W3C for addressing fragments of an XML document [29].

The central syntactic construct in XPath is the expression. An expression can be evaluated to an object that is one of the types node set, boolean, number, and string. Only expressions that evaluate to node sets are considered here. The most important expression returning a node set is a location path. A location path selects a set of nodes relative to some base node[†]. A location path consists of one or more location steps separated by a slash ("/"). Every location step selects a set of nodes relative to a node selected by the preceding step. Nodes are selected based on an axis that specifies the relationship between the base node and the selected nodes that must be satisfied, e.g. the `child` axis selects all nodes that are direct children of the base node (i.e. that are directly contained in the base node). A step also includes a node test that is used to filter the nodes selected by the axis based on their type, e.g. the `text()` node test selects only text nodes. Additionally, a step may comprise any number of qualifiers that are used to specify additional constraints for the selection of a node.

In XPath there are *forward axes* that only select nodes that are after the base node in document order and *reverse axes* that select nodes that are before the base node in document order. For every reverse axis in XPath there is a corresponding forward axis that is *symmetrical* to the reverse axis.

The following pairs of axes are defined in XPath:

- `child/parent`: The `child` axis selects all direct children of the base node, i.e. all nodes directly contained in the base node. The `parent` axis selects the parent node of the base node.
- `descendant/ancestor`: The `descendant` axis selects all nodes in the subtrees starting with the children of the base node, i.e. the children of the base node and all descendants of those children. The `ancestor` axis selects all the nodes that the base node is a descendant of, i.e. all nodes that are on the path from the base node to the root of the document tree.
- `descendant-or-self/ancestor-or-self`: This pair of axes is equivalent to the `descendant /ancestor` pair, except that also the base node is selected.
- `following-sibling/preceding-sibling`: The siblings of a base node are all nodes in the document tree that have the same parent node as the base node. The `following-sibling` axis selects all nodes that follow the base node in document order and that are siblings of the base node. The `preceding-sibling` axis selects all siblings that precede the base node in document order.
- `following/preceding`: The `following` axis selects all nodes that follow the base node in document order starting with the first following sibling, i.e. the descendants of all following siblings of the nodes selected by the `ancestor-or-self` axis relative to the base node (including those siblings).
- `self`: The `self` axis is symmetrical to itself and selects only the base node.

[†]If the path is an absolute path, nodes are selected relative to the document root.

- `attribute` and `namespace`: These special axes are used for selecting the attribute nodes of the base node and the namespaces in scope at the base node. They will not be discussed here.

For a streamed evaluation of XPath, reverse axes cannot be considered without keeping a full record of the document received until some point in time. This is the case, as backward axes select elements the start tags of which were processed before. As no full record of those past events is kept, information about them is not available. Therefore, this work relies on a removal of reverse axes from XPath expressions that is presented in Section 2.2.

The initial node set resulting from the selection of nodes via an axis and a node test can be further filtered by any number of *qualifiers*[‡]. The qualifiers are applied in turn: first, the initial node set is filtered by the first qualifier to give a new node set that is further filtered by the the second qualifier. This is repeated for all the qualifiers of a step. Filtering a node set with a qualifier is performed by evaluation of the expression contained inside the qualifier relative to every node contained in the node set. The resulting node set is composed of all the nodes in the initial set for which the expression inside the qualifier evaluated to the boolean value *true*. A prominent example for an expression inside a qualifier is a location path. The qualifier is satisfied if the location path selects a non-empty node set. With a location path inside a qualifier where at least one of the location steps has another qualifier the notion of *nested qualifiers* is introduced.

Figure 1 XML fragment

```
<a>
    <a>
        <c></c>
    </a>
    <b></b>
    <c></c>
</a>
```

Example 2.1. The expression `/descendant::a[child::b]/descendant::c` selects all `c` elements that are within an `a` element within the document (denoted as a leading "/"), if that `a` has a child element `b`. If processed against the data in Figure 1, the step `descendant::a` selects the outer and inner `a` elements, as they match the node test of the step and are in the descendant axis relative to the document root (that is the base node of the first step). The location path inside the qualifier `[child::b]` is then evaluated relative to both of the selected `a` elements. For the outer `a` the evaluation results in a non-empty node set, hence only the outer `a` is considered to be in the node set selected by the first location step `/descendant::a[child::b]`. The second step `descendant::c` is then evaluated relative to the outer `a` (the only node in the set resulting from the evaluation of the first step) to yield as a result the set containing both of the `c` nodes.

A location path that is evaluated relative to the document root (denoted by a leading "/") is called an *absolute location path*. A path that is not absolute is a *relative location path*.

The XPath expressions considered here are absolute location paths or a union or intersection of two absolute location paths. An absolute location path consists of steps. Every step is composed of an axis, a node test, and any number of qualifiers (including zero). Supported axes are `child`, `descendant`, `descendant-or-self`, `self`, `following-sibling`, and `following`. Inside

[‡]In the XPath specification qualifiers are called *predicates*.

qualifiers relative and absolute location paths without qualifiers are supported (i.e. no nested qualifiers), as well as unions, intersections, and identity-based joins of two such paths. The grammar contained in Figure 2 formally describes the supported language.

Figure 2 Grammar for the supported subset of XPath

```
expression    ::= /qpath | /qpath op /qpath .
qpath         ::= qstep | qstep/qpath .
qstep         ::= step | step qualifiers .
qualifiers    ::= [qualifier] | [qualifier] qualifiers .
qualifier     ::= path | path == path .
path          ::= relpath | /relpath .
relpath       ::= step | step/relpath .
step          ::= axis::node-test .
op            ::= union | intersect .
axis          ::= child | descendant | descendant-or-self | self |
                  following-sibling | following .
node-test     ::= node() | text() | * | label .
```

Some features of XPath are not considered in the rest of this thesis, most prominently operations on values of nodes and positional qualifiers. As an example for the various operators and functions of XPath, the evaluation of the union and intersect operator is given in Section 4.5.

2.2 Translating XPath into Forward XPath

As presented in the last section, XPath specifies several *axes* that can be classified by the nodes they select: *Forward axes* select nodes that are after the current node in *document order*[§], *reverse axes* select nodes that appear before the current node [29]. The evaluation of reverse axes on a stream in one pass presents a considerable problem: At any time of the evaluation only the current message, representing the current node, is known. Without keeping a full record of past messages, it is impossible to address reverse axes.

One approach for solving this problem is to rewrite XPath expressions with reverse axes. In [27] it is shown that any XPath expression of concern here can be rewritten into an XPath expression without reverse axes, called a *forward* expression. There, two sets of rewriting rules are presented, one with linear time for rewriting, but slightly more complex resulting XPath expressions, due to the introduction of identity-based joins. The second one has exponential time for rewriting, but yields possibly more efficient rewritten XPath expressions. Here, the first set of rules is used, as identity-based joins can be resolved efficiently, cf. Section 4.3.3. Therefore, it is assumed that all expressions containing reverse axes are rewritten into forward XPath expressions prior to the compilation.

The general approach to rewriting of reverse axes can be found in [27]. The following examples give an impression of how rewriting can be performed.

Example 2.2. Consider the path `/descendant::a/parent::b` that includes the reverse axis

[§] *Document order* is the order of nodes as they are encountered in a depth-first left-to-right traversal of the document tree, i.e. a node n_1 is before a node n_2 in document order if the start tag of n_1 is before the start tag of n_2 in the XML representation of the document.

`parent`. This expression is equivalent to `/descendant-or-self::b[child::a]` where the reverse axis has been replaced by its symmetrical counterpart.

Example 2.3. The location path `/descendant::a[parent::b]` includes a reverse axis inside a qualifier. This is readily rewritten into the equivalent path `/descendant-or-self::b/child::a`.

2.3 XML Streams

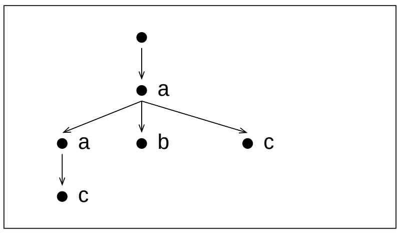
Streamed XML processing in contrast to processing an in-memory representation of XML data has some very interesting issues. Where in the latter case it is possible to randomly access any node in the document tree during processing, there exists only a strictly limited possibility of access in a streamed approach. The knowledge about the document is at any time restricted to the parts of the document received until that time. Of course, in this case it would be possible to collect the whole document before processing it. However, as shown in Section 1.1, there are environments where this buffered approach is impractical or even impossible to use. In a reasonable streamed approach the information about the input data received until a point in time should be used to maximum extent. Consider a query on an XML stream, where it is possible to decide after having received only a fraction of the input data that a particular node is a part of the result. A reasonable streamed approach outputs that node as soon as possible.

Figure 3 Three Representations of an XML Document

1. Serialized XML Document

```
<?xml version="1.0"?>
<a>
  <a>
    <c/>
  </a>
  <b/>
  <c/>
</a>
```

2. XML Tree (after the XPath Data Model)



3. XML Stream

```
<$>  <a>  <a>  <c>  </c>  </a>  <b>  </b>  <c>  </c>  </a>  </$>
```

Consider an XML document like the one in Figure 3.1. This document can be represented as a tree shown in Figure 3.2. The serialization of XML corresponds to a traversal of the XML tree in document order, i.e. a depth-first, left-to-right traversal, where every first occurrence of an inner node is encoded in a start tag with the label of the node and every last occurrence in an end tag. Every leaf is encoded by its value (e.g. any PCDATA in XML).

An XML document can be transmitted by a stream in two different ways. The first possibility is a character stream where every character is transmitted as-is, e.g. the start tag `<foo>` is transmitted as `'<', 'f', 'o', 'o', '>'`. A token stream in contrast sends the data as one token at a time, where a token consists of the information from multiple characters of the character stream that are logically bound together. The start tag `<foo>` is sent as one token by a token stream. Both streams are equivalent, however, as one can be simulated by the other. Only token streams will be considered in this document.

A streamed processor working on a token stream has full knowledge of the current token and partial (or also full) knowledge of all tokens preceding the current token, but it never has any knowledge of tokens following the current token (that will be received in the future).

2.4 Querying XML Streams

The main focus of this document is on querying streams of XML data. The data that is queried is a tree that has been serialized and is received by means of a token stream[¶]. Other than in relational databases where the data to be queried are relations that are very simple flat data structures, XML document trees can be complex structures. Therefore, a query language for XML has to take a different approach. The one that was chosen in XPath is navigational, where a query is specified by giving a path through the document that must be satisfied by a node to be a result of the query. This is very intuitive if there is an in-memory representation of the input document available. However, if the input data is received in terms of a token stream, XPath expressions (and especially location paths) have to be interpreted in a different way.

Example 2.4. Consider the query `/descendant::a[child::b]/descendant::c` which is processed against the XML document shown in Figure 3. The result of the query consists in both of the elements with label `c`. They are chosen as descendants of the outer `a` rather than the inner `a`, because the outer `a` also satisfies the qualifier, having a child element with label `b`.

At the beginning of the query only the first location step is active (i.e. it is trying to match), the rest of the steps are idle. Therefore, only `descendant::a` tries to match the start tag of the outer `a`. As the label of the element is the same as the node test of the step, it is successful and activates the next step inside the qualifier (`child::b`) and the next step outside the qualifier (`descendant::c`).

All of the three steps try to match the next tag (the start of the inner `a`). Only `descendant::a` is successful and notifies the other steps of this again. Because the first step matched a second time, `child::b` still tries to match the next tag (the start tag of the first `c`), as also the other two steps do. `descendant::c` is successful and therefore the first `c` is identified as a candidate for the result. However, at this point in time it is not known yet if this candidate also fulfills the qualifier. Therefore it must be stored until the value of the qualifier is determined. For the current position in the input document, the qualifier could be satisfied if there is a child with label `b` either inside the outer `a` or inside the inner `a`. Momentarily, the storage of this candidate contains only the start tag of the `c`, but other tags (or character data) can be added later. When the end tag of the `c` is encountered, it is stored and the storage of this candidate is closed, as the whole candidate has been received.

When receiving the end tag of the inner `a`, it is known that this element has no child with label `b` and therefore the qualifier cannot be satisfied based on the inner `a`. However, it still has the possibility to be satisfied based on the outer `a`.

All of the location steps try to match the start tag of the `b` that is received next. `child::b` is successful and as it is the only step in the qualifier, the qualifier is satisfied by the current element. Therefore, it is known that the candidate is a part of the result and the stored tags can be released and sent to the output.

The start tag of the second `c` is encountered afterwards and matched by `descendant::c`, that still tried to match because it was activated on the start tag of the outer `a`. As the qualifier was already satisfied for this outer `a`, the new candidate that is encountered is immediately known to be part of the result. Therefore, the current start tag does not have to be buffered, but can be directly written to the output. Also the next tag (the end tag of the second `c`) is output, as it is a part of the second candidate. The processing ends with the end tag of the outer `a`.

[¶]Of course it is also possible to construct an XML stream dynamically without an intermediate tree representation.

This rather simple example already shows the most important aspects of streamed querying:

- conditions and candidates: A node that matches the location path outside qualifiers is called a *candidate*. It can be part of the result if the qualifiers related to it are satisfied. However, for different candidates the same qualifiers have to be evaluated relative to different base nodes.
- past and future conditions: Conditions that are always determined before a candidate related to them is encountered are called *past conditions*. Conditions that are always determined after candidate creation are *future condition*. However, there are also conditions that are neither past nor future conditions, as no general rule about their determination can be given.
- multiple paths to a matched node: A node can be matched by a location step based on more than one matching of the preceding step. This introduces a disjunctive component in the relationship of candidates and conditions, as a candidate can become part of the result if the conditions of any one of the matchings of the preceding step are satisfied.

Figure 4 Example of a SAX ContentHandler

```
import org.xml.sax.ContentHandler;
import org.xml.sax.Attributes;

public class BookCounter implements ContentHandler {
  private int counter;
  public BookCounter() {
    counter = 0;
  }
  public void startElement(String qname, String url,
                           String lname, Attributes atts) {
    if(lname.equals("book")) counter++;
  }
  public void endDocument() {
    System.out.println("Number of book elements found: " + counter);
    counter = 0;
  }
  // other methods required by interface ContentHandler follow here
}
```

2.5 Simple API for XML (SAX)

As the name implies, SAX [23] is an application programming interface that is used to connect an application to a standard XML parser. As opposed to a DOM parser, a SAX parser creates an XML stream that is encoded in a sequence of method callbacks. Every single method call transmits another token of the XML stream. Tokens can be start and end of the document, start and end of an element, and text nodes, among others. These tokens are called *events* in SAX, as they are represented by a method call that can occur at any time. The development of the model presented in this thesis was specifically inspired by the way an XML document is processed using SAX. Therefore, a short introduction to SAX is given in this section.

SAX was originally developed for Java but support for SAX parsing is now available in a lot of other languages (e.g. PERL). The SAX approach is centered around an interface (*ContentHandler* in SAX2) that requires the implementation of one method for every kind of event that can occur (e.g. start-element, end-element, text). A software component that implements this interface by giving a definition of every required method can then be used in conjunction with a SAX parser that parses an XML document and provides a stream of events to the component by calling the appropriate methods. In the definitions of the required applications the events can be handled and appropriate actions can be taken.

Example 2.5. Consider an application that reads an XML document and counts the number of elements with label `book`. Such an application can easily be created using SAX by providing an implementation of the startElement callback method that inspects the label of the element and increases a counter if the label is `book`. At the end of the document the number of elements should be printed. This can be achieved by simply providing a definition of the endDocument callback method that prints the current value of the counter. Figure 4 shows the definition of a ContentHandler that exhibits this behaviour.

However, SAX is not restricted to the simple use of a ContentHandler that is provided to a SAX parser, but rather ContentHandlers can be chained, having one ContentHandler calling the appropriate methods of another one. This is a very powerful concept that allows clearly defined components to be plugged together to fulfill some complex task. The concept of chained ContentHandlers is very similar to the concept of chained transducers (cf. Section 4) used extensively in the model presented here. Hence, it is quite straightforward to implement a system that is specified in terms of chained transducers by using SAX. However, there is one important difference between chained transducers and chained SAX components. Whereas the former are only synchronized in that a transducer cannot execute transitions if the preceding transducer has provided some symbols to its input tape, the latter are strongly synchronized in the sense that as long as a ContentHandler is executing a message call, the preceding ContentHandler is suspended.

3 The SPEX Query System

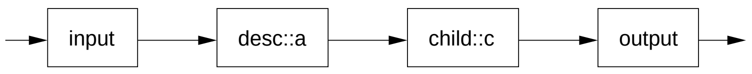
In this section, an informal presentation of the evaluation model for querying XML streams is given. A formalization of this model is presented in Section 4. Section 3.1 introduces the basic properties of query systems. Branching and synchronization are presented in Section 3.2. The handling of conditions in a query system is explained in Section 3.3. Section 3.4 gives a summary of the different transducer types.

3.1 Basic Properties

3.1.1 SPEX Networks

The task of querying is performed by a *SPEX network* that consists of interconnected transducers. A standard component of the network is a *SPEX transducer* that consists of a finite state control, two stacks, an input tape, and an output tape. Transducer are connected by piping the output of one transducer into the input of another one. Without additional components this creates chains of transducers. The first transducer in a chain is always the *input transducer* that is responsible for ensuring system synchronization. The last component, the *output transducer*, is responsible for creating correct output. An important type of transducer located in the chain between an input and output transducer is a step transducer that implements the logic of a location step construct from XPath. There are different types of step transducers for the different XPath axes, e.g. a *child transducer* for location steps with axis `child`, or a *following transducer* for the `following` axis. Figure 5 shows an example of a SPEX network for the query `/descendant::a/child::c`.

Figure 5 SPEX network for query `/descendant::a/child::c`



In general, SPEX networks can be more complex than the simple example in Figure 5 implies. For some XPath constructs (e.g. qualifiers, union) it is necessary to have one transducer sending messages to several other transducers and to have one transducer receiving input from more than other transducer. This is presented in more detail in Section 3.2.

3.1.2 SPEX Messages

As explained in the last section, communication in a query system is performed by sending messages to the input tapes of transducers. There are three different classes of messages passed inside a system:

- **Document messages** `<...>` correspond to the input of the system. Every document message represents a part of the input document. Examples for these messages are <l> for element start tags and </l> for element end tags.
- **Activation messages** `[...]` are used to activate transducers. At the beginning of the query most of the transducers are idle (i.e. not performing any task). When an activation

message is sent to a transducer some activity is started, e.g. a step transducers tries to match a part or all of the following document messages.

- **Condition determination messages** $\{\ldots\}$ are used to signal the truth value of conditions. The value of a newly created condition is not known until some time in the future (i.e. it is undetermined). As soon as the responsible transducer knows about the value of a condition, it sends a condition determination message.

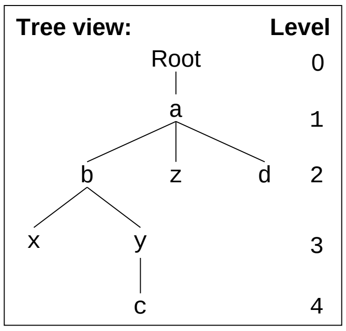
All of the document messages together represent the document tree of the input data. Every node in the tree is encoded by a start message followed by the messages of all child nodes followed by an end message. This is very similar to the XML serialization of a document tree where nodes are represented by nested start and end tags.

As long as there is no message on the input tape of a transducer it is blocked, as it is not able to execute any transition[†]. When a message arrives, a transducer immediately tries to perform a transition, sending one or more messages to the next transducer. These messages in turn enable the next transducer to execute. As long as there is no splitting of interconnections in the system (cf. Section 3.2), this leads to a correct behavior. However, if there is at least one split in the system, multiple transducers work in parallel and therefore synchronization among them is necessary. This is explained further in Section 3.2. In any case, the input transducer allows document messages to enter the system only one at a time. When a document message reaches the output transducer it must have been received by all transducers in the network, therefore enabling the input transducer to send the next document message.

The different classes of messages are handled by the system in different ways. Document messages are always forwarded even if the sending of another message (e.g. an activation message) is caused by the document message. Therefore, any number of activation and condition determination messages may appear before a document messages. Activation and determination messages are logically related to the document message that follows them. As document messages are always forwarded by transducers, they reach all transducers in the system. In contrast to this, activation messages are generally not forwarded, reaching only the next transducer in the chain[‡]. Like activation messages, condition determination messages are created by a SPEX transducer inside the system. However, like document messages, they are forwarded by transducers, hence reaching all transducers from the one creating the message up to the end of the chain.

Figure 6 example document tree

Input data:
```
<a>
  <b>
    <x/>
    <y><c/></y>
  </b>
  <z/>
  <d/>
</a>
```



[†]To execute a transition a transducer must read and remove a message from its input tape.

[‡]Note that there are special transducers that forward activation messages.

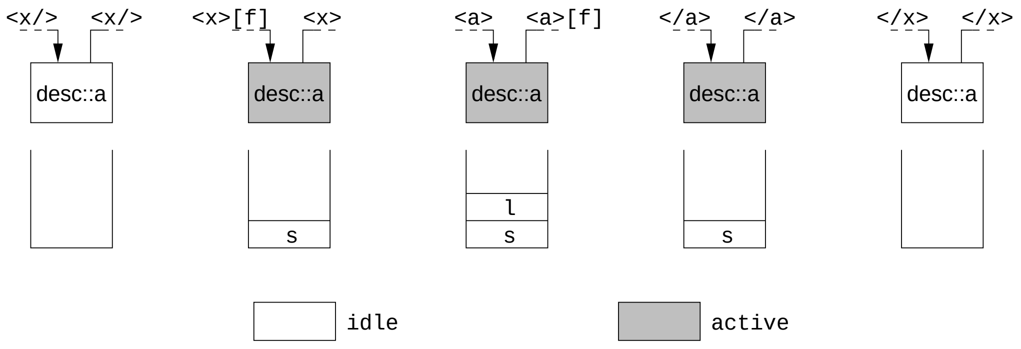
3.1.3 Keeping Track of the Tree Level

Definition 1 (Tree Level). An XML document tree consists of nodes nested in one another. The number of nestings of a specific node determines its **tree level** (i.e. its depth in the tree).

Figure 6 shows an XML fragment and a corresponding document tree, together with the levels of the nodes in the tree.

Most of the SPEX transducers need to keep track of the current tree level at some time. For this purpose they use a *count stack* that is a simple stack where fixed symbols are pushed and popped. A transducer does not use the count stack when it is idle. However, if it has been activated it will start to keep track of the current tree level by pushing a symbol on every start message (i.e. a document message representing a start tag of an element) and popping a symbol on every end message (i.e. a document message representing an end tag of an element). Levels that have a special meaning (e.g. the level on which a transducer is activated) are marked with a special symbol (e.g. `m`), whereas another symbol `l` is used for representing normal levels in the tree.

Figure 7 Counting the level in the document tree



Example 3.1. Figure 7 shows a sequence of configurations and message passings of a descendant transducer when processing the XML stream `<x/><x><a></a></x>`. After the `<x/>` document message the transducer is idle, only forwarding the message. Then it receives an activation message `[f]` and the corresponding start element message `<x>`. It switches to an active state and forwards the document message. Further, it puts an `s` symbol (for activation scope cf. Section 3.1.4) on the stack to mark the level in the document tree where it is activated. The next message received by the transducer is the start element message `<a>`. As it is active, it tries to match the `<a>` and is successful, outputting an activation message followed by the document message. In any case – matching or not –, the transducer puts an `l` symbol (for tree level) on the count stack, as the encountered document message is a start message. After that the transducer receives the end element message `</a>`, removing the topmost symbol from the count stack. As this was an `l` symbol, it performs no further action than outputting the document message. The last message that the transducer receives is the end element message `</x>`. Again, the transducer removes the topmost symbol of the count stack and forwards the message. However, as it removed an `s` from the stack, it leaves the scope that started with the reception of the activation message `[f]`, therefore switching back to an idle state.

Figure 8 XML fragment

```
<a>
 <a><b/></a>
 <b/>
</a>
```

3.1.4 Activations and Matching

Definition 2 (Activations). At the start of the query every transducer is said to be **idle** (except the first in the path) and can be **activated** by another transducer. The message that led to the activation of the transducer is the **activating message**. If the activating message is a start element or start document message, it has a corresponding **deactivating message** that is the corresponding end element (end document, resp.) message. All messages between the activating and deactivating messages compose the **activating node**. When a transducer is activated it enters an **activation scope** the end of which depends on the specificities of the transducer.

There is a tight relationship between an activation scope and its activating message. A transducer that is activated while it is already active (because it was activated before) is in nested activation scopes. E.g. when the location path `/descendant::a/child::b` is processed against the XML fragment shown in Figure 8, `child::b` is activated a first time on the start element message of the outer `a` and also on the start element message of the inner `a`. At that point `child::b` is in two activation scopes where the second one is nested inside the first.

Definition 3 (Matching). The **match scope** of a transducer is the set of nodes in the document tree that is selected by the axis of the transducer relative to an activation. A transducer (representing a location step) **matches** a node if it is in a match scope and its node test is satisfied by the node.

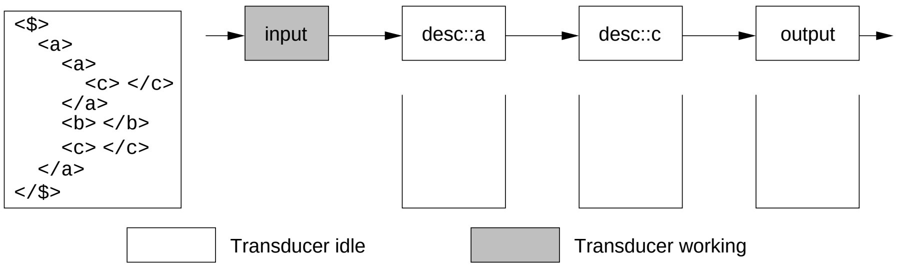
The nodes in the match scope are a subset of the nodes in the activation scope. E.g. the match scope of the child transducer includes all of the nodes directly contained in the activating node. If the activating message is not a start element (or start document) message, the match scope contains at most the activating message that is always included in the activation scope.

3.1.5 Processing a Simple Query

This section illustrates the working of a SPEX network by giving an example of the evaluation of a simple query. The example is presented stepwise by giving the configuration after each document message.

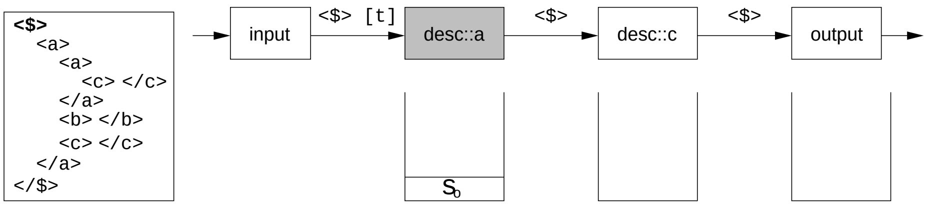
Example 3.2. Consider the query `/descendant::a/descendant::c`, which selects all `c` elements appearing under `a` elements in the document from Figure 3. Figure 9 shows a SPEX network for this query, where each box represents a transducer with the type given by the box label. The box to the left of the transducers shows the input data. The document message that is currently processed is shown in boldface. Below each transducer its count stack is shown. Transducers can be in two different kinds of states: idle and active. The box of a transducer is shown in gray, if the transducer is active. An idle transducer only forwards document messages, while an active transducer executes some task (e.g. a descendant transducer tries to match). In the initial configuration of the query system only the input transducer is active. All the other transducers in the system are idle and their stacks are empty.

Figure 9 Configuration before processing



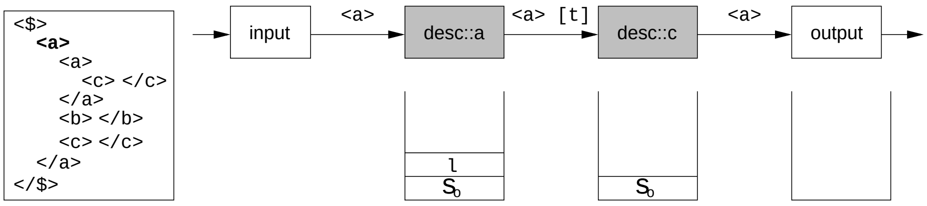
Step 1: processing of <$> The processing of the first message, the start-document message `<$>`, is shown in Figure 10. The input transducer writes an activation message `[t]` to its output tape and subsequently the start-document message `<$>`. Upon receiving the activation message, the `descendant::a` transducer switches to an active state where it tries to match further document messages, and puts a symbol s_0 on its count stack denoting the outermost activation scope of the transducer. When this symbol is on the top of the stack of `descendant::a` on any end-element or end-document message, the transducer will leave the active state, as the outermost activation scope is left, and no further document messages will be matched by this transducer. All transducers to the right of the `descendant::a` just forward the start-document message.

Figure 10 Configuration after Step 1



Step 2: processing of <a> The next message that is encountered is the start-element message `<a>` of the outermost `a` element, the processing of which is shown in Figure 11. The message `<a>` is received by the `descendant::a` transducer, which is in an active state and therefore tries to match the message. As it is successful, it writes an activation message `[t]` to its output tape and forwards the document message `<a>`. The `descendant::a` also puts an `l` symbol on its stack, which denotes another tree level encountered in the activation scope. The `descendant::c` transducer receiving the activation from the preceding transducer switches to an active state and puts an s_0 symbol on its count stack to mark the level of its initial activation just like the `descendant::c` transducer did when receiving the previous message.

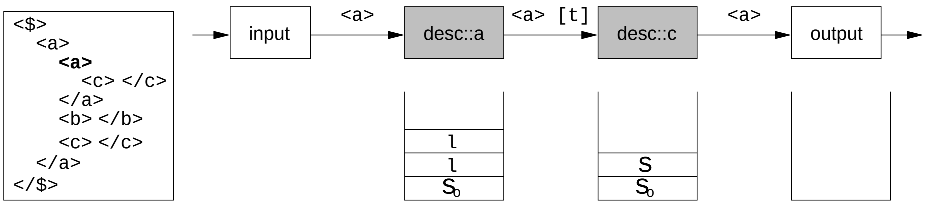
Figure 11 Configuration after Step 2



Step 3: processing of <a> The next message is the start-element message `<a>` of the inner `a` that is processed in a way similar to the previous `<a>`. The `descendant::a` matches and therefore

activates the `descendant::c` a second time. The `descendant::c` keeps track of this by putting an `s` symbol on its stack denoting the start of a nested activation scope. As the `descendant::c` is already in an active state, it also tries to match the current document message, but is not successful.

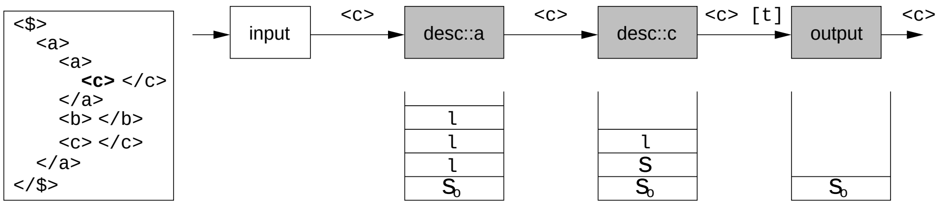
Figure 12 Configuration after Step 3



```
<$>
  <a>
    <a>
      <c> </c>
    </a>
    <b> </b>
    <c> </c>
  </a>
</$>
```

Step 4: processing of `<c>` The processing of the fourth message (the start-element message `<c>` of the first `c`) is shown in Figure 13. The `descendant::a` transducer is not successful in matching this message, but it still counts the tree level by pushing an `l` symbol. The `descendant::c` transducer also pushes an `l` on its stack, matches the current message `<c>`, and sends an activation `[t]` to the output transducer. Upon receiving this activation, the output transducer switches to an active state and keeps record of the activation scope by pushing an s_0 symbol on its stack. Further, it outputs the current message `<c>`, as it is part of the result.

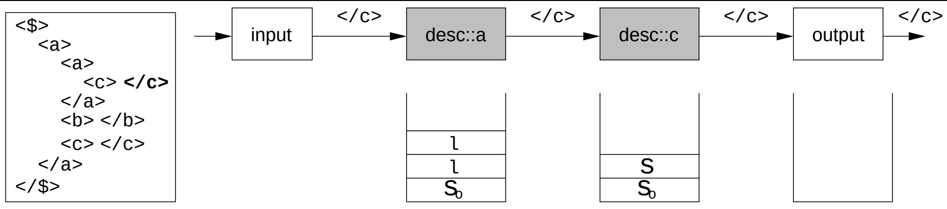
Figure 13 Configuration after Step 4



```
<$>
  <a>
    <a>
      <c> </c>
    </a>
    <b> </b>
    <c> </c>
  </a>
</$>
```

Step 5: processing of `</c>` The `descendant::a` and `descendant::c` transducers process the end-element message `</c>` by removing the `l` symbols from the top of their stacks, cf. Figure 14. The output transducer is still in an active state, therefore it outputs the current document message `</c>`. Additionally, as it is an end-element message and the top of the stack of the output transducer is an s_0 symbol, it switches back to an idle state and removes s_0 from its stack. This also means, that the encountered result has been written out completely.

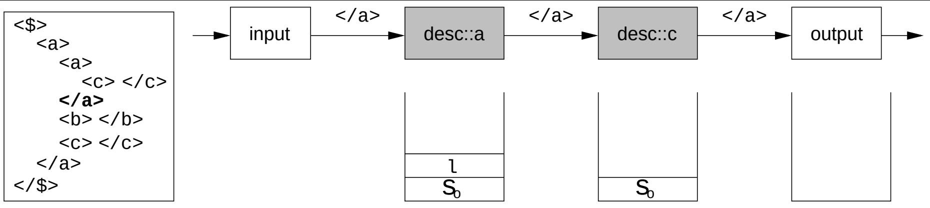
Figure 14 Configuration after Step 5



```
<$>
  <a>
    <a>
      <c> </c>
    </a>
    <b> </b>
    <c> </c>
  </a>
</$>
```

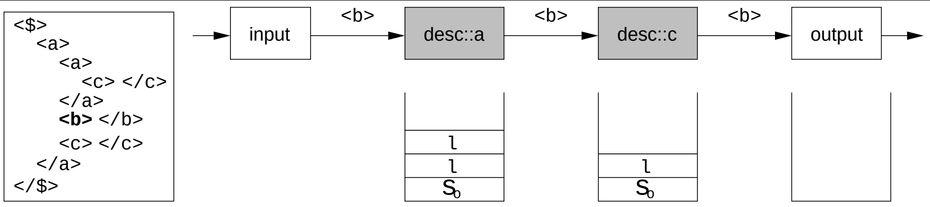
Step 6: processing of `</a>` The processing of the end-element message `</a>` of the inner `a` is shown in Figure 15. The `descendant::a` transducer removes the `l` symbol from the top of its stack, as the tree level is decreased when encountering an end-element message. The `descendant::c` encounters an `s` symbol on its stack that records the end of a nested activation scope, therefore removing it without leaving the active state.

Figure 15 Configuration after Step 6



Step 7: processing of `<b>` Figure 16 shows the processing of the start-element message `<b>`. Both of the `descendant::a` and `descendant::c` transducers are active and try to match the message. However, as the label of `<b>` doesn't match the node tests of the transducers, both of them are unsuccessful. Therefore they pass on the document message, pushing an `l` symbol on their count stack.

Figure 16 Configuration after Step 7



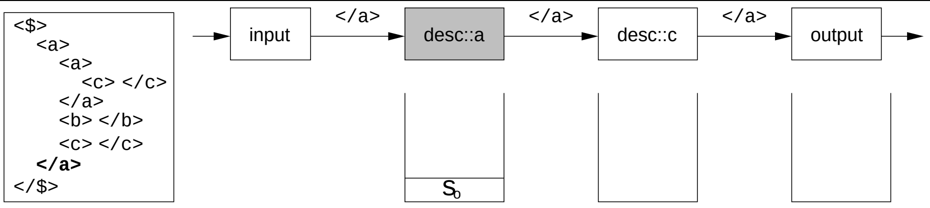
Step 8, 9, and 10: processing of `</b>`, `<c>`, and `</c>` The processing of the end-element message `</b>` is straightforward: the active transducers `descendant::a` and `descendant::c` remove the topmost symbol from the top of their count stack and pass on the message. No further action is performed, as in both cases an `l` symbol was removed.

The matching of the `descendant::c` transducer of the second `c` element leads to the same behavior as that of the first `c` : the output transducer is activated and outputs the current message `<c>`, as it is part of the result.

The processing of the message `</c>` that announces the end of the second `c` node, is equivalent to that of the first `c` node (cf. Step 5).

Step 11: processing of `</a>` When a transducer has an s_0 symbol on its stack and receives an end-element message, it leaves its outermost activation scope and therefore switches back to an idle state. This is the case in the `descendant::c` transducer when processing the end-element message `</a>` (cf. Figure 17). The `descendant::a` transducer still has an `l` symbol on its count stack and therefore stays in the active state removing the symbol from the top of its stack.

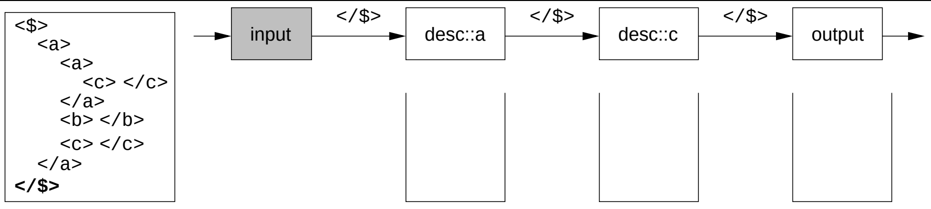
Figure 17 Configuration after Step 11



Step 12: processing of `</$>` The last message encountered is the end-document message `</$>` (cf. Figure 18). The `descendant::a`, that is the only active transducer, removes the

topmost symbol on its count stack and as this was an s_0, it switches back to an idle state. The input transducer switches back to an active state if the end-document message is received, hence being ready to process the next document.

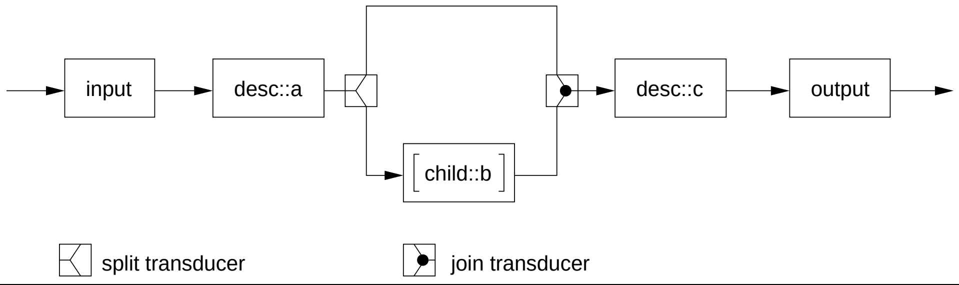
Figure 18 Configuration after Step 12



3.2 Branching and Synchronization

Up till now only simple chains of transducers were considered. However, for supporting more complex query constructs, e.g. qualifiers, it is necessary for some transducers to be able to send activation messages to more than one transducer. Two transducer types are introduced for this purpose. A *split transducer* is a special transducer that has two output tapes. Its operation is straightforward: Every message it receives (regardless of its type) is forwarded to both of the output tapes. The counterpart of the split transducer is the *join transducer* that uses two input tapes and a single output tape to merge the output of two other transducers. Instead of just forwarding every message received on an input tape to the output tape, the join transducer has to take care of the duplicates of messages created by a split transducer. Therefore, the processing of the join transducer is more complex. The details of this are presented below. An example of a SPEX network that uses split and join transducers is shown in Figure 19.

Figure 19 SPEX network for `/descendant::a[child::b]/descendant::c`



Note, that all of the interconnections are one-to-one relationships between an input tape of one transducer and the output tape of another transducer. Normally, this extends to a one-to-one relationship between transducers, as most of the transducers have just one input and one output tape. One-to-many relationships are only possible if one of the participating transducers is a split or join transducer.

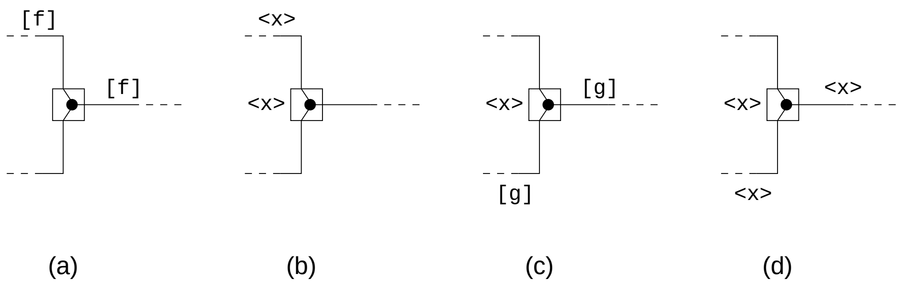
Further, the interconnections between transducers are directed, as a transducer has a read-only access to its input tape and a write-only access to its output tape. Hence, the communication flow is unidirectional. Additionally there should be no backward directed connections between transducers (i.e. there should be no cycles in a network). Therefore, a SPEX network is a directed acyclic graph (DAG).

Because of the introduction of branches in a SPEX network that allow parallel execution, mechanisms for the synchronization of transducers are necessary. There are three important requirements to the synchronization:

- The sending of document messages must be controlled by the input transducer, so as to guarantee that one and only one document message is in the system at any time.
- Messages must not be duplicated, i.e. no transducer should receive the same message two times.
- The order of events must not be changed. Especially, activation messages must always be received by a transducer before the document message that led to this activation.

As was already mentioned before, after sending a document message the input transducer waits until that message reaches the end of the transducer chain before sending the next message. However, without further mechanisms this solution would not fulfill all of the requirements, as after a split one branch might have a faster execution where a document message reaches the end of the chain while the same message is still handled by transducers of the second branch. Therefore, additional synchronization points are introduced with join transducers that work in the following way: if a document message is encountered on one of its input tapes for the first time, a join transducer does not forward this message. Only when the message is received another time from the second input tape, it is transfered to the output tape.

Figure 20 Example for the working of the join transducer



Example 3.3. Figure 20 illustrates the working of a join transducer. If the transducer receives an activation before a document message is encountered, it forwards the document message to its output tape (a). If it receives a specific document message for the first time (b) it keeps track of this (by entering in a special state – shown here by keeping record of the document message). However, activation messages are still passed (c). Only when the same document message is received for the second time, it is passed on to the output tape (d).

This ensures that, once a document message is passed on by a join transducer, it must have been processed by all transducers preceding the join. Hence, the output transducer is always the last transducer processing a specific document message. Duplicate removal and preservation of order is also guaranteed for document messages, as can easily be seen.

The special treatment of document messages in a join transducer is not necessary for activation and condition determination messages that are forwarded normally. This is correct for activation messages, as those are not forwarded by step transducers and between a split and the corresponding join there must be at least one step transducer on at least one of the lines. Therefore, the same activation message can only reach a join transducer from at most one input line.

Although the duplication of condition determination messages is non-critical (cf. Section 4), it can be ensured by having a special transducer on one of the branches after a split always remove determination messages.

3.3 Condition Handling

Conditions can be described as instances of qualifiers. The inclusion of a node in the result is determined by two different criteria: it must match the location path outside qualifiers and the qualifiers must be satisfied for the path to this specific node. A node that matches the location path is called a candidate. A candidate is related to a number of conditions by a boolean formula over those conditions (a *condition formula*). When the value of a formula is known, the corresponding candidate is either a result if the formula is satisfied or invalid otherwise.

There is also a relationship between an activation and the condition formula included in the corresponding activation message. An activation signals a matching at the specific position of the path. However, if the condition formula f of an activation still has an undetermined value, the match has only been performed under the reservation that f becomes true. In the case that f evaluates to true at some time, the matches signaled by the related activations are confirmed. If f evaluates to false, the matches are invalidated.

The transducers in a query system have different responsibilities. The step transducers outside qualifiers are responsible for determining candidates, but also for creating the relationship of candidates to conditions in the form of condition formulas. A condition formula for a specific candidate is created dynamically as step transducers try to match the incoming document. Steps inside qualifiers in contrast determine the value of conditions. However, step transducers inside qualifiers are not different from those outside qualifiers, the difference is in the interpretation of their work and the addition of the qualifier transducers that fulfill the logic of a qualifier.

3.3.1 Candidates and Conditions

Definition 4 (Candidates). A **candidate** is a node in the input document that might be a part of the result of the query, because all of the transducers of location steps outside qualifiers of the query location path matched. A candidate is created when the start message of its corresponding node is encountered and is only a **partial** candidate until the corresponding end message is received. Afterwards it is considered to be a **complete** candidate. A candidate is **undetermined** if the value of a corresponding condition (see below) is undetermined, else it is **determined** candidate. A determined candidate is a **result** if the corresponding conditions are true.

As already mentioned above, the status of a candidate is dependent on the value of the related qualifiers. However, a qualifier is evaluated relative to a node selected by the axis and node test of the qualifier's location step. For this reason the notion of *condition* is introduced here.

Definition 5 (Conditions). A **condition** is an instance of a qualifier relative to an activating node. A condition is **undetermined** at some point in time if its value is not known at that time, otherwise it is **determined**. A determined condition is either **true** if the qualifier is fulfilled relative to the activating node or **false** if it is known that the qualifier cannot be fulfilled any more.

The value of a condition is determined by the transducers of the location steps inside the corresponding qualifier. As soon as the last location step matches, the condition is determined with a value of *true*. Except for some special cases, the condition is known to be false if the corresponding activation scope of the first step in the qualifier is left.

As a location path can contain more than one qualifier, a candidate can be related to more than one condition. This can be represented with a logical formula.

Definition 6 (Condition Formulas). A **condition formula** is a formula of the propositional logic without negation where conditions are represented by variables.

A condition formula precises the relationship between a candidate and one or more conditions.

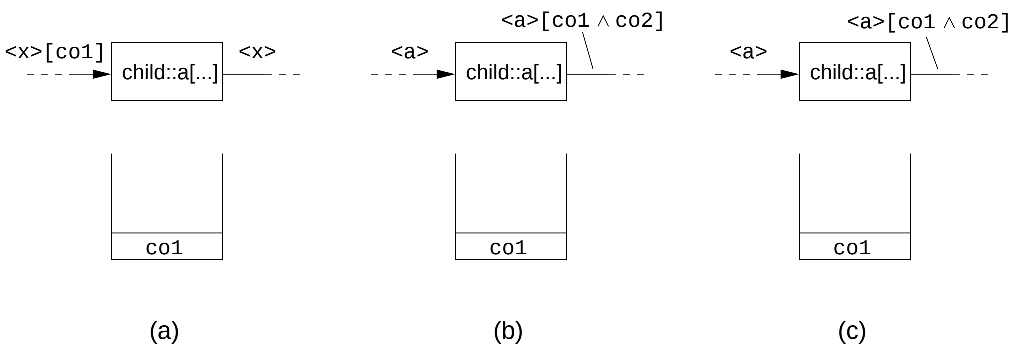
Example 3.4. When processing the query `/descendant::a[child::b]/descendant::c[child::d]` against the XML data in Figure 1, on the first `a` node a condition c_1 is created, on the second `a`, a condition c_2. For c_1 to be true, a `b` node must be contained directly inside the outer `a`, for c_2 directly inside the inner `a`. As soon as the start element message of the node with label `c` is encountered, a candidate Γ as well as another condition c_3 is created. Γ is a result if either c_1 or c_2 have a true value and also c_3 has a true value. This can be expressed in a logical formula $\Gamma = (c_1 \vee c_2) \wedge c_3$.

3.3.2 Setting up Condition Formulas

As mentioned above, condition formulas are created by step transducers while trying to match the location path they represent against the incoming stream.

When an active step transducer matches, it sends an activation message that includes the condition formula on the top of its condition stack. Additionally, if a location step has a qualifier, the corresponding step transducer creates a new condition on a match and sends an activation message with a condition formula that is a conjunction between the formula on top of its condition stack and the new condition[§].

Figure 21 Condition setup for a qualifier



Example 3.5. Figure 21 shows the setup of a condition in a child transducer that represents a location step with a qualifier. First, the transducer is activated with a condition *co1* (a), that every matching of the child transducer now depends on. Additionally, as this is a transducer that represents a step with a qualifier, it has to create a new condition if it matches. The activation message that is send by the transducer in the case of matching is a conjunction of the old formula on the stack and the newly created condition (b). Every time the transducer matches, it creates a new condition to include in the corresponding activation message (c). Note, that in this particular case *co1* is shown as an atomic condition. However, the formula a transducer is activated with can be arbitrarily complex.

A step transducer that receives an activation message stores the included condition formula on its condition stack. Depending on the axis of the transducer this is done in different ways. A

[§]Actually, in Section 4 the task of condition creation is delegated to a new transducer type for reasons of modularization, but for the moment it is assumed that the step transducer itself creates conditions.

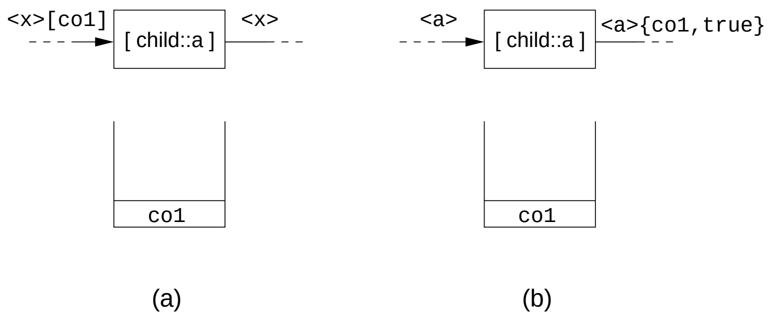
child transducer for example, only stores the new condition formula on the top of the condition stack. A descendant transducer, however, stores a disjunction of the previous entry and the new condition formula. This difference is due to the different relationships between match scopes in the child and descendant transducers (cf. Section 3.3.4).

When the last step transducer in the system matches with a specific condition formula, a new candidate has been identified that depends on this formula. As soon as the formula can be evaluated to a value of *true* (possibly even at the moment of candidate identification), the candidate is a result. Conversely, a candidate is invalid and can be removed when its condition formula is evaluated to *false*. Until this determination of the condition formula of a candidate, the document messages the candidate is composed of have to be stored.

3.3.3 Determination of Condition Values

Step transducers inside a qualifier work together to determine the value of conditions of that qualifier. This is done in a way similar to step transducers outside qualifiers. However, where the steps outside create a candidate if the last step matches, the last step inside a qualifier sends a determination message with a value of *true* for some conditions in this case. The conditions that are determined to be true are contained in the condition formula on top of the condition stack of the last step.

Figure 22 Determining the value of a condition



Example 3.6. Figure 22 shows a child transducer that is responsible for determining the value of a condition, because it is the last step in a qualifier. Like a child transducer outside a qualifier, this transducer puts the condition included with an activation on its condition stack (a). If it matches some time later, it sends a condition determination message with a value of true for the stored condition (b).

However, a condition formula on the stack of a transducer can be arbitrarily complex and can also contain conditions that were created for another qualifier (i.e. a qualifier of a location step that precedes the matching step). Therefore, it is necessary to add the information to a condition about the qualifier it is related to, so as to be able to extract from a condition formula all of the conditions related to a given qualifier.

The described behavior of the last step in a qualifier only covers the case where a condition can be determined to true. The case of a false condition must be handled differently. A condition is known to be false if it is not contained in any formula on a condition stack of any transducer inside the qualifier any more and the condition was not determined to be true before. There are cases where this can easily be detected, e.g. when all of the steps inside a qualifier are of type *descendant* or *child*, a condition is absent from all condition stacks in the qualifier if the first step in the qualifier leaves the activation scope that introduced the condition. In some other cases involving steps with axes *following* and *following-sibling*, this decision cannot be easily made.

3.3.4 Axis-Specific Behavior

The matching behavior and also the condition-formula handling of a step transducer depend on the axis of the supported location step. The notions of activation scope and match scope, defined in Section 3.1.4 are very important here. Therefore, they are summarized again.

The activation scope is defined relative to an activating message, i.e. a document message directly following an activation message[¶]. The activating message is directly related to the condition formula that is included with the corresponding activation message. This is reflected by the fact that during the whole activation scope the corresponding condition formula is stored somewhere on the condition stack. Actually, this is an important characteristic of an activation scope: it consists of the range of messages during which a transducer keeps record of the condition formula corresponding to the activating message.

The match scope of a transducer can be more precisely defined. It is the subset of messages from the activation scope that a transducer tries to match because of the position of the corresponding nodes in the document tree. The match scope of a transducer, but also its activation scope depend on the supported axis, e.g. the activation scope of a child transducer comprises the whole range of messages that represent the subtree of the input document tree starting at the activating node, and the match scope consists of the direct children of the activating node.

Whereas the matching behavior of step transducers can be directly derived from their match scopes, the condition formula handling depends on the possible relationships between the match scopes of nested activation scopes. There are two different possibilities:

- The match scopes of two nested activation scopes are disjunct, i.e. there is no message in the nesting activation scope that is contained in both match scopes.
- The match scopes of two nested activation scopes are also nested, i.e. the match scope of the nested activation scope is nested in the match scope of the nesting activation scope.

If the relationship between match scopes of a transducer can only be of the first type, as is the case with the child and self transducers, every nested activation leads to a new match scope. In this case, the transducer has to put the condition formula of the new activation on top of the old formula of the previous activation.

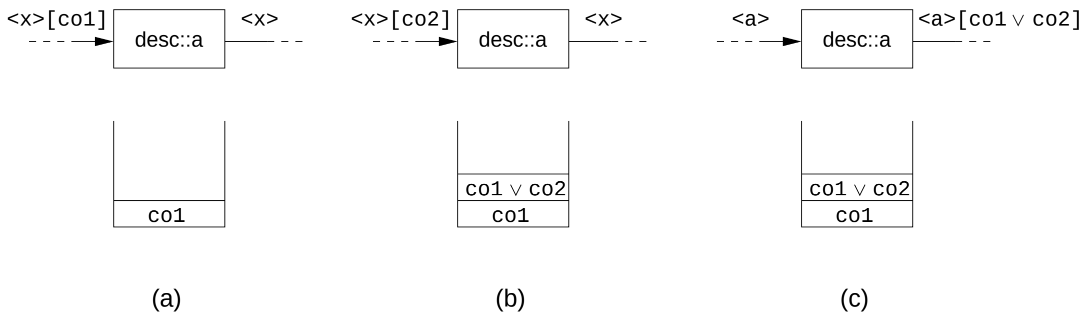
If the relationship can only be of the second type, as is the case with the descendant and following transducers, matchings that are performed after a nested activation are done for both the old and the new match scope. Therefore, the transducer has to put a disjunction of the condition formula of the old match scope and the formula of the new activation on top of the condition stack.

The worst case is if the relationships between two match scopes of nested activation scopes can be of both types: either disjunct or nested. Here, the decision of putting a disjunction of the old and the new formula or just the new formula on top of the condition stack depends on additional criteria. E.g. in the following-sibling transducer it is known that a nested match

[¶]Note the important difference between an activation message and an activating message that is a document message.

scope was encountered when the transducer gets an activation while it is already in the match scope. In all other cases the match scopes are disjunct.

Figure 23 Condition handling of the descendant transducer

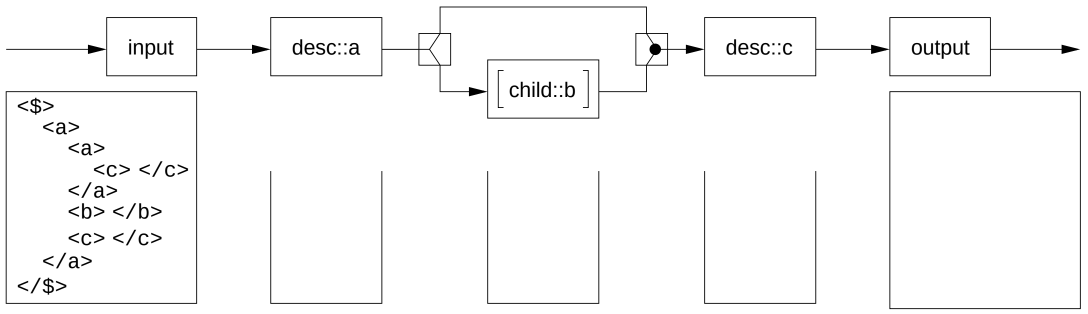


Example 3.7. As an example for the case where match scopes of nested activation scopes can only be nested, the condition handling of the descendant transducer is shown in Figure 23. If the condition stack of a descendant transducer is empty when it receives an activation, it simply puts the included condition (*co1* in Figure 23) on its condition stack (a). If it receives an activation message while there is already an entry on the condition stack, it pushes a disjunction of the old condition formula on the stack and the new formula received with the activation (b). Upon matching it sends an activation message that includes the condition formula on top of the condition stack (c).

3.3.5 Processing a Qualifier

In Section 3.1.5 an example of processing a simple query without a qualifier was introduced. There, the focus was on the explanation of state changes and use of the count stack of the transducers. This section illustrates the usage of conditions and condition formulas to process a query that includes a qualifier. Therefore, the count stacks and the states of the transducers are not shown here.

Figure 24 Configuration before processing

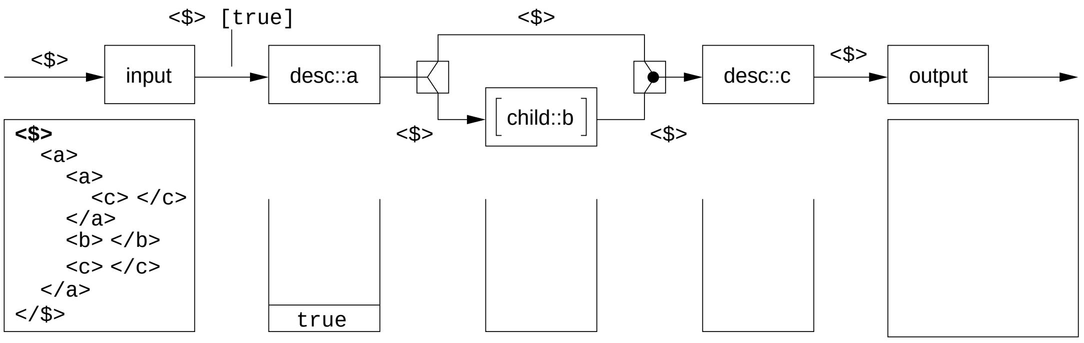


Example 3.8. Consider the query `/descendant::a[child::b]/descendant::c`, which selects all `c` elements appearing under `a` elements, that also have a child `b`, from the document in Figure 3. Figure 24 shows a SPEX network for the proposed query where each box represents the transducer with the type given by the box label. Directly below each step transducer its condition stack is shown. The box below the input transducer shows the input document and the position therein. The box below the output transducer shows the candidates currently in the system. Note, that in Section 4 a qualifier is formalized by a chain of several transducers.

Here it is referred to as a single qualifier transducer. In the initial network configuration, all stacks are empty.

Step 1: processing `<$>` The first encountered message is the start-document message `<$>` that is processed by the input transducer by writing an activation message `[true]` to its output tape. The condition formula contained within the activation message is `true`, as there are no qualifiers before the input transducer. The `descendant::a` transducer receives the activation message `[true]`, pushes the (trivial) condition formula on its condition stack and forwards the start-document message `<$>`. The other transducers are not activated yet, hence they just forward the message.

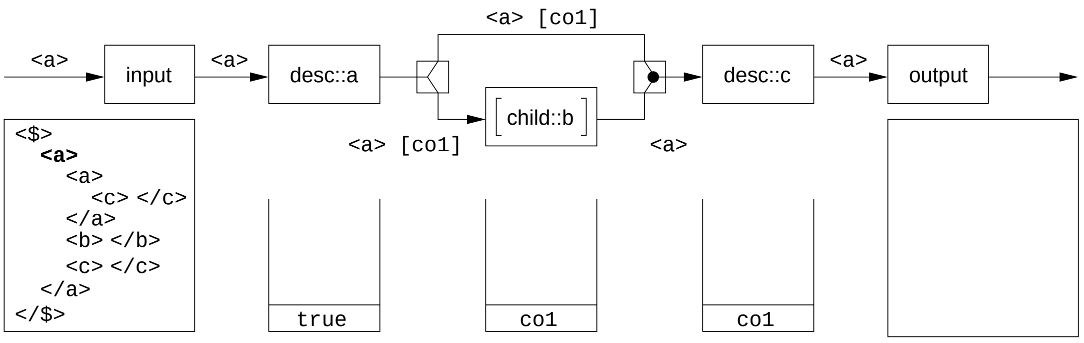
Figure 25 Configuration after Step 1



```
<$>
  <a>
    <a>
      <c> </c>
    </a>
    <b> </b>
    <c> </c>
  </a>
</$>
```

Step 2: processing `<a>` Figure 26 shows the processing of the second document message, the start-element message `<a>` corresponding to the outer `a`. The `descendant::a` was already activated and it succeeds to match the current document message. The location step corresponding to this transducer has a qualifier, represented by the qualifier transducer `[child::b]`, which now has the task to determine the value of a condition related to the current document message. A condition variable *co1* is created to represent this condition. The `descendant::a` transducer sends an activation message `[co1]` to its output tape, containing a formula that consists of a conjunction of the old formula on the condition stack and the newly created condition variable. As the old formula was the atomic formula `true` the new formula is simplified to *co1*. This activation message is forwarded through the split transducer to the `[child::b]` qualifier transducer and the `descendant::c` transducer. Both transducers treat the activation message in the usual way: they switch to an active state and push the formula contained within the message on their condition stacks. The join transducer has the task of synchronizing the network flow by waiting for both document messages to arrive on its input tapes and afterwards forwarding only one document message to the `descendant::c` transducer.

Figure 26 Configuration after Step 2

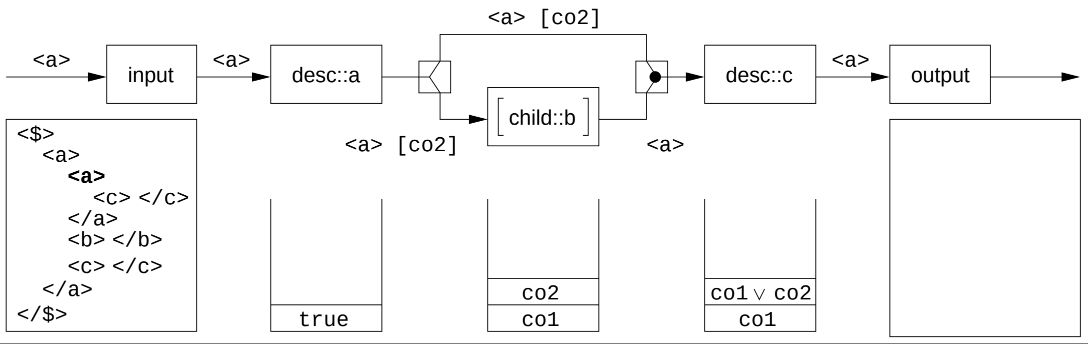


```
<$>
  <a>
    <a>
      <c> </c>
    </a>
    <b> </b>
    <c> </c>
  </a>
</$>
```

27

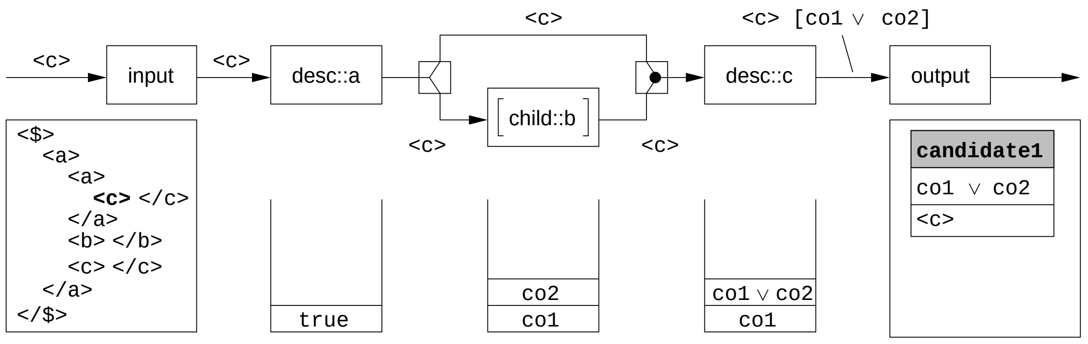
Step 3: processing `<a>` The processing of the third document message, the start-element message `<a>` corresponding to the inner `a`, is similar to the previous processing step. A new condition variable, *co2* is created, and passed through the network. The `descendant::c` transducer receives *co2* and pushes on its condition stack a disjunction between the top of its stack and the newly received formula: $co1 \vee co2$. In this way, a descendant transducer keeps track of the condition formulas of nested activations. That is, a document message `<c>`, matched starting with this tree level, can be part of the result if at least one of the condition variables *co1* and *co2* is fulfilled. The `descendant::c` transducer also tries to match the current document message but is not successful, as the node test doesn't match. However, if it would have matched at this moment, a correct behavior would have required that the `descendant::c` transducer sends an activation not with the topmost formula on its stack ($co1 \vee co2$), but rather with the old formula *co1*. This is due to the fact that the match scope of the new activation starts with the next tree level.

Figure 27 Configuration after Step 3



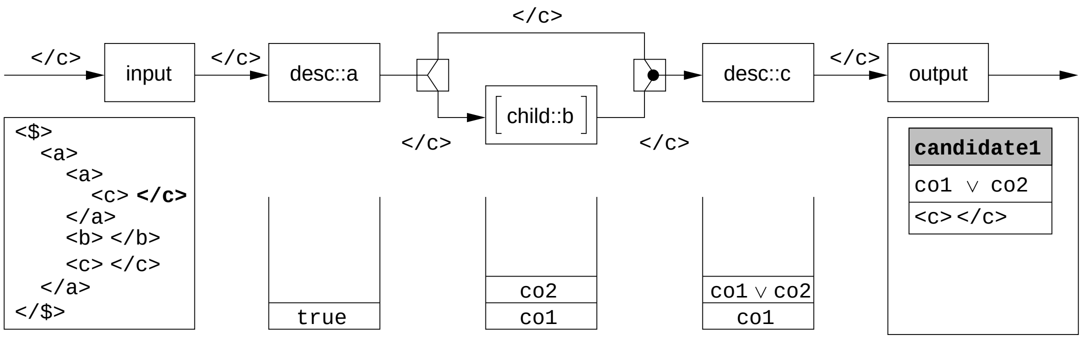
Step 4: processing `<c>` Figure 28 shows the processing of the fourth document message, the start-element message `<c>` corresponding to the first `c`. The `descendant::c` transducer matches and sends an activation message [$co1 \vee co2$] to the output transducer including the formula from the top of its stack. As the value of both *co1* and *co2* is still undetermined, the newly created *candidate1* is not yet determined as being part of the result. Therefore, it is stored in the candidate storage, waiting for its formula to be determined.

Figure 28 Configuration after Step 4



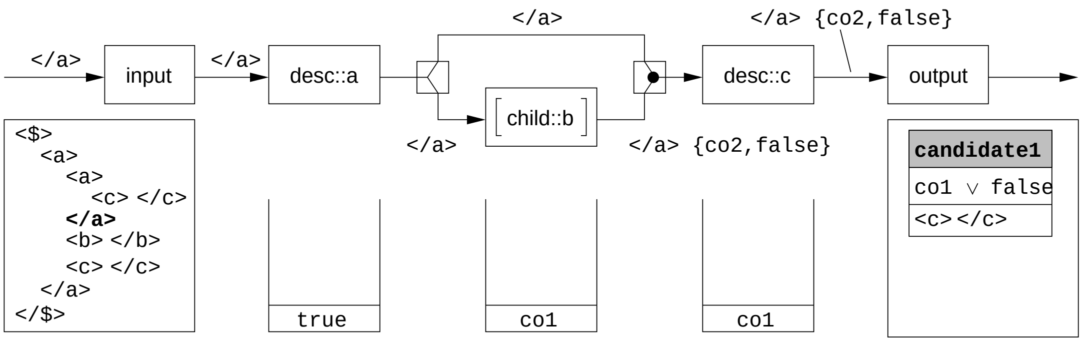
Step 5: processing `</c>` Afterwards, the end-element message `</c>` of the first `c` is processed by the network (cf. Figure 29). Most of the transducers take no other action than updating their count stack which is not shown here. Only the output transducer notes the end of *candidate1* with this end-element message and therefore closes the candidate after adding the message to its message store. Now, the candidate is complete.

Figure 29 Configuration after Step 5



Step 6: processing `</c>` Figure 30 shows the determination of the condition *co2* by the `child::b` transducer. The value of the condition is known to be false, as the inner `a` is left by the document message `</a>` without a `b` child encountered inside. The `child::b` removes *co2* from its stack and sends a condition-determination message {`co2,false`} with a false value for *co2*. The message is received by `descendant::c` as well as the output transducer. `descendant::c` updates its stack by replacing all instances of the condition variable *co2* with the value *false*. The output transducer searches through all of its candidates to find those that rely on the determined condition *co2* in order to output results or to remove invalid candidates. However, the only candidate in the storage, i.e. *candidate1*, still relies on the undetermined condition *co1*, hence its status can not be changed. Note, that in Figure 30 the entry *co1* ∨ *co2*, from the condition stack of the `descendant::c` transducer, has not been removed because of the determination of *co1*, but rather because the activation scope corresponding to this formula has been left.

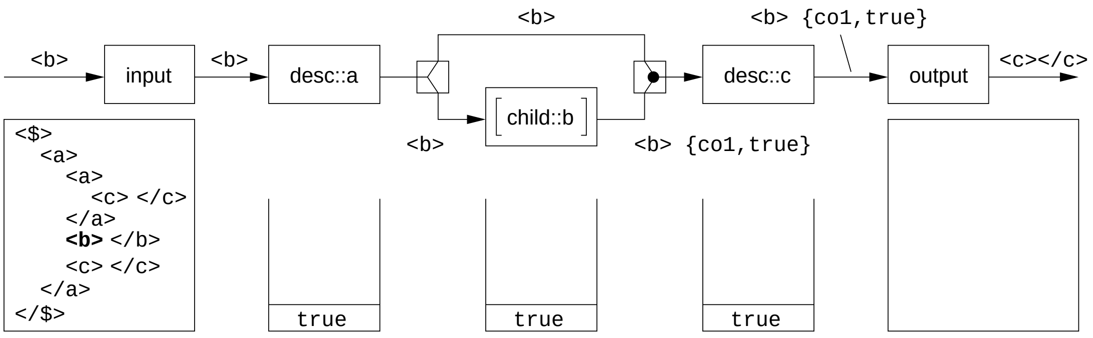
Figure 30 Configuration after Step 6



Step 7: processing `<b>` In Figure 31 the qualifier transducer `[child::b]` matches the document message `<b>`. Therefore it sends a condition-determination message {`co1,true`} for *co1* with a `true` value. Transducer `descendant::c` updates its stack by replacing *co1* with the value `true`. The output transducer updates the condition formula of *candidate1*. As this condition formula is now true, the candidate is known to be part of the result and therefore its stored messages are written to the output tape. The candidate is already complete, hence it is removed from the candidate storage.

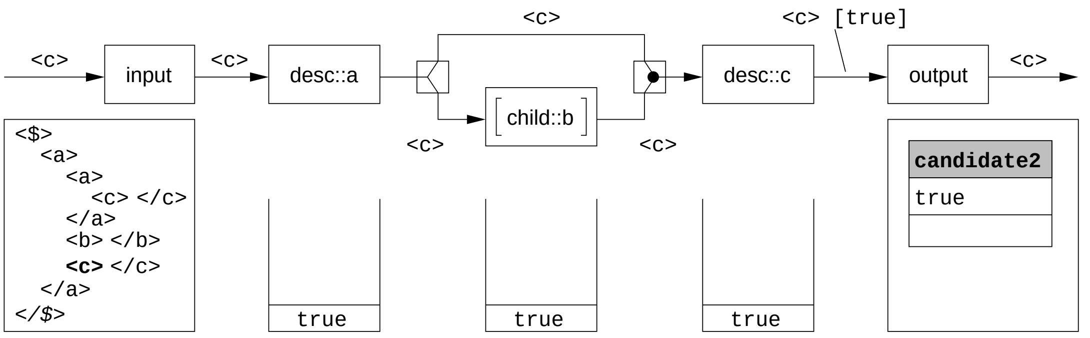
Step 8: processing `</b>` When the end-element message `</b>` is encountered, the only actions taken by the transducers are to update their count stacks (that are not shown here).

Figure 31 Configuration after Step 7



Step 9: processing `<c>` Figure 32 shows the processing of the start-element message `<c>` of the second `c`. This message is matched by the `descendant::c` transducer. The condition formula on top of its condition stack, i.e. the atomic formula `true`, is sent via an activation message `[true]` to the output transducer. There, a new candidate *candidate2* is created which is already determined. Therefore, it is not put in the candidate storage, but rather directly forwarded to the output tape. The candidate is shown in Figure 32 only for illustrative purposes. This behavior applies to all expressions with past conditions, i.e. conditions whose values are unknown at the time of candidate creation. Thus, the progressive aspect of the evaluation model is ensured.

Figure 32 Configuration after Step 9

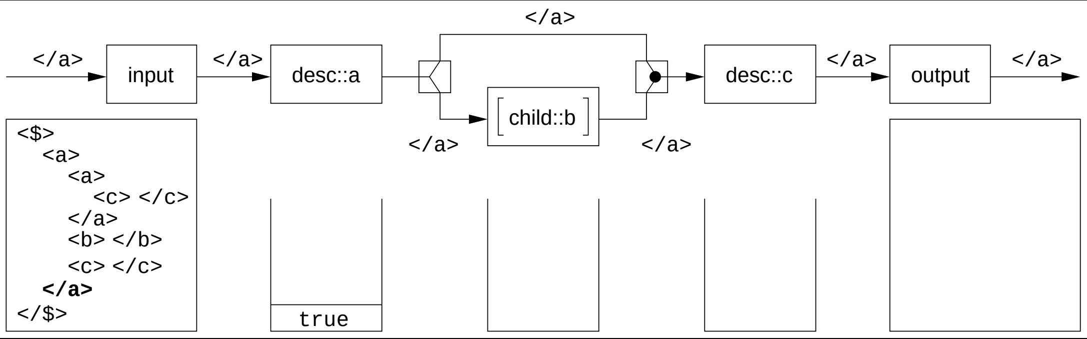


Step 10: processing `</c>` The end-element message `</c>` of the second `c` node signals the end of *candidate2*. As the candidate is determined to be a result, the document message `</c>` is output. Afterwards, the candidate is closed.

Step 11: processing `</a>` Figure 33 shows the processing of the end-element message `</a>` of the outer `a`. If *co1* would not have been determined to be true before, it would now be set to false. The `[child::b]` and `descendant::c` transducers leave the last activation scope removing the topmost entries of their condition stacks and switching back to an idle state. Only the `descendant::a` transducer is still active and looking for `a` elements.

Step 12: processing `</$>` The last message, the end-document message `</$>` signals the `descendant::a` transducer that its outermost activation scope is left. Therefore, also this transducer removes the last entry on its condition stack and switches to an idle state. Afterwards, all of the condition stacks of the transducers are empty and the system is ready to process the next input document.

Figure 33 Configuration after Step 11



3.4 Summary of Transducer Types

The different types of transducers that exist in a query system were already mentioned several times. They are summarized here:

- The **input transducer** receives the input of the query system, possibly performing a transformation to document messages, and forwards these messages to the next transducer one at a time.
- The **output transducer** has several very important tasks. Most notably, it has to buffer undetermined candidates until it is known whether they are a part of the result or not. The candidates that are part of the result have to be output in correct order (i.e. document order).
- A **split transducer** is a very simple transducer with one input tape and two output tapes. Every incoming message is duplicated and sent to both of the output tapes.
- A **join transducer** is more complex than its counterpart the split transducer. It consists of two input tapes and one output tape. Activation and condition determination messages arriving on any input tape are immediately forwarded, whereas a document message is held back until it also arrives on the second input tape[‖].
- The class of **step transducers** comprises the transducers used for location steps in a location path. There is one type of step transducer for every supported axis (e.g. child, descendant). The basic processing of all these transducers is similar. At query start a step transducer is idle (only forwarding document and condition determination messages). Normally (depending on its axis), when it gets activated it switches to an active state where it tries to match document messages. If it is successful, it sends an activation message to the next transducer. Occasionally, a step transducer leaves the active state when it leaves an activation scope. A step transducer knows when to enter and leave these states by counting the current level in the tree. Additionally, step transducers have to keep track of condition formulas passed with activation messages. This is explained in more detail in Section 3.3.
- **Qualifier transducers** are needed to fulfill the logic of a qualifier. They are inserted at certain places inside a SPEX network. More on qualifier transducers can be found in Section 4.3.

[‖]Remember that system synchronization guarantees that only one document message is inside the system at any time, although possibly with multiple copies.

- **Set transducers** carry the logic for performing a set operation on two node sets. Two types of set transducers exist: the union transducer for the support of the union operation and the intersect transducer for intersections. More information on set transducers can be found in Section 4.5.

4 Formalization

In this section various components, as they are used in the presented query system, are formally described. As informally presented in Section 3, an XPath expression is evaluated on a stream by means of a network of interconnected components, that provide different functionalities to the system, prominently those of the various XPath constructs. These components are defined as *SPEX transducers* and connected in a *SPEX network*.

4.1 The SPEX Transducer Network

A finite-state transducer is an abstract computing machine that consists of a finite state control, an input tape with a read-only input head, and an output tape with a write-only output head. A *pushdown transducer* is basically a finite-state transducer augmented with a pushdown store.

A SPEX pushdown transducer, as introduced here, is similar to a conventional pushdown transducer, except that it does not have accepting states, and uses two stacks instead of one: the condition stack used exclusively for keeping track of condition formulas on which query results depend and the count stack used to count the current level in the document tree. However, it is possible to define all SPEX transducers (except the output transducer) as pushdown transducers with only one stack, as the count stack and the condition stack are used in a highly synchronized way. The transducers presented here introduce two stacks only for reasons of conciseness.

Definition 7 (SPEX Transducer). A pushdown transducer T with two stacks, the *count* and the *condition* stack, is called a SPEX transducer:

$$T = (\mathcal{Q}, \Sigma, \Omega, \Gamma_{count}, \Gamma_{cond}, q_0, \delta), \text{ where}$$

- $\mathcal{Q}$ is a finite set of states,
- Σ is the input alphabet,
- Ω is the output alphabet,
- Γ_{count} is the alphabet of the count stack,
- Γ_{cond} is the alphabet of the condition stack,
- $q_0 \in \mathcal{Q}$ is the initial state, and
- δ is the transition function

$$\delta : \Sigma \times \mathcal{Q} \times \Gamma_{count} \times \Gamma_{cond} \rightarrow \mathcal{Q} \times (\Gamma_{count} \cup \{\epsilon\}) \times (\Gamma_{cond} \cup \{\epsilon\}) \times (\Omega \cup \{\epsilon\}).$$

Each transition depends on the next input symbol, the current state, the top of the count stack, and the top of the condition stack and possibly changes the state, the top of the two stacks, and the output tape.

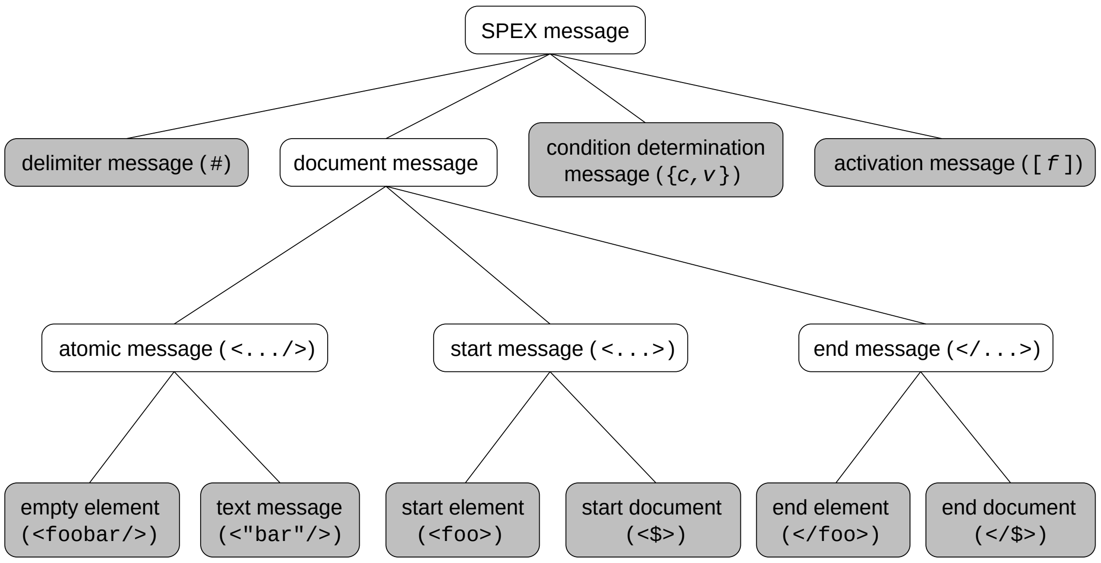
The transitions of SPEX transducers are specified in the following sections by means of configurations. A configuration is a triple $\phi \in \mathcal{Q} \times \Gamma^*_{count} \times \Gamma^*_{cond} = \Phi$. Hence, the transition relation is specified by a relation from $\Sigma^+ \times \Phi$ to $\Phi \times \Omega^*$, i.e. as a relation between configurations of the transducer under certain input symbols and possibly yielding some output symbols.

SPEX transducers can be varied by adding extra input or output tapes, as shown in Section 4.4. For conciseness, the notation $T^{in_1,\ldots,in_n}_{out_1,\ldots,out_m}$ is used to refer to a SPEX transducer with input tapes

$in_1, \ldots, in_n$ and output tapes $out_1, \ldots, out_m$. In Sections 4.2 through 4.6 several types of SPEX transducers are introduced.

A collection of SPEX transducers and their connections establish a SPEX network. A connection is used to transmit different kinds of messages between transducers: document messages, activation messages, condition determination messages, and delimiter messages.

Figure 34 SPEX message types



There are four kinds of messages in a SPEX network:

- A **Document message** corresponds to a message of the XML stream. As payload it carries the label of the corresponding element or the value of the corresponding text node. Document Messages are represented in the transducer specifications by angle brackets enclosing the payload of the message. They are further subdivided into the following types:
  - A **start message** signals the start of an element or the start of the document. It is represented by the label of the element inside angle brackets, e.g. `<a>`. The start document message uses the special symbol `<$>`.
  - An **end message** signals the end of an element or the end of the document. It is represented by the label of the element preceded by a slash and enclosed by angle brackets, e.g. `</a>`. Every end message corresponds to a previously received start message. The end document message uses the special symbol `</$>`.
  - An **atomic message** represents a node in the document tree that has no children, e.g. a text node. An atomic message is represented by a representation of the value of the message (e.g. the text of a text node in quotation marks) followed by a slash, e.g. `<"some text"/>`.
- An **activation message** `[`f`]` is used to activate transducers with a *condition formula* f that is carried by the message, and indicates that the activated transducers yield result under the reservation that f becomes `true`. A condition formula is an arbitrary formula of the propositional logic without negation, i.e. it consists of conjunctions and disjunctions of condition variables.

- A **condition determination message** $\{c, v\}$ is used to signal the truth value v of a condition variable c.
- A **delimiter message** is used as a grouping construct for messages, most prominently activation messages (see Sections 4.3.2 and 4.3.3 for an example of message grouping). It is represented by the special symbol $\#$.

Figure 34 summarizes the different kinds of messages that can appear inside a SPEX network.

A condition formula, as conveyed by an activation message, can be represented as a tree where the inner nodes are labeled with `and` and `or` boolean connectors and the leaves are carrying the condition variables. A condition variable represents an instance of a qualifier, and can have a truth value. In general, transducers do not look inside condition formulas, they just receive, store and send them as they are. However, there are transducers which need to decompose formulas, e.g. into a stream of condition variables, in order to filter out or to check some of them. As example consider the variable-filter transducer presented in Section 4.3.2. Other transducers need to process condition formulas in order to set condition variables to a value that was conveyed by a condition determination message, e.g. step transducers and the output transducer.

Definition 8 (SPEX Network). A SPEX Network SN of degree n is a tuple

$$SN = (DM, AM, CM, LM, G), \text{ where}$$

- DM, the set of document messages, is the input and output alphabet of the network,
- AM is the set of activation messages that are passed inside the network,
- CM is the set of condition determination messages that are passed inside the network,
- $LM = \{\#\}$ is the set of delimiter messages that are passed inside the network,
- $G = (N, V)$ is a directed acyclic graph with nodes N and vertices V.

  The nodes of G are n SPEX transducers: $N = \{T_1, \ldots, T_n\}$,

  $$T_i = (Q_i, \Sigma, \Omega, \Gamma_i^{count}, \Gamma_{cond}, q_i^0, \delta_i), i \in \{1, \ldots, n\}$$

  where $\Sigma = \Omega = DM \cup AM \cup CM \cup LM$, and $\Gamma_{cond} = \{f \mid [f] \in AM\}$, and Γ_{count}, q_i^0 and δ_i and Q_i depend on the type of the SPEX transducer.

  The vertices V represent the connections between SPEX transducers: $(T_i, T_j) \in V$, if an output tape of T_i is connected with an input tape of T_j.

Using the short-hand notation for transducers, the graph G can also be represented as a set of transducers, where the vertices are implicitly given by the identification between the input and output tapes of different transducers:

$$G = \{\mathsf{T_1}{}^{in_1}_{out_1}, \ldots, \mathsf{T_n}{}^{in_n}_{out_n} \mid (\mathsf{T_i}{}^{in_i}_{out_i}, \mathsf{T_j}{}^{in_j}_{out_j}) \in V \Leftrightarrow |out_i \cap in_j| = 1\},$$

where out_i (resp. in_i) is the ordered set of output (resp. input) tapes and $\mathsf{T_i}$ the type of transducer i.

The compilation of an XPath expression to a SPEX network is shown in Section 4.7, after introducing each SPEX transducer type by specifying its transition relation. For all the transducers, there are implicit transitions which forward document messages. These transitions are used only if there is no other defined transition to be executed.

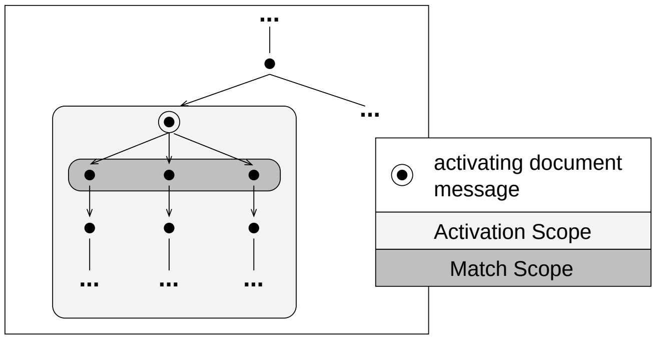
4.2 Step Transducers: $\mathsf{ST}^{in}_{out}(s)$

The task of a step transducer for a step s is to perform the matching of messages from the input document considering the axis and the node test of s. The specificities of the different axes are covered by different step transducers introduced below. The matching of a document message against a node test can easily be done by looking at its type and payload. Therefore, the execution of node tests is not considered in the specifications. For conciseness, in the transitions of a step transducer where the result of a node test must be considered, this result is checked by using different variables for the payload of document messages. A subscript of m on variables representing a label of an element (a value of a text node, resp.) is used to denote a label (value, resp.) that matches the node test of that transducer, e.g. $<l_m>$. A subscript of n is used for a non-matching node test.

4.2.1 Child Transducer

A child transducer, as presented in Figure 36, implements the logic of a child step from XPath. The activation scope of the transducer contains all document messages that are descendants of the activating document message in the tree in addition to the activating message itself. However, as shown in Figure 35 the match scope only contains those messages in the activation scope that are direct children of the activating node, i.e. that have a tree level of $n + 1$, where n is the tree level of the activating document message.

Figure 35 Activation and Match Scope of the `child` transducer



For matching only direct children of an activating message, the transducer must keep track of the tree level, i.e. the depth reached in the (unmaterialized) document tree, and distinguish between the levels of direct children and the other tree levels; these levels are marked with different symbols (`m` for match scope, `l` for level) on the count stack.

The transducer can be activated in two contexts: while waiting for being activated, as shown in transition (1) of Figure 36, or while trying to match, as result of a previous activation (7). Hence, it is able to match in several activation scopes originating from different activating document messages. Consider the expression `/descendant::a/child::b` on the stream shown in Figure 3: The child transducer for the step `child::b` is activated for the first `<a>` message while waiting (1), and for the second `<a>` message while trying to match `b` children of the first `a` (7).

In the `working` state, the transducer tries to match start messages (8,9) and atomic messages (10). If successful, it activates the next transducer by sending an activation message containing the topmost boolean formula from its condition stack (8,10). In any case, if the current tree level is increased by encountering a start message, the child transducer has to wait until it is back in the match scope again. This is done by pushing an `m` on the count stack and entering the `waiting` state(8,9), from where it will reenter the `working` state if it encounters the `m` on

Figure 36 Transitions of the child transducer

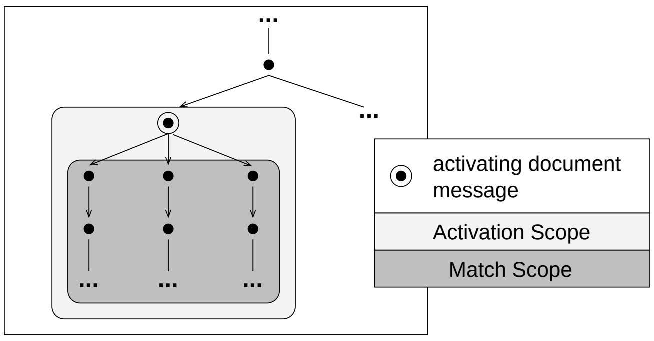
$\Gamma^{\texttt{count}}_{child}$ = {m,l} $\mathcal{Q}_{\texttt{child}}$ = {waiting,working,activate1,activate2} $q^0_{\texttt{child}}$ = waiting

1	([f]	, (waiting	, α,	β))	→ ((activate1,	α,	f|β),	ϵ	)
2	(<l>	, (waiting	, α,	β))	→ ((waiting	, l|α,	β),	<l>	)
3	(</l>	, (waiting	, l|α,	β))	→ ((waiting	, α,	β),	</l>	)
4	(</l>	, (waiting	, m|α,	β))	→ ((working	, α,	β),	</l>	)
5	(<l>	, (activate1,	α,	β))	→ ((working	, l|α,	β),	<l>	)
6	(<t/>	, (activate1,	α,	β))	→ ((waiting	, α,	β),	<t/>	)
7	([f]	, (working	, α,	β))	→ ((activate2,	α,	f|β),	ϵ	)
8	(<l_m>	, (working	, α,	f|β))	→ ((waiting	, m|α,	f|β),	[f];<l_m>	)
9	(<l_n>	, (working	, α,	β))	→ ((waiting	, m|α,	β),	<l_n>	)
10	(<t_m/>,	(working	, α,	f|β))	→ ((working	, α,	f|β),	[f];<t_m/>	)
11	(</l>	, (working	, l|α,	f|β))	→ ((waiting	, α,	β),	</l>	)
12	(</l>	, (working	, m|α,	f|β))	→ ((working	, α,	β),	</l>	)
13	(<l_m>	, (activate2,	α,	f_1|f_2|β))	→ ((working	, m|α,	f_1|f_2|β),	[f_2];<l_m>	)
14	(<l_n>	, (activate2,	α,	β))	→ ((working	, m|α,	β),	<l_n>	)
15	(<t_m/>,	(activate2,	α,	f_1|f_2|β))	→ ((working	, α,	f_2|β),	[f_2];<t_m/>	)
16	(<t_n/>	, (activate2,	α,	f|β))	→ ((working	, α,	β),	<t_n/>	)
17	({c,v}	, (*any_state*	, α,	β))	→ ((*any_state*	, α,	UPDATE(c,v,β)),	{c,v}	)

an end message (4). Similarly, if an end message is encountered in the `working` state, the transducer leaves the innermost activation scope, removing the topmost condition formula from the condition stack (11,12). Depending on the top of the count stack, the transducer stays in the `working` state if the next tree level is in the match scope of another activation (12), or switches to the `waiting` state otherwise (11). A condition determination message {c,v} is processed by a child transducer (as well as all other step transducers) by updating its condition stack, setting every occurrence of c to the value v (17).

Note, that in general, transitions of SPEX transducers consider only one input symbol. In some cases, it is more convenient for step transducers to look at two consecutive input messages, e.g. while getting activated they receive an activation message and a document message which led to the activation. These two messages are processed together and their combination results in different transitions of a step transducer. For consistency, such cases are treated by introducing extra states, e.g. `activate1` (5,6) and `activate2` (13-16).

Figure 37 Activation and Match Scope of the `descendant` transducer



4.2.2 Descendant Transducer

A descendant transducer, as shown in Figure 38, implements the logic of a descendant step from XPath. The activation scope of the transducer contains the activating message and all document messages that are descendants of it. For this transducer, the match scope contains all the nodes in the activation scope, except of the activating node itself (cf. Figure 37), as any node that is a descendant of the activating node is a candidate for matching.

Figure 38 Transitions of the descendant transducer

$\Gamma^{\mathtt{count}}_{desc}$ = $\{$`l`,$\mathtt{s}_0$,`s`$\}$ $\mathcal{Q}_{\mathtt{desc}}$ = $\{$`waiting`,`working`,`activate1`,`activate2`$\}$ $q^0_{\mathtt{desc}}$ = `waiting`

1	([f] ,	(`waiting` ,	α,	β))	$\rightarrow$	((`activate1`,	α,	$f\,\vert\,\beta$),	ϵ)
2	(<l> ,	(`activate1`,	α,	β))	$\rightarrow$	((`working` ,	$\mathtt{s}_0\,\vert\,\alpha$,	β),	<l>)
3	(<t/> ,	(`activate1`,	α,	$f\,\vert\,\beta$))	$\rightarrow$	((`waiting` ,	α,	β),	<t/>)
4	([f_1] ,	(`working` ,	α,	$f_2\,\vert\,\beta$))	$\rightarrow$	((`activate2`,	α,	$f_1 \vee f_2\,\vert\,f_2\,\vert\,\beta$),	ϵ)
5	(<l_m> ,	(`working` ,	α,	$f\,\vert\,\beta$))	$\rightarrow$	((`working` ,	`l`$\vert\,\alpha$,	$f\,\vert\,\beta$),	[f];<l_m>)
6	(<l_n> ,	(`working` ,	α,	β))	$\rightarrow$	((`working` ,	`l`$\vert\,\alpha$,	β),	<l_n>)
7	(<t_m/>,	(`working` ,	α,	$f\,\vert\,\beta$))	$\rightarrow$	((`working` ,	α,	$f\,\vert\,\beta$),	[f];<t_m/>)
8	(</l> ,	(`working` ,	`l`$\vert\,\alpha$,	β))	$\rightarrow$	((`working` ,	α,	β),	</l>)
9	(</l> ,	(`working` ,	`s`$\vert\,\alpha$,	$f\,\vert\,\beta$))	$\rightarrow$	((`working` ,	α,	β),	</l>)
10	(</l> ,	(`working` ,	$\mathtt{s}_0\,\vert\,\alpha$,	$f\,\vert\,\beta$))	$\rightarrow$	((`waiting` ,	α,	β),	</l>)
11	(<l_m> ,	(`activate2`,	α,	$f_1\,\vert\,f_2\,\vert\,\beta$))	$\rightarrow$	((`working` ,	`s`$\vert\,\alpha$,	$f_1\,\vert\,f_2\,\vert\,\beta$),	[f_2];<l_m>)
12	(<l_n> ,	(`activate2`,	α,	β))	$\rightarrow$	((`working` ,	`s`$\vert\,\alpha$,	β),	<l_m>)
13	(<t_m/>,	(`activate2`,	α,	$f_1\,\vert\,f_2\,\vert\,\beta$))	$\rightarrow$	((`working` ,	α,	$f_2\,\vert\,\beta$),	[f_2];<t_m/>)
14	(<t_n/> ,	(`activate2`,	α,	$f\,\vert\,\beta$))	$\rightarrow$	((`working` ,	α,	β),	<t_n/>)
15	($\{c,v\}$,	(*any_state* ,	α,	β))	$\rightarrow$	((*any_state* ,	α,	UPDATE(c,v,β)),	$\{c,v\}$)

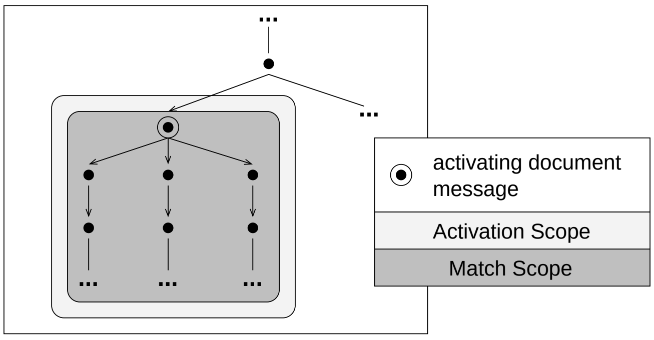
Like the child transducer, the descendant transducer can be activated while waiting for being activated (1), or while trying to match, as result of a previous activation (4). This transducer is able to match for several nested activation scopes at the same time. Keeping track of the tree level corresponding to each match scope of the transducer is done by pushing on the count stack an $\mathtt{s}_0$ symbol (initial activation scope) for the first activation (1), and an `s` symbol (activation scope) for the other activations (4). The reason for differentiating between the first and other activations is that leaving the first activation leads to changing the `working` state to the `waiting` state (10), while leaving the other activations leave the transducer in the same `working` state (9) since there remains at least one other activation scope which contained the current activation scope. However, in both of the two cases the topmost condition formula on the condition stack is removed, as the corresponding activation scope is left. Consider again the stream of Figure 3 with the query `/descendant::a/descendant::b`. For both `<a>` messages on the stream the `descendant::a` step transducer is activated, but for the first `<a>` $\mathtt{s}_0$ is pushed on the stack, for the second `<a>` `s`. Thus, when the end-element message `</a>` for the second `a` is encountered, `s` is popped from the stack, but the transducer remains in the `working` state, as it can still match for the first `a`.

An interesting property of this transducer, as presented in Section 3 and mirrored also in the descendant step logic, is that at the transducer activation time a disjunction of the received formula and the topmost formula from its condition stack is pushed on the condition stack (4). Therefore, at a time the topmost entry on the stack represents a formula equivalent to the disjunction of all the formulas stored on the stack. Note, that the disjunction can be normalized by removing multiple occurrences of the same conjuncts. In this way, a formula can contain only a single reference to a condition variable.

4.2.3 Descendant-or-self Transducer

The descendant-or-self transducer is working in a way very similar to that of the descendant transducer. The only difference is that the match scope of descendant-or-self also contains the

Figure 39 Activation and Match Scope of the `descendant-or-self` transducer



activating node (cf. Figure 39). This is reflected in the transitions in Figure 40, where the descendant-or-self transducer is also trying to match in the `activate1` state (2-5). Further, in the `activate2` state on a matching document message, in contrast to the descendant transducer where the newly received condition formula is not considered for an activation, the descendant-or-self transducer sends an activation with a condition formula that includes the formula added to the condition stack in transition 6 (13,15).

Note that in all step transducers atomic messages are handled in a different way than start messages, as an atomic message does not change the tree level. Therefore, the count stack is not changed when encountering an atomic message.

Figure 40 Transitions of the descendant-or-self transducer

$\Gamma^{\texttt{count}}_{dos}$ = $\{\texttt{l},\texttt{s}_0,\texttt{s}\}$ $\mathcal{Q}_{\texttt{dos}}$ = $\{\texttt{waiting},\texttt{working},\texttt{activate1},\texttt{activate2}\}$ $q^0_{\texttt{dos}}$ = `waiting`

1	([f] ,	(waiting ,	α,	β))	$\rightarrow$	((activate1,	α,	$f \mid \beta$),	ϵ)
2	(<l_m> ,	(activate1,	α,	$f \mid \beta$))	$\rightarrow$	((working ,	$\texttt{s}_0 \mid \alpha$,	$f \mid \beta$),	[f];<l_m>)
3	(<l_n> ,	(activate1,	α,	β))	$\rightarrow$	((working ,	$\texttt{s}_0 \mid \alpha$,	β),	<l_n>)
4	(<t_m/>,	(activate1,	α,	$f \mid \beta$))	$\rightarrow$	((waiting ,	α,	β),	[f];<t_m/>)
5	(<t_n/> ,	(activate1,	α,	$f \mid \beta$))	$\rightarrow$	((waiting ,	α,	β),	<t_n/>)
6	([f_1] ,	(working ,	α,	$f_2 \mid \beta$))	$\rightarrow$	((activate2,	α,	$f_1 \vee f_2 \mid f_2 \mid \beta$),	ϵ)
7	(<l_m> ,	(working ,	α,	$f \mid \beta$))	$\rightarrow$	((working ,	$\texttt{l} \mid \alpha$,	$f \mid \beta$),	[f];<l_m>)
8	(<l_n> ,	(working ,	α,	β))	$\rightarrow$	((working ,	$\texttt{l} \mid \alpha$,	β),	<l_n>)
9	(<t_m/>,	(working ,	α,	$f \mid \beta$))	$\rightarrow$	((working ,	α,	$f \mid \beta$),	[f];<t_m/>)
10	(</l> ,	(working ,	$\texttt{l} \mid \alpha$,	β))	$\rightarrow$	((working ,	α,	β),	</l>)
11	(</l> ,	(working ,	$\texttt{s} \mid \alpha$,	$f \mid \beta$))	$\rightarrow$	((working ,	α,	β),	</l>)
12	(</l> ,	(working ,	$\texttt{s}_0 \mid \alpha$,	$f \mid \beta$))	$\rightarrow$	((waiting ,	α,	β),	</l>)
13	(<l_m> ,	(activate2,	α,	$f \mid \beta$))	$\rightarrow$	((working ,	$\texttt{s} \mid \alpha$,	$f \mid \beta$),	[f];<l_m>)
14	(<l_n> ,	(activate2,	α,	β))	$\rightarrow$	((working ,	$\texttt{s} \mid \alpha$,	β),	<l_m>)
15	(<t_m/>,	(activate2,	α,	$f \mid \beta$))	$\rightarrow$	((working ,	α,	β),	[f];<t_m/>)
16	(<t_n/> ,	(activate2,	α,	$f \mid \beta$))	$\rightarrow$	((working ,	α,	β),	<t_n/>)
17	($\{c,v\}$,	(*any_state* ,	α,	β))	$\rightarrow$	((*any_state* ,	α,	UPDATE(c,v,β)),	$\{c,v\}$)

4.2.4 Self Transducer

The match scope of the self transducer comprises only the activating node itself. As matching of a node can be performed by looking only at the start message of the node, the task of the self transducer is much less complex than that of the other step transducers. Therefore, it does not need to keep track of the tree level after an activation, it tries to match immediately. This is reflected in the transitions of the self transducer shown in Figure 41.

When the self transducer receives an activation, it puts the corresponding condition formula on its condition stack and enters the `activate` state (1). There it tries to match the next document message and if successful, send an activation with the just recorded formula (2,4). In any case, it removes this formula from the condition stack and switches back to the `waiting` state. As the self transducer does not need to perform level counting, the count stack is not used.

Figure 41 Transitions of the self transducer

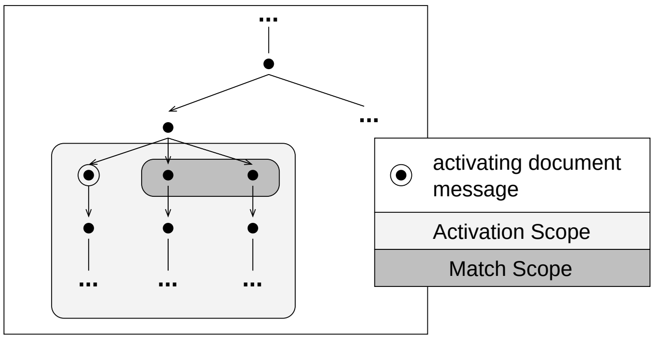
$\Gamma^{\texttt{count}}_{self} = \emptyset \quad \mathcal{Q}_{\texttt{self}} = \{\texttt{waiting},\texttt{activate}\} \quad q^0_{\texttt{self}} = \texttt{waiting}$

1	$(\texttt{[}f\texttt{]}\ ,\ (\texttt{waiting}\ ,\ \alpha,\ \beta))$	$\rightarrow$	$((\texttt{activate},\ \alpha,\ f\,\vert\,\beta),\ \epsilon)$
2	$(\texttt{<}l_m\texttt{>}\ ,\ (\texttt{activate},\ \alpha,\ f\,\vert\,\beta))$	$\rightarrow$	$((\texttt{waiting}\ ,\ \alpha,\ \beta),\ \texttt{[}f\texttt{];<}l_m\texttt{>})$
3	$(\texttt{<}l_n\texttt{>}\ ,\ (\texttt{activate},\ \alpha,\ f\,\vert\,\beta))$	$\rightarrow$	$((\texttt{waiting}\ ,\ \alpha,\ \beta),\ \texttt{<}l_n\texttt{>})$
4	$(\texttt{<}t_m\texttt{/>},\ (\texttt{activate},\ \alpha,\ f\,\vert\,\beta))$	$\rightarrow$	$((\texttt{waiting}\ ,\ \alpha,\ \beta),\ \texttt{[}f\texttt{];<}t_m\texttt{/>})$
5	$(\texttt{<}t_n\texttt{/>}\ ,\ (\texttt{activate},\ \alpha,\ f\,\vert\,\beta))$	$\rightarrow$	$((\texttt{waiting}\ ,\ \alpha,\ \beta),\ \texttt{<}t_n\texttt{/>})$
6	$(\{c,v\}\ ,\ (any\_state,\ \alpha,\ \beta))$	$\rightarrow$	$((any\_state,\ \alpha,\ \textsc{update}(c,v,\beta)),\ \{c,v\})$

4.2.5 Following-sibling Transducer

The match scope of the following-sibling transducer (as shown in Figure 42) contains all the direct siblings of the activating node that follow it in document order, i.e. all the nodes following the activating node on the same tree level, having the same parent.

Figure 42 Activation and Match Scope of the `following-sibling` transducer



The following-sibling transducer is the most complex of all the step transducers for several reasons. First, it reaches its match scope only after the activating node is closed, i.e. after the end message corresponding to the activating message is encountered. In the meantime, the following-sibling transducer must keep track of condition formulas by putting them on its count stack[†]. Only after the end of the activating node is reached by encountering the deactivating message, the following-sibling transducer is allowed to enter the `working` state and put the corresponding formula on its condition stack (5,6).

Second, similar to the child transducer, the following-sibling transducer matches only on a specific tree level. Therefore, when encountering a start message in the `working` state, it puts an `m` on the stack to signal that it just left its match scope and switches to the `waiting` state (10,11). Also, when encountering an end message in the `working` state, the following-sibling transducer knows that it leaves an activation scope, therefore removing the formula from the top of the condition stack (13-16). At this time, when there is an `l` on the count stack, the following-sibling transducer also leaves the `working` state (13). However, if there is any other symbol on the count stack it stays in the `working` state, as it leaves a match scope only to directly enter a new one (14-16).

[†]Condition formulas on the condition stack are used for activations to be send when a transducer matches.

Figure 43 Transitions of the following-sibling transducer

$\Gamma_{fs}^{\texttt{count}}$ = {l,m,a(f),n(f)} $\mathcal{Q}_{\texttt{fs}}$ = {waiting,working,activate1,activate2} $q_{\texttt{fs}}^0$ = waiting

1	([f] ,	(waiting ,	α,	β)) →	((activate1,	n(f)$\vert\alpha$,	β),	ϵ	)
2	(<l> ,	(waiting ,	α,	β)) →	((waiting ,	l$\vert\alpha$,	β),	<l>	)
3	(</l> ,	(waiting ,	l$\vert\alpha$,	β)) →	((waiting ,	α,	β),	</l>	)
4	(</l> ,	(waiting ,	m$\vert\alpha$,	β)) →	((working ,	α,	β),	</l>	)
5	(</l> ,	(waiting ,	n(f)$\vert\alpha$,	β)) →	((working ,	α,	$f\vert\beta$),	</l>	)
6	(</l> ,	(waiting ,	a(f_1)$\vert\alpha$,	$f_2\vert\beta$)) →	((working ,	α,	$f_1 \vee f_2\vert\beta$),	</l>	)
7	(<l> ,	(activate1,	α,	β)) →	((waiting ,	α,	β),	<l>	)
8	(<t/> ,	(activate1,	n(f)$\vert\alpha$,	β)) →	((working ,	α,	$f\vert\beta$),	<t/>	)
9	([f] ,	(working ,	α,	β)) →	((activate2,	a(f)$\vert\alpha$,	β),	ϵ	)
10	(<l_m> ,	(working ,	α,	$f\vert\beta$)) →	((waiting ,	m$\vert\alpha$,	$f\vert\beta$),	[f];<l_m>	)
11	(<l_n> ,	(working ,	α,	β)) →	((waiting ,	m$\vert\alpha$,	β),	<l_n>	)
12	(<t_m/>,	(working ,	α,	$f\vert\beta$)) →	((working ,	α,	β),	[f];<t_m/>	)
13	(</l> ,	(working ,	l$\vert\alpha$,	$f\vert\beta$)) →	((waiting ,	α,	β),	</l>	)
14	(</l> ,	(working ,	m$\vert\alpha$,	$f\vert\beta$)) →	((working ,	α,	β),	</l>	)
15	(</l> ,	(working ,	n(f_1)$\vert\alpha$,	$f_2\vert\beta$)) →	((working ,	α,	$f_1\vert\beta$),	</l>	)
16	(</l> ,	(working ,	a(f_1)$\vert\alpha$,	$f_2\vert f_3\vert\beta$)) →	((working ,	α,	$f_1 \vee f_3\vert\beta$),	</l>	)
17	(<l_m> ,	(activate2,	α,	$f\vert\beta$)) →	((waiting ,	α,	$f\vert\beta$),	[f];<l_m>	)
18	(<l_n> ,	(activate2,	α,	β)) →	((waiting ,	α,	β),	<l_n>	)
19	(<t_m/>,	(activate2,	a(f_1)$\vert\alpha$,	$f_2\vert\beta$)) →	((working ,	α,	$f_1 \vee f_2\vert\beta$),	[f_2];<t_m/>	)
20	(<t_n/> ,	(activate2,	a(f_1)$\vert\alpha$,	$f_2\vert\beta$)) →	((working ,	α,	$f_1 \vee f_2\vert\beta$),	[f_2];<t_n/>	)
21	({c,v} ,	(*any_state* ,	α,	β)) →	((*any_state* ,	α,	UPDATE(c,v,β)),	{c,v}	)

Third, as already mentioned in Section 3.3.4, the relationships between two match scopes in the following-sibling transducer can be of several different kinds. In the descendant transducer, there are two different kinds of relationships:

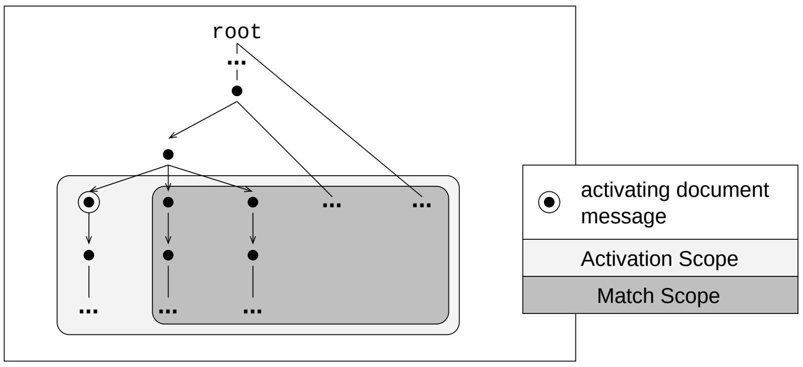
- One match scope is completely contained in the other. This situation is coped with by the descendant transducer by putting a disjunction of the condition formula of the outer match scope and the formula of the inner match scope on the condition stack.
- Two match scopes are disjunct. In the descendant transducer this also means that the corresponding activation scopes are disjunct. However, if two activations do not overlap, the corresponding condition formulas cannot be on the condition stack at the same time. Hence, no special treatment is necessary in this case.

The child transducer adds a third kind of relationship between match scopes:

- For two nested activation scopes, the corresponding match scopes are disjunct. This is treated in the child transducer by putting the condition formula corresponding to the inner activation scope on top of the formula corresponding to the outer scope.

In the following-sibling transducer all three kinds of relationships between match scopes can occur: when activated in the `waiting` state, it puts the corresponding condition formula on the condition stack marked by an `n` for a new match scope (1). When activated in the `working` state, the formula put on the stack is marked with an `a` to denote an additional formula for a nested match scope (9). If the following-sibling transducer encounters an end message with a condition formula marked with an `n` on its count stack (5,15), it pushes the formula on the condition stack as a new entry. If the encountered formula is marked by an `a`, the formula is added to the topmost formula on the condition stack by creating a disjunction (6,16,19,20).

Figure 44 Activation and Match Scope of the `following` transducer



4.2.6 Following Transducer

The following transducer shares the behavior of the following-sibling transducer only in respect to the time of entering the match scope. However, the following transducer is different in comparison to all the other transducer in that once it is in an activation scope (or match scope), it never leaves this scope until the end of the document (cf. Figure 44). This also implies that a formula put on the condition stack of the following transducer is not removed until the end of the document. Therefore, the space (and thus also time) complexity of the following transducer is worse than that of the other transducers (cf. Section 5). Moreover, the determination of condition variables of qualifiers that include steps with a `following` axis differs from that of the other axes (cf. Section 4.3.1).

The transition relation of the following transducer shown in Figure 45, is similar to that of the following-sibling transducer although less complex. This is due to the fact that there is only one kind of relationship between match scopes: they are always nested. When activated with a condition formula, the following transducer temporarily stores that formula on its count stack (1,7). If this formula is encountered on an end message in the `waiting` state, the transducer moves the formula from the count to the condition stack and enters the `working` state (4). If it is encountered while already in the `working` state, a disjunction of the old formula already on the condition stack before and the new formula from the count stack is put on the condition stack (12).

4.3 Qualifier Transducers

Recall that an XPath qualifier instance can be seen as a condition variable, on which candidates for the result depend. In a SPEX network for each qualifier such variables are created and – depending on the result of the qualifier – evaluated to a truth value. Therefore, in addition to step transducers representing location paths inside qualifiers, an XPath qualifier adds specific transducers for handling the determination of condition variables, i.e. for setting their truth value.

4.3.1 Variable-Creator Transducer: $\mathsf{VC}^{in}_{out}(q)$

After a step with a qualifier a *variable-creator transducer* is inserted into the SPEX network. Its transitions are shown in Figure 46. A variable-creator transducer is responsible for creating a condition variable c for each instance of the qualifier q. The creation is triggered by a received activation message $[f]$. At that time it outputs a new activation message, consisting of a

Figure 45 Transitions of the following transducer

$\Gamma^{\texttt{count}}_{foll}$ = {l,m,a(f),n(f)} $\mathcal{Q}_{\texttt{foll}}$ = {waiting,working,activate1,activate2} $q^0_{\texttt{foll}}$ = waiting

1	([f]	(waiting	α,	β))	$\rightarrow$	((activate1,	a(f)|α,	β),	ϵ)
2	(<l>	(waiting	α,	β))	$\rightarrow$	((waiting	l|α,	β),	<l>)
3	(</l>	(waiting	l|α,	β))	$\rightarrow$	((waiting	α,	β),	</l>)
4	(</l>	(waiting	a(f)|α,	β))	$\rightarrow$	((working	α,	f|β),	</l>)
5	(<l>	(activate1,	α,	β))	$\rightarrow$	((waiting	α,	β),	<l>)
6	(<t/>	(activate1,	a(f)|α,	β))	$\rightarrow$	((working	α,	f|β),	<t/>)
7	([f]	(working	α,	β))	$\rightarrow$	((activate2,	a(f)|α,	β),	ϵ)
8	(<l_m>	(working	α,	f|β))	$\rightarrow$	((working	l|α,	f|β),	[f];<l_m>)
9	(<l_n>	(working	α,	β))	$\rightarrow$	((working	l|α,	β),	<l_n>)
10	(<t_m/>,	(working	α,	f|β))	$\rightarrow$	((working	α,	f|β),	[f];<t_m/>)
11	(</l>	(working	l|α,	β))	$\rightarrow$	((working	α,	β),	</l>)
12	(</l>	(working	a(f_1)|α,	f_2|β))	$\rightarrow$	((working	α,	$f_1 \vee f_2$|β),	</l>)
13	(<l_m>	(activate2,	α,	f|β))	$\rightarrow$	((working	α,	f|β),	[f];<l_m>)
14	(<l_n>	(activate2,	α,	β))	$\rightarrow$	((working	α,	β),	<l_n>)
15	(<t_m/>,	(activate2,	a(f_1)|α,	f_2|β))	$\rightarrow$	((working	α,	$f_1 \vee f_2$|β),	[f_2];<t_m/>)
16	(<t_n/> ,	(activate2,	a(f_1)|α,	f_2|β))	$\rightarrow$	((working	α,	$f_1 \vee f_2$|β),	[f_2];<t_n/>)
17	({c,v} ,	(*any_state*	α,	β))	$\rightarrow$	((*any_state*	α,	UPDATE(c,v,β)),	{c,v})

conjunction between the received boolean formula and the newly created variable (1).

Figure 46 Transitions of the variable-creator transducer

$\Gamma^{\texttt{count}}_{vc}$ = {l,s} $\mathcal{Q}_{\texttt{vc}}$ = {working,activate} $q^0_{\texttt{vc}}$ = working

1	([f]	(working ,	α,	β))	$\rightarrow$	((activate,	α,	c|β),	[$f \wedge c$])
2	(<l>	(working ,	α,	β))	$\rightarrow$	((working ,	l|α,	β),	<l>)
3	(</l>	(working ,	s|α,	c|β))	$\rightarrow$	((working ,	α,	β),	{c,false};</l>)
4	(</l>	(working ,	l|α,	β))	$\rightarrow$	((working ,	α,	β),	</l>)
5	(<l>	(activate,	α,	β))	$\rightarrow$	((working ,	s|α,	β),	<l>)
6	(<t/>	(activate,	α,	c|β))	$\rightarrow$	((working ,	α,	β),	{c,false};<t/>)
7	({c,v},	(*any_state*,	α,	β))	$\rightarrow$	((*any_state*,	α,	β),	{c,v})

Further, a variable creator also determines the false value of a condition, i.e. if a condition is false, this is notified by the variable creator. It is known that a condition is false, if the last step inside the corresponding qualifier cannot match for that condition any more. Depending on the combination of axes on location steps inside a qualifier there can be three different cases:

- All the transducers of the steps inside the qualifier are of any type other than `following`, except the first one that is also not of type `following-sibling`:

  In this case it is known that all transducers on the qualifier have left the activation scope of the condition if the first one left this scope. As can be seen in Figures 35, 37, and 39, the activation scopes of all transducers that can appear directly after the variable creator are identical, they are bound to the activating node. Therefore, if the activating node is left by encountering the end message corresponding to the activating message, the variable creator has determined a false value of the corresponding condition.

  This is done by putting a special symbol `s` (activation scope) on the count stack on an activation and the just created condition on the condition stack (1,5). Upon encountering an end message with this symbol on the stack the variable creator removes the topmost condition on the condition stack and sends a condition determination message with a `false` value for that condition (3).

- All the transducers of the steps inside the qualifier are of any type other than `following` and the first one is of type `following-sibling`:

  Like in the previous case, here it is known that all transducers in the qualifier have left the activation scope corresponding to a condition if the following-sibling transducer that is the first transducer in the qualifier has left this scope. As shown in Figure 42, the activation scope of a following-sibling transducer comprises the activating node, as well as all of its following siblings. Therefore, it is known that a condition has a false value if the last of the siblings is closed, which can only be recorded by encountering the end message of the parent node.

  For the support of this case transition 5 from Figure 46 has to be changed to put an `s` on the count stack not for the current tree level n, but rather for the tree level of the parent node, which is level $n-1$. This can be achieved by putting an `l` on top of the `s`. The new transition 5' for this case is shown in Figure 47.

- There is a following transducer among the transducers for the steps inside the qualifier:

  This is the most complex case, as there are two possibilities that have to be taken care of. Let t be the first following transducer inside the qualifier. If t is not activated with a condition c before the first step leaves the activation scope c corresponds to, all of the transducers inside the qualifier are outside of an activation scope of c. Therefore, c is known to be false and a condition determination message is send for it. However, if t is activated with c, the condition cannot be determined to false until the end of the document, as a following transducer never leaves an activation scope before.

  To be able to cope with this case, a new transducer and a new message type have to be introduced. For reasons of conciseness this is not discussed further here.

Figure 47 Transition 5' of the variable-creator transducer

$$5'(\texttt{<}l\texttt{>},\ (\texttt{activate},\ \texttt{l|}\alpha,\ \beta)) \rightarrow ((\texttt{working},\ \texttt{l|s|}\alpha,\ \beta),\ \texttt{<}l\texttt{>})$$

Note, that at different times multiple condition determination messages for the same condition can be send, even with different values. However, this does not lead to incorrect behavior due to the way conditions are handled in transducers. This is discussed in more detail in Section 4.3.3.

4.3.2 Variable-Filter Transducer: $\mathsf{VF}^{in}_{out}(q)$

A *variable-filter transducer* is defined for a qualifier q and is sensitive to condition variables created for that qualifier. It processes incoming activation messages to select only the condition variables for qualifier q. For every selected condition variable an activation message with an atomic formula that contains only this variable is send. As there are transducer which need to know whether two following activation message were created by the same variable filter or by different ones, a variable filter sends a delimiter message after receiving an activation, before processing the corresponding formula, and one after the processing.

The specification of the variable filter transducer in Figure 48 comprises only one transition. A function $\textsc{filter}(q,f)$ is used that returns a sequence of activation messages for every condition variable for qualifier q in the condition formula f.

Figure 48 Transition of the variable-filter transducer

$$\Gamma^{\texttt{count}}_{vf(q)} = \emptyset \quad \mathcal{Q}_{\texttt{vf(q)}} = \{\texttt{working}\} \quad q^0_{\texttt{vf(q)}} = \texttt{working}$$

$$1\ ([f],\ (\texttt{working},\ \alpha,\ \beta)) \rightarrow ((\texttt{working},\ \alpha,\ \beta),\ \texttt{\#;FILTER}(q,f)\texttt{;\#})$$

4.3.3 Variable-Determinant Transducers: PD^{in}_{out}, ID^{in}_{out}

A *variable-determinant transducer* for a given qualifier q provides determination messages, which assign a `true` value to condition variables, representing instances of the qualifier q, if the instances are satisfied. There are several kinds of variable determinants, corresponding to the XPath expressivity to specify qualifiers, like path qualifiers, identity-based joins, value-based joins, etc. Here, only the first two cases are explained, but in the provided architecture other variable determinants can be added.

Path-determinant transducer PD^{in}_{out} : A path-determinant transducer is a variable determinant for a qualifier q containing a single path. As described in Figure 49, it gets activation messages from a variable filter $\mathsf{VF}^{in}_{out}(q)$, which filters out all variables from a boolean formula that do not belong to the qualifier q. Hence, such a path determinant gets on its input tape only activation messages with condition variables corresponding to an instance c of the qualifier q from a condition formula and sends determination messages for these ones (1). Additionally, a path determinant processes incoming condition determination messages by not passing them on to its output tape (2). This removes the duplicates of condition determination messages created by the the split transducer at the beginning of the qualifier.

Figure 49 Transitions of the path-determinant transducer

$$\Gamma^{\texttt{count}}_{pt} = \emptyset \quad \mathcal{Q}_{\texttt{pt}} = \{\texttt{working}\} \quad q^0_{\texttt{pt}} = \texttt{working}$$

$$\begin{array}{ll} 1 & ([c]\ ,\ (\texttt{working}\ ,\ \alpha,\ \beta)) \rightarrow ((\texttt{working}\ ,\ \alpha,\ \beta),\ \{c,\texttt{true}\}) \\ 2 & (\{c,v\},\ (\mathit{any\_state},\ \alpha,\ \beta)) \rightarrow ((\mathit{any\_state},\ \alpha,\ \beta),\ \epsilon\ ) \end{array}$$

Identity-determinant transducer IT^{in}_{out} : The identity-determinant transducer handles qualifiers q consisting of identity-based joins between paths, e.g. `/child::a[descendant::b == child::c/descendant::b]`. In the previous example, an `a` is selected if at least a descendant `b` of that `a` is among the `b`'s selected by the path `child::c/descendant::b`.

Checking the identity of two nodes selected by two different paths can easily be performed in a SPEX query system. Remember, that in a SPEX network the selection of a node by a path is given by the fact that it sends an activation message to the component following that path on the start message of the selected node. It can easily be detected if two nodes selected by two different paths $p1$ and $p2$ are identical simply by checking whether both $p1$ and $p2$ send an activation for the same document message. As there is only one document message in a SPEX network at the same time and activation messages are always send before the corresponding document message, the two paths of an identity determinant select the same node if the determinant receives two activation messages followed by a document message.

An identity determinant is directly preceded in the network by a join transducer that joins the paths of both arguments to the identity-based join. At the end of each of these paths there is a variable filter (except when one of them is an absolute path, see below). Therefore, instead of receiving a single activation, activation messages are grouped by the two variable filters. When a delimiter message '#' is encountered on the input tape while the transducer is in the `waiting` state (transition 1 in Figure 50), it enters the state `collect`, in order to collect the conditions of all activation messages encountered before the next delimiter message. These conditions – send by the same variable filter – are collected by putting them on the condition stack (3). After a subsequent delimiter message is encountered (4), the transducer looks out for a second block of activations signaled by another delimiter message. If there is none and a document message is encountered, the identity join failed for the last activation and the transducer switches back to the `waiting` state, removing the collected conditions from its condition stack (9). However, if another delimiter message is encountered, the identity determinant enters the `compare` state (7). There, if it gets an activation message with a condition c that is contained in its condition stack, it knows that the condition is fulfilled from both sides of the identity join, hence sending out a condition determination message with a value of `true` for c (14). If the end of the second group of conditions is encountered by reading a '#', the identity determinant switches back to the `waiting` state and clears its condition stack (15).

Figure 50 Transitions of the identity-determinant transducer

$\Gamma_{it}^{\text{count}} = \emptyset$ $\mathcal{Q}_{\text{it}}$ = {waiting,collect,lookout1,lookout2,activate1,activate2,compare} q^0_{it} = waiting

1	(# ,	(waiting ,	α,	β)) →	((collect ,	α,	β),	ϵ)
2	([true],	(waiting ,	α,	β)) →	((lookout1 ,	α,	β),	ϵ)
3	([c] ,	(collect ,	α,	β)) →	((collect ,	α,	c|β),	ϵ)
4	(# ,	(collect ,	α,	β)) →	((lookout2 ,	α,	β),	ϵ)
5	(# ,	(lookout1 ,	α,	β)) →	((activate1,	α,	β),	ϵ)
6	(<l> ,	(lookout1 ,	α,	β)) →	((waiting ,	α,	β),	ϵ)
7	(# ,	(lookout2 ,	α,	β)) →	((compare ,	α,	β),	ϵ)
8	([true],	(lookout2 ,	α,	β)) →	((activate2,	α,	β),	ϵ)
9	(<l> ,	(lookout2 ,	α,	β)) →	((waiting ,	α,	$\perp$),	ϵ)
10	([c] ,	(activate1,	α,	β)) →	((activate1,	α,	β),	{c,true})
11	(# ,	(activate1,	α,	β)) →	((waiting ,	α,	β),	ϵ)
12	(ϵ ,	(activate2,	α,	c|β)) →	((activate2,	α,	β),	{c,true})
13	(ϵ ,	(activate2,	α,	$\perp$)) →	((waiting ,	α,	β),	ϵ)
14	([c] ,	(compare ,	α,	β_1|c|β_2)) →	((waiting ,	α,	β),	{c,true})
15	(# ,	(compare ,	α,	β)) →	((waiting ,	α,	$\perp$),	ϵ)
16	({c,v} ,	(*any_state* ,	α,	β)) →	((*any_state* ,	α,	β),	ϵ)

The transitions explained above ensure correct behavior if both sides of the join are relative location paths. If one of the paths is absolute, additional transitions are necessary. Transition 2 shows the case where the first activation is received from the absolute path. The transducer enters the `lookout1` state to see if a block of activations comes from the second path. If the next message is a document message, there will be no further activations for this document message and the identity determinant switches back to the `waiting` state (6). If there is a delimiter message (5), the transducer goes to the `activate1` state, where it will send a condition determination message for every incoming activation (10) and reenter the `waiting` state when encountering the next delimiter message (11).

The case where the block of activations from the relative path is received before the activation of the absolute path is treated by the transitions 8, 12, and 13.

4.4 Split and Join Transducers

Split and join transducers allow for more complex SPEX networks than simple transducer chains, by providing support for parallel stream processing and synchronization primitives, encoded at the level of transducer transitions.

Split Transducer $\mathbf{SP}^{in}_{out_1,out_2}$: The split transducer has two output tapes instead of one. Its single task is to forward every message received on the input tape to both of the output tapes.

Join Transducer $\mathbf{JO}^{in_1,in_2}_{out}$: The join transducer is responsible for merging the messages on its two input tapes to be provided to just one output tape. There are two important issues about this task. First, the join transducer has to eliminate duplicate messages that are created by the split transducer. Second, in combination with the input transducer it ensures system synchronization. Activation messages cannot be duplicated, as they are always consumed on at least one branch after a split. The duplication of condition determination messages is prevented by having variable determinants remove determination messages (cf. Section 4.3.3). Therefore, both of the requirements are fulfilled by a special treatment of document messages. Other than activation and determination messages that are unconditionally passed through by the join transducer, the same document message has to be received on both input tapes before it is passed on. Of course, this immediately solves the message duplication problem. Together with the fact that SPEX transducers never send an activation or determination message after the corresponding document message, also the synchronization of two branches that are to be joined is ensured.

In the specification presented in Figure 51, the right-hand side of the transition relation is presented shortened as a 2-tuple consisting of the new state and the output tape. The left-hand side of transitions is also changed to show the current state and both input tapes. For reasons of conciseness the specification does not show the count and condition stacks, as they are not used by the join transducer.

Figure 51 Transitions of the join transducer

$\Gamma^{\text{count}}_{jt}$ = $\emptyset$ $\mathcal{Q}_{\text{jt}}$ = {none, left, right} q^0_{jt} = none

1	(<d>	, <d>	, none)	→	(none , <d>	)
2	(<d>	, [f]	, none)	→	(left , [f]	)
3	(<d>	, {c,v}	, none)	→	(left , {c,v}	)
4	([f]	, <d>	, none)	→	(right, [f]	)
5	({c,v}	, <d>	, none)	→	(right, {c,v}	)
6	([f]	, {c,v}	, none)	→	(none , [f];{c,v}	)
7	({c,v}	, [f]	, none)	→	(none , [f];{c,v}	)
8	([f_1]	, [f_2]	, none)	→	(none , [f_1];[f_2]	)
9	({c_1,v_1}	, {c_2,v_2}	, none)	→	(none , {c_1,v_1};{c_2,v_2}	)
10	(ϵ	, [f]	, left)	→	(left , [f]	)
11	(ϵ	, {c,v}	, left)	→	(left , {c,v}	)
12	(ϵ	, <d>	, left)	→	(none , <d>	)
13	([f]	, ϵ	, right)	→	(right, [f]	)
14	({c,v}	, ϵ	, right)	→	(right, {c,v}	)
15	(<d>	, ϵ	, right)	→	(none , <d>	)

The specification of the join transducer heavily relies on the fact that the same document message must appear on both input tapes before some other document message is received. In Figure 51 in the none state there are transitions for every possibly combination of messages on both input tapes. If a document message is present on both input tapes the transducer copies it to its output tape (1). If there is any other message but a document message on both input tapes (6-9), the transducer still expects a document message, as `none` has been received yet. Therefore, it just passes on the received messages and doesn't change state. If a document message was found on the left tape but not on the right one, the transducer outputs the message from the right tape and enters the `left` state (2,3). Likewise, if a document message was encountered on the right tape, but not on the left one, it outputs the message on the left tape and enters the `right` state (4,5). In the left state the left input tape of the transducer should always be empty. Every incoming message on the right tape is then forwarded to the output tape (10-12). If the received message is a document message, the transducer also switches back to the `none` state (12) where it is ready to process the block of messages corresponding to the next document message. The transitions in the `right` state are analogous to that of the `left` state.

Note, that split and join transducers with n output (input, resp.) tapes can be simulated by $n-1$ binary transducers.

4.5 Set Transducers

Set transducers represent set operations on node sets. A node set is a set of trees (a forest), each element of which is a subtree of the document tree conveyed by the input. Every element (tree of the forest) is represented in a SPEX network as the sequence of document messages starting with the document message directly following a specific activation message and ending with the corresponding end message. A set operation on two node sets that are returned by two location paths p_1 and p_2 is encoded in a SPEX network by merging the output of p_1 and p_2 with a join transducer and connecting the join transducer with the specific set transducer. The task of the set transducer is to decide whether a node signaled by an activation message is to be included in the resulting node set. This is done just by looking at the activation messages for a document message. The conditions activations have to fulfill for a node to be part of the resulting node set depends on the specific set operation. For a SPEX network, union and insersection transducers are defined.

Figure 52 Transitions of the union transducer

$$\Gamma^{\texttt{count}}_{union} = \emptyset \quad \mathcal{Q}_{\texttt{union}} = \{\texttt{waiting}, \texttt{activate}\} \quad q^0_{\texttt{union}} = \texttt{waiting}$$

$$\begin{array}{rl}
1 & ([f],\ (\texttt{waiting},\ \alpha,\ \beta)) \to ((\texttt{activate},\ \alpha,\ f\,|\,\beta),\ \epsilon) \\
2 & ([f_2],\ (\texttt{activate},\ \alpha,\ f_1\,|\,\beta)) \to ((\texttt{waiting},\ \alpha,\ \beta),\ [f_1 \vee f_2]) \\
3 & (\texttt{<}l\texttt{>},\ (\texttt{activate},\ \alpha,\ f\,|\,\beta)) \to ((\texttt{waiting},\ \alpha,\ \beta),\ [f];\texttt{<}l\texttt{>}) \\
4 & (\texttt{<}t\texttt{/>},\ (\texttt{activate},\ \alpha,\ f\,|\,\beta)) \to ((\texttt{waiting},\ \alpha,\ \beta),\ [f];\texttt{<}t\texttt{/>}) \\
5 & (\{c,v\},\ (any\_state,\ \alpha,\ \beta)) \to ((any\_state,\ \alpha,\ \beta),\ \{c,v\})
\end{array}$$

Union Transducer $\mathsf{SE}^{in}_{out}(union)$: The union transducer supports the union operation of XPath. Here, a node is part of the resulting set of candidates if at least one activation is received for it. A node is part of the result if any of the formulas of corresponding activation messages are fulfilled. Therefore, if there are two activations with condition formulas f_1 and f_2 for a single document message, the resulting activation message includes a condition formula that is a disjunction of f_1 and f_2. In Figure 52 an activation in the `waiting` state causes the union transducer to switch to the `activate` state, storing the condition formula of the activation message on the condition stack (1). There are two possibilities in the `activate` state: the next message can be a document message which causes the transducer to send an activation message with the condition formula on top of the condition stack (3,4), or it can be another activation message causing the transducer to send an activation with a formula that is a disjunction of the formula on the condition stack and the formula of the second activation (2).

Figure 53 Transitions of the intersection transducer

$$\Gamma^{\texttt{count}}_{intersect} = \emptyset \quad \mathcal{Q}_{\texttt{intersect}} = \{\texttt{waiting},\ \texttt{activate}\} \quad q^0_{\texttt{intersect}} = \texttt{waiting}$$

$$\begin{array}{llll}
1 & ([f]\ ,\ (\texttt{waiting}\ ,\ \alpha,\ \beta)) & \rightarrow & ((\texttt{activate},\ \alpha,\ f\,|\,\beta),\ \epsilon) \\
2 & ([f_2]\ ,\ (\texttt{activate},\ \alpha,\ f_1\,|\,\beta)) & \rightarrow & ((\texttt{waiting}\ ,\ \alpha,\ \beta),\ [f_1 \wedge f_2]) \\
3 & (\texttt{<}l\texttt{>}\ ,\ (\texttt{activate},\ \alpha,\ f\,|\,\beta)) & \rightarrow & ((\texttt{waiting}\ ,\ \alpha,\ \beta),\ \texttt{<}l\texttt{>}) \\
4 & (\texttt{<}t\texttt{/>}\ ,\ (\texttt{activate},\ \alpha,\ f\,|\,\beta)) & \rightarrow & ((\texttt{waiting}\ ,\ \alpha,\ \beta),\ \texttt{<}t\texttt{/>}) \\
5 & (\{c,v\},\ (any\_state,\ \alpha,\ \beta)) & \rightarrow & ((any\_state,\ \alpha,\ \beta),\ \{c,v\})
\end{array}$$

Intersection transducer $\mathsf{SE}^{in}_{out}(intersect)$: The intersection transducer (cf. Figure 53) differs from the union transducer in that it only outputs an activation message if two activations with formulas f_1 and f_2 have been received for the same document message. Further, the created activation message can only lead to a result if both of the formulas f_1 and f_2 evaluate to true. Therefore, if there is no activation in the `activate` state, the intersection transducer does not send an activation message (3,4), and if there is another activation in `activate` the transducer sends an activation message that is a conjunction of the formula on its condition stack and the formula included with the second activation.

Note that the `exception` set operation can not be trivially added to a SPEX network, since there is no assumed order between the activation messages arriving from separate branches at a join transducer and hence implicitly at a set transducer.

4.6 Input and Output Transducers

Input Transducer IN^{in}_{out}: The input transducer is the first component in a SPEX network, which has the task of forwarding one document message at a time, and writing an activation message to the output tape on the start-document message. After a document message reaches the end of the network, the next document message is forwarded, thus ensuring the presence of only one document message in the network at a time.

Output Transducer OU^{in}_{out}: The output transducer is the last component in a network, which manages the candidates for the result. Its tasks are the identification and storage of candidates, the evaluation of formulas in order to decide the status of the related candidates, and the output of results in document order. The output transducer is more complex than the other transducers introduced here and is not totally restricted to the constraints a SPEX transducer has, like the stack access for the candidate store.

An output transducer uses the following data structures for its operation:

- The count stack is used in the same manner as in the other transducers, i.e. for level counting purposes. However, for some special combination of candidate determination it is necessary to access the bottom of the stack. Therefore, the count stack is not always strictly used as a stack. For reasons of conciseness, this data structure will still be called count stack.
- The message store is a queue (i.e. a data structure where entries are appended at the end of the list and can only be removed from the head) that is used for storing document

messages that are part of some result. Entries in the message store are pairs that consist of a document message m and a condition formula f.

Condition formulas can be absent from a message store entry. For this case the symbol ϵ is used to denote an empty condition formula. A message store entry is the start of a candidate if the condition formula of this entry is not empty. The end of a candidate is not specially marked in the store, as it can be found by counting levels in a similar way as most of the transducers do.

Instead of keeping track of candidates by introducing a table where candidates are stored with the document messages they consist of, only a single message store is introduced here. This has the advantage that first, every document message has to be stored at most once, and second, document order of results can easily be ensured, as candidate starting points are stored in the buffer in document order.

In the specification of the output transducer two functions are used:

- The UPDATE(c,v,β) function iterates over all the entries in the message store β to set all occurences of the condition variable c to the value v. This function is equivalent to the UPDATE function used by step transducers.
- Informally, the CHK(s,α,β) function checks the message store β for determined candidates and outputs (or removes) candidates as long as the candidate that is next in document order is determined. Depending on the specific operations performed, different configurations are returned. The formal specification of this functions will be given later.

In the specification of the output transducer shown in Figure 54 there are five different basic states used for different activities.

State `waiting`: The output transducer is in the `waiting` state (cf. Figure 54(a)) if there is no partial candidate in the system. This state can be left when encountering an activation or condition determination message depending on the specificities of these messages and the content of the message store.

State `storing1`: The output transducer is in the `storing1` state (cf. Figure 54(b)) if there is exactly one undetermined partial candidate in the system. In this state the transducer has to put document messages into the message store. It can be left if the candidate is completed, if it is determined, or if a nested candidate is encountered.

State `storing2`: The output transducer is in the `storing2` state (cf. Figure 54(c)) if there is more than one partial candidate in the system (i.e. any number of nested candidates) and the first of these is undetermined. Basically, this state is equivalent to the `storing1` state. However, because of the CHECK function which introduces *sporadic* jumps between states, the distinction of the `storing1` and `storing2` states is necessary to keep a consistent count stack.

State `output1`: The output transducer is in the `output1` state (cf. Figure 54(d)) if the candidate that is first in document order is part of the result, but not complete yet. In this case the transducer must output all document messages of the candidate until it is completed. Further, in the `output1` state there are no more partial candidates (nested in the one that is being output) in the system. Therefore, the output transducer does not store document messages. The `output1` state can only be left if the candidate is completed or another candidate is encountered.

Figure 54 Transitions of the output transducer

```
 1 ([true], (waiting   , ⊥,          β)) → ((activate2, ⊥,               β), ϵ     )
 2 ([f]   , (waiting   , ⊥,          β)) → ((activate1, ⊥,        β|(ϵ,f)), ϵ     )
 3 ({c,v} , (waiting   , α,          β)) → CHK((waiting,α,UPDATE(c,v,β)),⊥)
 4 (<l>   , (activate1, α, β|(ϵ,f))) → ((storing1 , α,     β|(<l>,f)), ϵ     )
 5 ({c,v} , (activate1, α,          β)) → ((activate7, α, UPDATE(c,v,β)), ϵ     )
 6 (<l>   , (activate2, α,          β)) → ((output1  , α,               β), <l>   )
 7 ({c,v} , (activate2, α,          β)) → ((activate2, α, UPDATE(c,v,β)), ϵ     )
 8 (<l>   , (activate7, α, β|(ϵ,f))) → CHK((storing1,α,UPDATE(c,v,β|(<l>,f))),⊥)
```

(a) waiting state transitions

```
 9 (<l>   , (storing1  ,   α,          β)) → ((storing1 , l|α,     β|(<l>,ϵ)), ϵ     )
10 (</l>  , (storing1  , l|α,          β)) → ((storing1 ,   α,    β|(</l>,ϵ)), ϵ     )
11 (</l>  , (storing1  ,   ⊥,          β)) → ((waiting  ,   ⊥,    β|(</l>,ϵ)), ϵ     )
12 ([f]   , (storing1  ,   α,          β)) → ((activate3,   α,       β|(ϵ,f)), ϵ     )
13 ({c,v}, (storing1  ,   α,          β)) → CHK((storing1,α,UPDATE(c,v,β)),⊥)
14 (<l>   , (activate3,   α, β|(ϵ,f))) → ((storing2 , m|α,     β|(<l>,f)), ϵ     )
15 ({c,v}, (activate3,   α,          β)) → ((activate8,   α, UPDATE(c,v,β)), ϵ     )
16 (<l>   , (activate8,   α, β|(ϵ,f))) → CHK((storing2,α,UPDATE(c,v,β|(<l>,f))),⊥)
```

(b) storing1 state transitions

```
17 (<l>   , (storing2  ,   α,          β)) → ((storing2 , l|α,     β|(<l>,ϵ)), ϵ     )
18 (</l>  , (storing2  , l|α,          β)) → ((storing2 ,   α,    β|(</l>,ϵ)), ϵ     )
19 (</l>  , (storing2  , m|α,          β)) → ((storing1 ,   α,    β|(</l>,ϵ)), ϵ     )
20 ([f]   , (storing2  ,   α,          β)) → ((activate4,   α,       β|(ϵ,f)), ϵ     )
21 ({c,v}, (storing2  ,   α,          β)) → CHK((storing2,α,UPDATE(c,v,β)),⊥)
22 (<l>   , (activate4,   α, β|(ϵ,f))) → ((storing2 , l|α,     β|(<l>,f)), ϵ     )
23 ({c,v}, (activate4,   α,          β)) → ((activate9,   α, UPDATE(c,v,β)), ϵ     )
24 (<l>   , (activate9,   α, β|(ϵ,f))) → CHK((storing2,α,UPDATE(c,v,β|(<l>,f))),⊥)
```

(c) storing2 state transitions

```
25 (<l>   , (output1   ,   α,          β)) → ((output1   , l|α,              β), <l> )
26 (</l>  , (output1   , l|α,          β)) → ((output1   ,   α,              β), </l>)
27 (</l>  , (output1   ,   α,          β)) → CHK((waiting,α,β),⊥)
28 ([f]   , (output1   ,   α,          β)) → ((activate5,    α,       β|(ϵ,f)), ϵ   )
29 ({c,v}, (output1   ,   α,          β)) → ((output1   ,   α, UPDATE(c,v,β)), ϵ   )
30 (<l>   , (activate5,   α, β|(ϵ,f))) → ((output2   , m|α,     β|(<l>,f)), <l> )
31 ({c,v}, (activate5,   α,          β)) → ((activate5,    α, UPDATE(c,v,β)), ϵ   )
```

(d) output1 state transitions

```
32 (<l>   , (output2   ,   α,          β)) → ((output2   , l|α,     β|(<l>,ϵ)), <l> )
33 (</l>  , (output2   , l|α,          β)) → ((output2   ,   α,    β|(</l>,ϵ)), </l>)
34 (</l>  , (output2   , m|α,          β)) → ((output1   ,   α,    β|(</l>,ϵ)), </l>)
35 ([f]   , (output2   ,   α,          β)) → ((activate6,    α,       β|(ϵ,f)), ϵ   )
36 ({c,v}, (output2   ,   α,          β)) → ((output2   ,   α, UPDATE(c,v,β)), ϵ   )
37 (<l>   , (activate6,   α, β|(ϵ,f))) → ((output2   , l|α,     β|(<l>,f)), <l> )
38 ({c,v}, (activate6,   α,          β)) → ((activate6,    α, UPDATE(c,v,β)), ϵ   )
```

(e) output2 state transitions

State `output2`: The output transducer is in the `output2` state (cf. Figure 54(e)) if there are several partial candidates in the system and the first of these is part of the result and therefore being output. This state is combination of the `output1` and `storing2` states in that the output transducer has both to output document messages and put them into the message store.

Additionally, there are several temporary states (`activate1` to `activate9`) that are used in the same manner as the `activate` states of other SPEX transducers. For conciseness, transitions for atomic messages are not shown in Figure 54. However, it is straightforward to extend this specification for these messages.

The functions used in the output transducer are specified in a way similar to the transition function of a SPEX transducer (cf. Figure 55).

CHK($(s,\alpha,\beta),\gamma$): The task of the CHK function (cf. Figure 55(a)) is to check for candidates in the message store β, to output candidates that are part of the result by writing their messages to the list γ, and to remove unused entries from the message store. Additionally, it occasionally causes a state change in the output transducer and a modification of its count stack. If the first entry of the condition store β is the start of a candidate with a condition formula that is true (1), the OUT function is called to output this candidate and remove it from the message store. On the configuration returned by the OUT function the CHK function is called again to check for another candidate. In the case of a false condition formula of the first candidate in the store (2) the RM function is called to remove this candidate from the store. Again the CHK function is recursively called on the return value of the RM function to check for another candidate. If the message store is empty (4) or the condition formula of the first candidate is undetermined (3), the configuration that was passed as argument is returned unchanged.

OUT($s,\alpha,\beta_1,\beta_2,\gamma$): The OUT function shown in Figure 55(b) is responsible for outputting the candidate that is first in document order (i.e. the first candidate in the message store). It does this by iterating over the message store β_1 and uses the count stack α to keep track of the start and end of candidates. It outputs the messages in the store until the end of the first candidate is reached. The second message store β_2 is used to save entries that are not to be removed. Therefore, if the function is not inside a nested candidate, the current message store entry is not copied to β_2, as it is to be removed from the store (1-3). However, the store entries that are a part of some nested candidates are still needed in the store, hence they are copied to β_2 (4-7).

Normally, the OUT function returns when the end of the candidate to be output is reached. However, if the candidate is not completed yet (i.e. not all of its document messages have been received) the iteration over the message store abruptly stops when the end of the store is reached. Depending on the state that was passed on the call of the function and the number of partial candidates at the time of the function call the state has to be changed. E.g. when the output transducer was in the `storing1` state and the (only) candidate it was storing events for is also the first candidate in the store and is determined to be a result, all messages stored for the candidate have to be released and the output transducer has to enter the `output1` state, as the candidate is not completed yet. Further, the count stack has to be reorganized.

RM($s,\alpha,\beta_1,\beta_2,\gamma$): The RM function specified in Figure 55(c) is called to remove the first candidate from the message store. It works in a way very similar to the OUT function, except that no output is created.

Figure 55 Functions used in the specification of the output transducer

1 $\texttt{CHK}((\mathit{any\_state},\ \alpha,\ (d,\texttt{true})\texttt{|}\beta),\ \gamma) \rightarrow \texttt{CHK}(\texttt{OUT}(\mathit{any\_state},\ \texttt{n}_1\texttt{|}\alpha,\ \beta,\ \bot,\ \gamma\texttt{|<}l\texttt{>}))$

2 $\texttt{CHK}((\mathit{any\_state},\ \alpha,\ (d,\texttt{false})\texttt{|}\beta),\ \gamma) \rightarrow \texttt{CHK}(\texttt{RM}(\mathit{any\_state},\ \texttt{n}_1\texttt{|}\alpha,\ \beta,\ \bot,\ \gamma)\ )$

3 $\texttt{CHK}((\mathit{any\_state},\ \alpha,\ (d,f)\texttt{|}\beta),\ \gamma) \rightarrow ((\mathit{any\_state},\ \alpha,\ (d,f)\texttt{|}\beta),\ \gamma)$

4 $\texttt{CHK}((\mathit{any\_state},\ \alpha,\ \bot),\ \gamma) \rightarrow ((\mathit{any\_state},\ \alpha,\ \bot),\ \gamma)$

(a) Function CHK: $\Phi \times \Omega^* \rightarrow \Phi \times \Omega^*$

1 $\texttt{OUT}(\mathit{any\_state},\ \alpha,\ (\texttt{<}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{OUT}(\mathit{any\_state},\ \texttt{1|}\alpha,\ \beta_1,\ \beta_2,\ \gamma\texttt{|<}l\texttt{>})$

2 $\texttt{OUT}(\mathit{any\_state},\ \texttt{1|}\alpha,\ (\texttt{</}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{OUT}(\mathit{any\_state},\ \alpha,\ \beta_1,\ \beta_2,\ \gamma\texttt{|</}l\texttt{>})$

3 $\texttt{OUT}(\mathit{any\_state},\ \texttt{n}_1\texttt{|}\alpha,\ (\texttt{</}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow ((\mathit{any\_state},\ \alpha,\ \beta_2),\ \gamma\texttt{|</}l\texttt{>})$

4 $\texttt{OUT}(\texttt{waiting},\ \alpha,\ (\texttt{<}l\texttt{>},f)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{OUT}(\texttt{waiting:2},\ \texttt{n}_2\texttt{|}\alpha,\ \beta_1,\ \beta_2\texttt{|(<}l\texttt{>},f),\ \gamma\texttt{|<}l\texttt{>})$

5 $\texttt{OUT}(\texttt{waiting:2},\ \alpha,\ (\texttt{<}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{OUT}(\texttt{waiting:2},\ \texttt{1|}\alpha,\ \beta_1,\ \beta_2\texttt{|(<}l\texttt{>},\epsilon),\ \gamma\texttt{|<}l\texttt{>})$

6 $\texttt{OUT}(\texttt{waiting:2},\ \texttt{1|}\alpha,\ (\texttt{</}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{OUT}(\texttt{waiting:2},\ \alpha,\ \beta_1,\ \beta_2\texttt{|(</}l\texttt{>},\epsilon),\ \gamma\texttt{|</}l\texttt{>})$

7 $\texttt{OUT}(\texttt{waiting:2},\ \texttt{n}_2\texttt{|}\alpha,\ (\texttt{</}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{OUT}(\texttt{storing2},\ \alpha,\ \beta_1,\ \beta_2\texttt{|(</}l\texttt{>},\epsilon),\ \gamma\texttt{|</}l\texttt{>})$

8 $\texttt{OUT}(\texttt{storing1},\ \alpha,\ (\texttt{<}l\texttt{>},f)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{OUT}(\texttt{storing1:2},\ \texttt{n}_2\texttt{|}\alpha,\ \beta_1,\ \beta_2\texttt{|(<}l\texttt{>},f),\ \gamma\texttt{|<}l\texttt{>})$

9 $\texttt{OUT}(\texttt{storing1},\ \alpha_1\texttt{|n}_1\texttt{|}\alpha_2,\ \bot,\ \beta_2,\ \gamma) \rightarrow ((\texttt{output1},\ \alpha_2,\ \beta_2),\ \gamma)$

10 $\texttt{OUT}(\texttt{storing1:2},\ \alpha,\ (\texttt{<}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{OUT}(\texttt{storing1:2},\ \texttt{1|}\alpha,\ \beta_1,\ \beta_2\texttt{|(<}l\texttt{>},\epsilon),\ \gamma\texttt{|<}l\texttt{>})$

11 $\texttt{OUT}(\texttt{storing1:2},\ \texttt{1|}\alpha,\ (\texttt{</}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{OUT}(\texttt{storing1:2},\ \alpha,\ \beta_1,\ \beta_2\texttt{|(</}l\texttt{>},\epsilon),\ \gamma\texttt{|</}l\texttt{>})$

12 $\texttt{OUT}(\texttt{storing1:2},\ \texttt{n}_2\texttt{|}\alpha,\ (\texttt{</}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{OUT}(\texttt{storing2},\ \alpha,\ \beta_1,\ \beta_2\texttt{|(</}l\texttt{>},\epsilon),\ \gamma\texttt{|</}l\texttt{>})$

13 $\texttt{OUT}(\texttt{storing2},\ \alpha,\ (\texttt{<}l\texttt{>},f)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{OUT}(\texttt{storing2:2},\ \texttt{n}_2\texttt{|}\alpha,\ \beta_1,\ \beta_2\texttt{|(<}l\texttt{>},f),\ \gamma\texttt{|<}l\texttt{>})$

14 $\texttt{OUT}(\texttt{storing2:2},\ \alpha,\ (\texttt{<}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{OUT}(\texttt{storing2:2},\ \texttt{1|}\alpha,\ \beta_1,\ \beta_2\texttt{|(<}l\texttt{>},\epsilon),\ \gamma\texttt{|<}l\texttt{>})$

15 $\texttt{OUT}(\texttt{storing2:2},\ \texttt{1|}\alpha,\ (\texttt{</}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{OUT}(\texttt{storing2:2},\ \alpha,\ \beta_1,\ \beta_2\texttt{|(</}l\texttt{>},\epsilon),\ \gamma\texttt{|</}l\texttt{>})$

16 $\texttt{OUT}(\texttt{storing2:2},\ \texttt{n}_2\texttt{|}\alpha,\ (\texttt{</}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{OUT}(\texttt{storing2},\ \alpha,\ \beta_1,\ \beta_2\texttt{|(</}l\texttt{>},\epsilon),\ \gamma\texttt{|</}l\texttt{>})$

17 $\texttt{OUT}(\texttt{storing2:2},\ \alpha_1\texttt{|n}_1\texttt{|}\alpha_2,\ \bot,\ \beta_2,\ \gamma) \rightarrow ((\texttt{output2},\ \alpha_2,\ \beta_2),\ \gamma)$

(b) Function OUT: $\mathcal{Q} \times \Gamma^*_{count} \times \Gamma^*_{cond} \times \Gamma^*_{cond} \times \Omega^* \rightarrow \Phi \times \Omega^*$

1 $\texttt{RM}(\mathit{any\_state},\ \alpha,\ (\texttt{<}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{RM}(\mathit{any\_state},\ \texttt{1|}\alpha,\ \beta_1,\ \beta_2,\ \gamma)$

2 $\texttt{RM}(\mathit{any\_state},\ \texttt{1|}\alpha,\ (\texttt{</}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{RM}(\mathit{any\_state},\ \alpha,\ \beta_1,\ \beta_2,\ \gamma)$

3 $\texttt{RM}(\mathit{any\_state},\ \texttt{n}_1\texttt{|}\alpha,\ (\texttt{</}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow ((\mathit{any\_state},\ \alpha,\ \beta_2),\ \gamma)$

4 $\texttt{RM}(\texttt{waiting},\ \alpha,\ (\texttt{<}l\texttt{>},f)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{RM}(\texttt{waiting:2},\ \texttt{n}_2\texttt{|}\alpha,\ \beta_1,\ \beta_2\texttt{|(<}l\texttt{>},f),\ \gamma)$

5 $\texttt{RM}(\texttt{waiting:2},\ \alpha,\ (\texttt{<}l\texttt{>},f)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{RM}(\texttt{waiting:2},\ \texttt{1|}\alpha,\ \beta_1,\ \beta_2\texttt{|(<}l\texttt{>},\epsilon),\ \gamma)$

6 $\texttt{RM}(\texttt{waiting:2},\ \texttt{1|}\alpha,\ (\texttt{</}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{RM}(\texttt{waiting:2},\ \alpha,\ \beta_1,\ \beta_2\texttt{|(</}l\texttt{>},\epsilon),\ \gamma)$

7 $\texttt{RM}(\texttt{waiting:2},\ \texttt{n}_2\texttt{|}\alpha,\ (\texttt{</}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{RM}(\texttt{waiting},\ \alpha,\ \beta_1,\ \beta_2\texttt{|(</}l\texttt{>},\epsilon),\ \gamma)$

8 $\texttt{RM}(\texttt{storing1},\ \alpha,\ (\texttt{<}l\texttt{>},f)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{RM}(\texttt{storing1:2},\ \texttt{n}_2\texttt{|}\alpha,\ \beta_1,\ \beta_2\texttt{|(<}l\texttt{>},f),\ \gamma)$

9 $\texttt{RM}(\texttt{storing1},\ \alpha_1\texttt{|n}_1\texttt{|}\alpha_2,\ \bot,\ \beta_2,\ \gamma) \rightarrow ((\texttt{waiting},\ \bot,\ \beta_2),\ \gamma)$

10 $\texttt{RM}(\texttt{storing1:2},\ \alpha,\ (\texttt{<}l\texttt{>},f)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{RM}(\texttt{storing1:2},\ \texttt{1|}\alpha,\ \beta_1,\ \beta_2\texttt{|(<}l\texttt{>},\epsilon),\ \gamma)$

11 $\texttt{RM}(\texttt{storing1:2},\ \texttt{1|}\alpha,\ (\texttt{</}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{RM}(\texttt{storing1:2},\ \alpha,\ \beta_1,\ \beta_2\texttt{|(</}l\texttt{>},\epsilon),\ \gamma)$

12 $\texttt{RM}(\texttt{storing1:2},\ \texttt{n}_2\texttt{|}\alpha,\ (\texttt{</}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{RM}(\texttt{storing1},\ \alpha,\ \beta_1,\ \beta_2\texttt{|(</}l\texttt{>},\epsilon),\ \gamma)$

13 $\texttt{RM}(\texttt{storing2},\ \alpha,\ (\texttt{<}l\texttt{>},f)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{RM}(\texttt{storing2:2},\ \texttt{n}_2\texttt{|}\alpha,\ \beta_1,\ \beta_2\texttt{|(<}l\texttt{>},f),\ \gamma)$

14 $\texttt{RM}(\texttt{storing2:2},\ \alpha,\ (\texttt{<}l\texttt{>},f)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{RM}(\texttt{storing2:2},\ \texttt{1|}\alpha,\ \beta_1,\ \beta_2\texttt{|(<}l\texttt{>},\epsilon),\ \gamma)$

15 $\texttt{RM}(\texttt{storing2:2},\ \texttt{1|}\alpha,\ (\texttt{</}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{RM}(\texttt{storing2:2},\ \alpha,\ \beta_1,\ \beta_2\texttt{|(</}l\texttt{>},\epsilon),\ \gamma)$

16 $\texttt{RM}(\texttt{storing2:2},\ \texttt{n}_2\texttt{|}\alpha,\ (\texttt{</}l\texttt{>},\epsilon)\texttt{|}\beta_1,\ \beta_2,\ \gamma) \rightarrow \texttt{RM}(\texttt{storing2},\ \alpha,\ \beta_1,\ \beta_2\texttt{|(</}l\texttt{>},\epsilon),\ \gamma)$

17 $\texttt{RM}(\texttt{storing2:2},\ \alpha_1\texttt{|n}_1\texttt{|}\alpha_2\texttt{|m|}\alpha_3,\ \bot,\ \beta_2,\ \gamma) \rightarrow ((\texttt{storing2},\ \alpha_2,\ \beta_2),\ \gamma)$

18 $\texttt{RM}(\texttt{storing2:2},\ \alpha_1\texttt{|n}_1\texttt{|}\alpha_2,\ \bot,\ \beta_2,\ \gamma) \rightarrow ((\texttt{storing1},\ \alpha_2,\ \beta_2),\ \gamma)$

(c) Function RM: $\mathcal{Q} \times \Gamma^*_{count} \times \Gamma^*_{cond} \times \Gamma^*_{cond} \times \Omega^* \rightarrow \Phi \times \Omega^*$

Figure 56 A denotational semantics for forward XPath expressions

$\mathcal{C}$: Expression $\rightarrow$ (Network, Tape) $\rightarrow$ (Network, Tape)

$$\mathcal{C}[\![\ /qpath\ ]\!](\sigma,t) = \mathcal{C}[\![qpath]\!](\sigma,t)$$

$$\mathcal{C}[\![/qpath_1\ op\ /qpath_2]\!](\sigma,t) = ((\sigma_4,t_6) \mid \sigma_1 = \sigma \cup \{\mathsf{SP}^{t}_{t_1,t_2}\}, (\sigma_2,t_3) = \mathcal{C}[\![qpath_1]\!](\sigma_1,t_1), (\sigma_3,t_4) = \mathcal{C}[\![qpath_2]\!](\sigma_2,t_2), \sigma_4 = \sigma_3 \cup \{\mathsf{JO}^{t_3,t_4}_{t_5}, \mathsf{SE}^{t_5}_{t_6}(op)\})$$

$$\mathcal{C}[\![\ qstep/qpath\ ]\!](\sigma,t) = \mathcal{C}[\![qpath]\!](\mathcal{C}[\![qstep]\!](\sigma,t))$$

$$\mathcal{C}[\![\ step\ qualifiers\ ]\!](\sigma,t) = \mathcal{C}[\![qualifiers]\!](\mathcal{C}[\![qstep]\!](\sigma,t))$$

$$\mathcal{C}[\![\ qualifier\ qualifiers]\!](\sigma,t) = \mathcal{C}[\![qualifiers]\!](\mathcal{C}[\![qualifier]\!](\sigma,t))$$

$$\mathcal{C}[\![\ axis::node\text{-}test\ ]\!](\sigma,t) = (\{\sigma, \mathsf{ST}^{t}_{t_1}(axis, node\text{-}test)\}, t_1)$$

$$\mathcal{C}[\![\ [\ path\ ]\ ]\!](\sigma,t) = ((\sigma_3,t_7) \mid \sigma_1 = \sigma \cup \{\mathsf{VC}^{t}_{t_1}(q),\ \mathsf{SP}^{t_1}_{t_2,t_3}\},\ (\sigma_2,t_4) = \mathcal{C}[\![path]\!](\sigma_1,t_2), \sigma_3 = \sigma_2 \cup \{\mathsf{VF}^{t_4}_{t_5}(q),\ \mathsf{PT}^{t_5}_{t_6},\ \mathsf{JO}^{t_3,t_6}_{t_7}\})$$

$$\mathcal{C}[\![\ [\ path_1\texttt{==}path_2\ ]\ ]\!](\sigma,t) = ((\sigma_4,t_{10}) \mid \sigma_1 = \sigma \cup \{\mathsf{VC}^{t}_{t_1}(q), \mathsf{SP}^{t_1}_{t_2,t_3}, \mathsf{SP}^{t_2}_{t_4,t_5}\}, (\sigma_2,t_6) = \mathcal{C}[\![path_1]\!](\sigma_1,t_4), (\sigma_3,t_7) = \mathcal{C}[\![path_2]\!](\sigma_2,t_5), \sigma_4 = \sigma_3 \cup \{\mathsf{JO}^{t_6,t_7}_{t_8}, \mathsf{VF}^{t_8}_{t_9}(q), \mathsf{IT}^{t_9}_{t_{10}}, \mathsf{JO}^{t_2,t_{10}}_{t_{11}}\})$$

$$\mathcal{C}[\![\ /relpath\ ]\!](\sigma,t) = ((\sigma_2,t_3) \mid t_1 = getInitTape(\sigma), setInitTape(\sigma,t_0), \sigma_1 = \sigma \cup \{\mathsf{SP}^{t_0}_{t_1,t_2}\}, (\sigma_2,t_3) = \mathcal{C}[\![relpath]\!](\sigma_1,t_2))$$

$$\mathcal{C}[\![\ step/relpath\ ]\!](\sigma,t) = \mathcal{C}[\![relpath]\!](\mathcal{C}[\![step]\!](\sigma,t))$$

4.7 Compiling XPath Expressions into SPEX Networks

The compilation of an XPath expression into a SPEX network comprises two important steps. First, the expression is translated into a forward XPath expression by using the first of the rewriting algorithms presented in [27]. Second, the rewritten expression is translated into a SPEX network, as described here.

The compilation of a forward XPath expression into a SPEX network is given by means of a denotational semantics, specified as a function $\mathcal{C}$ shown in Figure 56. The function $\mathcal{C}$ is defined by induction on the structure of XPath expressions, as introduced in Section 2. This is possible due to the compositional nature of XPath expressions and SPEX networks. The brackets $[\![\ ]\!]$ are used to stress that the semantics of the expressions they enclose is defined.

$\mathcal{C}$ maps an XPath expression, a given initial SPEX network configuration, and its output tape to an updated SPEX network and an output tape, the new network being constituted from the initial network and the network representation of the given expression. A network is represented by a set of interconnected SPEX transducers, as defined in Section 4.

The input and output transducers are added to a SPEX network after the internal components are set up by means of $\mathcal{C}$. Hence, a SPEX network corresponding to a path p is $(\{\sigma, \mathsf{OU}^{t_1}_{t_2}\} \mid (\sigma,t_1) = \mathcal{C}[\![p]\!](\mathsf{IN}^{in}_{t},t))$.

The functions *getInitTape* and *setInitTape* get or set, respectively the initial input tape of the network, which represents the output tape of the input transducer. They are used to add to the network transducers that represent absolut XPath expressions, e.g. for paths inside qualifiers.

5 Complexity Results

Each XML stream is described by its size s and the depth d of the (unmaterialized) document tree that is associated with the stream. It is assumed that $d \ll s$, i.e. the depth is significantly smaller than the size of the stream. An XPath query q is specified by $q(n)$ where n is its length. S_t (T_t, resp.) denotes the space (the time, resp.) complexity of a given transducer t for processing the query $q(n)$ on the given XML stream.

Lemma 5.1 (SPEX Network Construction). *The compilation time and the degree of a SPEX network for $q(n)$ is linear in n.*

Proof. The compilation consists in two steps. First, $q(n)$ is translated in a reverse-axis-free query $q'(n')$. As proven in [27], q' has a size linear in n ($n' = c \times n$ where c is some constant) and is rewritten in time linear in n. Second, $q'(n')$ is translated by means of a denotational semantics, as presented in Section 4.7, into a network of SPEX transducers. The number of SPEX transducers created for each XPath construct that may appear in q' is constant, hence the degree of the network corresponding to $q'(n')$ is linear in n'. The time for adding each transducer to the network is constant, hence the compilation time is linear in n' and thus in n. □

The space complexity for SPEX transducers is determined by the amount of memory used by their count and condition stacks. A count stack allows only entries of constant space and it can have a maximum number of d entries, since it counts the nested levels in the tree conveyed in the stream. Therefore, the space needed for a count stack is linear in d. The time for pushing and popping a count stack entry is constant, hence for processing the entire stream the time complexity for managing a count stack is linear in s. Below, only results concerning the condition stacks are shown.

As presented in Section 4, an activation message carries a condition formula. In a system without `following` and `following-sibling` steps, a step with a qualifier can match nested document messages at most d times. For that step, qualifier instances, represented by condition variables, are created. Thus, the number of condition variables at a time in the system is at most d and a condition formula can refer to at most d condition variables at a time. In the following lemmas a condition formula is considered to have size linear in d.

Lemma 5.2 (Step Transducers (1)). *If there is no transducer with type* `following` *or* `following − sibling` *in the system, the space and time needed by a step transducer* ST *for processing a stream with size s and depth d is $S_{\mathsf{ST}} = O(d^2)$, $T_{\mathsf{ST}} = O(d \times s)$.*

Proof. In a system without `following` and `following-sibling` transducers, a condition stack allows entries, represented by condition formulas, which have size linear in d. A descendant transducer can activate the next transducer, as result of matching, with a formula consisting of a disjunction of the entire stack, normalized in the manner outlined in 4.2.2. Therefore, a condition stack entry of the next transducer, which is a condition formula, occupies a space linear in d if there is no `following` or `following-sibling` transducer in the system. There can be d entries on a condition stack, due to the maximum of d nested activations in all step transducers. Hence, the space needed by a step transducer is quadratic in d: $S_{\mathsf{ST}} = O(d^2)$.

The processing time needed for one incoming message is determined by the received formulas within activation messages. A transducer needs time for copying a formula to and from its condition stack. This time is linear in d. For the entire stream, the processing time is linear in s: $T_{\mathsf{ST}} = O(d \times s)$. □

Lemma 5.3 (Step Transducers (2)). *If there is at least one transducer with type* `following` *or* `following − sibling` *in the system, the space and time needed by a step transducer* ST *for processing a stream with size* s *and depth* d *is* $S_{\mathsf{ST}} = O(d \times s)$, $T_{\mathsf{ST}} = O(s^2)$.

Proof. Analog to the proof of Lemma 5.2, except that `following` and `following-sibling` transducers can create condition formulas with a size linear in s. □

Lemma 5.4 (Variable-Creator Transducer). *The space and time used by a variable-creator transducer for processing a stream with size* s *and depth* d *is* $S_{VC} = O(d)$, $T_{VC} = O(d \times s)$.

Proof. A variable-creator instantiates for each received activation a new condition variable with a stamp given by a certain qualifier. Each condition variable is stored on an entry on the condition stack, and the condition stack can have at most d entries, which is the maximum number of nested activations. Hence, the space needed for the condition stack is linear in d: $S_{VC} = O(d)$.

The time complexity of a variable-creator transducer is given by the time needed for copying formulas from input to output, hence time linear in d for each activation message. Since there can be as many activation messages as document messages, the time needed for processing the entire stream is $T_{VC} = O(d \times s)$. □

Lemma 5.5 (Input, Variable-Filter, Path-Terminator, Split and Join Transducers). *The space and time needed by input, variable-filter, path-terminator, split and join transducers for processing a stream with size* s *and depth* d *is* $S_t = O(1)$, $T_t = O(d \times s)$, *where* $t \in \{IN, VF, PT, SP, JO\}$.

Proof. Input, variable-filter, path-terminator, split and join transducers do not use their push-down store, hence their space requirement is constant: $S_t = O(1)$. Copying an activation message from input to output requires time linear in d, as the size of activation messages is linear in d. Since there can be as many activation messages as document messages, the time needed for copying all the s activation messages is $T_t = O(d \times s)$. The input transducer does not encounter activation messages, hence $T_{IN} = O(s)$. □

Lemma 5.6 (Set and Identity-Terminator Transducers). *The space and time used by set and identity-terminator transducers for processing a stream with size* s *and depth* d *is* $S_{SE} = S_{IT} = O(d)$, $T_{SE} = S_{IT} = O(d \times s)$.

Proof. A set or an identity-terminator transducer needs space only on the condition stack. More specifically, it needs one entry on the stack, representing a formula, which has size linear in d. Hence, $S_{SE} = S_{IT} = O(d)$.

The processing time of one message from the input stream includes the time for handling a related activation message.

- For a set transducer, that is the time for copying the message on the condition stack, which is linear in d, and the time for creating a disjunction or conjunction of two formulas with variable duplicate elimination, cf. Section 4, which is also d. Hence, for the entire stream, $T_{SE} = O(d \times s)$.
- For an identity-terminator, that is the time for copying the activation message on the condition stack, which is linear in d, and the time for choosing common condition variables, which is also linear in d. Hence, for the entire stream, $T_{IT} = O(d \times s)$.

□

Lemma 5.7 (Output Transducer). *The space and time needed by an output transducer for querying a stream with size s and depth d is $S_{OU} = O(d \times s)$, $T_{OU} = O(d \times s)$.*

Proof. The output transducer needs space for candidates and for their formulas. A formula has size linear in d, a candidate can be of size linear in s. Further, there can be s candidates in the system. However, for storing the messages of candidates a single buffer can be used. In the worst case, the whole document is stored, leading to a space requirement for the messages of candidates of s. As condition formulas have to be stored for every candidate, the space requirement for the condition formula of a single candidate is d, hence $S_{OU} = O(d \times s)$.

Managing the candidates and their related formulas implies time for copying a formula for each candidate. There can be s candidates, hence for each of them a formula of size linear in d has to be copied on a store. Hence, $T_{OU} = O(d \times s)$. □

Theorem 5.1 (Querying XML Streams with SPEX Networks (1)). *If there is no* `following` *or* `following − sibling` *transducer in the system, the space $S_{\texttt{net}}$ and the time $T_{\texttt{net}}$ needed by a SPEX network* `net` *for querying a stream with size s and depth d is $S_{\texttt{net}} = O(d \times s) \approx O(s)$, $T_{\texttt{net}} = O(d \times s) \approx O(s)$.*

Proof. The space and time used by a SPEX network `net` for querying a stream with size s and depth d is given by the sum of the space, respectively of the time, needed by its components. Since the output transducer imposes the most consuming space and time, which are linear in d times s, the space and time complexity is: $S_{\texttt{net}} = O(d \times s)$ and $T_{\texttt{net}} = O(d \times s)$. □

Theorem 5.2 (Querying XML Streams with SPEX Networks (2)). *If there is a transducer of type* `following` *or* `following − sibling` *in the system, the space $S_{\texttt{net}}$ and the time $T_{\texttt{net}}$ needed by a SPEX network* `net` *for querying a stream with size s and depth d is $S_{\texttt{net}} = O(s^2)$, $T_{\texttt{net}} = O(s^2)$.*

Proof. Analog to Theorem 5.1. □

6 Implementation

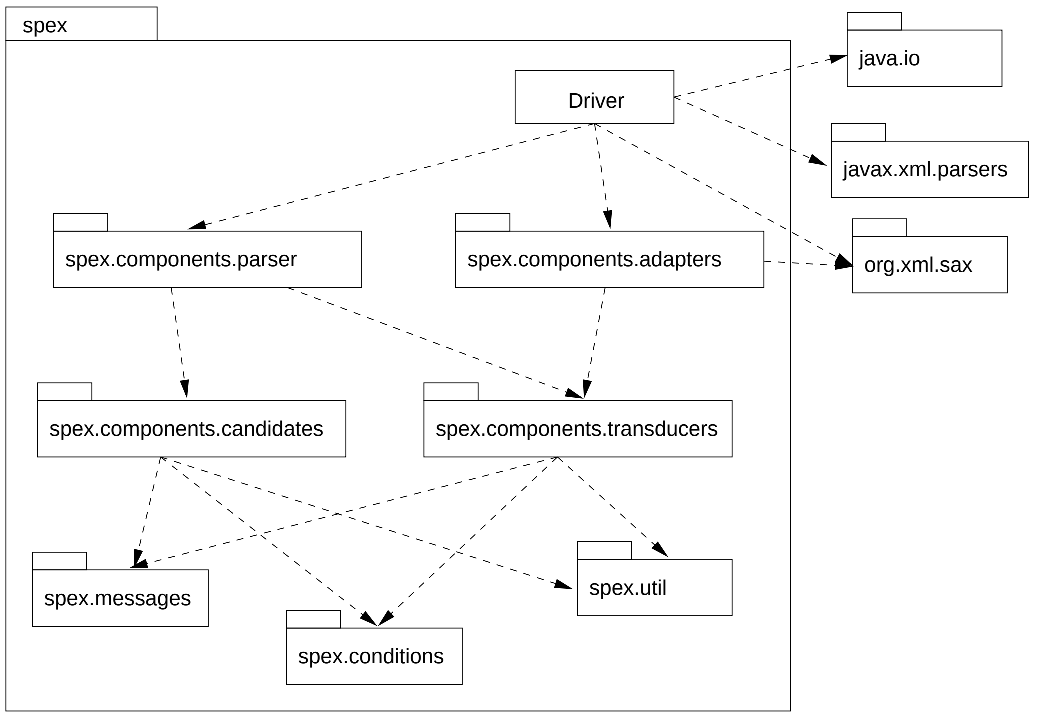
For an experimental evaluation of the developed processing model a prototype was implemented in Java. Most of the system, is a straightforward implementation of the transducer specifications of Section 4 to the target language Java. There are some exceptions to this that are explained in Section 6.3. Section 6.1 shows the design of the prototype using UML diagrams [28]. Experimental results are shown in Section 7.

6.1 Design

The `spex` package shown in Figure 57 comprises all the components of the SPEX system. The `spex.Driver` class uses the `spex.components.parser` and `spex.components.adapters` as well as some external packages to provide a command-line tool for the task of querying a stream that is generated out of an XML file. The external classes used are

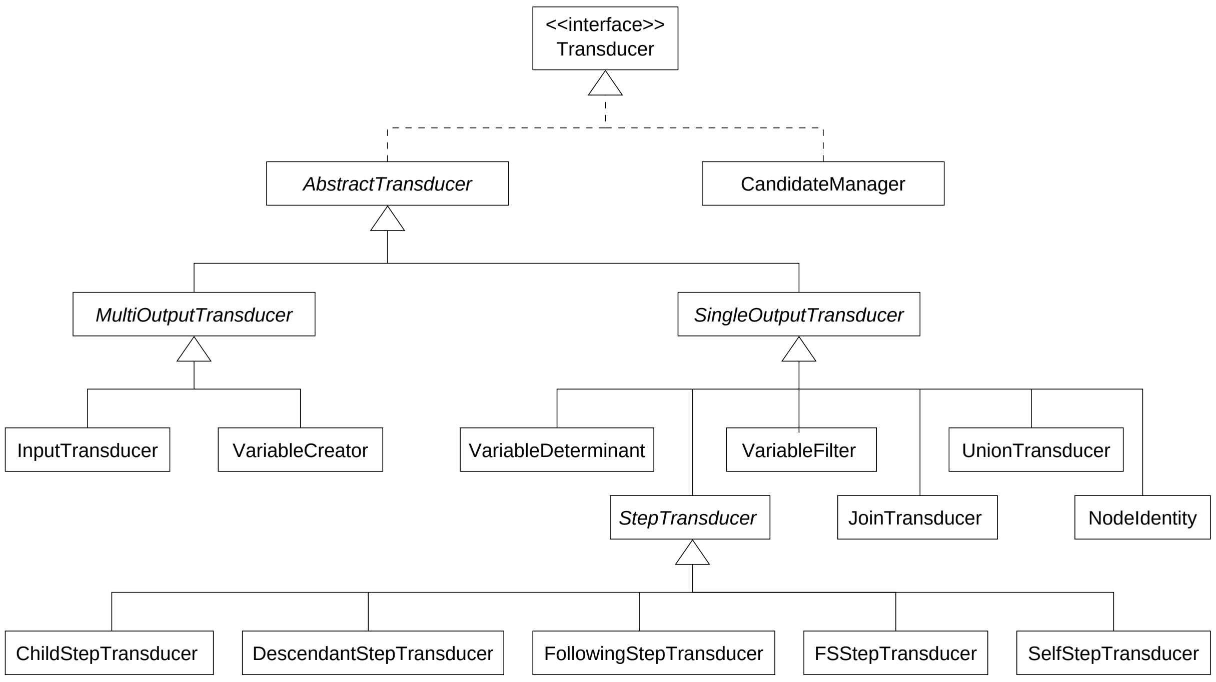
- `org.xml.sax` for the creation of a SAX input stream out of an XML source,
- `javax.xml.parser` for an abstract interface to XML parsers,
- the SAX-parser implementation Xerces [9] and the adaptation of the Ælfred parser [7] used in Saxon [8].

Figure 57 Overview of the packages



Classes in the `spex.components.parser` package are responsible for parsing XPath queries and translating them to SPEX networks. This translation is specified using the Java Compiler Compiler (JavaCC) from WebGain [5]. The `spex.components.adapters` package contains several simple adapter classes to convert to and from SAX events and to provide text output.

Figure 58 Classes in the package `spex.components.transducers`



The SPEX network created by the translator is composed of transducers which are specified in the `spex.components.candidates` and `spex.components.transducers` packages. These transducers in turn make use of the SPEX messages defined in the `spex.messages` package, the conditions of the `spex.conditions` package, and some utilities contained in the `spex.util` package.

Figure 58 shows the inheritance hierarchy of all the transducers in the SPEX prototype. The `Transducer` interface defines the basic outline of a SPEX transducer. It is directly implemented by the `CandidateManager` class which represents the output transducer from Section 4.6. The second class implementing the `Transducer` interface, the abstract class `AbstractTransducer`, is the base class of all other SPEX transducers. It is subclassed by the (still abstract) classes `SingleOutputTransducer` and `MultiOutputTransducer` that represent SPEX transducers with

Figure 59 Classes in the package `spex.messages`

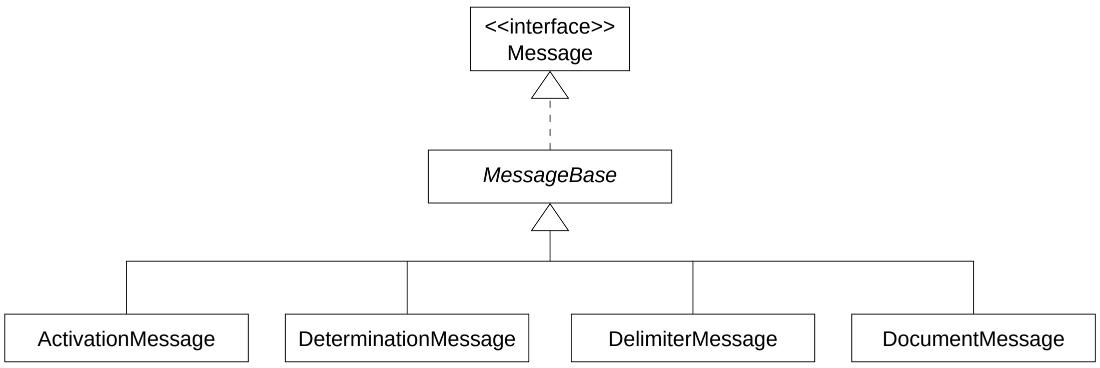
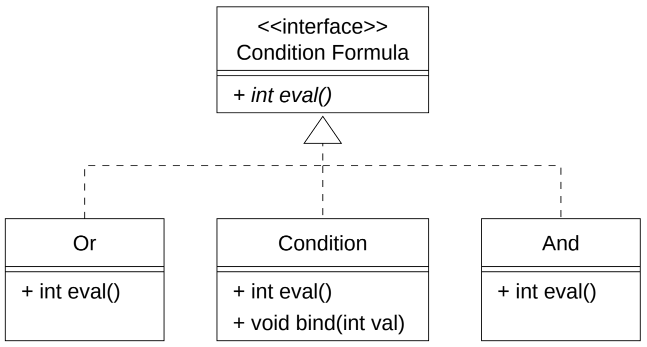


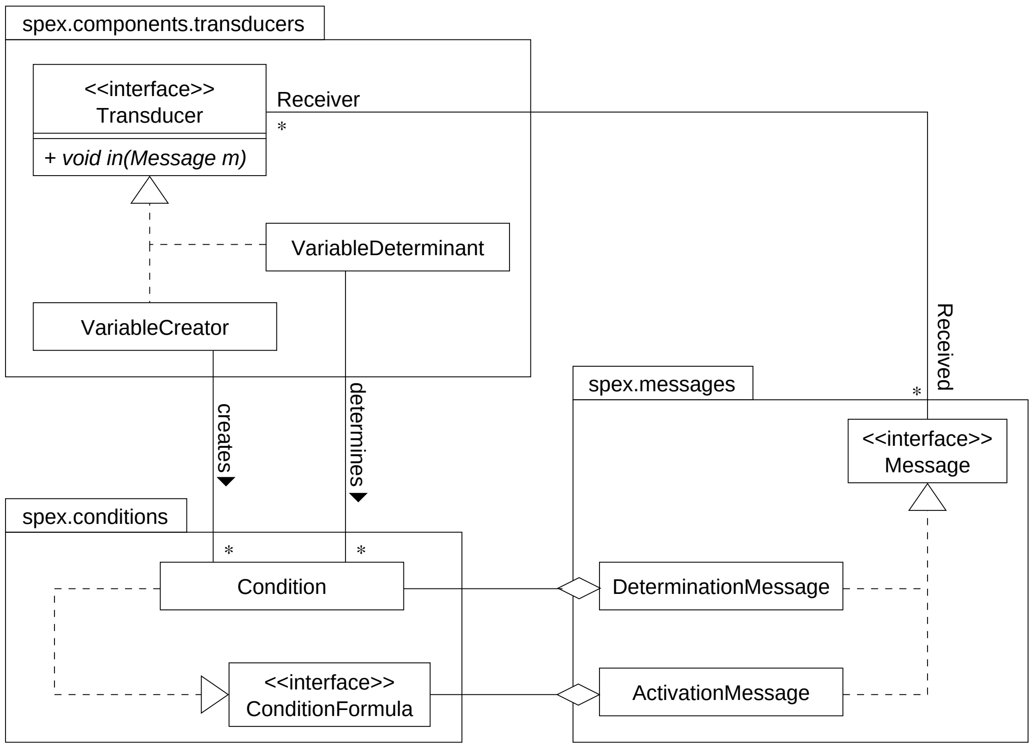
Figure 60 Classes in the package `spex.conditions`



one input tape and multiple input tapes, respectively. As there are only two cases where a split transducer of Section 4 is placed, the split transducer itself is not specified here. Rather its (trivial) functionality is integrated in the `InputTransducer` and `VariableCreator` classes derived from `MultiOutputTransducer`. Several final classes like the `VariableDeterminant` and the `UnionTransducer` are derived from the `SingleOutputTransducer` class. There is another level in the inheritance tree for the step transducers of Section 4.2 represented by the abstract class `StepTransducer`. For every supported XPath axis there is one subclass of `StepTransducer`.

The classes defined in the `spex.messages` package are shown in Figure 59. The interface `Message` which specifies the basic properties of a SPEX message is implemented by the abstract class `MessageBase` that provides a base class for the different messages. There is one class for every SPEX message defined in Section 4.1.

Figure 61 Relationships between packages



The `spex.conditions` package shown in Figure 60 contains the definitions of conditions and condition formulas. The properties of a condition formula are specified by the `ConditionFormula` interface. This interface is directly implemented by the different parts a condition formula can consist of: conjunctions (class `Or`), disjunctions (class `And`), and atomic conditions (class `Condition`). Figure 60 also shows the methods of the classes in the `spex.conditions` package. Every `ConditionFormula` can be evaluated to a truth value (there are three possible values: true, false, and undetermined). Additionally, the `bind()` operation assigns a value to a condition.

Figure 61 shows a more complex UML diagram that illustrates the dependencies between the different packages. Messages can be sent to transducers by calling their `in()` method. There are two transducers in the `spex.components.transducers` package that are directly associated to conditon formulas. The `VariableCreator` is responsible for creating the conditions related to a specific qualifier. Together with a corresponding `VariableDeterminant` the `VariableCreator` also determines the value of these conditions[†]. Most of the other transducers are indirectly related to condition formulas, as they receive activation and determination messages that include condition formulas or conditions.

6.2 Transducer Implementation

Implementation of transducers was done by a simple translation of the transducer definitions of Section 4 into Java code. Some utility methods are used to be able to conveniently specify the transitions in Java. Figure 62 shows a part of the transition-function implementation of the transducer with type `following`.

Figure 62 Transition function of the transducer with type `following`

```
/**
 * method containing the transition function
 * of the following transducer
 */
public void in(Message m) {

    // transitions 1 to 4
    if(state == WAITING) {
        // transition 1
        if(isActivation(m)) {
            ConditionFormula f = ((ActivationMessage)m).getFormula();
            state = ACTIVATE1;
            countStack.push(new Activation(f));
            return;
        }
        // transition 2
        if(isStart(m)) {
            countStack.push(new Level());
            out(m);
            return;
        }
        ...
    }
    ...
}
```

[†]The true value of a condition is determined by the `VariableDeterminant`, a false value is determined by the `VariableCreator`.

6.3 Specific Issues

Although the implementation is a straightforward translation of the system specified in Section 4, there are several issues where the implementation differs from the specification.

- **Connection of transducers**: The implemented transducers do not use real input and output tapes, but rather they rely on SAX-like method callbacks for communication. A transducer passes messages to another transducer by calling its `in()` method. Therefore, connections between two transducers t_1 and t_2 are established by having t_1 storing a reference to t_2. A transducer with multiple output tapes is implemented by storing a list of transducer references instead of just one. In the case of an outgoing message the `in()` method of every transducer in this list is called. A transducer t has multiple input tapes if there is more than one transducer in the system that stores a reference to t.
- **System synchronization**: In Section 4 synchronization between transducers was ensured by having the input transducer wait for a document message to reach the output transducer before sending the next message. However, the message passing by method callbacks provides an implicit synchronization: A transducer t_1 passes a message m to a transducer t_2 by calling the `in()` method of t_2. t_1 is now suspended until t_2 finishes processing m.
- **Conditions**: Conditions are implemented as objects that have an identity and include a field that contains the value of the condition. Rather than passing around conditions as variables that are either bound to a value or unbound, references to condition objects are included in condition formulas. Hence, if a transducer determines the value of a condition it sets the value of the condition object accordingly. All other transducers know about this change immediately, as they store references to the same condition object. Therefore, no updating of condition stacks is necessary (i.e. the UPDATE function specified in Section 4 is not needed). Condition determination messages are still used, but they are interpreted as signals to inform about the changing of a variable value, rather than specifying a binding of a variable to a value.
- **Output transducer**: The `CandidateManager` class represents the output transducer of Section 4. Its basic transitions are exactly as defined in Section 4.6. However, the functions defined there are implemented differently for reasons of efficiency.

7 Experimental results

For evaluating the performance of the SPEX approach, a rather straightforward prototype (SPEX processor) has been implemented in Java and tested against databases of various sizes and characteristics (cf. Table 1):

- the MONDIAL geographical database [21] as a rather small and highly structured XML document,
- an excerpt of the lexical WordNet database [17] as a medium sized, flat, and highly repetitive RDF representation,
- and the structure resp. content files of the Open Directory Project (DMOZ) [3] as large, flat RDF documents.

Five types of queries have been considered for comparing the efficiency of the SPEX prototype with existing XPath-like implementations (cf. Table 2):

- **non-matching**: a query that does not match any of the elements in the data to be queried, e.g. `//non-existing`,
- **simple**: simple structural queries that do not create nested results, e.g. `//province/city`,
- **nested**: a structural query that creates nested results, e.g. `//*/*`,
- **future**: queries with structural qualifiers that create future conditions (i.e. where candidates are related to conditions whose values are unknown at the time of candidate creation), e.g. `//country[province]/name`,
- **past**: queries with structural qualifiers that create past conditions (i.e. where candidates are related to conditions whose values are known at the time of candidate creation), e.g. `//country[province]/religions`.

Queries were performed on a Pentium III 1GHz, 512 MB system running under SuSE Linux 7.3. The Java sources were compiled with the Jikes compiler from IBM [6] and the Java Virtual Machine used is from the IBM Developer Kit for Linux 1.3.0 [4]. Table 3 shows the run times of the SPEX processor for evaluating the queries shown in Table 2 against the MONDIAL and WordNet databases, as well as both of the files containing the structure and content of DMOZ. Table 4 shows the run times of Fxgrep [24], a functional XML querying tool, and the Saxon [8] and Xalan [10] XSLT processors for the same queries. However, Table 4 only contains run times for the MONDIAL and WordNet databases. It was not possible to get a result for the DMOZ files, as the memory usage of the Fxgrep, Saxon, and Xalan processors showed to be beyond the limitations of the system used for experiments. Therefore, comparisons between the processors

Table 1 Characteristics of data

data file	size (MB)	nr. of elts.	max. depth	parse time (sec)
Mondial	1.17	24,184	5	0.8
WordNet	13.56	272,775	3	2.3
DMOZ structure	289.83	3,940,716	3	34.3
DMOZ content	974.38	13,233,278	3	91.3

Table 2 Queries used in experiments

query	Mondial	WordNet
no match	`//non-existing`	`//non-existing`
simple	`//province/city`	`//Noun/wordForm`
nested	`//*/*`	`//*/*`
past	`//country[province]/religions`	`//Noun[wordForm]/wordForm2`
future	`//country[province]/name`	`//Noun[wordFrom2]/wordForm`

query	DMOZ structure	DMOZ content
no match	`//non-existing`	`//non-existing`
simple	`//Topic/Title`	`//Topic/catid`
nested	`//*/*`	`//*/*`
past	`//Topic[editor]/newsGroup`	`//Topic[catid]/link`
future	`//Topic[editor]/Title`	`//Topic[link]/catid`

Table 3 Run times of the SPEX processor

data	**output**	**run times (in sec)**				
		no match	**simple**	**nested**	**past**	**future**
Mondial	yes	–	1.7	3.7	1.4	1.3
	no	1.1	1.2	1.3	1.3	1.3
WordNet	yes	–	7.3	20.3	7.2	7.6
	no	3.8	4.1	5.5	5.6	5.8
DMOZ structure	yes	–	73.9	304.9	68.5	73.8
	no	53.7	58.7	68.7	67.5	72.2
DMOZ content	yes	–	200.2	931.7	336.3	234.6
	no	168.3	184.4	215.1	216.3	234.1

Table 4 Run times of Saxon, Fxgrep, and Xalan

processor	**data**	**run time (in sec)**				
		no match	**simple**	**nested**	**past**	**future**
Saxon	Mondial	1.5	2.1	3.3	1.7	1.6
	WordNet	6.2	57.7	113.7	7.7	8.0
Fxgrep	Mondial	2.2	2.4	4.6	2.1	2.1
	WordNet	24.9	27.0	41.0	25.7	29.4
Xalan	Mondial	2.5	3.6	21.1	2.8	2.7
	WordNet	4.9	126.9	> 1500	6.3	6.3

Table 5 Space requirements of processors

processor	data	space requirement (in MB)				
		no match	simple	nested	past	future
SPEX	Mondial	9	9	10	9	9
	WordNet	9	9	9	9	9
	DMOZ structure	9	9	11	9	9
	DMOZ content	9	9	11	9	9
Saxon	Mondial	13	16	21	16	16
	WordNet	59	85	108	82	82
Fxgrep	Mondial	18	18	18	18	18
	WordNet	88	83	83	81	81
Xalan	Mondial	19	21	23	20	20
	WordNet	60	80	167	85	85

are performed only for the smaller Mondial and WordNet databases. The space requirements of the processors for evaluating the various queries against the data files are shown in Table 5.

The results illustrated in Figures 63 to 66 show that the SPEX prototype achieves a very competitive performance on the Mondial and WordNet databases. For some queries it even outperforms the other processors. Further, it has a consistent behaviour for different queries, unlike Saxon and Xalan that show very good processing times for the *non-matching*, *past*, and *future* queries on the WordNet file, but have a very bad performance for the *simple* and *nested* queries. Note also that this section only compares the total run times of the processors. There is another big advantage of the SPEX processor: the extremely short answer times. In all of the test cases SPEX produced output immediately, whereas the output of the other processors is delayed by the need for an in-memory representation of the input document.

A further comparison of the processors on larger files like the DMOZ files could not be performed as the time and memory consumption of Saxon, Xalan, and Fxgrep on the 300 MB DMOZ structure document proved to be beyond the limitations of the system used for the tests. The SPEX prototype, however, uses a constant amount of memory (between 9 and 11 MB, including the Java Virtual Machine, cf. Figures 65 and 66) for all of the given queries and documents.

Run times of queries are strongly influenced by the number of selected elements. However, this is not due to some additional processing, but rather due to the high costs of terminal output. Although output of all the processors was discarded rather than displayed on a terminal[†], it could be noted that especially the *nested* query, that creates a high number of elements as output, has an increased run time. Therefore, the SPEX processor was also tested without creating output[‡]. Table 3 shows the run times of the SPEX processor creating output and not creating output. For some queries (esp. *nested*) the difference in run times is dramatical, e.g. when creating output the *nested* query on the DMOZ structure file takes about five times as long compared to a processing where no output is created. Figure 67 shows a comparison for the *nested* query. Note that the axes of Figures 67, 68, and 69 are shown in a logarithmic scale. In contrast to the *nested* query where run times differ by a constant factor, there is almost no difference in run times for the *future* query shown in Figure 68. Figure 69 compares the run times of all the queries when no output is created. It suggests that run times are linear in the size of the input data.

[†]This was achieved by redirecting output to the special UNIX device `/dev/null`.

[‡]For this purpose results were processed as normal but they were not written to the Java stream `System.out`.

Figure 63 Comparison of run times for the Mondial database

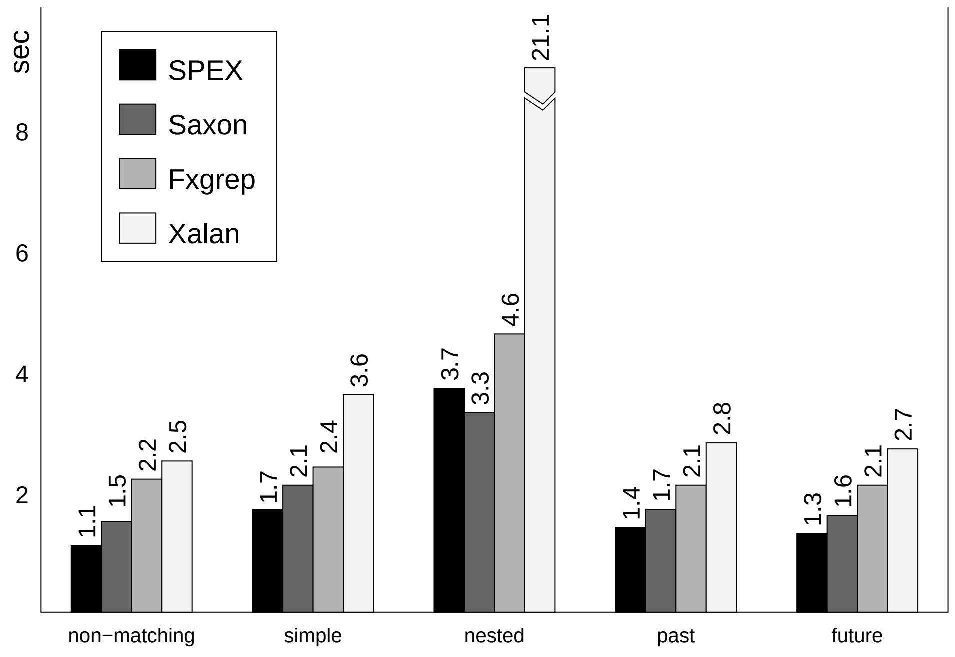


Figure 64 Comparison of run times for the WordNet database

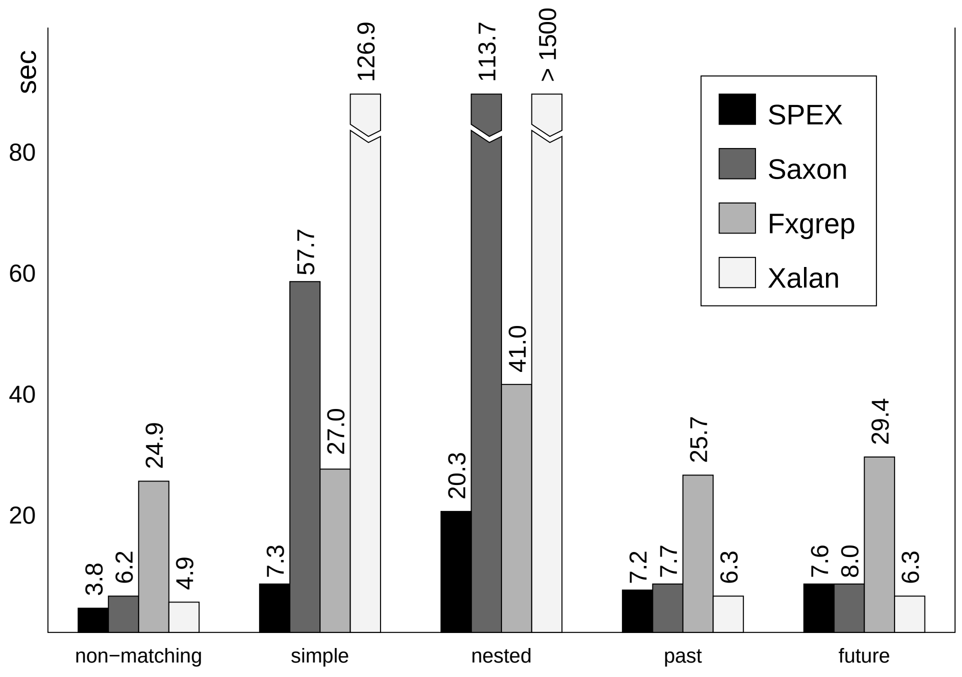


Figure 65 Comparison of space requirements for the Mondial database

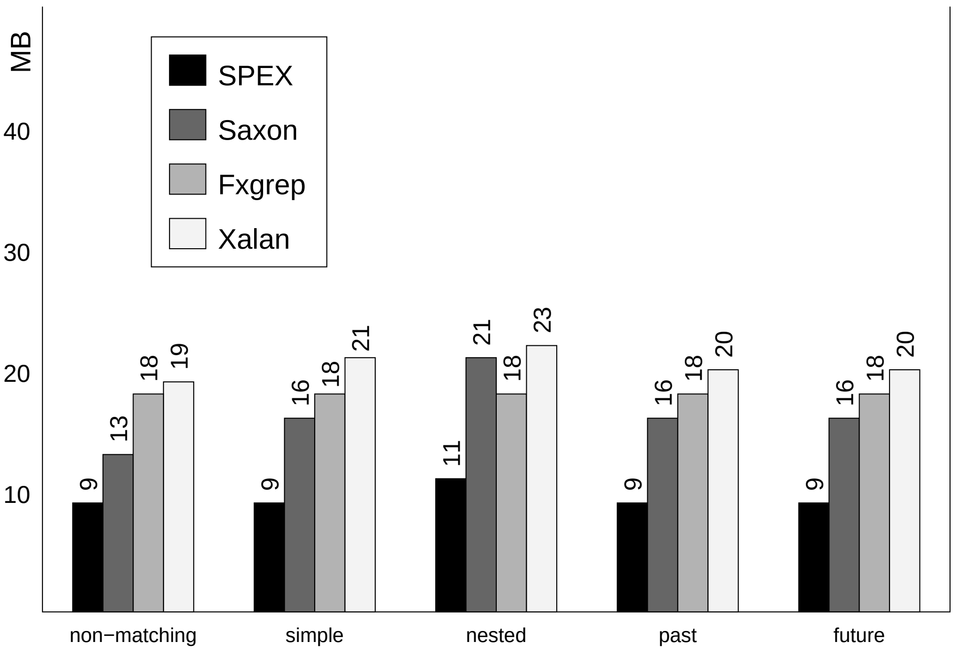


Figure 66 Comparison of space requirements for the WordNet database

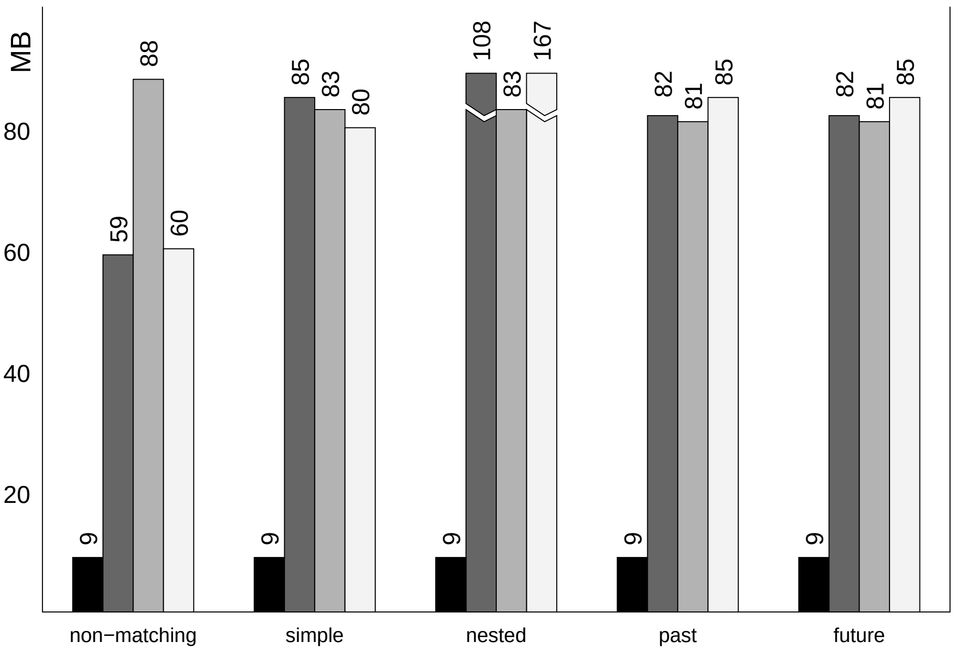


Figure 67 Comparison of SPEX run time with output vs. no output for query *nested*

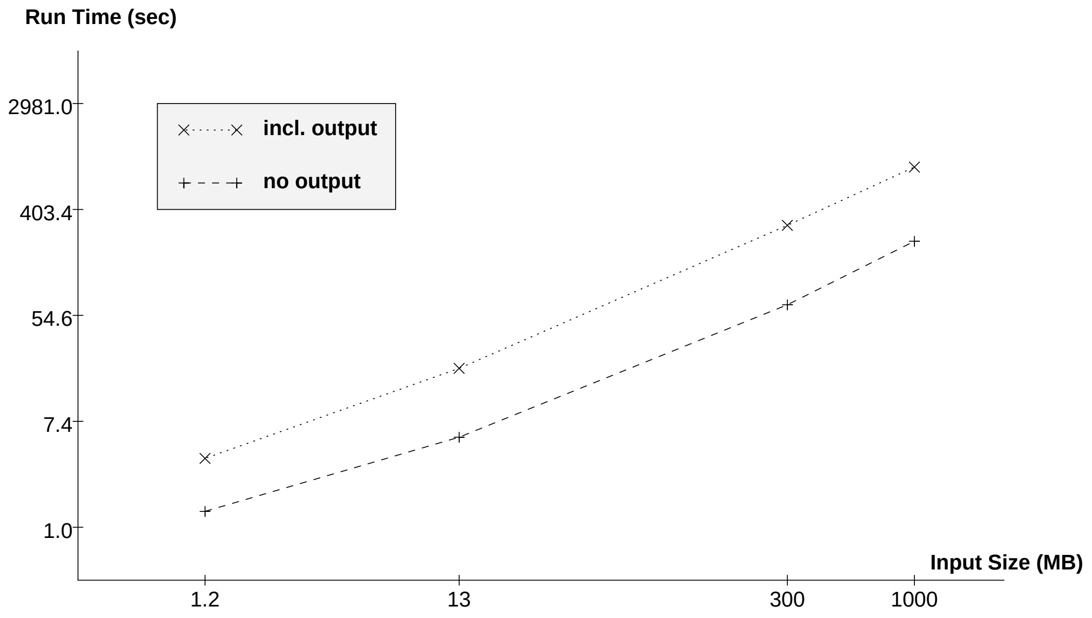


Figure 68 Comparison of SPEX run time with output vs. no output for query *future*

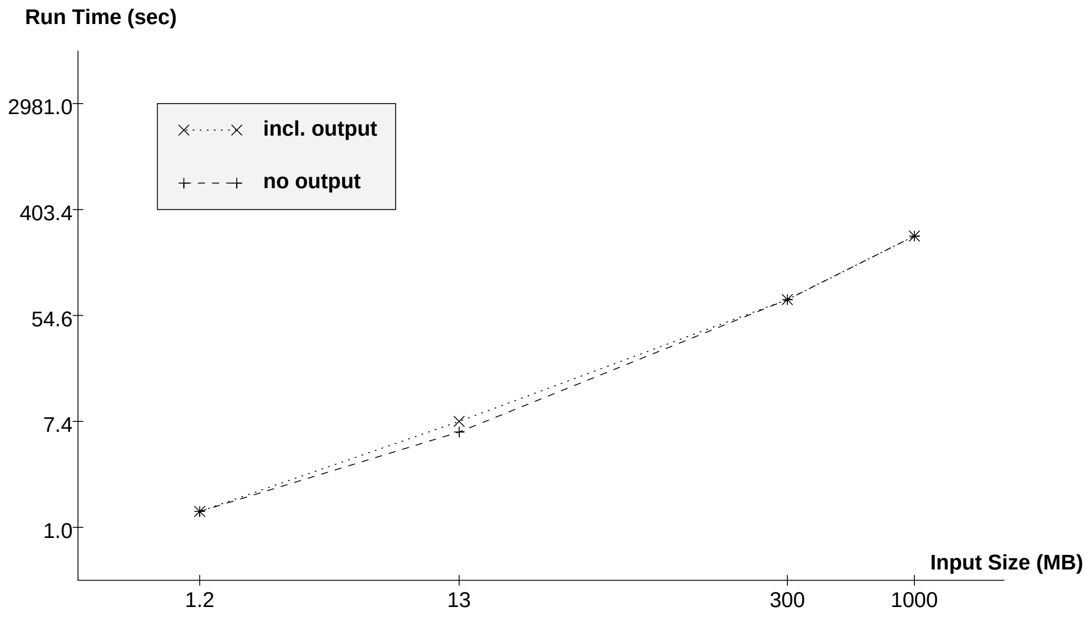
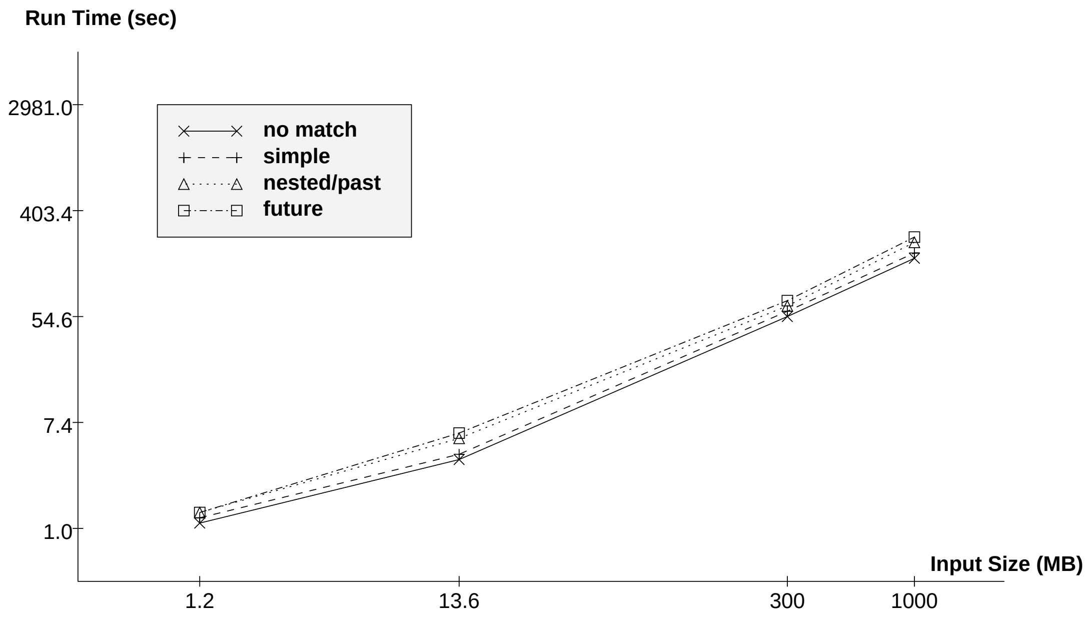


Figure 69 Comparison of queries for SPEX the processor (without output)



8 Related work

There are several other attempts towards the goal of querying XML streams. However, all of them support less powerful query languages like regular path expressions or very restricted subsets of XPath or XQuery.

The query operator X-Scan from the Tukwila data integration system [19] supports path expressions corresponding to `child` and `descendant` steps from XPath. These expressions are compiled into nondeterministic finite automata (NFA). A deterministic finite automaton (DFA) is constructed from the NFA by using the standard algorithm for creating a DFA out of an NFA. The drawback of this approach is that the constructed DFA has an exponential number of states compared to the initial NFA.

In X-Scan, a lot of the states created in the construction of the DFA may be unreachable, giving a great potential for optimization by simplification of the constructed DFA. Therefore an improved version [18] considers an evaluation model based on the on-demand creation of the DFA, called lazy DFA.

The XFilter engine [11] and its successor YFilter [16] are used in the context of selective dissemination of information system (SDI) as filters for deciding if a specific XPath expression matches a document. The processing model is based on a finite state machine, which uses a hashtable as central index structure. Keys for this hashtable are the nodetests of the location steps contained in the query. A location step matches if it is stored in the hashtable with a key that corresponds to the label of the actual element and the step also matches the current tree level. E.g. for a path `/descendant::a/child::b`, if the first step matches on tree level n, an entry is added to the hashtable with a key of `b` and a tree level of $n+1$. If some time later a `b` node is encountered, the current tree level is compared with the levels of the entries with key `b` in the hashtable. All of those entries where the levels are equal to the current level are considered to match.

XTrie [14] improves the model of XFilter by proposing an index structure for the path queries based on a trie. Note that these engines can treat only qualifiers referring to the data value, attributes and position of individual elements.

The XML Stream Machine (XSM) [20] is an approach to support a subset of XQuery for querying XML streams. Like the model presented in this thesis, the task of querying is performed by a network of transducers. In addition to location paths also variable bindings, concatenation of elements, and element construction are supported. However, the only supported axis is the `descendant` axis. Further, it restricts the type of data to be queried by only allowing input data that conforms to non-recursive DTDs[†]. Note that the combination of both restrictions, only supporting the `descendant` axis and not allowing nested elements of the same type, strongly simplifies the task of a query system. Particularly, it is not necessary to count the tree level here, as the end of a node can be found by just looking for the end tag of the node. This works, as there can be no nested node of the same type.

Although there is a demand in the XSLT community to provide support for progressive XSLT processing (sometimes called incremental processing), there are few achievements in this direction. The Xalan 2 XSLT processor [10] offers the possibility of incremental processing at the cost of an increased total processing time. However, it still constructs a complete in-memory representation of the input document. As far as possible, the query is performed in parallel to the tree construction, effectively giving a progressive approach.

[15] investigates a restriction of XPath called *Sequential XPath (SXPath)* that enables the query-

[†]A recursive DTD allows elements of a type t to (directly or indirectly) contain other elements of the same type.

ing of XML streams with an amount of memory linear in the depth of the input document. Therefore, SXPath only allows forward axes on location steps outside qualifiers and simple qualifiers that can be decided on immediately (e.g. checking of attributes).

A compilation of this thesis into a research paper can be found in [26]. In [25] the model presented here is slightly changed to support regular path expressions with qualifiers. There, a new type of step transducer, a *closure transducer* is introduced that supports the positive closure operator "+".

9 Conclusion

In this paper, an approach to a streamed and progressive evaluation of XPath expressions against XML streams is described, based upon a rewriting introduced in [27] of general XPath into forward XPath, i.e. XPath without reverse axes such as `ancestor`. Such an approach has the following advantages:

- It needs less memory than standard approaches that store the document tree in memory. This is beneficial for mobile devices with limited memory, for data-centric applications that handle large amounts of data, for continuous services, and for selective dissemination of information.
- Response times are much lower, as results are provided as soon as they are available. In contrast, an approach that creates an in-memory representation of the input before starting the query delays answering until the entire document has been received. This is especially undesirable if data is received over slow network links.
- Depending on the combination of query and input data (e.g. simple queries on large amounts of data) total processing time can be lower, as there is no overhead for setting up an in-memory representation of the input.
- For environments where exact answers are not required, it is possible to provide support for approximate query answering that might further reduce memory consumption, answer times, and total processing times.

The salient feature of the evaluation model is the processing of XPath expressions with qualifiers using communicating pushdown transducers.

Furthermore, without `following` steps in the query, the memory space needed by this method is quadratic in the depth of the tree of the streamed document. Only the last transducer in the system, the output transducer, has a worst case space requirement which is linear in the size of the input. The worst case is only encountered in special cases (e.g. the result consists of the whole document). If the query contains a step with axis `following` or `following-sibling`, space requirements increase. However, this seems to be inherent in the nature of the `following` step that can apply to all of the nodes of the input document in the worst case. The compilation of XPath expressions into transducer networks is linear in time and space.

Experiments with a prototype implementation demonstrate a remarkable efficiency: Applied to XML databases of different characteristics the prototype outperforms in most cases the query processors used for comparison. In all cases, the performance of the prototype is comparable to that of a standard, state-of-the-art XPath processor.

As future research the following topics have to be investigated:

- Migration to XQuery and/or XSLT: The processing model presented here can be extended to support XQuery and XSLT. An important point of investigation here is the support of variables.
- Multi-query optimization: The same transducer network can be used for processing several queries. The processing of common subparts of the different queries can be done by the same SPEX transducers.
- SPEX network reduction: It would be desirable to optimize a SPEX query system by reducing the corresponding SPEX network to a single transducer.

- Approximate query answering: As mentioned above, there are environments where exact answering of queries is not needed, and therefore the investigation of approximate query answering techniques would be valuable.
- Using Schemas/DTDs for optimization: The presented approach works without a priori knowledge of the structure and content of the input. If there is a knowledge about the input document before starting the query (e.g. by a Schema or DTD), the processing can be highly optimized.

References

[1] Astronomical Data Center. Home page. available at http://adc.gsfc.nasa.gov/.

[2] Cocoon 2.0: XML Publishing Framework. available at http://xml.apache.org/cocoon/index.html.

[3] dmoz – The Open Directory Project. located at http://dmoz.org/.

[4] IBM Developer Kit for Linux Version 1.3.0. available at http://www-106.ibm.com/developerworks/java/jdk/linux130/.

[5] Java Compiler Compiler (JavaCC) Version 1.1. available at http://www.webgain.com/products/java_cc/.

[6] Jikes Version 1.13. available at http://oss.software.ibm.com/developerworks/opensource/jikes/.

[7] Ælfred xml parser. available at http://www.opentext.com/services/content_management_services/xml_sgml_solutions.html#aelfred_and_sax.

[8] Saxon 6.5.2. available at http://saxon.sourceforge.net/.

[9] Xerces-Java XML Parser Version 1.4.4. available at http://xml.apache.org/xerces-j/index.html.

[10] Xalan-Java Version 2.4 D1, Apache Project. available at http://xml.apache.org/xalan-j/index.html.

[11] M. Altinel and M. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. of 26th Conference on Very Large Databases (VLDB)*, 2000.

[12] Francois Bry and Peer Kroeger. A Molecular Biology Database Digest. Technical report, 2001.

[13] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams: A New Class of Data Management Applications. Technical Report CS-02-04, Brown Computer Science, February 2002.

[14] C. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proc. of International Conference on Data Engineering (ICDE)*, 2002.

[15] A. Desai. Introduction to Sequential XPath. In *Proc. of IDEAlliance XML Conference*, 2001.

[16] Y. Diao, P. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and Scalable Filtering of XML Documents. In *Proc. of the 18th International Conference on Data Engineering*, 2002.

[17] C. Fellbaum, editor. *WordNet – An Electronic Lexical Database*. MIT Press, 1998. available at http://www.cogsci.princeton.edu/~wn/.

[18] T. J. Green, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata and Stream Indexes. Technical report, University of Washington, 2001.

[19] A. Levy, Z. Ives, and D. Weld. Efficient Evaluation of Regular Path Expressions on Streaming XML Data. Technical report, University of Washington, 2000.

[20] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proc. of 28th Conference on Very Large Databases (VLDB)*, 2002.

[21] W. May. Information Extraction and Integration with FLORID: The MONDIAL Case Study. Technical Report 131, University of Freiburg, Institut for Computer Science, 1999. available at http://www.informatik.uni-freiburg.de/~may/Mondial/.

[22] S. McGrath. XPipe. available at http://xpipe.sourceforge.net/.

[23] D. Megginson. SAX: The Simple API for XML, 1998. available at http://www.saxproject.org/.

[24] A. Neumann. Fxgrep: The Functional XML Querying Tool, March 2000. available at http://informatik.uni-trier.de/~neumann/Fxgrep.

[25] D. Olteanu, T. Kiesling, and F. Bry. An Evaluation of Regular Path Expressions with Qualifiers against Data Streams. submitted for publication, 2002.

[26] D. Olteanu, T. Kiesling, F. Bry, and T. Furche. Streamed and Progressive Evaluation of XPath based on Pushdown Transducers. submitted for publication, 2002.

[27] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Workshop on XML-Based Data Management (XMLDM) at EDBT*, 2002.

[28] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[29] W3C. XML Path Language (XPath) Version 1.0. W3C Recommendation, 1999.

[30] W3C. Document Object Model (DOM) Level 2 Core Specification. W3C Recommendation, 2000.

[31] W3C. XQuery 1.0: An XML query language. W3C Working Draft, 2002.

[32] W3C. XSL Transformations (XSLT) Version 2.0. W3C Working Draft, 2002.