Using XSLT Stylesheets to Transform XPath Queries

Sven Groppe, Stefan Böttcher, Reiko Heckel, Georg Birkenheuer

University of Paderborn, Faculty 5, Fürstenallee 11,
D-33102 Paderborn, Germany
{sg, stb, reiko, birke}@uni-paderborn.de

Abstract. Often, XML documents stored in an XML database must be transformed by an XSL processor into a client-specific format before queries are submitted. In applications where XML documents are large and the query result is relatively small, it is of considerable advantage to retrieve and process only that part of the document, which is used by the query. Our contribution transforms an XPath query defined over the client's XML format into a query over the format used by the database, such that the evaluation of the latter query returns a reduced document containing only as much data as is needed to evaluate the original query.

1 Introduction

1.1 Problem Definition and Motivation

XSL [18] processors transform whole XML documents into another XML format according to XSLT stylesheets. Whenever applications use XPath [19] queries to retrieve a relevant section of a whole transformed XML document, we propose to transform only that section of the original XML document, which is necessary to answer the XPath query, instead of transforming the whole XML document (see Figure 1). For this purpose, we introduce a query transformation algorithm, which transforms a given XPath query XP_{transf} to an XPath query XP_{orig} according to a given XSLT stylesheet S. We then evaluate XP_{orig} on the original document D to retrieve a smaller resultant XML fragment $XP_{orig}(D)$, which is then transformed by an XSLT processor to $S(XP_{orig}(D))$ and at last queried according to the query XP_{transf} in order to retrieve the final result $XP_{transf}(S(XP_{orig}(D)))$. The *algorithmic problem of query transformation* to be solved by our proposed query transformation algorithm is to determine XP_{orig} according to a given XPath query XP_{transf} and an XSLT stylesheet S such that it meets the following conditions. The resultant XML fragment of $XP_{orig}(D)$ must be as small as possible, but must also guarantee the equivalence of $XP_{transf}(S(XP_{orig}(D)))$ and $XP_{transf}(S(D))$, i.e. both return the same result for every XML document D.

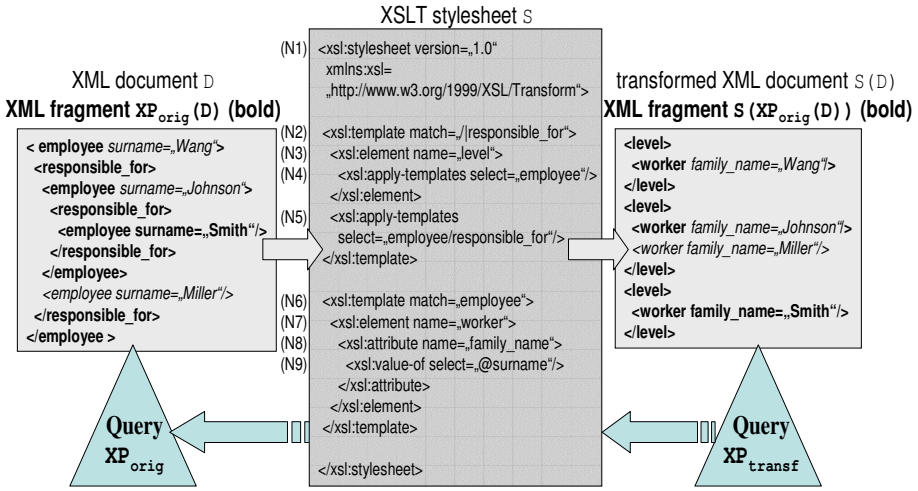


Figure 1. Example of the transformation from XML format F_{orig} into F_{transf} by an XSLT stylesheet

There are three main advantages of our approach in comparison to transforming the whole XML document.

First, we avoid the problem of data replication. Without using our approach, every application retrieves a whole replicated transformed XML document. Whenever there is a change in the original XML document and the applications have to work on the current XML data, the original document must be transformed and transmitted again to the applications. Synchronisation problems occur during the time, which is needed to transform the XML document. Within our approach, all queries will be directed to one single XML document and relevant sections will be transformed and transmitted to the applications on-demand, such that the applications work on the current data every time.

Second, a smaller data volume is transmitted in distributed scenarios, as the resultant XML fragment contains only the relevant data to answer a query and therefore, the resultant XML fragment is smaller than the whole XML document. This saves transportation costs, which has considerable advantages especially in distributed scenarios communicating via small bandwidth like WAP or mobile data clients, such as mobile phones or PDA's.

Third, as the CPU time required for an XSLT based transformation primarily depends on the amount of transformed XML data and our approach significantly reduces the amount of data to be transformed, our approach reduces the CPU load of those processors, which transform the XML document. Therefore, our approach has considerable advantages in comparison to transforming the whole XML document, whenever servers perform frequent modifications of fragments of the stored XML document or whenever distributed applications often query for the data stored on other servers. Without using our approach, all accessed servers or the clients themselves have to transform the whole XML document on every change into all specific XML formats for the clients. This causes a high CPU load on the servers and the clients, which is avoided by using our approach.

Whenever XML data is transformed according to an XSLT stylesheet, our approach can be used to enable on-demand transformations, especially our approach can be incorporated into XML databases. Furthermore, our approach can be included in relational databases, which are enabled to generate XML data from the relational data and then transform the generated XML data by an XSLT processor.

The remainder of this paper is organized as follows. The following subsection describes related work and subsets of XSLT and XPath for which our optimization applies. After introducing necessary terms in Section 2.1, Section 2.2 describes the transformation of an XSLT stylesheet into a graph and how this graph is used for the search of paths generating output that is relevant to a given query $\texttt{XP}_{\texttt{transf}}$ (Section 2.3). Section 2.4 outlines how the query $\texttt{XP}_{\texttt{orig}}$ is constructed from input nodes along the detected paths of the search described in Section 2.3. Finally, we summarize the results and draw the conclusions.

1.2 Relation to other Work and our Focus

A slightly different approach, called *query reformulation,* is known from relational databases. In comparison to our approach, applying the reformulated query transforms the retrieved data into the target format in one step [12]. Within our approach, we separate the transformation of XML data from applying the query in order to use standard XSL processors and standard XPath evaluators wherever possible. As within the approach of query reformulation in relational databases, our approach does not use a direct connection to a database for transforming the query.

For the transformation of XML queries into queries based upon other data storage formats, at least two major research directions can be distinguished. Firstly, the mapping of XML queries to object oriented or relational databases (e.g. [8], [12]), and secondly, the transformation of XML queries or XML documents into other XML queries or XML documents (e.g. [1]). We follow the second approach; however, we focus on XSL [18] for the transformation of both data and XPath [19] queries.

Within related contributions to schema integration, two approaches to data and query translation can be distinguished. While the majority of contributions (e.g. [2], [10]) map the data to a unique representation, we follow [9] and map the queries to those domains where the data resides.

The contribution in [11] contains query reformulation according to path-to-path mappings. We go beyond this, as we use XSLT as a more powerful mapping language. [17] describes how XSL processing can be incorporated into database engines, but focuses on efficient XSL processing. The complexity of XPath query evaluation on XML documents is examined in [13]. In comparison, we use an evaluation based on output nodes of XSLT and consider query transformation. Altinel and Franklin present in [3], an algorithm to filter XML documents according to a given query and analyze the performance, but the algorithm does not contain query transformation.

[16] projects XML documents to a sufficient XML fragment before processing XQuery queries. [16] contains a static path analysis of XQuery queries, which computes a set of projection paths formulated in XPath from an arbitrary XQuery expression. In comparison to this approach and among other things, we describe a path

analysis within XSLT stylesheets depending on an input XPath query. Furthermore, we analyze paths within recursive calls (of templates).

In contrast to all these approaches, we focus on the transformation of XPath queries according to an XSLT stylesheet.

Within this paper, we go beyond our previous contributions of [14] and [15], as we support a larger subset of XSLT and a larger subset of XPath for the XPath query transformation (i.e. most axes of XPath are now supported). We present our algorithm visually, and in a more systematic way through graph transformation rules.

1.3 Considered Subsets of XPath and XSLT

Due to space limitations with this paper, we restrict XPath queries XP_{transf}, such that they do not contain the axes `following`, `following-sibling`, `namespace`, `preceding` and `preceding-sibling` and they do not contain the node tests `comment()` and `processing-instruction()`. Filters must be always of the form `[@Attr="constant"]`, i.e. an attribute is tested against a constant value.

Similarly, we restrict XSLT, i.e., we consider the following nodes of an XSLT stylesheet only:

- `<xsl:stylesheet>`,
- `<xsl:template match=M name=N>`,
- `<xsl:element name=N>`,
- `<xsl:attribute name=N>`,
- `<xsl:apply-templates select=I>`,
- `<xsl:text>`,
- `<xsl:value-of select=I>`,
- `<xsl:for-each select=I>`,
- `<xsl:call-template name=N>`,
- `<xsl:attribute-set name=N>`,
- `<xsl:if test=T>`,
- `<xsl:choose>`,
- `<xsl:when test=T>`,
- `<xsl:otherwise>`,
- `<xsl:processing-instruction>`,
- `<xsl:comment>` and
- `<xsl:sort>`,

where `I` and `M` contain an XPath expression without function calls, `T` is a boolean expression, and `N` is a string constant.

Whenever attribute values are generated by the XSLT stylesheet, we assume (in order to keep this presentation simple) that this occurs in only one XSLT node (i.e. `<xsl:text>` or `<xsl:value-of select=I>`).

2 Query Transformation as a Search Problem in the Stylesheet Graph

Figure 1 shows an example of our approach: The XSLT stylesheet `S` transforms the representation of the hierarchy of a company (XML document `D`) into a flat model, i.e. into the transformed XML document `S(D)`. Assume that we must answer an XPath query

XP_{transf}`=/level/worker[@family_name="Smith"]`

on the transformed XML document S(D). It is sufficient to transform only a *resultant XML fragment* $XP_{orig}(D)$ (i.e. the bold face part of the left box of Figure 1) for answering XP_{transf}, where XP_{orig} is a query in the XML format F_{orig} of the original document D computed by our new query transformation algorithm.

Notice that standard XPath evaluators only return a query result as a node set, not as a resultant XML fragment. This *resultant XML fragment* $XP_{orig}(D)$ is defined to contain all nodes of the original XML document D, which contribute to the successful evaluation of the query XP_{orig} given in XML format F_{orig}, and all ancestors of these nodes up to the root.

In the example, it is sufficient to transform the bold face part of the left box in Figure 1 for answering XP_{transf}. The bold face part is the resultant XML fragment of the query

XP_{orig}=(/|//responsible_for)(/employee/responsible_for)*
/employee[@surname="Smith"]

where A* is a short notation for an arbitrary number of paths A.[1]

A query XP_{transf} only selects a certain part of the transformed XML document S(D) (i.e. $XP_{transf}(S(D))$), which is generated by the XSLT processor at certain *output nodes* of the XSLT stylesheet S. In the example of Figure 1, all the elements level in S(D) are generated by the node N3 of S (see Figure 1), and all the elements worker are generated by node N7. These output nodes N3 and N7, of the XSLT stylesheet S are reached after a sequence of nodes (which we call *path*) of S have been executed. In the example, one path from the root node of S to the nodes N3 and N7 is <N1,N2,N3,N4,N6,N7>. While executing these paths, the XSLT processor also processes *input nodes* (e.g. node N4), each of which selects a certain node set of the input XML document D. When considering all executed input nodes of a path, the input nodes altogether select a whole node set of the input XML document D. In the path <N1,N2,N3,N4,N6,N7>, this whole node set can be described using the query /employee. The determination of the whole node set (described using a query XP_{orig}), which is selected on *any* paths of the XSLT stylesheet S which generate output relevant to the query XP_{transf}, has the following advantage: we then can select a *smaller*, but *sufficient* part $XP_{orig}(D)$ of the input XML document D, where the transformed XML fragment $S(XP_{orig}(D))$ contains all the information required to answer the query XP_{transf} correctly, i.e. $XP_{transf}(S(XP_{orig}(D)))$ is equivalent to $XP_{transf}(S(D))$.

The first step (described in sections 2.1. to 2.3.) involves the transformation of the XSLT stylesheet into an equivalent stylesheet graph. This stylesheet graph is used to search for the elements, attributes and attribute values generated by the XSLT

[1] Notice that although standard XPath evaluators do not support A*, we can retrieve a superset of A* by replacing A*/ with //.

stylesheet, and which are needed to answer the XPath query XP_{transf}. The search itself is described by means of attributed graph transformation rules (see [6]).

Within the second step (to be delayed until Section 2.4), we determine the transformed query XP_{orig} of the XSLT stylesheet. The query XP_{orig} summarizes the XPath expressions which are applied to the input XML document along all paths of the XSLT stylesheet where the search of the first step was successful.

2.1 Absolute and Relative Parts of XPath Expressions

For the computation of the input query XP_{orig} and for the construction of the stylesheet graph, we use the concepts of *relative* and *absolute parts* of an XPath expression, which are defined as follows.

Definition: An XPath expression `I` can be divided into a *relative part* `rp(I)` and an *absolute part* `ap(I)` (both of which may be empty) in such a way that `rp(I)` contains a relative path expression, `ap(I)` contains an absolute path expression, and the union of `ap(I)` and `rp(I)` is equivalent to `I`.
Example: The relative part of `I=(/E1|E2/E3|E4)/E5` is `rp(I)=(E2/E3|E4)/E5`, the absolute part is `ap(I)=/E1/E5`.

2.2 Stylesheet Graph

In order to compute the node set of the input XML document that is relevant to the query XP_{transf}, we represent an XSLT stylesheet (e.g., that of Figure 1) by a typed attributed graph (see [6]). We visualize the typed attributed graph as an object diagram in the notation of the Unified Modeling Language (UML), as shown in Figure 3. Figure 2 shows the class diagram defining the types of vertices and edges for the graph and the graph transformation rules discussed later. In brief, edges with diamond-shaped arrow heads represent composition (physical containment) relations. `Node` is a type of node, which takes up the content of nodes of the XSLT stylesheet. `Graph` objects are used by the output path search (see Section 2.3). A `Graph` can be nested, where the containment association represents nesting. Furthermore, `Graph` objects can only be associated with zero or one `Node`. Besides the containment association between `Graph` objects, we define the `contains` relation to be the transitive closure of the containment association. The class `IPE` is used while computing input path expressions (see Section 2.4).

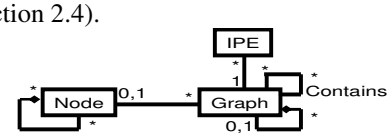


Figure 2. Class diagram without attributes

The following rules transform an XSLT stylesheet into the corresponding stylesheet graph:

a. For each node in the XSLT stylesheet, we insert a separate object of class `Node` into the stylesheet graph (In the example, the names `Ni` of the inserted objects `Ni:Node` in the stylesheet graph of Figure 3 correspond to the labels of the nodes in the XSLT stylesheet of Figure 1). For each attribute A within the node in the XSLT stylesheet, we insert an attribute into the stylesheet graph with the same name and value as A. In order to keep all the information of a node in the XSLT stylesheet, we store the tag name of the node in the XSLT stylesheet in an attribute `tag` in the stylesheet graph (For example, for the node `<xsl:template match="/|responsible_for">` in the XSLT stylesheet we insert an object `N2:Node` with attribute `match=/|responsible_for` and `tag=template` in the stylesheet graph).

b. The object `N1` in the stylesheet graph that corresponds to the node `<xsl:stylesheet>` of the XSLT stylesheet is the *start node* of the stylesheet graph. Therefore, we insert an attribute `start = true` in `N1` (see Figure 3).

c. For each node in the stylesheet graph, we check whether or not the node belongs to the so called *normal nodes*, to the *output nodes* or to the *input nodes*:
   1) A node in the stylesheet graph belongs to the *output nodes*, if the corresponding node in the XSLT stylesheet generates an element `E` (`<xsl:element name=E>`), generates an attribute `@A` (`<xsl:attribute name=A>`) or generates a text node (`<xsl:text>content</xsl:text>`). We assign the value `output` to the attribute `type` in every output node of the stylesheet graph. When a text node is generated in the output node, we assign the value `content` to the attribute `value` in the node of the stylesheet graph.
   2) A node in the stylesheet graph belongs to the *input nodes*, if the corresponding node in the XSLT stylesheet selects a node set `I` of the input XML document. This is the case for `<xsl:apply-templates select=I/>`, `<xsl:value-of select=I/>`, `<xsl:for-each select=I>`, and for `<xsl:if test=T>` and `<xsl:when test=T>`, when `I` occurs in a boolean expression `T`. Then, we copy `I` to the attribute `select` and assign the value `input` to the attribute `type` in the node of the stylesheet graph.
   3) Otherwise we assign the value `normal` to the attribute `type` in the node of the stylesheet graph.

d. Let `n1` and `n2` be the nodes in the stylesheet graph which correspond to the nodes `S1` and `S2` in the XSLT stylesheet. `n1` contains `n2` (visually represented by a composition relation), if
   1) `S2` is a child node of `S1` within the XSLT stylesheet, or
   2) `S1` is a node `<xsl:call-template name=N>` and `S2` is a node `<xsl:template name=N>` with an attribute `name` set to the same `N`, or
   3) `S1` is a node with an attribute `xsl:use-attribute-sets=N` and `S2` is a node `<xsl:attribute-set name=N>` with an attribute `name` set to the same `N`, or
   4) `S1` is a node `<xsl:apply-templates select=I/>` and `S2` is a

node <xsl:template match=M>, and the template of S2 can possibly be called from the selected node set I. This is the case, if ap(I)|//rp(I) and ap(M)|//rp(M) are possibly not disjointed, which can be checked by a fast (but incomplete) tester (e.g. the tester presented in [7]).

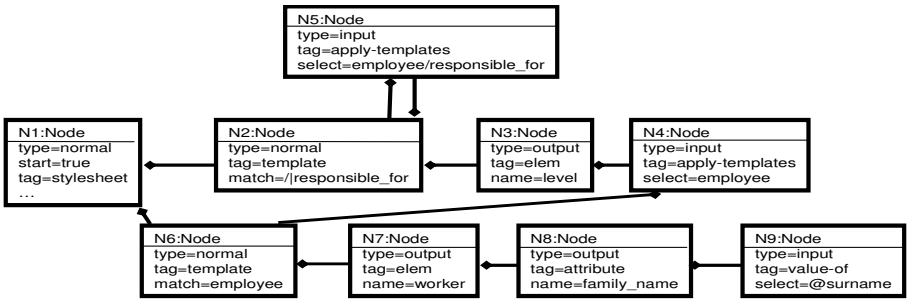


Figure 3. Stylesheet graph of the XSLT stylesheet of Figure 2

2.3 Output Path Search in the Stylesheet Graph

In this step of the query transformation algorithm, we search for all of the output nodes of the stylesheet graph, which may contribute towards the answer to the query XP_{transf}. For example, for XP_{transf}=/level/worker[@family_name="Smith"] and the XSLT stylesheet of Figure 1 (or its stylesheet graph shown in Figure 3, respectively), we search for the output nodes which generate the elements (firstly level, then worker), the attributes (@family_name of the element worker), and the attribute values (the value of @family_name of the element worker).

The search is described in terms of graph transformation rules within Appendix A. The rules are parameterized and must be instantiated according to the query XP_{transf}. For this purpose, we initialize a token counter i with 0 and then parse XP_{transf} from left to right. Depending on the currently parsed token according to the table in Figure 4, we increment a token counter i and instantiate the graph transformation rules. The *last parsed token number* is defined to be the last value of i after parsing the complete XP_{transf}.

For example, for XP_{transf}=/level/worker[@family_name="Smith"], we obtain the parsed token intervals shown in Figure 6. We look up each parsed token in Figure 4 in order to determine the graph transformation rules within Appendix A, which have to be applied. In this case, we instantiate the following graph transformation rules (cf. Figure 4 and Appendix A):

Step(1), Go(1)	(for "/"),
Element(2, "level")	(for "level"),
Step(3), Go(3)	(for "/"),

Element(4, “worker”) (for “`worker`”),
Go(4), Attribute(6, “family_name”), Go(5), ValueSource(6), ValueText(6), FilterSource(6), FilterContent(6) (for “`[@family_name='Smith']`”).

The last parsed token number is 6 in this example.

Parsed Token	Increment i	Graph Transformation Rules to instantiate
Location Step		
/	1	Step(i), Go(i)
Axes		
ancestor	1	Parent(i), Ancestor(i)
ancestor-or-self	1	AncestorOrSelf(i)
attribute	2	Attribute(i,_), Go(i-1), ValueSource(i), ValueText(i), AttributeAndValue(i,_)
child	1	child(i)
descendant	1	child(i), Go(i), Descendant(i)
descendant-or-self	1	Step(i), Go(i), Descendant(i)
parent	1	Parent(i)
self	1	Step(i)
Node Tests		
node()	1	Step(i)
*	1	Step(i)
name	1	NameNodeTest(i,name)
text()	1	Text(i)
Abbreviated Syntax		
//	1	Step(i), Go(i), Descendant(i)
Elem	1	Element(i,Elem)
*	1	Element(i,_)
@Attr	2	Attribute(i,Attr), Go(i-1) ValueSource(i), ValueText(i), AttributeAndValue(i,Attr)
@*	2	Attribute(i,_), Go (i-1), ValueSource(i), ValueText(i) , AttributeAndValue(i,_)
.	1	Step(i)
..	1	Parent(i)
Filter		
[@Attr=const]	2	Go(i-2), Attribute(i,Attr), Go(i-1), ValueSource(i), ValueText(i), FilterSource(i,Attr,const), FilterConstant(i,Attr,const)

Figure 4. Table of graph transformation rules to instantiate for a parsed token. The underscore (“_”) is a dummy for an unspecified value.

After this instantiation, which encodes the necessary control of the transformation process, the graph transformation rules are applied freely, and for as long as possible. The rules replace an occurrence of their left-hand side pattern with a copy of their right-hand side within each application. The same happens to all rules without any parameter to be instantiated, like Init(), PassNode(), ContainsBasic(), ContainsRecur-

siv() and Loop(). The search for XP_{transf} within the stylesheet graph is executed by applying all instantiated graph transformation rules.

The search within the stylesheet graph of the running example is shown in Figure 5, and starts with the node G1 of type :Graph. The search generates the nodes G01, G2, …, G11. The labels L0, L1, …, L6 within the nodes of type :Graph represent the token of XP_{transf}, which is currently recognized (cf. Figure 6). We call the output of a successful search, where the complete XP_{transf} has been recognized, a *detected path*. The detected path is described by a sequence of :Graph objects. Within the running example, the detected path is G0, G1, G2, G4, G5, G6, G7, G8 and G11 with an attached path G3 containing a loop and an attached path G9, G10, where the filter [@family_name='Smith'] has been detected (cf. Figure 5).

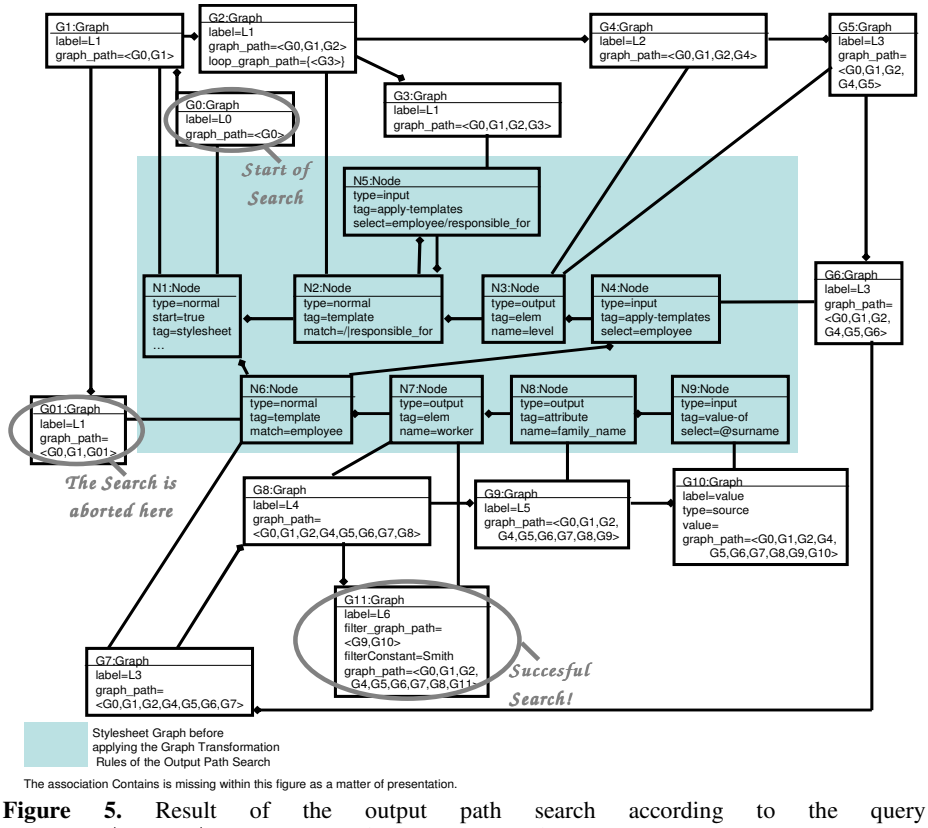


Figure 5. Result of the output path search according to the query XP_{transf}=/level/worker[@family_name="Smith"]

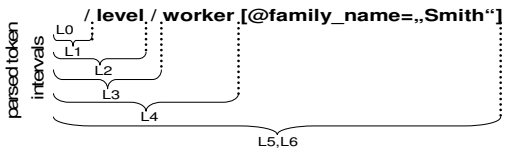


Figure 6. Intervals of parsed tokens.

2.4 Computing Input Path Expressions

While executing the detected paths computed in Section 2.3, the XSLT processor also processes *input nodes* (e.g. node N4 in Figure 1 and Figure 5). Each input node selects a certain node set of the input XML document D. The node set is described by an XPath expression within the attribute select of the input node, which we call a *local input path expression* I.

As outlined in Section 2, we have to combine all the local input path expressions of input nodes along a detected path. For this purpose, we use two different attributes in nodes of class IPE in the stylesheet graph:

The *current input path expression* (current ipe) contains the whole input path expression of the detected path down to (and including) the current XSLT node. We guarantee that the XSLT processor processes the current XSLT node with a subset of the XML nodes of the original XML document described by current ipe, for each step of the detected path.

The *completed input path expression* (completed ipe) contains all such input path expressions, which are selected within the detected path before the current node, but which will not be used further in the computation of a current ipe.

The different combination steps for the input path expressions of current ipe and completed ipe are described in the graph transformation rules of Appendix B. Figure 7 shows the stylesheet graph of the running example after applying the graph transformation rules for computing the input path expressions.

The completed ipe is always initialized with the empty set. For the example within Figure 7, the current ipe is initialized with /|//responsible_for. In general, the XSLT processor starts executing the detected path with the node set described by the match attribute M of the first template <xsl:template match=M> within the detected path. The template can match nodes of the node set rp(M) occurring within an arbitrary depth of the XML document because of built-in templates. Therefore, we initialize current ipe with ap(M)|//rp(M) (see InitCollectInputPath(n) in Appendix B where n is the parsed token number).

The rules Pass() and PassOldInput() of Appendix B, pass situations where the input path expressions remain unchanged. Basic() describes the basic computation step, where the next XSLT node is an input node. The input path expressions of attached paths are computed according to the type of the attached path by the graph transformation rules InitBranch(), CollectBranch(), InitFilter(), CollectFilter(), InitLoop() and CollectLoop().

The complete input path expression, which is used as query XP_{orig} on the input XML document, is the union of all the completed ipes and the current ipes of the last node of each of the k detected paths (1..k),

$$XP_{orig} = \text{completed ipe}_1 \mid \text{current ipe}_1 \; |\ldots| \; \text{completed ipe}_k \mid \text{current ipe}_k.$$

If there is no detected path (i.e. k=0), then XP_{orig} is empty.

In the example of Figure 7, we have `k=1` and the `completed ipe` and `current ipe` of the detected path (`k=1`) is stored in `I13:IPE`. Therefore, XP_{orig} is `(/|//responsible_for)(/employee/responsible_for)* /employee[@surname="Smith"]`.

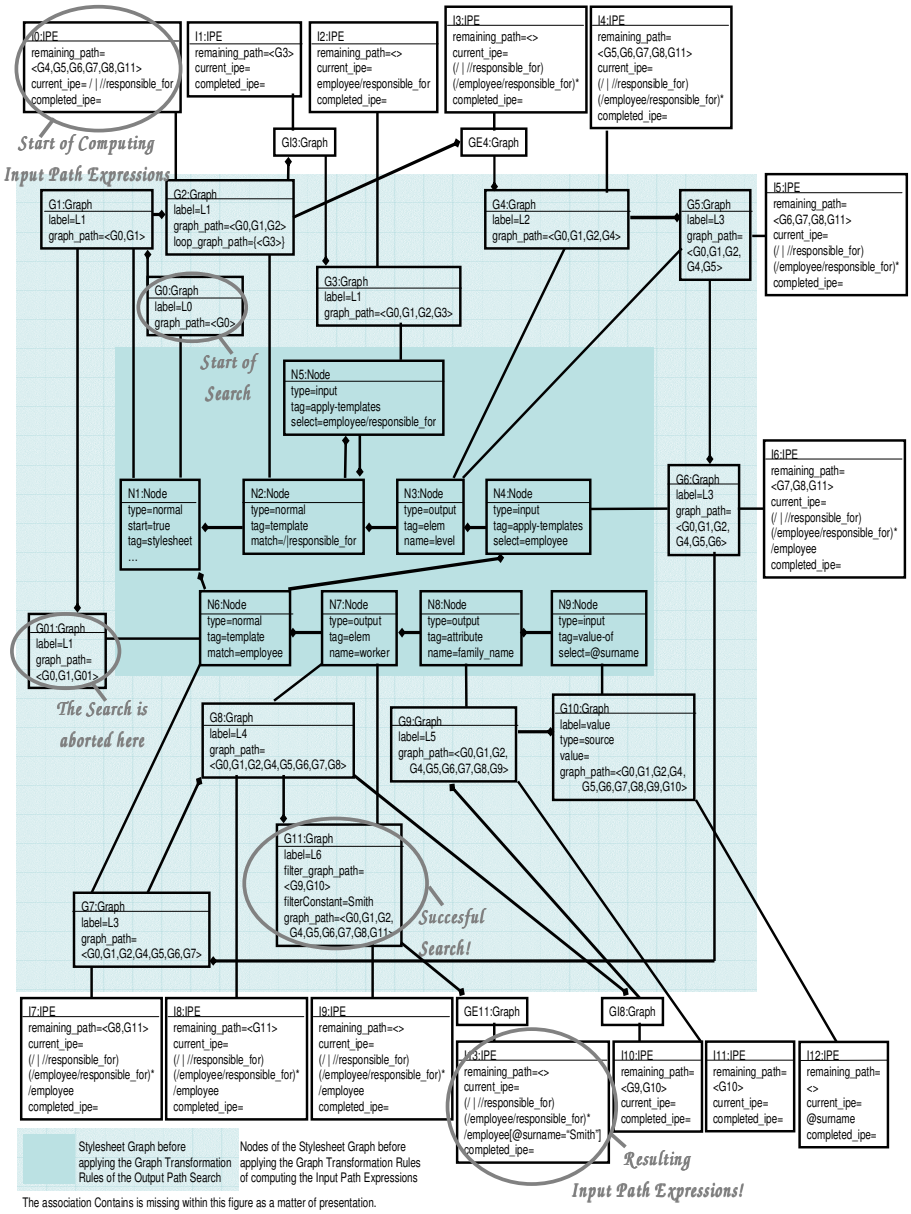


Figure 7. Stylesheet graph after applying the graph transformation rules for computing the input path expressions

2.5 Performance Analysis

We have implemented two prototypes. The first prototype applies the graph transformation rules of Appendix A and Appendix B directly. Within another optimized prototype, we have implemented a direct depth-first search for a smaller subset of XPath in order to optimize the speed of the transformation process. Within [15], we present the results of the experiments of the optimized prototype in comparison to the standard approach, which transforms the entire XML document in order to answer a query. The experimental evaluation shows that our approach to queries on transformed XML data has considerable advantages when using queries where the selectivity is not too big, and for queries on XML documents that are not too small. Furthermore, we have shown that our approach is scalable and becomes more efficient for larger XML documents. Within an XML DBMS, which has to support XSL processing, the use of our approach can be switched on and off depending on the file size of the original XML document, and estimations of selectivity of the transformed query.

3 Summary and Conclusions

Whenever XML data D given in an XML format F_{orig} can be transformed by an XSLT stylesheet S into an XML format F_{transf}, and a query expressed in terms of format F_{transf} has to be applied, our goals are as follows: to avoid replicas, to reduce the processing costs for document transformation by an XSLT processor and to reduce data transmission costs in distributed scenarios.

Within our approach, we transform a given query XP_{transf} into a query XP_{orig}, by using a given XSLT stylesheet S. XP_{orig} can be applied to the input XML document D in order to retrieve a smaller fragment $XP_{orig}(D)$ which contains all the relevant data. $XP_{orig}(D)$ can be transformed by the XSLT stylesheet into $S(XP_{orig}(D))$, from which the query XP_{transf} selects the relevant data.

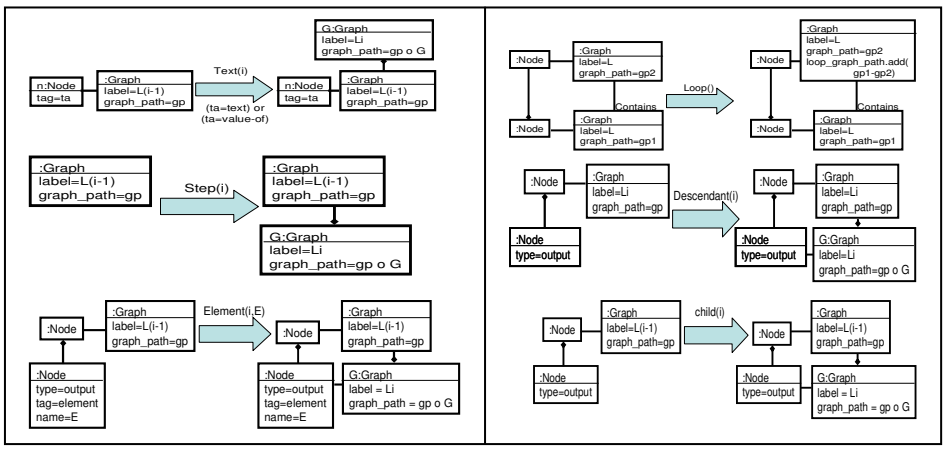
However, the approach is not limited to the given subsets of XSLT and XPath, and we consider it to be promising to extend it to a larger subset of XPath and XSLT.

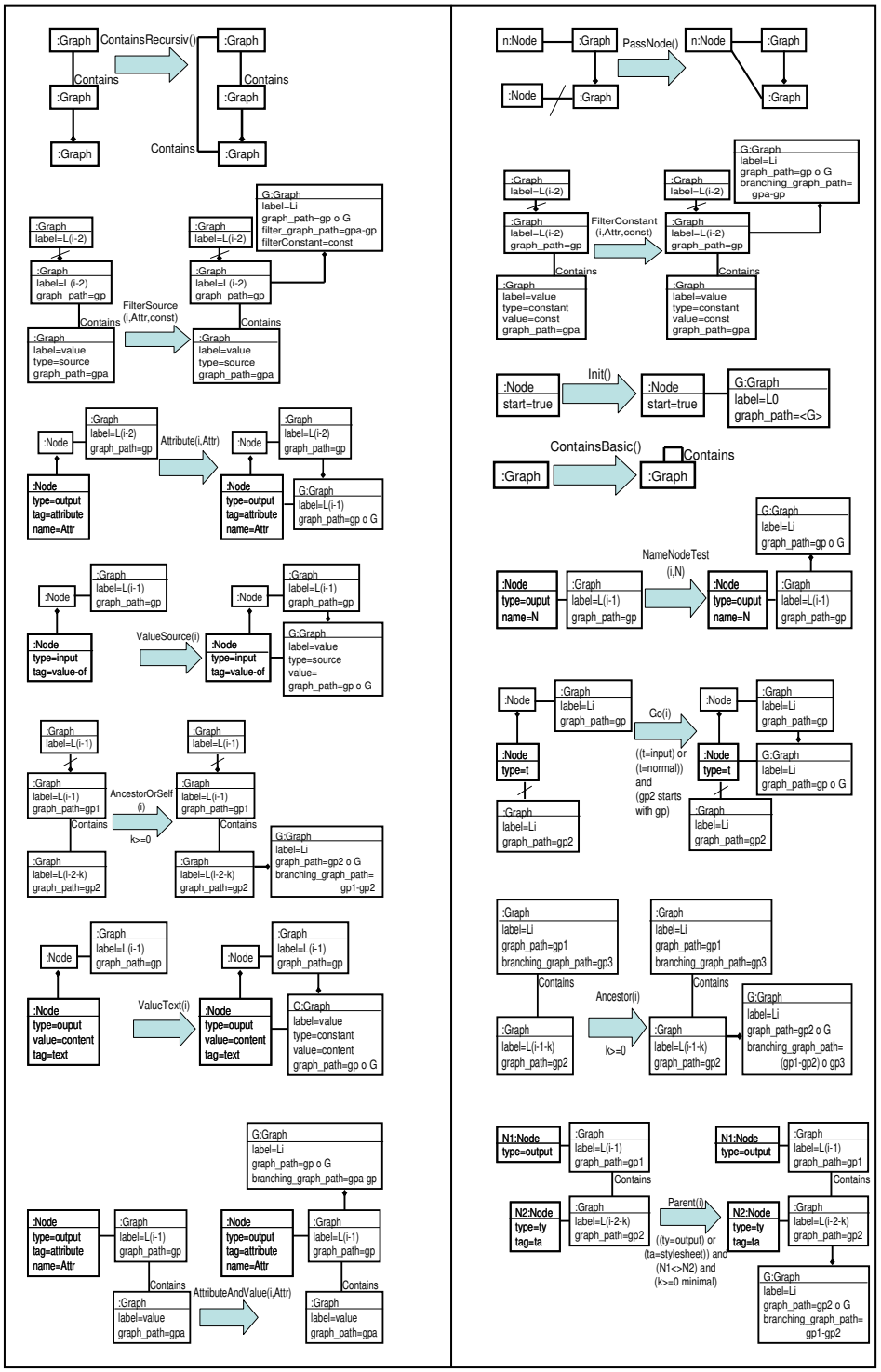
4 References

1. S. Abiteboul, On views and XML. In *PODS*, pages 1-9, 1999.
2. S. Abiteboul, S. Cluet, and T. Milo, Correspondence and translation for heterogeneous data. In *Proc. of the 6th ICDT,* 1997.
3. Altinel, M., and Franklin, M. J., Efficient Filtering of XML documents for Selective Dissemination of Information, In *Proceedings of 26th International Conference on Very Large Databases,* Cairo, Egypt, 2000.
4. Apache Software Foundation, Xalan-Java, http://xml.apache.org/xalan-j/index.html, 2003.
5. Apache Software Foundation, 2003. Xerces2 Java Parser 2.5.0 Release, http://xml.apache.org/xerces2-j, 2003.
6. L. Baresi, and R. Heckel, Tutorial Introduction to Graph Transformation: A Software Engineering Perspective, In Corradini, A.,H. Ehrig, H.-J. Kreowski and G. Rozenberg (Edi-

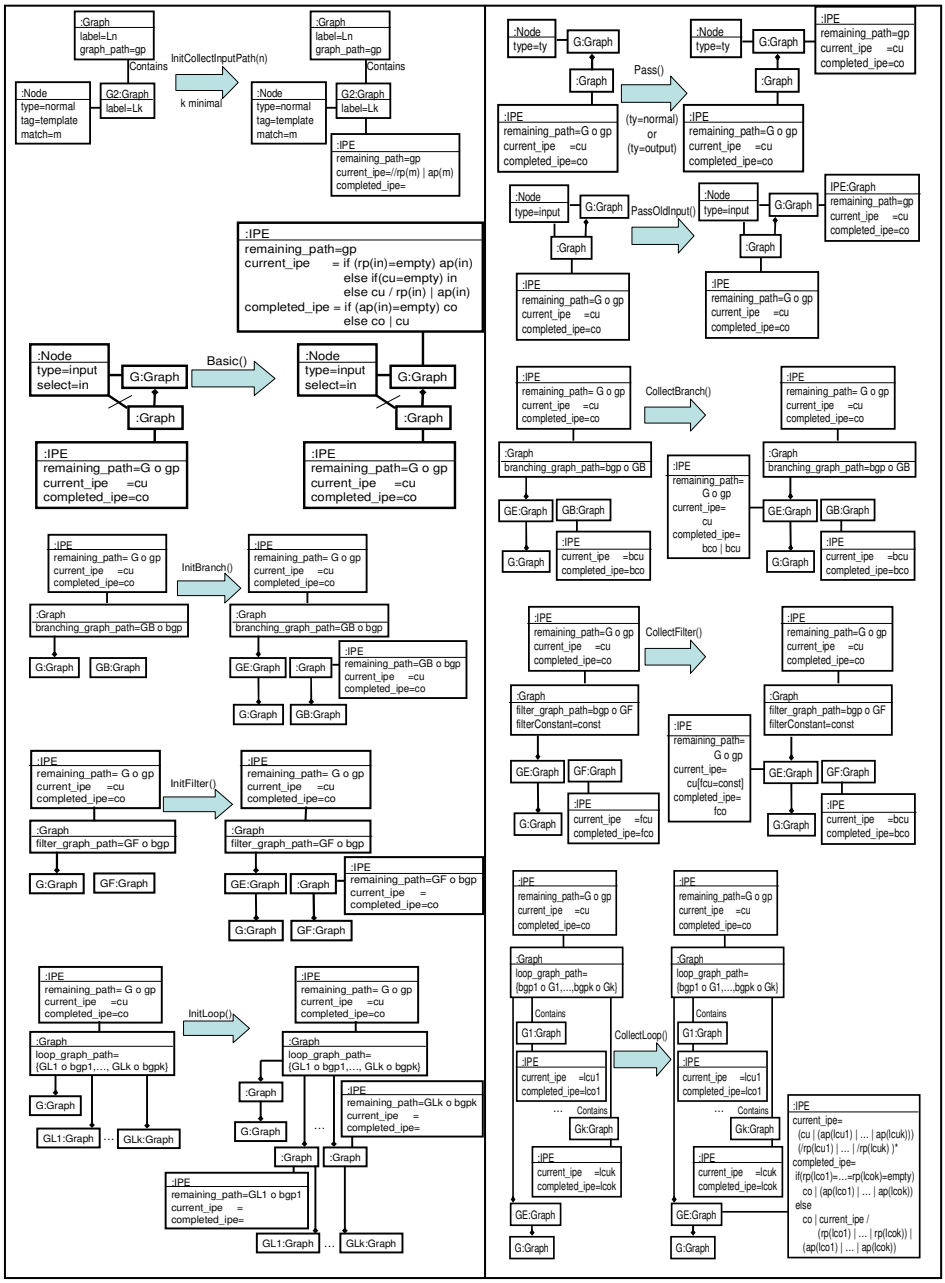
toren): *Proc. 1st Int. Conference on Graph Transformation (ICGT 02)*, Barcelona, Spain, Volume 2505 of Lecture Notes in Comp. Science. Springer-Verlag, Oktober 2002.
7. S. Böttcher, and A. Türling, Checking XPath Expressions for Synchronization, Access Control and Reuse of Query Results on Mobile Clients. *Workshop: Database Mechanisms for Mobile Applications*, Karlsruhe, Germany, 2003.
8. R. Bourret, C. Bornhövd, and A.P. Buchmann, A Generic Load/Extract Utility for Data Transfer Between XML Documents and Relational Databases. *2nd Int. Workshop on Advanced Issues of EC and Web-based Information Systems (WECWIS)*, San Jose, California, 2000.
9. C.-C. K. Chang, and H. Garcia-Molina, Approximate Query Translation Across Heterogeneous Information Sources. *VLDB 2000, 2000.*
10. S. Cluet, C. Delobel, J. Simon, and K. Smaga, Your mediators need data conversion! In *Proc. of the 1998 ACM SIGMOD Conf., 1998.*
11. S. Cluet, P. Veltri, and D. Vodislav, Views in a Large Scale XML Repository. In *Proceedings of the 27th VLDB Conference*, Roma, Italy, 2001.
12. Deutsch, A., and Tannen, V., Reformulation of XML Queries and Constraints, In *ICDT 2003*, LNCS 2572, pp. 225-241, 2003.
13. Gottlob, G., Koch, C., and Pichler, R., The Complexity of XPath Query Evaluation, In *Proceedings of the 22th ACM SIGMOD-SIGACT-SIGART symposium of Principles of database systems (PODS 2003)*, San Diego, California, USA, 2003.
14. S. Groppe, and S. Böttcher, XPath Query Transformation based on XSLT stylesheets, *Fifth International Workshop on Web Information and Data Management (WIDM'03)*, New Orleans, Louisiana, USA, 2003.
15. Groppe, S., Böttcher, S., and Birkenheuer, G., Efficient Querying of transformed XML documents, *6th International Conference of Enterprise Information Systems (ICEIS 2004)*, Porto, Portugal, 2004.
16. Marian, A., and Siméon, J., Projecting XML Documents. In *Proceedings of the 29th VLDB Conference*, Berlin, Germany, 2003.
17. G. Moerkotte, Incorporating XSL Processing Into Database Engines. In *Proceedings of the 28th VLDB Conference*, Hong Kong, China, 2002.
18. W3C, Extensible Stylesheet Language (XSL). http://www.w3.org/Style/XSL/, 2001.
19. W3C, XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath/, 1999.

Appendix A(Graph Transformation Rules for Output Path Search)




:Graph
ContainsRecursiv()
Contains
:Graph
:Graph
:Graph
Contains
Contains
:Graph
:Graph
Contains
G:Graph
label=Li
graph_path=gp o G
filter_graph_path=gpa-gp
filterConstant=const
:Graph
label=L(i-2)
graph_path=gp
:Graph
label=L(i-2)
graph_path=gp
:Graph
label=L(i-2)
graph_path=gp
:Graph
label=L(i-2)
graph_path=gp
Contains
Contains
FilterSource
(i,Attr,const)
:Graph
label=value
type=source
graph_path=gpa
:Graph
label=value
type=source
graph_path=gpa
:Node
:Graph
label=L(i-2)
graph_path=gp
Attribute(i,Attr)
:Node
:Graph
label=L(i-2)
graph_path=gp
:Node
type=output
tag=attribute
name=Attr
:Node
type=output
tag=attribute
name=Attr
G:Graph
label=Li
graph_path=gp o G
:Node
:Graph
label=L(i-1)
graph_path=gp
:Node
:Graph
label=L(i-1)
graph_path=gp
:Node
type=input
tag=value-of
ValueSource(i)
:Node
type=input
tag=value-of
G:Graph
label=value
type=source
value=
graph_path=gp o G
:Graph
label=L(i-1)
:Graph
label=L(i-1)
:Graph
label=L(i-1)
graph_path=gp1
AncestorOrSelf
(i)
k>=0
:Graph
label=L(i-1)
graph_path=gp1
Contains
Contains
:Graph
label=L(i-2-k)
graph_path=gp2
:Graph
label=L(i-2-k)
graph_path=gp2
G:Graph
label=Li
graph_path=gp2 o G
branching_graph_path=
gp1-gp2
:Node
:Graph
label=L(i-1)
graph_path=gp
:Node
:Graph
label=L(i-1)
graph_path=gp
:Node
type=ouput
value=content
tag=text
ValueText(i)
:Node
type=ouput
value=content
tag=text
G:Graph
label=value
type=constant
value=content
graph_path=gp o G
G:Graph
label=Li
graph_path=gp o G
branching_graph_path=gpa-gp
:Node
type=output
tag=attribute
name=Attr
:Graph
label=L(i-1)
graph_path=gp
:Node
type=output
tag=attribute
name=Attr
:Graph
label=L(i-1)
graph_path=gp
Contains
Contains
:Graph
label=value
graph_path=gpa
AttributeAndValue(i,Attr)
:Graph
label=value
graph_path=gpa
n:Node
:Graph
PassNode()
n:Node
:Graph
:Node
:Graph
:Graph
:Graph
G:Graph
label=Li
graph_path=gp o G
branching_graph_path=
gpa-gp
:Graph
label=L(i-2)
:Graph
label=L(i-2)
:Graph
label=L(i-2)
graph_path=gp
FilterConstant
(i,Attr,const)
:Graph
label=L(i-2)
graph_path=gp
Contains
Contains
:Graph
label=value
type=constant
value=const
graph_path=gpa
:Graph
label=value
type=constant
value=const
graph_path=gpa
:Node
start=true
Init()
:Node
start=true
G:Graph
label=L0
graph_path=<G>
ContainsBasic()
Contains
:Graph
:Graph
G:Graph
label=Li
graph_path=gp o G
NameNodeTest
(i,N)
:Node
type=ouput
name=N
:Graph
label=L(i-1)
graph_path=gp
:Node
type=ouput
name=N
:Graph
label=L(i-1)
graph_path=gp
:Node
:Graph
label=Li
graph_path=gp
Go(i)
:Node
type=t
:Graph
label=Li
graph_path=gp2
((t=input) or
(t=normal))
and
(gp2 starts
with gp)
:Node
type=t
:Graph
label=Li
graph_path=gp
G:Graph
label=Li
graph_path=gp o G
:Graph
label=Li
graph_path=gp2
:Graph
label=Li
graph_path=gp1
branching_graph_path=gp3
:Graph
label=Li
graph_path=gp1
branching_graph_path=gp3
Contains
Contains
Ancestor(i)
:Graph
label=L(i-1-k)
graph_path=gp2
k>=0
:Graph
label=L(i-1-k)
graph_path=gp2
G:Graph
label=Li
graph_path=gp2 o G
branching_graph_path=
(gp1-gp2) o gp3
N1:Node
type=output
:Graph
label=L(i-1)
graph_path=gp1
N1:Node
type=output
:Graph
label=L(i-1)
graph_path=gp1
Contains
Contains
N2:Node
type=ty
tag=ta
:Graph
label=L(i-2-k)
graph_path=gp2
Parent(i,j)
((ty=output) or
(ta=stylesheet)) and
(N1<>N2) and
(k>=0 minimal)
N2:Node
type=ty
tag=ta
:Graph
label=L(i-2-k)
graph_path=gp2
G:Graph
label=Li
graph_path=gp2 o G
branching_graph_path=
gp1-gp2

Appendix B (Graph Transformation Rules for Computing Input Path Expressions[2])



[2] The analogous special cases with attribute remaining_path=<> and missing node `G:Graph` for InitBranch(), CollectBranch(), InitFilter(), CollectFilter, InitLoop() and CollectLoop() are not listed here. InitBranch(), InitFilter() and InitLoop() must be applied before the other rules.