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

Approximate Streamed Evaluation of XPath under Memory Constraints

Dominik Schwald

Projektarbeit

Abgabe der Arbeit: 31.07.2003
Betreuer: Prof. Dr. François Bry
Dipl. Ing. Dan Olteanu

Abstract

This paper presents techniques for approximate evaluation of regular path queries against XML streams.

Querying unbounded XML streams enforces new demands to query evaluators: As shown in [2], the memory needed for answering queries on unbounded XML streams is unbounded as well. Since this is not acceptable for real world applications, an approximate evaluation is thus required.

Preliminary research showed that it is necessary to have several different approximation techniques for ensuring 'good' query answers under memory constraints, because the definition of a 'good' query answer, depends heavily on the application purpose. E.g. some applications can not accept false positives, but false negatives.

The contributions of this paper consists in:
(1) Motivating the need for application-sensitive approximate query answering by showing two application scenarios where different approximations are needed.
(2) Introducing several approximation techniques and the mean to compose them via a language for approximation techniques called SAL (SPEX Approximation Language).
SAL enhances the query evaluator SPEX[12] with the facility of approximate answers to regular path queries on XML streams. Real world application scenarios help to explain the topic throughout the whole paper.

Zusammenfassung

Dieser Artikel präsentiert Techniken für die genäherte Auswertung regulärer Pfad Anfragen auf XML Datenströme.

Anfragen auf unendliche XML Datenströme stellen neue Anforderungen an Anfrageauswerter: Die Anfrage auf einen unendlichen XML Datenstrom benötigt auch unendlich viel Speicher [2].

Frühere Forschung zeigt, dass es nötig ist mehrere unterschiedliche Techniken der genäherten Anfrageauswertung zu haben, um trotz beschränkten Speichers 'gute' Antworten zu erhalten. Dies liegt daran, dass die Definition einer 'guten' Antwort stark von dem Ziel der verwendeten Anwendung abhängt. Beispielsweise können manche Anwendungen Fehler erster Art nicht tolerieren, sondern nur Fehler zweiter Art.

Der Gegenstand dieses Artikels ist:
(1) Die Motivation der Notwendigkeit anwendungsabhängiger genäherter Anfragen. Dies wird anhand zweier Anwendungenscenarien erklärt, in denen unterschiedliche Näherungsverfahren nötig sind.
(2) Die Vorstellung verschiedener Techniken für genäherte Anfrageauswertung, sowie die Möglichkeit diese Techniken mittels einer Sprache für Näherungsverfahren zu kombinieren. Diese Sprache wird SAL (SPEX Approximation Language) genannt.

SAL erweitert den Anfrageauswerter SPEX [12] um die Möglichkeit der genäherten Anfrageauswertung von regulären Pfad Anfragen auf einen XML Datenstrom. Echte Anwendungsscenarien helfen über den ganzen Artikel hinweg den Sachverhalt zu erklären.

Contents

1 Introduction

This paper presents techniques for approximate evaluation of regular path queries against XML streams.

Querying data streams is a field of growing importance and several applications provide information via a XML stream, like stock exchange or meteorology data. Since steams are unbounded, this enforces new demands to query evaluators: As shown in [2] the memory needed for answering queries on unbounded XML streams is unbounded as well. This is caused, because stream fragments need to be stored in memory until one knows if they are a answer to the query or not. Since unbounded memory usage is not acceptable for real world applications, an approximate evaluation is thus required. The goal of the approximate evaluation is to enforce memory constraints.

In preliminary research we tried to find the best general approximation technique for all application scenarios, but found out, that it is necessary to have several different approximation techniques. The reason for the existence of several approximation techniques and for enabling their composition resides in the simple fact that there is no general application-independent measure for the quality of an approximation. This result resembles subtle the result of *No free lunch theorem*[14] regarding the employment of heuristics for solving optimization problems: if an heuristic is fine tuned for a specific optimization problem, then it can not provide results of a similar quality for another optimization problem.

The contributions of this paper consists in:
(1) Motivating the need for application-sensitive approximate query answering by showing two application scenarios where different approximations are needed.
(2) Introducing several approximation techniques and the mean to compose them via a language for approximation techniques called SAL (SPEX Approximation Language).
SAL enhances the query evaluator SPEX[12] with the facility of approximate answers to regular path queries on XML streams. The data model of SAL models the input data, that is potential answers of the query evaluator, their characteristics and connections. SAL resembles the XPath language with data model, composability and evaluation strategy.

Real world application scenarios help to explain the topic throughout the whole paper.

Related work
In [13] Divesh Srivastava proposes approximate phrase matching in XML. This is done in order to improve flexibility like allowing users to specify

which tags and annotations to ignore when matching a phrase. Exact and approximate answers are premitted and returned as ranked results. This kind of "approximation" differs a lot from the way approximation is used in this paper (as detailed in Section 2.2).

The STREAM (Stanford Strream Data Manager) Project [8] queries flat-approximation (tuple data) streams. Approximation is used in this project in order to handle *blocking operators* like joins. This is done by a sliding window.

AURORA [1] is a DBMS for monitoring purposes. Input and output are streams. This project gives approximate answers to ensure real time processing: If ressources are low, aurora does *loadshedding*, i.e. discards new incoming data until the system ressources are avaliable again.

Overview

This paper is structured as follows: Section 2 motivates for the topic of approximate query evaluation on XML streams. In Section 3, two application scenarios are introduced, showing different approximation techniques at work in some 'real world' applications. Section 4 describes and explains the desired characteristics of a language for approximation techniques. In Section 5 a closer look is taken to the SPEX query evaluator, a query evaluator for XML streams. Section 6 then introduces an approximation language for the query evaluator SPEX, called SAL: SPEX Approximation Language.

2 Motivation

2.1 Why querying XML-Streams?

A XML stream is the depth-first, left-to-right traversal of a (possible unbounded) XML document ([4]).

Nowadays XML is the de facto standard format for data exchange on the web. Therefore XML data streams can be found in several applications, like real-time measurements and continuous services which select information, e.g. stock exchange or meteorology data and highly structured streams containing meta data about video streams (MPEG-7 [10]).

2.2 What means 'approximate evaluation'?

When talking about approximation it is very important to know *why* approximation is used, in order to understand what is meant by approximation.

In current research [13] Divesh Srivastava uses approximate for flexible querying: If the query structure does not exactly match the data structure, then one of these structures can be relaxed in order to still retrieve some answers. Based on the relaxation degree, the answers retrieved are ranked. I.e. approximation is used here, by modifying the data or the query, to sharpen the matching of data structure and query structure.

This kind of approximate evaluation differs from the kind of approximation that is used in the present paper: Approximate evaluation is employed here, for enforcing memory constraints. We consider one query and one data, but the result for the query might be different from the correct result in such a way, that some results might miss, and/or some might not be answers to the query. As long as no memory limits are hit, exact answers to the query are returned.

2.3 Why querying under memory constraints?

Querying under memory constraints is unavoidable, because real applications do have real (hardware given) memory constraints. In order to avoid that the query evaluator crashes due to 'out-of-memory' errors, memory constraints are a good idea. Finally it enables query evaluation on small devices (mobile, PDA, . . .) with very limited memory.

2.4 Why using approximation?

Because of Section 2.3 it is not always possible to give exact answers to a XML query, but approximate answers. Using approximation makes sense in a lot of applications, because it allows to query unbounded streams without knowing something about the stream or the query. Also it is often better to get some answers to the query, that just nothing. If an approximation technique, that fits to the used application is used, than it is possible to get 'good' query answers despite memory constraints.

2.5 Why is it necessary to have several approximation techniques?

When approximation is applied, one tries to
(1) avoid cases where more than the desired amount of memory is allocated.
(2) get 'good' query answers, in contrast to just discarding new data blindly when no more memory is available.
Therefore, one tries to tune approximation techniques to the application purpose.

The reason for the existence of several approximation techniques and for enabling their composition resides in the simple fact that there is no general application-independent measure for the quality of an approximation. A very important point is the question what the meaning of a result to the query is: For some applications false positives are acceptable (e.g. querying a movie database for action films, getting a love-movie would not harm too much). Whereas for other applications this is not (e.g. an e-mail spam filter that automatically deletes an important e-mail because it was (wrongly) detected classified as spam).

This result was drawn from our preliminary research in the attempt to find the best approximation method and resembles subtle the result of *No free lunch theorem*[14] regarding the employment of heuristics for solving optimization problems: if an heuristic is fine tuned for a specific optimization problem, then it can not provide results of a similar quality for another optimization problem.

3 Application scenarios

This section introduces two application scenarios that are used throughout the paper. An application scenario is an example of a real world application that queries a XML stream. Several examples will show the benefits of approximate answers and introduce different approximation techniques.

First the application scenario is described in general, followed by the data model, given in DTD [5]. Then examples motivate why approximation is needed and how approximation could be applied. For this purpose, each example introduces a XPath [6] query, since XPath is (together with XQuery [3]) the language of choice for querying XML.

3.1 Power Plant

The first application scenario, power plant (PP), describes a stream based logging system in a power plant. The power plant is controlled by a lot of computers, human interaction is not necessary, but used in order to improve security.

The XML stream is used to model information about the things that happen currently at the PP. Viz, every time a controlling unit interacts with the power plant or gets new data, this information is encoded in the XML stream. Also every human interaction is logged in the XML stream of the PP. If a warning occurs somewhere, this is also stored in the XML stream.

```
DTD:

<!ELEMENT powerplant (info?,warning?)*>
<!ELEMENT info       (controller|human)>
<!ELEMENT controller (temp?,volum?,text?)>
<!ATTLIST controller id CDATA #REQUIRED>
<!ELEMENT temp       (#PCDATA)>
<!ELEMENT volum      (#PCDATA)>
<!ELEMENT text       (#PCDATA)>
<!ELEMENT human      (status|warning)>
<!ATTLIST human      id CDATA #REQUIRED>
<!ELEMENT status     (#PCDATA)>
<!ELEMENT warning    (controller*)>

Example of a possible input stream fragment:

<powerplant>
```

```
[...]
  <info>
    <controller id='19'>
      <temp>179</temp>
      <volum>422</volum>
    </controller>
  </info>
  <info>
    <human id='41'>
      <status>normal</status>
    </human>
  </info>
  <warning>
    <controller id='42'>
      <text>cooling water low</text>
    </controller>
  </warning>
[...]
```

3.1.1 Example: Retrieve last information before the occurrence of a warning

In order to analyze warnings and improve the system, the information received just before the occurrence of a warning is of high interest.

To get all information received just before a warning occurred, the following query can be used:
Select all `info` *elements, if they are followed by a* `warning` *element on the same level.*
In XPath this could be expressed:

```
/descendant::info[following-sibling::warning]
```

Why approximation is needed

In this example, approximation is needed, e.g. to limit the result of the XPath query to the latest `n info` elements.[1] Without approximation, all `info` elements with a following-sibling `warning` element would be a result to the XPath query.

[1]This could also be done with XPath:
`/descendant::info[following-sibling::warning][position() > last()-n]`
Anyway using approximation here could be seen an implementation of the XPath selection of the last `n info` elements.

How approximation can be applied

In this scenario, approximation is used to realize a *sliding window* over the incoming XML stream. The size of the sliding window is limited by `n info` elements. Every time a new `info` element occurs in the stream, it is stored in memory. If the number of stored `info` elements would reach `n+1`, the oldest element is discarded, so that the number of stored `info` elements will always remain `n`. (Of course the number is zero when the system starts and then can grows until maximum `n` while querying the stream.)
Remark: Querying for the first `n info` elements after a warning, would be a *fixed window* over the stream.
In XPath, this could be expressed:

```
/descendant::warning/following-sibling::info[position() <= n]
```

3.1.2 Modified Example: Retrieve last information before the occurrence of a warning with a maximum memory of `n` characters

While the last example could also be solved using only XPath, already a small modification will change this and prove the exigence of approximation.

Instead of being interested in the last `n info` elements, now we only allow the last `n` characters to be stored. This has some advantages compared with the above example:
(1) Not the number of `info` elements decides how much can be stored, but the available space. If possible, this example will return more answers.
(2) This is an exact memory constraint. While the last `n info` elements does not say anything about the used memory, the number of maximum used characters is a real memory bound.

In order to get this data from the XML stream we could use the same query as above:
Select all `info` *elements, if they are followed by a* `warning` *element on the same level.*
In XPath this could be expressed:

```
/descendant::info[following-sibling::warning]
```

Why approximation is needed

In this example, approximation is needed, to limit the result of the XPath query, not to use more than `n` characters Without approximation, all `info` elements with a following-sibling `warning` element would be a result to the XPath query.

Evaluating the XPath query on an unbounded stream leads into storing the whole stream in memory. Since real machines have real (hardware given) memory limits, approximation necessary is order to avoid *out-of-memory* errors.

How approximation can be applied

In this scenario, approximation is used to realize a *sliding window*. The size of the sliding window is limited to `n` characters. As soon as the character limit is reached, the oldest `info` element is discarded, so that the stored `info` elements will always have a a maximum memory usage of `n` characters.

3.2 Unix Process Status

This application scenario describes a process status monitoring application. A computer generates a XML stream containing information of the currently running processes on this machine. This is done by parsing the output out of the unix `ps` command (detailed in [7]), that provides information of the currently running processes. Since a process can invoke one or more child processes, that itself can again invoke child processes etc., the XML-Stream is highly structured and reflects the parent-child relationship of the processes. The data contains the processID, the user and several other important information about the process.

```
DTD:

<!ELEMENT processChain (process)*>
<!ELEMENT process      (command, args?, flag, state, user,
                        processID, parentID, c, priority,
                        nice, address, size, wchan, start,
                        terminal, time, process*) >
<!ELEMENT command      (#PCDATA) >
<!ELEMENT args         (arg+)    >
<!ELEMENT arg          (#PCDATA) >
<!ELEMENT flag         (#PCDATA) >
<!ELEMENT state        (#PCDATA) >
<!ELEMENT user         (#PCDATA) >
<!ELEMENT processID    (#PCDATA) >
<!ELEMENT parentID     (#PCDATA) >
<!ELEMENT c            (#PCDATA) >
<!ELEMENT priority     (#PCDATA) >
<!ELEMENT nice         (#PCDATA) >
<!ELEMENT address      (#PCDATA) >
<!ELEMENT size         (#PCDATA) >
<!ELEMENT wchan        (#PCDATA) >
<!ELEMENT start        (#PCDATA) >
<!ELEMENT terminal     (#PCDATA) >
<!ELEMENT time         (#PCDATA) >


Example of a possible input stream fragment:

<processChain>
  <process>
```

```
<command>init</command>
<args>
  <arg>[5]</arg>
</args>
<flag>004</flag>
<state>S</state>
<user>root</user>
<processID>1</processID>
<parentID>0</parentID>
<c>0</c>
<priority>69</priority>
<nice>0</nice>
<address>-</address>
<size>112</size>
<wchan>do_sel</wchan>
<start>Jul13</start>
<terminal>?</terminal>
<time>00:00:04</time>
<process>
  <command>[keventd]</command>
  <flag>002</flag>
  <state>S</state>
  <user>root</user>
  <processID>2</processID>
  <parentID>1</parentID>
  <c>0</c>
  <priority>69</priority>
  <nice>0</nice>
  <address>-</address>
  <size>0</size>
  <wchan>contex</wchan>
  <start>Jul13</start>
  <terminal>?</terminal>
  <time>00:00:02</time>
</process>
[...]
```

3.2.1 Example: Retrieve processes that invoked certain child processes

In order to use this XML stream for analyzing the running system, it could be of interest what processes invoked child processes consuming more that e.g. 10000 bytes of memory. This could be reached with the XPath query: *Select all process, that have a descendant process, having a child size (the consumed memory) greater than 10000 (bytes).*

```
descendant::process[descendant::process[child::size > 10000]]
```

Why approximation is needed

Since the evaluation of this query could need more memory than avaliable it is necessary to employ some approximation techniques for this example.

How approximation can be applied

A reasonable application technique would be to output the newest outer nested process (new in matters of the appearance in the XML-input stream, not of the time the process is running) when memory is needed. This approximation technique would free space on the one hand and return a 'good' query answer. This is because of three things:
(1) Potential answers are output, rather than discarded, so that in the worst case, a potential answer is output, although this process did not invoke any child processes consuming more than 10000 bytes. Therefore it is not possible to overlook any process, that really invoked a child process consuming more than 10000 bytes.
(2) It is reasonable to choose an outer-most process, because the more 'out' a process is, the more child processes it can contain.
(3) To chose the the newest outer-most process instead of the oldest (new and old in matters of the appearance in the XML-input stream) is because of the fact, that an older process already had more time where a potential child process consuming more than 10000 bytes could occur, than a newer one.

4 Desired Characteristics

Based on real application scenarios described in Section 3 and characteristics of a general-purpose query language [9], this section describes and explains the desired characteristics of a language for approximation techniques (AL) for ensures limited memory usage of a regular path query evaluator (QE) against XML streams.

Based on these characteristics, in the next section we introduce a new AL for the SPEX QE, called SAL (SPEX Approximation Language).

4.1 Provide Primitive Constructs (Approximation Techniques for Query Answering)

An AL should provide primitive constructs that explore the available input informations from the QE during processing. The available information from the QE is at least in from of potential answers and their relationships.

A potential answer is a (not necessary well-formed) fragment of the input XML stream, such that it is still undecided whether this fragment is part of the answer or not.

4.2 Precise Construct Semantics

An AL should have some formalization of its semantics. The formalization needs to be sufficient to support reasoning about AL, such as determining result structure and explaining how the language works.

The need for a clean and clear semantics is best met by
(i) producing a model for the input data (potential answers, as computed by the QE) and
(ii) by defining how a complex construct is composed out of several primitive constructs.

4.3 Compositional Semantics

AL constructs should have referential transparency. That is, the meaning of an AL construct should be the same wherever it appears. Furthermore, constructs with equal result types should be allowed to appear in the same contexts.

4.4 Free Memory when needed by the Query Evaluator

The QE should employ an AL primitive (or complex) construct for deallocating memory when there is a need for free avaliable memory.

The effect of employing (evaluating) such an AL construct is in discarding/outputting potential answers of the QE and freeing their corresponding memory.

4.5 Time and Space Complexity

The evaluation of an AL construct should free memory for the QE without using itself too much memory. It does not make sense to use approximation techniques that need exponential memory, while the initial goal is to free memory, because no more is avaliable.
It is assumed that the evaluation of AL constructs should be done efficiently, i.e. in polynomial time and space in the size of the input, as provided by the QE.

5 The SPEX Query Evaluator

This section introduces the QE SPEX, a streamed and progressive evaluator of XPath against XML streams. It is necessary to know some basics how query evaluation is done with SPEX in order to understand how approximation can be applied (this will be detailed in Section 6).

The general goals for QEs are:

- Avoid storing the XML stream in memory, as it can be too large or volatile,
- access each XML stream item at most once, for efficiency reasons,
- deliver results on-the-fly, so as not to delay further result processing.

All these goals are achieved by SPEX.

5.1 Terms and Definitions

Definition 1 (Potential Answer). *A potential answer (PA) is a (not necessary well formed) fragment of the XML stream.*

Definition 2 (Answer Condition). *An answer condition (AC) is a three state variable. Possible states are `undetermined`, `true`, `false`.*

For each XPath predicate ACs are created during processing. They convey the information whether a predicate fulfilled or not for a particular instance of the query. Every PA depends on ACs: Initially every AC is `undetermined`. When an AC becomes `true`, this means that a predicate of the XPath query was fulfilled. If an AC becomes `false`, this means that a predicate of the XPath query was not fulfilled.

5.2 Example: SPEX at work

In this example it will be explained how query evaluation is done with SPEX. Consider given a XML stream and a XPath query. The XPath query will be evaluated by SPEX and on several breakpoints of the XML stream, a snapshot of SPEX's current evaluation status will be shown.

5.2.1 The XPath Query

The XPath query for the example is:

```
/descendant::a[descendant::b[following-sibling::c]/descendant::d]/descendant::c
```

Select all `c` *elements that are descendants of an* `a` *element, that itself has a descendant* `b` *element, having a following-sibling* `d` *element and a descendant* `d` *element.*

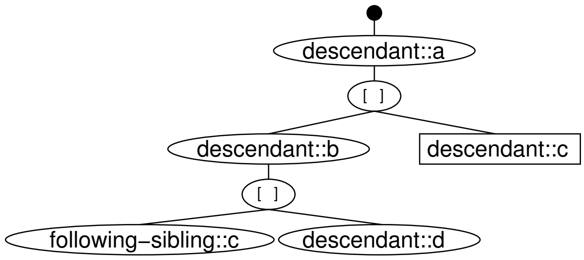

Figure 1: Evaluation plan for the XPath query

Figure 1 shows the evaluation plan for this XPath query. The symbols used in Figure 1 have the following meaning:
An ellipse represents a step of the XPath query. Inside the ellipse, the path the node should belong to is given. The path is understood relative to the parent node (or the root node, if there is no parent node).
A rectangle represents a node, but a special one: These nodes are answers to the XPath query.
The ellipses with `[ ]` inside represent a predicate of the XPath query.

5.2.2 The XML stream

In the following, we consider the XML document fragment:

<a_1>
<c_1>
<a_2>
<b_1>
<d_1>
</d_1>
<c_2> [...]
</c_2>
</b_1>
<b_2>

<d$_2$> [...]
</d$_2$>
</b$_2$>
<a$_3$>
<b$_3$>
</b$_3$>
<c$_3$>
</c$_3$>
</a$_3$>
<c$_4$>
[...]

Figure 2 shows the XML stream in a tree view. The tree view is very useful, as one can see immediately the parent-child and sibling relations.

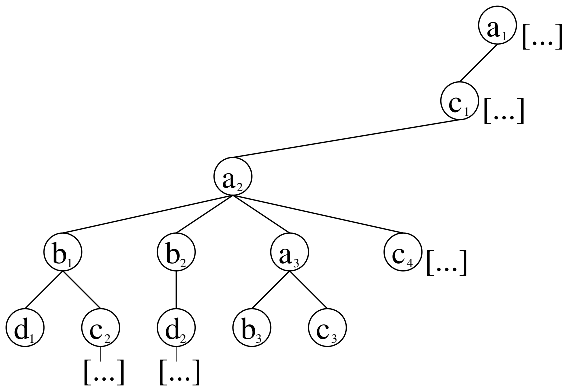

Figure 2: Tree view of the XML stream

The XML stream corresponds to a preorder, depth-first, left-to-right traversal of the above tree. In order to allow easier distinction of the XML tags, every tag has been marked with an additional number. E.g. <a$_1$> is a normal <a> tag, the number is only for ensuring, that one knows immediately what tag is being looked at. The corresponding XML stream is:

<a$_1$> <c$_1$> <a$_2$> <b$_1$> <d$_1$> </d$_1$> <c$_2$> [...] </c$_2$> </b$_1$> <b$_2$> <d$_2$> [...] </d$_2$> </b$_2$> <a$_3$> <b$_3$> </b$_3$> <c$_3$> </c$_3$> </a$_3$> <c$_4$> [...]

Splitting the XML stream

In order to see the query evaluation step-by-step, the evaluation is shown here in several steps. After the evaluation on a XML stream fragment, a snapshot of the current status (the PA-AC graph structure) of SPEX will be shown and explained.

5.2.3 Step 1 - The opening tag of c_1

Since this is the first XML stream fragment considered, no earlier evaluation was done by SPEX, i.e. there are no PAs or ACs stored in memory when starting the evaluation process. The first XML stream fragment considered is:
`<a1> <c1>`

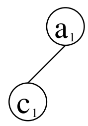

Figure 3: Tree view of the first XML stream fragment

Figure 3 shows the first fragment of the XML stream. Figure 4 shows the PAs and ACs after the evaluation.

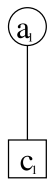

Figure 4: PA-AC graph structure in SPEX after Step 1

Explanation of Figure 4

The squares are the PAs: The XML fragments of all PAs are stored in memory.
The circles represent ACs: Only the states of ACs are stored in memory (`undetermined, true, false`), nothing more. For convenience we write as AC the element for which the predicate instance is considered. The fill color of the circle representing an AC shows its state: White for undetermined

ACs, black for ACs that are true, and gray if the AC is partly fulfilled (if an AC is false, it disappears, so there is no special fill color for this state).

While evaluating the query the following things happen:
(1) An a element (<a_1>) was encountered. This lead to the creation of the AC a_1. This AC still is `undetermined`, since it depends on the fulfillment of a predicate of the XPath query:
`[descendant::b[following-sibling::c]/descendant::d]`
(2) A `c` element (<c_1>) was encountered. This lead to the creation of the PA $\boxed{c_1}$, because this `c` element is a descendant of an `a` element. Since the PA $\boxed{c_1}$ depends on the fulfillment of the AC a_1, it is connected to this AC. The XML subtree rooted by $\mathtt{c}_1$ is stored in memory until the occurrence of its closing tag </c_1>.

The PA-AC graph structure shows only the root element of a PA, e.g. $\boxed{\mathrm{c}_1}$.

5.2.4 Step 2 - The opening tag of $\mathtt{b}_3$

Figure 5 gives the tree view of the second XML stream fragment considered, while Figure 6 shows the current evaluation status. The XML stream was processed until (including) the opening tag of $\mathtt{b}_3$:
<a$_1$> <c$_1$> <a$_2$> <b$_1$> <d$_1$> </d$_1$> <c$_2$> [...] </c$_2$> </b$_1$> <b$_2$> <d$_2$> [...] </d$_2$> </b$_2$> <a$_3$> <b$_3$>

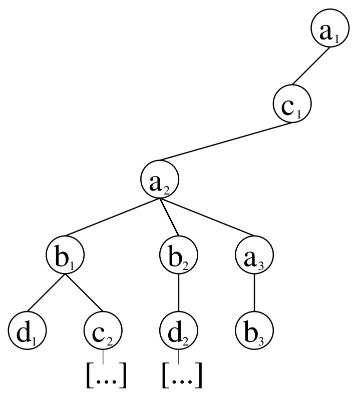

Figure 5: Tree view of the XML stream until the opening tag of b_3

Explanation of Figure 6

Now we have two PAs: $\boxed{c_1}$ and $\boxed{c_2}$. The PA $\boxed{c_1}$ depends on the AC a_1, which itself needs that *at least* one of the AC b_1 *or* b_2 *or* b_3 to become true.

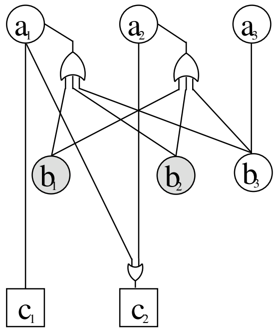

Figure 6: PA-AC graph structure in SPEX after the evaluation of Step 2

`OR` connections are modeled by an `or` gate in all figures.
The PA $\boxed{c_2}$ depends on AC a_1 *or* on the AC a_2. AC a_2 depends like a_1 on the fulfillment of *either* b_1 *or* b_2 *or* b_3. Finally, there is an AC, that is not yet connected to a PA: a_3.

The PAs b_1 and b_2 are both partly fulfilled: A `d` descendant of each `b` has been already encountered. They both still need a following-sibling c to be completely fulfilled. A partly fulfilled AC has a gray background.

The current status of the PAs is the following: As soon as AC a_1 becomes `true`, both PAs $\boxed{c_1}$ and $\boxed{c_2}$ are answers to the query. If AC a_2 becomes `true`, only PA $\boxed{c_2}$ is an answer to the query.
The PA $\boxed{c_2}$ depends on a_1 or a_2. In the figure this is modeled by an `or` gate.

5.2.5 Step 3 - The closing tag of $\mathtt{b}_3$

In this breakpoint, the only new thing is, that the element b_3 was closed. The impact of this to the PA-AC graph structure is shown in Figure 7. The tree view of the XML stream is the same as in Figure 5, because the only new added tag is the closing tag of $\mathtt{b}_3$:
<a1> <c1> <a2> <b1> <d1> </d1> <c2> [...] </c2> </b1> <b2> <d2> [...] </d2> </b2> <a3> <b3> </b3>

Explanation of Figure 7

Because $\mathtt{b}_3$ was closed, it is sure, that the AC b_3 will not fulfill (b_3 has no `descendant d`). Therefore the AC b_3 is set to false and thus disappeared from Figure 7. The AC a_3 is remains undetermined, because there can come other `b` and `c` elements, that can create new ACs and PAs, that are connected to AC a_3.

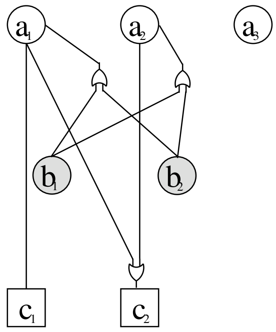

Figure 7: PA-AC graph structure in SPEX after the evaluation of Step 3

5.2.6 Step 4 - The opening tag of c_3

Now the XML stream was processed until the opening tag of c_3. Figure 8 gives the tree view of this XML stream, while Figure 9 shows the current evaluation status. The XML stream is:
<a$_1$> <c$_1$> <a$_2$> <b$_1$> <d$_1$> </d$_1$> <c$_2$> [...] </c$_2$> </b$_1$> <b$_2$> <d$_2$> [...] </d$_2$> </b$_2$> <a$_3$> <b$_3$> </b$_3$> <c$_3$>

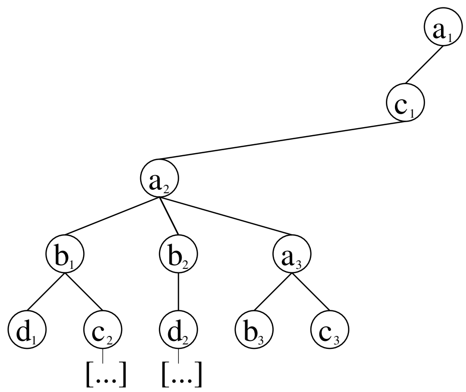

Figure 8: Tree view of the XML stream until the opening tag of c_3

Explanation of Figure 9

With the opening tag of c_3, a new PA $\boxed{c_3}$ is created. This PA depends on the fulfillment of either a_1 *or* a_2 *or* a_3.

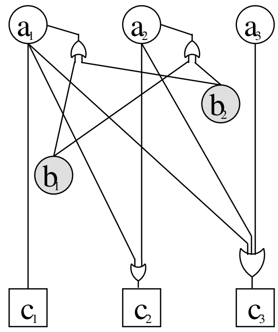

Figure 9: PA-AC graph structure in SPEX after the evaluation of Step 4

5.2.7 Step 5 - The closing tag of $\mathtt{a}_3$

Now the XML stream was processed until the closing tag of $\mathtt{a}_3$. This does not change the tree view, so Figure 8 still gives the tree view of this XML stream. Figure 10 shows the current evaluation status. The XML stream is: <a$_1$> <c$_1$> <a$_2$> <b$_1$> <d$_1$> </d$_1$> <c$_2$> [...] </c$_2$> </b$_1$> <b$_2$> <d$_2$> [...] </d$_2$> </b$_2$> <a$_3$> <b$_3$> </b$_3$> <c$_3$> </c$_3$> </a$_3$>

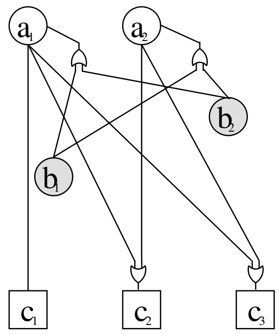

Figure 10: PA-AC graph structure in SPEX after the evaluation of Step 5

Explanation of Figure 10

With the closing tag of $\mathtt{a}_3$, the AC a_3 turned out to be false (since it had no `descendant b`), thus it disappeared. The PA $\boxed{c_3}$ is still a potential answer to the XPath query and it depends on the fulfillment of either AC a_1 or AC a_2.

5.2.8 Step 6 - The opening tag of $\mathtt{c}_4$

Now the whole XML stream fragment was processed. Figure 2 shows the tree view of the XML stream. Figure 11 shows evaluation status of SPEX after encountering the opening tag of c_4. The XML stream is:
`<a1> <c1> <a2> <b1> <d1> </d1> <c2> [...] </c2> </b1> <b2> <d2> [...] </d2> </b2> <a3> <b3> </b3> <c3> </c3> </a3> <c4>`

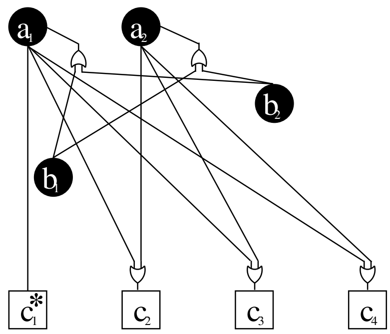

Figure 11: PA-AC graph structure in SPEX after the evaluation of the stream fragment

Explanation of Figure 11

As soon, as the opening tag of $\mathtt{c}_4$ appears on the XML stream, two things happen:
(1) $\boxed{c_4}$ is a new PA, and
(2) $\mathtt{c}_4$ is a following-sibling of b_1 and of b_2. Therefore b_1 and b_2 turn to `true`, thus, all ACs depending on b_1 or b_2 turn to `true`. In this case, ACs a_1 and a_2 turn to true. Now all PAs depending on either AC a_1 or AC a_2 are answers to the XPath query. Here, every PA depends on AC a_1, thus now every PA is an answer to the query.
Remark 1: SPEX outputs the PAs in document order, i.e., SPEX starts immediately to output the PA $\boxed{c_1}$ (this is modeled by the star in $\boxed{c_1}$ in the figure). Because the element c_1 is not yet closed, SPEX can not output any other PAs, although all other PAs are for sure answers to the query. PA $\boxed{c_2}$, $\boxed{c_3}$ and $\boxed{c_4}$ need to be kept in memory and output later when the outputting of $\boxed{c_1}$ is finished (i.e. the end tag of $\mathtt{c}_1$ appears), so as to deliver answers in document order.
Remark 2: The PAs $\boxed{c_2}$, $\boxed{c_3}$ and $\boxed{c_4}$ are already included in the PA $\boxed{c_1}$. Thus, they are called nested answers.

5.2.9 Conclusion

Throughout this example we have seen possible connections between ACs and PAs:

- relations between ACs, e.g., a_1 depends on b_1 or b_2.
- relations between ACs and PAs, e.g., $\boxed{\mathrm{c}_1}$ depends on a_1 or a_2.
- relations between PAs (like nesting), e.g. $\boxed{\mathrm{c}_1}$ includes $\boxed{\mathrm{c}_2}$, $\boxed{\mathrm{c}_3}$ and $\boxed{\mathrm{c}_4}$.

All these relations are modeled in SAL and primitives for exploring these structures are proposed.

6 SAL - the SPEX Approximation Language

This section introduces an AL for SPEX called SAL (SPEX Approximation Language).

Section 6.1 gives a short general overview how SAL works. The data model is given in Section 6.2, followed by Section 6.3 that takes a closer look on Sal. Sections 6.4 and 6.5 define SAL constructs. Finally grammar and semantics of all SAL construct are given in Section 6.6.

6.1 How SAL works

SAL operates on PAs, with several attributes, that are connected to predicates of the XPath query by ACs. The information used by SAL is modeled in a directed acyclic graph, DAG, where the leaf nodes are PA, the others nodes are ACs (PA-AC graph structure). This is a part of a complete answer aggregate (CAA) [11].

The goal of SAL is to get rid of one or more PA. This goal could be achieved in two ways, as detailed in Section 6.5:

- By just discarding or outputting PAs, or
- by determining some ACs (i.e. set to `true` or `false`), which leads to discarding or outputting of PAs.

A SAL approximation technique (SALAT) operates on the PA-AC graph structure. Internally this is represented as a list. The items of this list can be PA items or AC items. How such a PA-AC graph structure is created while processing an XML stream is shown in Section 5.2.
Figure 12 shows an example PA-AC graph structure (taken from the "SPEX at work" Example in the previous section).

Item list

As mentioned above, the item list can contain two kind of items:

- PA items
- AC items

When introducing the data model in Section 6.2, it will be shown what kind of information these items contain.

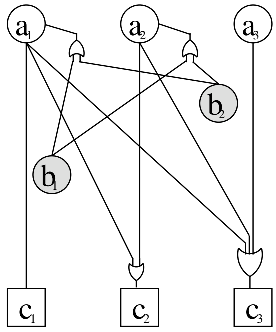

Figure 12: Example PA-AC graph structure

6.2 Data Model

6.2.1 Potential Answer Item - PA Item

A PA item represents a PA with its properties. A PA item has the following properties:

1. Identifier (Number)
   A consecutive number, that uniquely identifies the PA. A PA item with a lower identifier than another appeared earlier in the XML input stream of SPEX.

2. Number of XML elements (Number)
   The number of XML elements currently stored in this PA.

3. Number of characters (Number)
   The number of characters currently stored in this PA.

4. Children PA items (List)
   The list of children PA items, themselves PA items, that are included in this PA (in case of nesting).

5. Parent PA item (PA item)
   The parent PA item of this item.

6. Connected AC items (List)
   The list of AC items that are connected with this PA item.

The six properties are stored in a tuple that represents a PA item:

```
(   identifier,
    number of XML elements,
```

```
    number of characters,
    children PA items,
    parent PA item,
    connected AC items
)
```

6.2.2 Answer Condition Item - AC Item

An AC item represents an AC. Each AC item is associated with a predicate from the initial XPath query. An AC item hat the following properties:

1. Identifier (Number)
   A consecutive number that uniquely identifies the AC. An AC item that has a higher identifier than another was created after the AC item with the lower one.

2. Connected PA items (List)
   The list of PA items connected with this item.

3. Parent AC items (List)
   The list of AC items of the previous predicate (i.e. the predicate appearing earlier in the XPath query) connected with this item

4. Children AC items (List)
   The list of AC items of the next predicate (i.e. the predicate appearing later in the XPath query) connected with this item

5. Predicate information (List)
   Information from the associated predicate:

   - *Identifier (Number)*
     A consecutive number that uniquely identifies the predicate. A predicate with a lower identifier than another appeared earlier in the XPath query.
   - *Number of created ACs (Number)*
     The number of ACs that were created at this predicate until now while evaluating the XPath query with SPEX.
   - *Number of fulfilled ACs (Number)*
     The number of ACs that were created at this predicate until now while evaluating the XPath query with SPEX.
   - *Number of failed ACs (Number)*
     The number of ACs that were determined to false by SPEX at this predicate until now.

- *Number of undecided ACs (Number)*
  The number of ACs that were determined to true by SPEX at this predicate until now.
- *Number of ACs set to true due to SAL action (Number)*
  The number of ACs that were determined to true by SAL at this predicate until now.
- *Number of ACs set to false due to SAL action (Number)*
  The number of ACs that were determined to false by SAL at this predicate until now.

The five properties are stored in a tuple that represents an AC item:

```
(   identifier,
    connected PA items,
    parent AC items,
    children AC items,
    (
        identifier
        number of created ACs
        number of fulfilled ACs
        number of failed ACs
        number of undecided ACs
        number of ACs set to true due to SAL action
        number of ACs set to false due to SAL action
    }
)
```

6.3 A closer look on SAL

This section takes a closer look on SAL. Definitions and semantics are given for SAL constructs in general, followed by an introduction of the Sal namespaces.

6.3.1 Composition of SAL Approximation Constructs

A SAL Approximation Technique (SALAT) consists in two parts: The SAL selection part and the SAL action part. With the SAL selection part, an item list (PA or AC item list) is selected from the initial item list of all PAs. In the SAL action part some action is performed on the PA or AC items of that list. We introduced an operator `>` :

selection $>$ *action*:The SAL action is applied on the result list of the SAL selection part.

Definition 3 (SAL Approximation Technique). *A SAL approximation technique (SALAT) has the form:*
`selection` $>$ `action`
where `selection` *stands for the selection part and* `action` *stands for the action part of a SAL approximation technique.*

6.3.2 SAL Selections

A selection in SAL is a composition of various selection constructs, implemented by functions. The function $\mathcal{S}$ is considered for giving the semantics of various selection constructs. $\mathcal{S}[\![c]\!](l)$ means the result of the evaluation of the selection construct c on the item list l.

Definition 4 (Semantics). $\mathcal{S}$ *has the type:*
$\mathcal{S} \rightarrow$ *selection construct* $\rightarrow$ *item list* $\rightarrow$ *item list*

There are four types of semantic functions in SAL:

1. $\mathcal{S} \rightarrow$ selection construct $\rightarrow$ PA item list $\rightarrow$ PA item list
2. $\mathcal{S} \rightarrow$ selection construct $\rightarrow$ PA item list $\rightarrow$ AC item list
3. $\mathcal{S} \rightarrow$ selection construct $\rightarrow$ AC item list $\rightarrow$ PA item list
4. $\mathcal{S} \rightarrow$ selection construct $\rightarrow$ AC item list $\rightarrow$ AC item list

In the following we define a composition operator `/` :

Definition 5 (Composition, selection part). *Given two approximation constructs c_1 and c_2, then a valid composition is either c_1/c_2 or c_2/c_1, with the semantics:*
$\mathcal{S}[\![c_1/c_2]\!](l) = \mathcal{S}[\![c_2]\!](\mathcal{S}[\![c_1]\!](l))$
$\mathcal{S}[\![c_2/c_1]\!](l) = \mathcal{S}[\![c_1]\!](\mathcal{S}[\![c_2]\!](l))$

6.3.3 SAL Predicates

Every selection in SAL could be extended by a SAL predicate. A SAL predicate could be seen as an `if` construct, that is applied on an item list and returns those items that return a nonempty list for the SAL predicate. That is, the item list returned by a SAL selection with a SAL predicate is a subset of the list that would have been returned without the SAL predicate (if all items in the list return a nonempty list to the SAL predicate, the result without the SAL predicate is the same as with the SAL predicate).

Definition 6 (SAL Predicate). *Given a selection constructs* `c` *and a SAL selection* `s`*, with* `s` *expecting that kind of item list that is the return type of* `c`*, then the* `s` *is a legal SAL predicate of* c *with the semantics:*

$$\mathcal{S}[\![c[s]]\!](l) = \{x_1 | x_1 \in \mathcal{S}[\![c]\!](l) \wedge \mathcal{Q}[\![s]\!]x_1\}$$
$$\mathcal{Q}[\![s]\!]x = \mathcal{S}[\![s]\!]x \neq \emptyset$$

With introducing the operators `and`, `or` and `not` to predicates, SAL is enhanced with union (`or`) and intersection (`and`).

6.3.4 SAL Actions

Like selection constructs, a SAL action works on an item list, but instead of returning an item list, action constructs perform an action on the given list, that leads to the discard or output of PAs.

There are two kind of actions, one for each type of item lists:

1. actions on a PA item list
2. actions on an AC item list

6.3.5 SAL Namespaces

As introduced above there are several sets of constructs in SAL. We introduced a namespace for each set of constructs in order to identify them uniquely.

Definition 7 (SAL Namespaces). *An SAL namespace is a collection of names, that are used in SAL constructs as prefixes.*

Definition 8 (SAL Construct Call). *A SAL construct is called with its namespace, a colon and its name:*
`SAL construct := namespace:name`

There are six namespaces in SAL:

1. `pa`
   Constructs in this namespace have the semantics:
   $\mathcal{S} \rightarrow$ selection construct $\rightarrow$ PA item list $\rightarrow$ PA item list

2. `ac`
   Constructs in this namespace have the semantics:
   $\mathcal{S} \rightarrow$ selection construct $\rightarrow$ AC item list $\rightarrow$ AC item list

3. `paToAc`
   Constructs in this namespace have the semantics:
   $\mathcal{S} \rightarrow$ selection construct $\rightarrow$ PA item list $\rightarrow$ AC item list

4. `acToPa`
   Constructs in this namespace have the semantics:
   $\mathcal{S} \rightarrow$ selection construct $\rightarrow$ AC item list $\rightarrow$ PA item list

5. `paAction`
   Actions in this namespace get a PA item list as input and apply their changes to SPEX by discarding or outputting PAs.

6. `acAction`
   Actions in this namespace get an AC item list as input and apply their changes to SPEX at ACs, i.e. set to true or false.

6.3.6 Sure Memory-Deallocation

In order to guarantee that SAL deallocates some memory, an approximation technique has to meet several conditions. If an approximation technique fulfills these conditions, it is called a SMDAT (*Sure Memory-Deallocation Approximation Technique*). Since it is not possible ensure in general that determining some ACs will lead to memory-deallocation, sure memory-deallocation approximation techniques have to return a nonempty PA item list after the selection part. This is achieved by *only* using `pa` constructs that never will return an empty list for a nonempty input list (cf. Section 6.4.1).

In cases of approximation techniques with bridging construct one can not ensure that memory is deallocated for sure:
If the latest bridging construct is a paToAc, as mentioned above, determining some ACs does not have to lead to sure memory-deallocation. But also if the last bridging construct is an acToPa, memory-deallocation is not guaranteed, since it is possible, to have an AC that is not connected to any PA, i.e. it is possible that the bridging construct returns an empty PA item list.

6.4 SAL Selection Constructs

This section described the SAL constructs of the selection part of a SALAT.

6.4.1 PA Constructs

These constructs belong to the namespace `pa` (cf. Section 6.3.5). PA constructs have the semantics:

$$\mathcal{S} \rightarrow \text{selection construct} \rightarrow \text{PA item list} \rightarrow \text{PA item list}$$

The constructs from this namespace could be split into three sets:

- Basic constructs
- Sorting constructs
- Parent-child relation constructs

For examples in this section it is assumed, that there are the following PA items when approximation is called:

- c_1 : `<a>example for SAL!</a>`
  Number of XML-Elements : 1
  Number of characters : 23
  Children PA items :
  Parent PA item :

- c_2 : `<a><a>foo</a></a>`
  Number of XML-Elements : 2
  Number of characters : 17
  Children PA items : c_3
  Parent PA item :

- c_3 : `<a>foo</a>`
  Number of XML-Elements : 1
  Number of characters : 10
  Children PA items :
  Parent PA item : c_2

- c_4 : `<a>this element is not yet closed`
  Number of XML-Elements : 0
  Number of characters : 33
  Children PA items :
  Parent PA item :

The following constructs introduced in this section will never return an empty list for a nonempty input list ,i.e. they could be used to compose a SMDAT:

- reverse
  changes only the order of an item list, not the content

- take
  returns a list with at least one PA item

- sortByChars
  changes only the order of an item list, not the content

- sortByElems
  changes only the order of an item list, not the content

- sortByDocOrder
  changes only the order of an item list, not the content

The following constructs do not ensure that for a nonempty input list a nonempty list is returned:

- parent
  returns an empty list if there are only PA items that do not have parent PA items in the item list

- child
  returns an empty list if there are only PA items that do not have child PA items in the item list

- ancestor
  returns an empty list if there are only PA items that do not have ancestor PA items in the item list

- descendant
  returns an empty list if there are only PA items that do not have descendant PA items in the item list

Basic constructs

This section describes constructs, that do very basic operations on a PA item list, like reversing the list or selecting the first few items. Especially in combination with the sorting constructs introduced later, these constructs are very useful.

reverse

This construct reverses the list. This construct is can be used to compose a SMDAT.

Applying this construct after a sorting construct allows you to reverse sort order (i.e. sort ascending respectively sort in inverted document order).

Syntax	pa:reverse
Semantics	$\mathcal{S}[\![\texttt{pa:reverse}]\!]$ (l) = `reverse l`
Complexity	$O(n)$

Example	Get the PA items in reversed document order.
Input List	$[c_1, c_2, c_3, c_4]$
SALAT (selection part only)	pa:reverse
Output List	$[c_4, c_3, c_2, c_1]$

```
Implementation of reverse:

reverse []     = []
reverse (x:xs) = reverse (xs) @ [x]
```

take(n : Integer)

This construct selects the first n ($n \in \mathbb{N} \setminus \{0\}$) PA items form the list.

The selection part of the SALAT in the PP scenario with the sliding window uses `reverse` in order to get the PA items sorted in reversed document order and `take 1` to select the oldest PA to be discarded.

Syntax	pa:take(n)
Semantics	$\mathcal{S}[\![\texttt{pa:take(n)}]\!]$ (l) = `take n l`
Complexity	$O(n)$

Example	Keep only the first 2 PA.
Input List	$[c_1, c_2, c_3, c_3, c_4]$
SALAT (selection part only)	pa:take(2)
Output List	$[c_1, c_2]$

```
Implementation of take(n):

take []     _        = []
take (x:xs) n | n<=1 = [x]
take (x:xs) n | n>1  = [x] @ take (xs) n-1
```

Sorting constructs

Constructs described in this section do not increase or decrease the PA item list, they just change its order. Therefor the list items are compared on a special criteria in order to sort the list.

- sortByChars
  Sorts the list descending by the number of characters.
- sortByElems
  Sorts the list descending by the number of XML-Elements.
- sortByDocOrder
  Sorts the list in document order.

The constructs are implemented by functions. These functions use quicksort[2] as sorting algorithm. Therefor the complexity is always the same: $O(n \cdot log(n))$ for the average case and $O(n^2)$ for the worst case.

```
Quicksort implementation:
quicksort compare []     = []
quicksort compare (x:xs) =
        quicksort compare elems_lt_x ++ [x] ++ quicksort compare elems_greq_x
                where
                      elts_lt_x   = [y | y <- xs, compare(y,x) <  0]
                      elts_greg_x = [y | y <- xs, compare(x,y) >= 0]
```

The function `compare` compares two elements a and b on a specific criteria and returns `0` if they are equal, a negative value if a is lower than b and a positive value if a is greater than b.

sortByChars

This construct sorts the list descending by the number of characters.

This construct is useful in applications where the size of PAs matters. If an application is interested in being able to store as much PAs as possible, the biggest PA will be discarded or output in order to use the gained space for several new PAs with fewer space usage. On the other hand an application could also be more interested in big results, i.e. disadvantages small PA items in order to have space for the big ones.

Syntax	pa:sortByChars
Semantics	$\mathcal{S}[\![$`pa:sortByChars`$]\!]$ (l) = `quicksort compareByChars l`
Complexity	$O(n \cdot log(n))$ average case, $O(n^2)$ worst case

[2] Hoare, 1962

Example	Select the PA item using the most space.
Input List	$[c_1, c_2, c_3, c_4]$
SALAT (selection part only)	pa:sortByChars/pa:take(1)
Output List	$[c_4]$

```
Implementation of compareByChars:

compareByChars (a,b) | getNumberOfChars(a) == getNumberOfChars(b) =  0
compareByChars (a,b) | getNumberOfChars(a) <  getNumberOfChars(b) = -1
compareByChars (a,b) | getNumberOfChars(a) >  getNumberOfChars(b) =  1

getNumberOfChars (_,_,x,_,_,_) = x
```

sortByElems

This construct sorts the list descending by the number of XML-elements.

This construct is useful if the number of XML-elements is a kind of quality measure for the PA, i.e. a PA containing more XML-elements usually contains more interesting information than ones with fewer XML-elements, so discarding the one having the fewest XML-elements is reasonable.

Syntax	pa:sortByElems
Semantics	$\mathcal{S}[\![\texttt{pa:sortByElems}]\!]\ (l) =$ `quicksort compareByElems l`
Complexity	$O(n \cdot log(n))$ average case, $O(n^2)$ worst case

Example	Select the PA item with the fewest XML-elements.
Input List	$[c_1, c_2, c_3, c_4]$
SALAT (selection part only)	pa:sortByElems/pa:reverse/pa:take(1)
Output List	$[c_4]$

```
Implementation of compareByElems:

compareByElems (a,b) | getNumberOfXmlElements(a) == getNumberOfXmlElements(b) =  0
compareByElems (a,b) | getNumberOfXmlElements(a) <  getNumberOfXmlElements(b) = -1
compareByElems (a,b) | getNumberOfXmlElements(a) >  getNumberOfXmlElements(b) =  1

getNumberOfXmlElements (_,x,_,_,_,_) = x
```

sortByDocOrder

This construct sorts the list in document order.

If document order is only secondary for the ranking of a PA, it is necessary to apply document order to a previous selection (see the example below).

Syntax	pa:sortByDocOrder
Semantics	$\mathcal{S}[\![$`pa:sortByDocOrder`$]\!]$ (l) = `quicksort compareByDocOrder l`
Complexity	$O(n \cdot log(n))$ average case, $O(n^2)$ worst case

Example	Select the PA out of the two PA items using the most space.
Input List	$[c_1, c_2, c_3, c_4]$
SALAT (selection part only)	pa:sortByChars/pa:take(2)/ pa:sortByDocOrder/pa:take(1)
Output List	$[c_1]$

```
Implementation of compareByDocOrder:

compareByDocOrder (a,b) | getPAIdentifier(a) == getPAIdentifier(b) =  0
compareByDocOrder (a,b) | getPAIdentifier(a) <  getPAIdentifier(b) = -1
compareByDocOrder (a,b) | getPAIdentifier(a) >  getPAIdentifier(b) =  1

getPAIdentifier (x,_,_,_,_,_) = x
```

Parent-child constructs

These functions work on the semantical (parent-child) relationship of PA items.

- parent
  Selects the parent PA items of all PA items in the PA item list.

- ancestor
  Selects the ancestor PA items of all PA items in the PA item list.

- child
  Selects the child PA items of all PA items in the PA item list.

- descendant
  Selects the descendant PA items of all PA items in the PA item list.

parent

Selects the parent PA items of all PA items in the PA item list.

Syntax	pa:parent
Semantics	$\mathcal{S}[\![$`pa:parent`$]\!]$ (l) = `parent l`
Complexity	$O(n)$

Example	Select the parent PA item of the PA using the fewest space.
Input List	$[c_1, c_2, c_3, c_4]$
SALAT (selection part only)	pa:sortByChars/pa:take(1)/pa:parent
Output List	$[c_2]$

```
Implementation:

parent []                                        = []
parent (x:xs) | getParentPAItem(x)==null =
   parent(xs)
parent (x:xs) | getParentPAItem(x)!=null =
   getParentPAItem(x) @ parent(xs)

getParentPAItem (_,_,_,_,x,_) = x
```

ancestor

Selects the ancestor PA items of all PA items in the item list. This is the transitive and reflexive closure of `parent`.

Syntax	pa:ancestor
Semantics	$\mathcal{S}[\![\texttt{pa:ancestor}]\!]\ (l) = \texttt{ancestor l}$
Complexity	$O(n \cdot d)$

Example	Select the ancestor PA items of the PA using the fewest space.
Input List	$[c_1, c_2, c_3, c_4]$
SALAT (selection part only)	pa:sortByChars/pa:take(1)/pa:ancestor
Output List	$[c_2]$

```
Implementation:

ancestor []                                        = []
ancestor (x:xs) | getParentPAItem(x)==null =
   ancestor(xs)
ancestor (x:xs) | getParentPAItem(x)!=null =
   getParentPAItem(x) @ ancestor(getParentPAItem(x):xs)
```

child

Selects the child PA items of all PA items in the item list.

Syntax	pa:child
Semantics	$\mathcal{S}[\![\texttt{pa:child}]\!]\ (l) = \texttt{child l}$
Complexity	$O(n)$

Example	Select the children PA items from the initial PA item list.
Input List	$[c_1, c_2, c_3, c_4]$
SALAT (selection part only)	pa:child
Output List	$[c_3]$

`Implementation:`

```
child []                                              = []
child (x:xs) | getChildrenPAItems(x)==null =
   child(xs)
child (x:xs) | getChildrenPAItems(x)!=null =
   getChildrenPAItems(x) @ child(xs)

getChildrenPAItems (_,_,_,x,_,_) = x
```

descendant

Selects the descendant PA items of all PA items in the PA item list. This is the transitive and reflexive closure of child.

Syntax	pa:descendant
Semantics	$\mathcal{S}[\![\texttt{pa:descendant}]\!]\ (l) = \texttt{descendant l}$
Complexity	$O(n \cdot d)$

Example	Select the descendant PA items from the initial PA item list.
Input List	$[c_1, c_2, c_3, c_4]$
SALAT (selection part only)	pa:descendant
Output List	$[c_3]$

`Implementation:`

```
descendant []                                              = []
descendant (x:xs) | getChildrenPAItems(x)==null =
   descendant(xs)
descendant (x:xs) | getChildrenPAItems(x)!=null =
   getChildrenPAItems(x) @ descendant(getChildrenPAItems(x):xs)
```

6.4.2 Bridging constructs

In opposite to PA or AC constructs (cf. Sections 6.4.1, 6.4.3), bridging constructs are implemented by functions that return a different kind of list than they get as input. These constructs are used to get from PA items to AC items and back.

Bridging constructs: PA item list to AC item list

These constructs belong to the namespace `paToAc` (cf. Section 6.3.5). PA item list to AC item list constructs have the semantics:

$$\mathcal{S} \rightarrow \text{selection construct} \rightarrow \text{PA item list} \rightarrow \text{AC item list}$$

These constructs are a requirement for the usage of AC constructs in addition to the previously defined PA constructs.

connectedACs

This construct returns all ACs that are connected with the PA items in the item list.

Syntax	paToAc:connectedACs
Semantics	$\mathcal{S}[\![\texttt{paToAc:connectedACs}]\!]\ (\mathrm{l}) =$ `connectedACs (l)`
Complexity	$O(n)$

```
Implementation:

connectedACs []     = []
connectedACs (x:xs) = getConnectedACs(x) @ connectedACs(xs)

getConnectedACs (_,_,_,_,_,x) = x
```

Bridging constructs: AC item list to PA item list

These constructs belong to the namespace `acToPa` (cf. Section 6.3.5). These constructs have the semantics:

$$\mathcal{S} \rightarrow \text{selection construct} \rightarrow \text{AC item list} \rightarrow \text{PA item list}$$

These constructs are a requirement in order to get back to PAs, after dealing with ACs.

connectedPAs

This construct returns a PA item list containing all PA items that are connected with the ACs in the input list.

Syntax	acToPa:connectedPAs
Semantics	$\mathcal{S}[\![\texttt{acToPa:connectedPAs}]\!]$ (l) = `connectedPAs (l)`
Complexity	$O(n)$

```
 Implementation of connectedPAs:

connectedPAs []         = []
connectedPAs (x:xs)     = getConnectedPAs(x) @ connectedPAs(xs)

getConnectedPAs (_,c,_,_,_) = c
```

6.4.3 AC constructs

These constructs belong to the namespace `ac` (cf. Section 6.3.5). AC constructs have the semantics:

$$\mathcal{S} \rightarrow \text{selection construct} \rightarrow \text{AC item list} \rightarrow \text{AC item list}$$

The constructs from this namespace could be split into three sets:

- Basic constructs
- Sorting constructs
- Predicate related constructs

Remark: The following constructs introduced in this section do never return an empty list for a nonempty input list. (Anyway, since determining some ACs does not ensure, that PAs are discarded or output it is not possible to compose a SMDAT using AC constructs.)

- reverse
  changes only the order of an item list, not the content
- take
  returns a list with at least one AC item
- sortByConnectedPAs
  changes only the order of an item list, not the content
- sortByAge
  changes only the order of an item list, not the content
- sortByConnectedACs
  changes only the order of an item list, not the content

The following constructs do not ensure that for a nonempty input list a nonempty list is returned:

- selectPredicate(`predicateNumber`)
  returns an empty list if there is no AC item from the predicate `predicateNumber` *in the item list*
- selectMostFulfillingPredicate
  returns an empty list if there is no AC item from the most fulfilling predicate in the item list
- selectMostFailingPredicate
  returns an empty list if there is no AC item from the most failing predicate in the item list

Basic constructs

These constructs provide some basic operations on AC item lists. Especially in combination with the sorting constructs introduced later these constructs are very useful.

reverse

This construct reverses the list.

Syntax	ac:reverse
Semantics	$\mathcal{S}[\![$`ac:reverse`$]\!]$ (l) = `reverse l`
Complexity	$O(n)$

take(n : Integer)

This construct selects the first n ($n \in \mathbb{N}$) AC items form the list.

Syntax	ac:take(n)
Semantics	$\mathcal{S}[\![$`ac:take(n)`$]\!]$ (l) = `take n l`
Complexity	$O(n)$

Sorting constructs

Constructs described in this section do not increase or decrease the item list, they just change its order. Therefor the list items are compared on a special criteria in order to sort the list.

- sortByConnectedPAs
  Sorts the list descending by the number of connected PAs.
- sortByAge
  Sorts the list descending by the age of the ACs, i.e. the AC that exists longest will be the first in the list.
- sortByConnectedACs
  Sorts the list descending by the number of connected ACs (both directions are counted: as well to the previous predicate, as to the next predicate)

Like the sorting constructs for PA item lists, these constructs are implemented by functions using quicksort as sorting algorithm. Therefore the complexity is $O(n \cdot log(n))$ for the average case and $O(n^2)$ for the worst case.

sortByConnectedPAs

This construct sorts the AC item list descending by the number of connected PAs.

An application might be interested in the AC, that is connected to the fewest PAs, because determining this AC with a SALAT does only change very few at the system.

Syntax	ac:sortByConnectedPAs
Semantics	$\mathcal{S}[\![$`ac:sortByConnectedPAs`$]\!]$ (l) = `quicksort compareByConnectedPAs l`
Complexity	average case: $O(n \cdot log(n))$, worst case: $O(n^2)$

```
Implementation of compareByConnectedPAs:

compareByConnectedPAs (a,b)
     | getNumberOfConnectedPAs(a) == getNumberOfConnectedPAs(b) =  0
compareByConnectedPAs (a,b)
     | getNumberOfConnectedPAs(a) <  getNumberOfConnectedPAs(b) = -1
compareByConnectedPAs (a,b)
     | getNumberOfConnectedPAs(a) >  getNumberOfConnectedPAs(b) =  1

getNumberOfConnectedPAs c = count getConnectedPAs(c)

getConnectedPAs (_,x,_,_,_) = x

count []     = 0
count (x:xs) = 1 + count (xs)
```

sortByAge

This construct sorts an AC item list descending by the age of the ACs, i.e. the consecutive number.

Using this construct is useful in finding the oldest AC (i.e. the one that spent the longest time in the system and is still undetermined).

Syntax	ac:sortByAge
Semantics	$\mathcal{S}[\![$`ac:sortByAge`$]\!]$ (l) = `quicksort compareByACIdentifier l`
Complexity	average case: $O(n \cdot log(n))$, worst case: $O(n^2)$

```
Implementation

compareByACIdentifier (a,b)
     | getACIdentifier(a) == getACIdentifier(b) =  0
compareByACIdentifier (a,b)
     | getACIdentifier(a) <  getACIdentifier(b) = -1
compareByACIdentifier (a,b)
     | getACIdentifier(a) >  getACIdentifier(b) =  1
```

```
getACIdentifier (x,_,_,_,_) = x
```

sortByConnectedACs

This construct sorts the AC item list descending by the number of ACs (of the previous and the next predicate) an AC is connected to.

Using this construct is useful to get an impression of the importance of an AC to other ACs.

Syntax	ac:sortByConnectedACs
Semantics	$\mathcal{S}[\![$`ac:sortByConnectedACs`$]\!]$ (l) = `quicksort compareByConnectedACs l`
Complexity	average case: $O(n \cdot log(n))$, worst case: $O(n^2)$

```
Implementation

compareByConnectedACs (a,b)
     | getConnectedACs(a) == getConnectedACs(b) =  0
compareByConnectedACs (a,b)
     | getConnectedACs(a) <  getConnectedACs(b) = -1
compareByConnectedACs (a,b)
     | getConnectedACs(a) >  getConnectedACs(b) =  1

getConnectedACs (_,_,previousACs,nextACs,_) =
     count(previousACs) + count(nextACs)
```

Predicate related constructs

Constructs from this section are used to change the AC item list by some criteria based on predicate information. Since a predicate often has more than one AC, this constructs do not make very fine graded selections, since all ACs belonging to the same predicate are treated equally.

selectPredicate `(predicateNumber : Integer)`

Only the ACs that belong to the predicate `n` ($n \in \mathbb{N}$) will remain in the AC item list.

This construct is useful if a predicate is more important than another one for the user.

Syntax	ac:selectPredicate `(predicateNumber)`
Semantics	$\mathcal{S}[\![$`ac:selectPredicate (predicateNumber)`$]\!]$ (l) = `selectPredicate predicateNumber l`
Complexity	$O(n)$

```
Implementation of selectPredicate
```

```
selectPredicate _ []                                                   = []
selectPredicate n (x:xs) | getPredicateNumber(getPredicateInfo(x))==n =
     (x) @ selectPredicate n (xs)
selectPredicate n (x:xs) | getPredicateNumber(getPredicateInfo(x))!=n =
     selectPredicate n (xs)

getPredicateInfo (_,_,_,_,i) = i

getPredicateNumber (x,_,_,_,_,_,_) = x
```

selectMostFulfillingPredicate

Only ACs, that belong to the predicate that fulfills its ACs best (i.e. the number of fulfilled ACs by the number of created ACs is bigger than at every other predicate).

Applications that want to set ACs to true that fulfill the query very likely, need this construct.

Syntax	ac:selectMostFulfillingPredicate
Semantics	$\mathcal{S}[\![$`ac:selectMostFulfillingPredicate`$]\!]$ (l) = `selectMostFulfillingPredicate l`
Complexity	average case: i$O(n^2 \cdot log(n))$, worst case: $O(n^3)$

```
Implementation of selectMostFulfillingPredicate:

selectMostFulfillingPredicate l =
     selectPredicate getPredicateNumber(getPredicateInfo(
       take 1 quicksort compareByPredicateFulfillment l))

compareByPredicateFulfillment (a,b) |
   getPredicateInfo(getFulfillment(a)) == getPredicateInfo(getFulfillment(b))
     =  0
compareByPredicateFulfillment (a,b) |
   getPredicateInfo(getFulfillment(a)) >  getPredicateInfo(getFulfillment(b))
     =  1
compareByPredicateFulfillment (a,b) |
   getPredicateInfo(getFulfillment(a)) <  getPredicateInfo(getFulfillment(b))
     = -1

getFulfillment(_,created,fulfilled,_,_,_,_) = fulfilled/created
```

selectMostFailingPredicate

Only ACs, that belong to the Predicate that fails the most ACs (by the number of ACs this predicate created) will remain in the AC item list.

Application that want to set ACs to false that fail the query very likely, need this construct.

Syntax	ac:selectMostFailingPredicate
Semantics	$\mathcal{S}[\![\texttt{ac:selectMostFailingPredicate}]\!]$ (l) = `selectMostFailingPredicate l`
Complexity	average case: i$O(n^2 \cdot log(n))$, worst case: $O(n^3)$

```
Implementation of selectMostFailingPredicate:

selectMostFailingPredicate l =
     selectPredicate getPredicateNumber(getPredicateInfo(
       take 1 quicksort compareByPredicateFailing l))

compareByPredicateFailing (a,b) |
   getPredicateInfo(getFailing(a)) == getPredicateInfo(getFailing(b)) =  0
compareByPredicateFailing (a,b) |
   getPredicateInfo(getFailing(a)) >  getPredicateInfo(getFailing(b)) =  1
compareByPredicateFailing (a,b) |
   getPredicateInfo(getFailing(a)) <  getPredicateInfo(getFailing(b)) = -1

getFailing(_,created,_,failed,_,_,_) = failed/created
```

6.5 SAL Action Constructs

In opposite to the selection constructs (cf. Section 6.4) , used in the selection part of a SALAT this section specifies the action part of a SALAT, i.e. the action constructs.

6.5.1 AC actions

These constructs belong to namespace acAction (cf. Section 6.3.5).

toFalse

This action sets all ACs in the item list to false.

toTrue

This action sets all ACs in the item list to true.

6.5.2 PA actions

These constructs belong to the namespace paAction (cf. Section 6.3.5).

discard

This action discards all PAs in the item list, i.e. drops them from SPEX and out of memory. SALAT with paAction:discard as action part ensure, that approximation does not lead to an output of a PA where it is still uncertain if this PA does fulfill the XPath query (alpha error = 0). On the other hand, this might lead to false negatives, i.e. PAs that would fulfill the XPath query are discarded.

output

This action outputs all PAs in the item list from SPEX. SALAT with paAction:output as action part ensure, that no PAs are discarded while it is still uncertain if they are answers to the query or not(beta error = 0). On the other hand this might lead to false positives, i.e. the output of PAs that do not fulfill the XPath query.

6.6 Getting Together: Grammar and Semantics

The following sections give the grammar (Section 6.6.1) and the semantics of SAL (Section 6.6.2).

6.6.1 Grammar

This section defines the grammar of SAL.

$SALAT$ ::= $PASelect$?`>`$paAction$
  | $PASelect$?`/`$paToAc$`/`$ACSelect$`>`$acAction$

$PASelect$ ::= $BasePASelect$`/`$PASelect$
  | $paToAc$`/`$ACSelect$`/`$acToPa$`/`$PASelect$

$ACSelect$ ::= $BaseACSelect$`/`$ACSelect$
  | $acToPa$`/`$PASelect$`/`$paToAc$`/`$ACSelect$

$paToAc$ ::= `connectedACs`
$acToPa$ ::= `connectedPAs`
$BasePASelect$ ::= `reverse`
  | `take` ($Number$)
  | `sortByChars`
  | `sortByElements`
  | `sortByDocumentOrder`
  | `parent`
  | `ancestor`
  | `child`
  | `descendant`

$BaseACSelect$ ::= `reverse`
  | `take` ($Number$)
  | `sortByConnectedPAs`
  | `sortByAge`
  | `sortByConnectedACs`
  | `selectPredicate` ($Number$)
  | `selectMostFulfillingPredicate`
  | `selectMostFailingPredicate`

$paAction$ ::= `discard | output`

$paAction$ ::= `toFalse | toTrue`

$Number$ ::= $NonZeroNumber$
  | $NonZeroNumber$ $AnyNumber+$

$NonZeroNumber$ ::= `1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`

$AnyNumber$ ::= `0 |` $NonZeroNumber$

6.6.2 Semantics

This section gives the semantics of every SAL construct.

Composition of selection constructs

$\mathcal{S}[\![c_1/c_2]\!](l) = \mathcal{S}[\![c_2]\!](\mathcal{S}[\![c_1]\!](l))$

PA Selection Constructs

$$
\begin{array}{ll}
\mathcal{S}[\![\texttt{pa:reverse}]\!]\ (\mathrm{l}) & = \texttt{reverse (l)} \\
\mathcal{S}[\![\texttt{pa:take(n)}]\!]\ (\mathrm{l}) & = \texttt{take (n) (l)} \\
\mathcal{S}[\![\texttt{pa:sortByChars}]\!]\ (\mathrm{l}) & = \texttt{quicksort sortByChars (l)} \\
\mathcal{S}[\![\texttt{pa:sortByElems}]\!]\ (\mathrm{l}) & = \texttt{quicksort sortByElems (l)} \\
\mathcal{S}[\![\texttt{pa:sortByDocOrder}]\!]\ (\mathrm{l}) & = \texttt{quicksort sortByDocOrder (l)} \\
\mathcal{S}[\![\texttt{pa:parent}]\!]\ (\mathrm{l}) & = \texttt{parent (l)} \\
\mathcal{S}[\![\texttt{pa:ancestor}]\!]\ (\mathrm{l}) & = \texttt{ancestor (l)} \\
\mathcal{S}[\![\texttt{pa:child}]\!]\ (\mathrm{l}) & = \texttt{child (l)} \\
\mathcal{S}[\![\texttt{pa:descendant}]\!]\ (\mathrm{l}) & = \texttt{descendant (l)}
\end{array}
$$

paToAc Bridging Constructs

$$\mathcal{S}[\![\texttt{paToAc:connectedACs}]\!]\ (\mathrm{l}) = \texttt{connectedACs (l)}$$

acToPa Bridging Constructs

$$\mathcal{S}[\![\texttt{acToPa:connectedPAs}]\!]\ (\mathrm{l}) = \texttt{connectedPAs (l)}$$

AC Selection Constructs

$$
\begin{array}{ll}
\mathcal{S}[\![\texttt{ac:reverse}]\!]\ (\mathrm{l}) & = \texttt{reverse (l)} \\
\mathcal{S}[\![\texttt{ac:take(n)}]\!]\ (\mathrm{l}) & = \texttt{take (n) (l)} \\
\mathcal{S}[\![\texttt{ac:sortByConnectedPAs}]\!]\ (\mathrm{l}) & = \texttt{quicksort compareByConnectedPAs (l)} \\
\mathcal{S}[\![\texttt{ac:sortByAge}]\!]\ (\mathrm{l}) & = \texttt{quicksort compareByACIdentifier (l)} \\
\mathcal{S}[\![\texttt{ac:sortByConnectedACs}]\!]\ (\mathrm{l}) & = \texttt{quicksort compareByConnectedACs (l)} \\
\mathcal{S}[\![\texttt{ac:selectPredicate(n)}]\!]\ (\mathrm{l}) & = \texttt{selectPredicate (n) (l)} \\
\mathcal{S}[\![\texttt{ac:selectMostFulfillingPredicate}]\!]\ (\mathrm{l}) & = \texttt{selectMostFulfillingPredicate (l)} \\
\mathcal{S}[\![\texttt{ac:selectMostFailingPredicate}]\!]\ (\mathrm{l}) & = \texttt{selectMostFailingPredicate (l)}
\end{array}
$$

Functions

```
reverse []     = []
reverse (x:xs) = reverse (xs) @ [x]
```

```
take []      _         = []
take (x:xs) n | n<=1 = [x]
take (x:xs) n | n>1  = [x] @ take (xs) n-1


parent []                                = []
parent (x:xs) | getParentPAItem(x)==null =
   parent(xs)
parent (x:xs) | getParentPAItem(x)!=null =
   getParentPAItem(x) @ parent(xs)



child []                                   = []
child (x:xs) | getChildrenPAItems(x)==null =
   child(xs)
child (x:xs) | getChildrenPAItems(x)!=null =
   getChildrenPAItems(x) @ child(xs)


ancestor []                                = []
ancestor (x:xs) | getParentPAItem(x)==null =
   ancestor(xs)
ancestor (x:xs) | getParentPAItem(x)!=null =
   getParentPAItem(x) @ ancestor(getParentPAItem(x):xs)


descendant []                                   = []
descendant (x:xs) | getChildrenPAItems(x)==null =
   descendant(xs)
descendant (x:xs) | getChildrenPAItems(x)!=null =
   getChildrenPAItems(x) @ descendant(getChildrenPAItems(x):xs)




connectedACs []     = []
connectedACs (x:xs) = getConnectedACs(x) @ connectedACs(xs)

connectedPAs []         = []
connectedPAs (x:xs)     = getConnectedPAs(x) @ connectedPAs(xs)

selectPredicate _ []                                                 = []
selectPredicate n (x:xs) | getPredicateNumber(getPredicateInfo(x))==n =
     (x) @ selectPredicate n (xs)
selectPredicate n (x:xs) | getPredicateNumber(getPredicateInfo(x))!=n =
     selectPredicate n (xs)
```

```
selectMostFulfillingPredicate l =
     selectPredicate getPredicateNumber(getPredicateInfo(
       take 1 quicksort compareByPredicateFulfillment l))

selectMostFailingPredicate l =
     selectPredicate getPredicateNumber(getPredicateInfo(
       take 1 quicksort compareByPredicateFailing l))
```

Help Functions

```
quicksort compare []     = []
quicksort compare (x:xs) =
        quicksort compare elems_lt_x ++ [x] ++ quicksort compare elems_greq_x
                where
                      elts_lt_x   = [y | y <- xs, compare(y,x) <  0]
                      elts_greg_x = [y | y <- xs, compare(x,y) >= 0]

compareByChars (a,b) | getNumberOfChars(a) == getNumberOfChars(b) =  0
compareByChars (a,b) | getNumberOfChars(a) <  getNumberOfChars(b) = -1
compareByChars (a,b) | getNumberOfChars(a) >  getNumberOfChars(b) =  1

compareByElems (a,b) | getNumberOfXmlElements(a) == getNumberOfXmlElements(b) =  0
compareByElems (a,b) | getNumberOfXmlElements(a) <  getNumberOfXmlElements(b) = -1
compareByElems (a,b) | getNumberOfXmlElements(a) >  getNumberOfXmlElements(b) =  1

compareByDocOrder (a,b) | getProNumber(a) == getProNumber(b) =  0
compareByDocOrder (a,b) | getProNumber(a) <  getProNumber(b) = -1
compareByDocOrder (a,b) | getProNumber(a) >  getProNumber(b) =  1

getPAIdentifier        (x,_,_,_,_,_) = x

getNumberOfXmlElements (_,x,_,_,_,_) = x

getNumberOfChars       (_,_,x,_,_,_) = x

getChildrenPAItems     (_,_,_,x,_,_) = x

getParentPAItem        (_,_,_,_,x,_) = x

getConnectedACs        (_,_,_,_,_,x) = x


getACIdentifier    (x,_,_,_,_) = x
```

```
getConnectedPAs    (_,x,_,_,_) = x

getParentACItems   (_,_,x,_,_) = x

getChildrenACItems (_,_,_,x,_) = x

getPredicateInfo   (_,_,_,_,x) = x


compareByConnectedPAs (a,b)
     | getNumberOfConnectedPAs(a) == getNumberOfConnectedPAs(b) =  0
compareByConnectedPAs (a,b)
     | getNumberOfConnectedPAs(a) <  getNumberOfConnectedPAs(b) = -1
compareByConnectedPAs (a,b)
     | getNumberOfConnectedPAs(a) >  getNumberOfConnectedPAs(b) =  1

getNumberOfConnectedPAs c = count getConnectedPAs(c)

count []     = 0
count (x:xs) = 1 + count (xs)

compareByACIdentifier (a,b)
     | getACIdentifier(a) == getACIdentifier(b) =  0
compareByACIdentifier (a,b)
     | getACIdentifier(a) <  getACIdentifier(b) = -1
compareByACIdentifier (a,b)
     | getACIdentifier(a) >  getACIdentifier(b) =  1

compareByConnectedACs (a,b)
     | getConnectedACs(a) == getConnectedACs(b) =  0
compareByConnectedACs (a,b)
     | getConnectedACs(a) <  getConnectedACs(b) = -1
compareByConnectedACs (a,b)
     | getConnectedACs(a) >  getConnectedACs(b) =  1

getConnectedACs (_,_,previousACs,nextACs,_) =
     count(previousACs) + count(nextACs)

compareByPredicateFulfillment (a,b) |
   getPredicateInfo(getFulfillment(a)) == getPredicateInfo(getFulfillment(b))
     =  0
compareByPredicateFulfillment (a,b) |
   getPredicateInfo(getFulfillment(a)) >  getPredicateInfo(getFulfillment(b))
     =  1
compareByPredicateFulfillment (a,b) |
   getPredicateInfo(getFulfillment(a)) <  getPredicateInfo(getFulfillment(b))
     = -1

getFulfillment(_,created,fulfilled,_,_,_,_) = fulfilled/created
```

```
compareByPredicateFailing (a,b) |
   getPredicateInfo(getFailing(a)) == getPredicateInfo(getFailing(b)) =  0
compareByPredicateFailing (a,b) |
   getPredicateInfo(getFailing(a)) >  getPredicateInfo(getFailing(b)) =  1
compareByPredicateFailing (a,b) |
   getPredicateInfo(getFailing(a)) <  getPredicateInfo(getFailing(b)) = -1

getFailing(_,created,_,failed,_,_,_) = failed/created

getPredicateNumber (x,_,_,_,_,_,_) = x
```

7 Conclusion

The contributions of this paper are the motivation for the need of approximate answers to XPath queries on XML streams and the desired characteristics of approximation languages in general and the approximation language SAL for the query evaluator SPEX in particular.

Future work
In the near future, experimental results of query evaluation under memory constraints are intended.
Also another view of approximation is explored: The idea is to take a look on the XPath query and the schema of an XML stream in order to decide weather approximation is necessary. If so, to try to change the schema in such a way, that the query can be evaluated without the need of approximation.

References

[1] D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, August 2003. http://www.cs.brown.edu/research/aurora/.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of 21st ACM Symposium of Principles of Database Systems (PODS)*, 2002.

[3] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. Xquery 1.0: An xml query language w3c working draft. August 2003. http://www.w3.org/TR/xquery/.

[4] T. Bray, J. Paoli, C. Sperberg-McQueen, and E. Maler. Extensible markup language (xml) 1.0 (second edition) w3c recommendation. October 2006. http://www.w3.org/TR/REC-xml.

[5] T. Bray, J. Paoli, C. Sperberg-McQueen, and E. Maler. Extensible markup language (xml) 1.0 (second edition) w3c recommendation - dtd. October 2006. http://www.w3.org/TR/REC-xml#dt-doctype.

[6] J. Clark and S. DeRose. Xml path language (xpath) version 1.0 w3c recommendation. November 1999. http://www.w3.org/TR/xpath.

[7] T. O. Group. The single unix® specification (version 2), ps - report process status. 1997. http://www.opengroup.org/onlinepubs/007908799/xcu/ps.html.

[8] T. S. Group. Stream: The stanford stream data manager. *IEEE Data engeneering Bulletin*, 26(1), 2003. http://www-db.stanford.edu/stream/.

[9] D. Maier. Database desiderata for an xml query language. *QL'98 - Query Languages 1998*, November 1998.

[10] J. M. Martínez. Overview of the mpeg-7 standard (version 5.0). March 2001. http://ipsi.fraunhofer.de/delite/Projects/MPEG7/Documents/w4031mpeg7overview.htm.

[11] H. Meuss and K. U. Schulz. Complete answer aggregates for tree-like databases: A novell approach to combine querying and navigation. *ACM Transactions on Information Systems (TOIS)*, 53(6), April 2001.

[12] D. Olteanu, T. Kiesling, and F. Bry. An evaluation of regular path expressions with qualifiers against xml streams. In *19th International Conference on Data Engineering (ICDE)*, 2003.

[13] D. Srivastava. Pix: Exact and approximate phrase matching in xml. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 2003.

[14] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, April 1997.