XML to XML through XML

Pim Lemmens, Geert-Jan Houben

Eindhoven University of Technology
Department of Computer Science
PO Box 513, 5600 MB Eindhoven, the Netherlands

E-mail: {W.J.M.Lemmens, G.J.Houben}@tue.nl

Abstract

XML documents are used to exchange data. Data exchange implies the transformation of the original data to a different structure. Often such transformations need to be adapted to some specific situation, like the rendering to non-standard platforms for display or the support of special user preferences. Adaptations are, in fact, transformations of transformations. This report describes a system for specifying XML transformations that allows the transformations themselves to be transformed in the same way as the original data.

XML to XML through XML 1

Abstract 1

1 Introduction 3

2 XML transformations 4

3 Specifying transformations 6

3.1 Templates and Parameters 6

3.2 Input driven specifications 7

3.3 Output driven specifications 8

4 Components of a transformation specification 9

4.1 Filtering 9

4.2 The transformation specification 11

4.3 List of HTL primitives 13

4.4 HTL Templates 14

5 Implementation issues 15

6 Conclusion 16

7 References 17

Appendix A. HTL syntax 18

Appendix B. HTL semantics 24

1 Introduction

XML is rapidly changing the way the World Wide Web is used. The quarterly reports issued by XML.org [ZapThink, LLC] nowadays contain information on more than 400 industry standards in this area. As more and more web data appear in the form of XML documents the need is felt to have adequate mechanisms to adapt such documents to different circumstances. A user might want to extract some relevant data from a document that contains far more data than she can use. Another user may want to check whether some prospective document does indeed contain the information he requires. These uses need some filtering of the available documents. In some cases it may be necessary to change the contents of a document, e.g. by modifying some element or attribute values to satisfy certain constraints imposed by the user, the platform used or the author of the system using the documents. One would for example like to replace prices in dollars by prices in some other currency. The best known use of transformations of XML, however, is to make the data suitable for presentation, e.g. within a web browser. The XML transformation language XSL(T) [W3C XSL working group, 1999a] was originally developed specifically for this purpose.

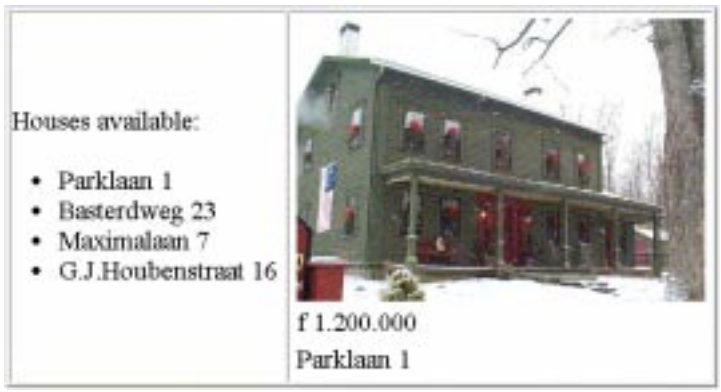

Figure 1

As an example of the use of transformations of XML documents, consider a web site of a real estate agency (Figure 1). This web site contains a number of HTML forms that allow users to formulate their wishes. For example, one form offers a selection of regions with houses for sale, and price ranges for the available houses. The regions and prices a user has chosen are translated into a query for data representing suitable houses from the agency's database. The processing of this query amounts to applying some filter to an extensive data collection. Also, maps of the relevant regions are retrieved from a site offering regional maps. Here a document has to be searched from among a large number of available documents, which means conceptually that all elements of the set of maps should be tested for compliance to certain criteria, i.e. (a description of) the contents of a document leads to a 'yes' or 'no' decision on whether to include the document in the result. Next, the data retrieved by the user query may be of many different kinds, texts as well as pictures or video clips and sound bites. Not all platforms, however, are able to deal with all these different kinds of data, so the query should be modified (transformed) in accordance with the platform used. Also, the retrieved data should be made accessible in a user friendly way: if the set of retrieved houses is large the set could be presented as an index of houses to choose from, perhaps in the form of a panel of thumbnail images, each with a link to the data of the associated house. In the end the data on the separate houses should be presented in a specific layout or be made accessible through specific navigation mechanisms. In HTML terms this implies the arrangement of data into tables, lists and links, using anchors for navigation. This amounts to yet another transformation. So, for this relatively simple example already four kinds of document transformations are required.

This report introduces a method to specify transformations of XML documents, in which the form of the specification corresponds to the form of the document concerned, i.e. the schema specification has the same layout as its instantiation in the document. This isomorphy of transformation specification to the documents themselves allows the transformations themselves to be transformed easily. Thus the same tool may be used for adaptation of queries and style sheets to specific user and platform profiles as well as for filtering the available data and testing the documents for their compliance to the specified

criteria and the integration of data from different sources and the styling of the output documents. In this report we introduce the language HTL that we devised for this purpose.

2 XML transformations

XML documents contain trees of XML nodes, which come in three classes: elements, attributes and texts. Each document has one, and only one root node, of which all other nodes are descendants. Element nodes have a name, zero or more attributes and contents, which may be empty or consist of any number of element or text children. Attribute nodes have a name and a value, which is a string of characters. They always are children of element nodes but cannot have children themselves. Finally, text nodes only have a value, which is a character string, and no children. There are a number of special attributes, among which those named 'ID' or 'IDREF', which may be used to form links between elements. Any element may have an 'ID' attribute - which should have a value which is unique among 'ID' attributes - and any number of 'IDREF' attributes, whereby it will link to elements with 'ID' attributes equal to the values of those 'IDREF' attributes.

The database that contains the data on the houses the real estate agent has for sale may be some XML file, part of which may look like:

```
<houseDB>
  ..................
  <house ID="12345">
    <picture src="http://www.somehost.com/house123.jpg"/>
    <price currency="dollars">120,000</price>
    <street>Basterdweg</street>
    <houseNr>23</houseNr>
    <city>Eindhoven</city>
  </house>
  ......................
</houseDB>
```

The root node of this document is called 'houseDB'. Children of this element node are any number of house nodes, each of which contains a number of sub-nodes, like 'picture', 'price' and 'city'. In this case 'picture' is an empty node, which only contains an attribute called 'src'. A number of nodes, among which the 'city' node, only have contents and no attributes, while the 'price' node has both.

There are many reasons to apply transformations to XML documents. Driven by a query in some query language, they may be used for extraction of relevant data, like pictures and addresses of houses having a price within a specified range and located in a specified city. Driven by a style sheet specification, an XML transformation may generate a presentation, like the one depicted in figure 1, for a given body of data. Such transformations may also be used to adapt a document to a different context, for example in the case of electronic data interchange. Even though they need essentially the same data, different users may have different preferences when extracting data, as essentially the same data may be provided in different forms and the platforms for showing these data may have different capabilities. For example, one would not like to download a video film onto a simple PDA, while one would also not like to see a two-tone monochrome image on a desktop PC capable of showing full screen video, when such data are available. This example shows how presentations on different platforms may have to look different, even when showing exactly the same data, e.g. because of screen size limitations.

In order to provide the right data in the right form to many different platforms and many different users, it is advisable not only to have the capability to transform documents that contain the data that will be required, but to transform the queries and style sheets as well: “transform the transformations”. Otherwise one would need many versions of the same query and of the same style sheet, one for every combination of platform capabilities and user preferences. Having one query and one style sheet for each type of requirement and adapting these on the fly to the specific circumstances of a given event,

may sometimes be the only feasible solution. As the documents involved (raw data, queries, style sheets, etc.) are of many different types and will be used for many different operations one should either provide many different transformation mechanisms, or find one that serves every purpose. The latter is possible if all documents concerned are in the same standard format, which is the case if they are all written in XML. It is more complicated if part of the information is provided in a different format, like for instance the XPath expressions used in XSL.

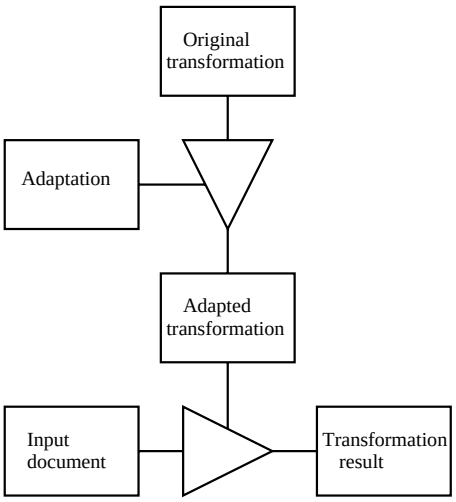

Figure 2

If we use transformations that themselves may be the object of transformations to provide the data required by a specific user on a specific platform, then this could lead to the following sequence of steps: first transforming the user query to a form adapted to the specific requirements, then executing that query and finally using a style sheet that itself has been adapted to generate the presentation to be used. The query and style sheet adaptations may themselves be performed in a number of smaller steps, e.g. first to a form that takes into account user preferences and then to a form that accounts for platform requirements. Such smaller adaptations should only change the relevant aspects of a query or style sheet document and leave the rest unchanged. In this way we get an XML document that is transformed a number of times before being used to realize the desired transformation. For this final transformation this XML document can be translated into XSLT. In order to allow a transformation to be transformed itself using an XML interpreter, it had better not contain anything but element tags and simple attribute and element values, i.e. no values that contain any hidden structure of which the components are not directly accessible by the XML interpreter.

An alternative to specifying XML transformations as described above is offered by XSL. Transforming XSL specifications is, however, not an easy task. XSL represents XML structures in the form of XPath expressions [W3C XSL & Linking Working Groups, 1999b], which are strings consisting of node names separated by slashes ('/') and additional expressions within straight brackets ('[' and ']'). In order to transform XPath expressions these strings should be analysed and rebuilt. In addition to the software for analysing XML constructs, this would require facilities to separately access the elements of XPath expressions, a facility that XSLT does not provide. Also, in XSL transformations all templates are children of the root node, irrespective of which part of the input document they represent: this leads to a "flat" structure of XSL documents which is very different from the structure of the documents to be transformed. There are also many different ways possible to use XSL templates to access the elements of an XML document to be transformed. So, XSL documents differ in many respects from the documents on which they operate. This makes it very hard to use the same mechanism for adaptation of XSL documents to specific requirements as for transformation of other documents.

One could also consider XQuery [Chamberlin et al. 2000]. In XQuery the specification of the document to be queried, the elements of that document that should be considered, the conditions for retrieval of these elements and the output to be generated, are all formulated in one text node. Having the complete specification in one text node implies the need for a specific interpreter in addition to the XML parser already needed. XQuery also uses XPath expressions, leading to the same problem as with XSL. As the specification of the retrieval parts of queries in XQuery are not isomorphic to the structures to be retrieved and the syntactic elements of a query do not correspond to XML elements, it

is not straightforward to analyse an XQuery query and change or reorder some of its components (e.g. using XSLT or some DOM-type document analyser). An XML interpreter would only be useful to transform the specification of the output part of the query, as markup is only used in the output to add structure.

To modify the components of a transformation specification (using an XML-aware transformation tool), they should all be separately accessible to the tool and their structure should as much as possible be isomorphic to the data they represent. Specifically, this means that:

- Elements of the transformation specification should correspond to (sets of) elements in the input or output documents.
- Relations among elements in the transformation specification should correspond to relations among elements in the documents concerned.
- Element or attribute names in the transformation specification should be the same as those in the documents.
- Structures or elements that have repeated occurrences in the documents, however, like lists of elements of the same type, may be represented with a single occurrence in the specification.

In this chapter we discussed the requirements for a uniform system that may be used to transform XML documents, but that may also be used to adapt the transformation specifications to specific needs. We noticed that in order to achieve both goals several requirements should be met: all relevant objects should be separately accessible by an XML interpreter, i.e. each should correspond to a DOM node [Apparao et al. 1998]. Further the specification of the structure of the (input and output) data should mirror the structure of the data themselves as close as possible. Yet another requirement, the need for the separation of input and output specifications, will be discussed in the next chapter.

3 Specifying transformations

Specifying XML transformations may be done in several different ways, using several different tools. XSL uses templates that match a specified node in a XML network, and switches from node to node, each time invoking a different template. Output is generated from within these templates. Xquery queries consist of two parts: the 'FOR .. WHERE' part for the specification of the data required from the data sources and the 'RETURN' part for the specification of the way the output document should be structured. In this chapter these different aspects, the use of templates and the use of separate source data specifications and output document specifications will be discussed.

3.1 Templates and Parameters

A transformation takes one or more input documents and produces an output document that contains items directly taken from the input document or items derived (calculated) from such items. Input items are elements, relations among elements, attribute name-value pairs or text node contents taken as a single value. The input items used are selected on the basis of their position in the input document, their structure or their value. XSLT provides template constructs for matching specific items and for specifying the operations to be performed on them. Matching criteria are specified by Xpath expressions. Matching may however also be done using schema templates that contain elements that represent element sets in the input document. In XSLT templates are collections of operations, comparable to functions in imperative programming languages like C and JAVA. Instead of taking sets of operations as a basis for specification we use data structures, and a transformation is specified by linking input and output structures directly or through processing instructions. Thus, the difference between XSLT and our HTL language is the difference between processing sets of nodes or node trees.

Structures may have substructures that one would like to treat separately. In XSLT this is done by invoking templates that match such substructures from a template that matched a superstructure. In HTL substructures may be represented by parameters within a structure, placeholders that reference other templates. In XSLT one also has parameters, but these are used to transfer data from one template / processing unit to another, where they are needed for internal processing. HTL also has this functionality, as parameters may also be used to import external data. A HTL template to be used in a query for the houses application mentioned above may look like:

```
<house>
  <price><htl:param name="PriceTag"></price>
  <street id="streetID"/>
  <houseNR id="numberID"/>
  <city><htl:param name="City"></city>
</house>
```

This template can be used to specify elements in the database that each contain data on a single house. It has two parameters, called 'PriceTag' and 'City' respectively, that represent values, e.g. from a user form specifying the price and city conditions of this request. After execution of the query with these specific values inserted, the components of the addresses (street and number) of the relevant houses are available for the generation of the output through the 'streetID' and 'numberID' references.

As mentioned above, XSLT templates are used to control the traversal of the input document tree as a process. Output is generated on the fly during this process. This poses some limitations on the output structures to be generated as it is not possible to specify explicitly where data is to be added to the output document. So, in XSLT output documents are built strictly sequentially. The alternative offered by HTL is to use templates to specify output structures, and the construction of an output tree using the contents of input documents. The elements from the input document are accessed in more or less arbitrary order in this case. Thus, depending on the way the transformation is specified, the order in which transformation operations are performed will either depend on the actual structure of the input documents or on the specification of the structure of an output document. These different ways are treated in the following two sections.

3.2 Input driven specifications

In XSLT transformations the order in which the input document tree is traversed is depth first, in the order of the input document. Of each element first the children are accessed, then in turn the children of each these children, etc., after which the next sibling in the document order is processed. This order may be changed by specifying jumps in the evaluation order. All operations are defined with respect to the current context, which centers on the node that is currently being processed. That means that parents, children and siblings of the current node, as well as its attributes are easily accessible. Elements from other parts of the tree are harder to access, but it is always possible to specify a path to an element starting from the document root. Templates only specify a local environment, i.e. an element of a certain type, possibly with a parent and one or more children or attributes. Tree traversal means switching from one template to another, either in the default order, as described above, or following explicitly specified context switches. The template context, i.e. the 'match' attribute of the 'xsl:template' element, is specified using XPath expressions. This approach is typical for input-driven specifications.

Output operations are local to a specific template, and thus are processed in the order in which the templates are processed. Although clever interleaving of context switches and output operations and the use of variables to store intermediate output substructures for later output generation may somewhat alleviate this problem, there are insurmountable limits to the possibilities of building new structures using elements of a specific input document in this way. Furthermore, as there is not a one-to-one relationship between the elements of the input document and the elements of the transformation specification it is hard to adapt a given transformation specification to a specific purpose by a suitable transformation of an XSLT specification, since templates only match a (small) part of the full input tree, and a single template may be used at many positions of the document tree. As a consequence, there are many different ways to specify the transformation of a given type of document. So it is very hard to identify and find the elements that have to be changed for a given adaptation. Additionally, the (sub)structure specifications are in the form of (XPath) expressions which are not isomorphic to the structures themselves and of which the components are not separately accessible. So adaptation of XPath expressions would require complicated and very specific string transformations.

3.3 Output driven specifications

The specification formalism proposed here uses XML trees to prescribe the structure of the output document. This is comparable to what a query language like XQuery does in its return statement. Specifications in HTL always have just one template that contains the root of the output document and there are any number of templates for other parts. The output document is built in a hierarchical way, starting from a main template which may contain references to other templates, which in turn may invoke other templates, etc. It is thus built in template order, which is not necessarily the same as the order indicated by the relations among the elements of the result tree.

The parts to be retrieved from the input documents are specified in the form of a retrieval schema. An element of such a schema may correspond to any number of instance elements in an input document. Output templates may contain references to schema elements. Input document elements are thus accessed in a context-independent way, in the order in which the expressions that contain them occur in the output template hierarchy. As one schema element may correspond to multiple instance elements in an actual document an atomic term in an output expression may correspond to any number of elements in the input, depending on the schema definition, its relations to other elements in the document or its value. For instance, if the database mentioned in the real estate example contained six houses that satisfy the criteria of the query, a pointer to the price element contained within the schema element representing the retrieved houses would indicate not one, but six prices.

```
<htl:select idref=”expensive”>
  <p>Upper range:
    <htl:value-of><street/></htl:value-of>
    <htl:value-of><houseNr/></htl:value-of>
    <htl:value-of><city/></htl:value-of>
  </p>
</htl:select>
```

This template selects certain houses that satisfy the conditions given in the template in which the element with id ‘expensive’ occurs, which presumably selects the more expensive houses. It outputs for each of these houses a paragraph containing the text: “Upper range:”, followed by the contents of the ‘street’, ‘houseNr’ and ‘city’ elements. The contents of the ‘htl:value-of’ elements may be considered as implicit templates that dictate what should be selected for output.

So, just as multiple elements of a given type may result from a single element in an input specification, multiple instance elements may also appear in the output where only one element is specified in the schema. A query for a house in a given city within a given price range may deliver a number of pages, each one describing a single house. As a consequence, for each element in an output specification that results in multiple instances there is defined some access structure which allows each of the instances to be displayed, either alongside each other, or alternatively controlled by a separate index with links to the different pages, or as a guided tour with 'next' and 'previous' buttons on each page.

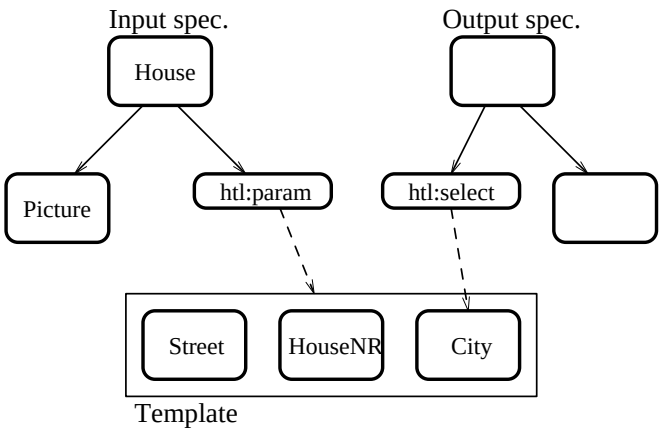

Figure 3

Transformations specified in this way provide as much freedom in specifying the output format as XML allows, while still permitting full access to all parts of the input (Fig. 3). The price to be paid is

the extra specification of the input schema ('input spec.' in fig. 3), but this is limited only to those parts that will actually be used in the output with as much of their context as needed to uniquely identify them. For instance, where an input document would contain a number of elements describing books and another number of elements describing CDs, and each of these would have their own price elements, but you are only interested in book prices, you would specify as much of the book structure as necessary to identify only those price elements belonging to the books. Otherwise it would have sufficed to specify only those elements from the document that are named 'price'.

4 Components of a transformation specification

The basic operations of a general purpose transformation mechanism are filtering, selection, reordering and addition.

- **Filtering**. Filtering means selection according to certain criteria of elements from the input document(s) that are to be used in the generation of the output document. These criteria are specified in the form of a schema that describes the structure of the data required (“target structure”) and in which constraints are given for their values. The filter specification will thus specify the part of the input data that is relevant for the transformation concerned and exclude the irrelevant data. Any data not specified therein will not be accessible for the other parts of the transformation specification.
- **Selection.** In general, application of this filter specification delivers sets of elements for each schema element. Of these sets, some elements should possibly be treated differently from other elements. The selection operation takes care of this, by selecting elements from such a set, or from the set of children of a specified element, according to given criteria. In this way e.g. the prices of the houses in the real estate example may be divided into an upper and a lower range or specific elements from a set may be used at different places in the output specification.
- **Reordering and addition.** The second stage of the transformation process, after having selected the required elements and the attributes we need, involves putting them in a different framework. It is possible that we need to have them in a different order, with added information like specific captions for pictures or columns in tables, and values calculated from the data retrieved. In this stage the new output structure is specified, which contains references to the input schema and selections from the elements thus obtained for the values to be inserted or calculated.

So the basic questions to be answered are, first, whether a specific element is present in the specified input document, and, second, what should be done with it. The complete transformation specification consists of two parts. The first part, the filter specification, will only specify the elements of the input that will be used for the construction of the output document. The second part, the transformation specification, describes what the output document will look like and will thus specify selection from, reordering of and addition to the elements specified in the first part. The specifications in the second part contain references to the schema elements and attributes of the first through which they gain access to items from the input data sources. Filter specification and transformation specification will each consist of two kinds of elements: elements from our specification language HTL that indicate certain constraints or operations, and elements that represent elements in the input or in the output documents. The first elements will have a HTL name space identification, while the latter will have different name spaces, corresponding to the name spaces of the input sources and the output documents. These are the elements that directly refer to input or output elements. The example in section 3.1 shows a fragment from a filter specification, while the one in section 3.3 is taken from some transformation part.

4.1 Filtering

The input part of a transformation specification contains a sequence (set) of templates that represent parts of the input document. As mentioned, the set does not need to represent the document completely, it should only allow the parts to be used in the output to be accessed. So e.g. the document root will not necessarily be represented within some template. Only templates that are referred to, directly or indirectly, by the output generation part are useful here. Attribute values or element contents represent constraints on the values that should appear in the input. If, for instance, the 'city' element would have as contents 'Eindhoven' only houses located in the city of Eindhoven would be selected. Thus selections

from the available data are specified. Tag names of elements identifying the input part, the output part, the parameters, the links and the expressions will all be from the 'htl' name space, the other element names reflect the names as they appear in the respective documents. Elements that are needed by the output part should be marked by an explicit 'htl:id' attribute to identify them as a target for the relevant operations. So for our real estate agency example a definition of an input may look like:

```
<htl:template>
  <doc>
    <house htl:id="houseID">
      <price>
        <htl:from>50000</htl:from>
        <htl:to>100000</htl:to>
      </price>
      <htl:param IDREF="addressID" />
      <picture />
    </house>
  </doc>
</htl:template>
<htl:template htl:id="addressID">
  <street /> <houseNr />
  <city />
</htl:template>
```

This input specification contains a template for accessing house data and a template for the address of the house, which may be expanded within the house element. The 'htl:from' and 'htl:to' elements specify that only houses in a price range from $ 50,000 to $ 100,000 should be selected. The 'htl:param' element will be replaced by the contents of the template with the 'htl:id' value of 'addressID'. And only house elements that have at least a 'street' element as well as 'houseNr' and 'city' elements will be accepted. Here the 'doc' element presumably represents the root element, but it is also possible that the input documents contain any number of 'doc' elements, each of which will be considered for inclusion in the input set.

The filter specification is a collection of templates that specifies constraints to which the results of the retrieval process should comply. The components of these templates are element, attribute and relation specifications. Each of the specification templates narrows down the possible contents of the retrieval results. There may be constraints on the possible values of elements and attributes, there may be constraints on the position of an element within the document (e.g. expressed as a path from the root element to the element concerned) and there may be constraints with regard to the relations among elements (such as a specific parent or specific children of the element under consideration). The filter specification however does not need to have any relation to the data model of the data sources. If there is no correspondence at all the retrieval process will yield no data. If there is a correspondence the query will deliver those data from the data source(s) that correspond to (part of) the specification and reject the others. Non-specified properties of an input element will never lead to its rejection, so the retrieved elements may have any number of attributes or relations that do not occur in the filter specification. These are, however, not available to the output generation process. Attributes are element type specific, i.e. attributes of the same name of elements with different names are different attributes. Relations are directed associations between two element specifications. They may be specified directly or via templates and parameter specifications, in which the contents of a template replace the parameter element. These associations come in two versions: unconditional and conditional. An unconditional relation requires the relation to be actually present in the input document in order for it to be retrieved. Conditional relation specifications may occur in an element-relation loop and thus allow chains of arbitrary length to be specified.

The correspondence between the filter specification and the selected source data is defined by:

- The filter specification templates that are not referred to from other templates are taken as the base templates.
- Element matching starts from the child elements of these templates.
- Only elements that have a tag name that corresponds to a name of a filter specification element will be retrieved from the source documents.
- If the filter specification contains an unconditional (child or link) relation only those input elements that have a corresponding relation will be retrieved. The retrieved elements as presented to the transformation process will have the same relation.
- If the filter specification contains an attribute for some element only those elements having this attribute will be retrieved from the data sources. The retrieved elements will have the same attribute. Elements not having this attribute will be ignored.
- If the filter specification prescribes a value for an element or an attribute only those element having that value or an attribute with the specified name and value will be retrieved. Elements having other values, or in which the specified attribute has a different value will be ignored.
- If the filter specification specifies a range of values for an element or an attribute only elements that have a value from within this range or elements with the specified attribute having a value from within the range will be retrieved. Elements having values outside of this range or with attributes having values outside the specified range will be ignored.
- Part of the filter specification may be specified in the form of separate sub-trees, comparable to procedures in imperative programming languages, which may be used at different positions within the specification. If such a template contains constraints of any kind these constraints will apply to every use of the template.
- If the filter specification contains loops at least one of the parameters in such a loop will conditionally invoke the associated template. Such a loop may be compared to a recursive invocation of a procedure in an imperative programming language. It corresponds to repetitive occurrences in the data source of the data specified in the loop.

Every element in the filter specification corresponds to a set of elements in the data sources and a one-to-one unconditional relation between elements in the specification corresponds to a set of parent-child associations in the sources. A conditional relation does not have to correspond to some relation in the sources.

4.2 The transformation specification

The transformation specification consists of the elements in the output part and the templates invoked from there. It specifies the form of the transformation output. The values of elements or attributes in the result may correspond to values retrieved by the filter specification. In addition to objects from the filter specification the transformation specification may also specify objects that do not occur in the data sources, such as values that have been derived from these or constants declared within the transformation specification. Like the filter specification, the transformation specification will contain two kinds of elements and relations: those corresponding to the elements and relations that should appear in the transformation result, and the ones from the HTL namespace that will be used to derive values to be included in this result.

One of the templates that represent output elements and relations should be identified as the main template, the template from which to start generating the output tree. One (possibly the same) template should contain the root element of the document to be generated, as no document can be generated without a root. Parameter tags indicate either (output document) templates to be substituted or values received from outside the transformation engine. Operations include selection of structures or values from the (filtered) input and (boolean, arithmetic or string) expressions for manipulations of input or parameter values.

So there will be links from the output part to the input part, but not the other way around. The links are associated with selection, name-of or value-of tags, which will have an 'IDREF' attribute pointing to some other element. The following example shows what such a transformation specification may look like. Consider the real estate application mentioned before, where we would like to query a (possibly

large) database of houses to find those within the price range specified in the input part. The result of the query should be presented in HTML with for each house one paragraph containing its picture, its price and the city where it is located. Using the input specification shown above, this query could be formulated as follows:

```
<htl:transform>
  ......
  <htl:output>
    ......
    <body>
      <htl:select IDREF="houseID">
        <p>
          <htl:value-of><picture/></htl:value-of>
          price: <htl:value-of><price/></htl:value-of>
          city: <htl:value-of><city/></htl:value-of>
          .......
        </p>
      </htl:select>
    </body>
    ......
  </htl:output>
</htl:transform>
```

The htl:output element will occur alongside the template definitions as a child of the (htl:transform) root element. The htl:select element in this example selects house elements that satisfy the criteria given in the filter specification. It establishes a context of one of the elements selected at a time, for the enclosed operations. The htl:value-of elements output values of elements that exist within that context. The result will be a number of paragraphs, enclosed in HTML and tags, that each contain a picture, a price (preceded by 'price:') and a city (preceded by 'city:') of one of these houses.

The correspondence of the transformation specification to the actual output data is subject to the following rules:

- Selection elements refer to filter specification elements. They may be nested, in which case the outermost selection represents a set of elements from an input document, and the inner selections are selections from this set.
- Elements within selection templates, with names and relations that correspond to input elements represent the values and relations of those input elements, which may be used in expressions or be reflected in the output document.
- Selection elements may contain (HTL) elements that specify operations to be performed on each element of the associated set.
- The transformation specification may also contain (non-HTL) elements and element or attribute values that should appear unchanged in the output.

So elements within selections may have multiple instantiations in the output. Elements specified outside such elements, like the <body> element in the example above, will have at most one instantiation.

4.3 List of HTL primitives

The operations mentioned above may all be specified using HTL, a language designed for full compliance to the XML specification. As HTL is also XML, HTL transformations may be transformed by HTL transformations. HTL primitives are:

- 'htl:transform' is the root element of the transformation specification. It contains the output specification and any number of template definitions.
- 'htl:output' marks the transformation specification specification
  - Optional attribute: 'method', with possible values 'xml', 'html' or 'text'
- 'htl:template' specifies a part of the input or output document that may be referred to from some other part
  - Template elements have contents and possibly an 'htl:id' attribute by which they may be identified from HTL elements having an 'idref' attribute.
  - They may also have a mode attribute that controls their interpretation. Possible values are ‘conditional’, ‘negate’, ‘reject’
- 'htl:from', 'htl:to' used to specify a range of input document element or attribute values
- ‘htl:operator’ used to specify (logical, arithmetical or relational) expressions
  - Required attribute ‘op’ indicates which operator
- 'htl:attribute', when used in the filter specification, specifies a constraint on the allowed values of some attribute
  - Required attribute: 'name'
- 'htl:param' refers to a template that will replace the parameter element
  - Required attribute: either 'template' or 'name'
- An element named 'htl:any' stands for an arbitrary input element.
- An element with name 'htl:descendant' stands for any descendant of the element in which it is contained.

Transformation specification elements:

- 'htl:select' and ‘htl:reject’ define a subset of a set of input elements. ‘htl:select’ selects the elements according to the associated template. htl:reject selects those elements from a given context that would not be selected by a htl:select with the same associated template.
  - Select and reject elements have an associated template, which will either be defined inside the select element or be referred to by an optional 'idref' attribute, which may refer to any element from the template.
  - They may have a context attribute to explicitly specify the context ('root', 'parent', 'any', 'self') in which the template should be evaluated
  - A document attribute may be used to specify the URL of a document to be retrieved. If so, the context will automatically be 'root'.
- 'htl:value-of' specifies an expression to be evaluated
  - Value-of elements may have an associated template. If they have, they produce a concatenation of the values of elements or attributes that are specified in the template. If not, they produce the value of the current element.
- 'htl:name-of' may be used to insert the name of a specific element.
  - A name-of element may have an associated template. If it has it produces the name of the first element from the node list that would result from evaluation of the template. Otherwise, it supplies the name of the current element.
- 'htl:copy' copies the current context and all its descendants to the output.

- 'htl:attribute', in the transformation specification, specifies an attribute of a given non-htl element
  - Required attribute: 'name'
- 'htl:id' attributes allow templates and elements declared within templates to be referred to from other HTL elements.
- Non-htl elements will be interpreted either as constraints on input data, when used inside a template that defines a selection, a value-of or name-of expression, or output elements to be passed unchanged to the output
- Literal strings will be interpreted as either values to be used in constraints, or as values in expressions used to calculate output values.

The workhorse of the language is the 'htl:select' element. It takes care of the selection of the required elements from the input document and their insertion at the position where they are needed in the output. The 'htl:value-of' element is also useful, but it only operates in the context established by the selection.

For a more extensive definition of the HTL language, see appendices A and B.

4.4 HTL Templates

Templates are defined either explicitly, at the top level of the transformation specification, or within a select element, or implicitly within value-of or name-of elements. They represent parts of an input or output document. They are used to specify which parts of a document should be used in a selection ('htl:select') or another operation (like htl:value-of'). Templates may also be used to specify a part of an other template, either one specifying (part of) the input or one specifying (part of) the output, from which they are referred to by 'htl:param' elements. In HTL they may specify the structure of the output document or they may play the role that XPath expressions play in XSL. The difference is that XPath expressions are just strings, while templates are true XML constructs, and thus are amenable to transformation by XML transformation systems like XSL. A subset of the templates will contain an element which represents a document root, while the others represent pieces to be inserted into other templates. Such templates may in turn contain parameter elements, which may refer to other templates or to the referring template itself. By making references conditional arbitrary length chains may be represented in this way.

A template will contain any number of elements, with or without descendants. It will be marked by a 'htl:template' tag. 'htl:param' elements, when not used to indicate external values, will point to 'htl:template' elements, but 'htl:select' , 'htl:value-of' or 'htl:name' elements may refer to elements within templates to indicate the element to be selected. The descendants of such an element then indicate conditions for selection of the element concerned and the ancestors indicate how the element should be reached from some other element. The templates used by the elements mentioned thus specify conditions for acceptance or rejection of a specific element from the input document. The declaration of these templates may be implicit, they may be directly contained within the 'htl:value-of' or 'htl:name-of' elements themselves, without an enclosing 'htl:template' element. In this case, when the enclosing element does not have an 'idref' attribute the template root elements are addressed. But they may also be referred to by 'idref' attributes of the 'htl:select', 'htl:value-of' or 'htl:name-of' elements. So a template may have different meanings, dependent on the way it is accessed. Take the following template, for example:

```
<htl:template htl:id="t123">
  <doc>
    <house htl:id="h456">
      <street/>
      <houseNr/>
      <city/>
    </house>
  </doc>
</htl:template>
```

When this template is accessed through a htl:select or htl:value-of element having an idref attribute of "h456" this will indicate the 'house' elements which are children of a 'doc' element and have children called 'street', 'houseNr' and 'city'. When accessed through idref "t123" it means the 'doc' elements having one or more 'house' children, which each have 'street', 'houseNr' and 'city' children. The template may be used in different contexts, which are specified by a context attribute of the referring elements. The value of this attribute may be ‘self’, which means that that template should be evaluated relative to the current context element, or ‘root’, which means it will be evaluated relative to the document root, or ‘parent’, which means: relative to the parent of the current context, or ‘any’, which looks for occurrences anywhere in the document.

A template contains a sequence of XML elements that can have varying interpretations, dependent on its use. Templates may define the structure of (parts of) output documents. When used by 'htl:select' elements templates are used in the way XPath expressions in XSLT are used. In this case they will produce a string or a node list, although that list will be different in each case, depending on which element of the template is addressed. When used by 'htl:value-of' or 'htl:name-of' templates will result in strings representing, respectively, the value or the name of the first element of the node list concerned. Templates containing htl:operator elements also provide the functionality that arithmetical or logical expressions can offer.

5 Implementation issues

Ideally, the HTL language would be implemented by specially constructed interpreters, but as XSLT engines like Xalan [Xalan 2000] are readily available and are reasonably efficient the actual transformations may easily be performed by an XSLT engine. Therefore the specifications need to be converted to XSLT documents. For this purpose a XSLT transformation has been defined. Its functionality is described in appendix B. So in order to execute HTL constructs like the ones shown in this report they will first be transformed, using an XSLT style sheet, and the resulting XSLT transformation will then be applied to the input documents (cf. [Jelliffe, 2001]). As mentioned before, templates may have different roles and thus may be translated in a number of ways, either into xsl:templates or into Xpath expressions. Htl:select, htl:name-of and htl:value-of elements will be translated into XSL sub-templates, HTL output templates will be mirrored in the XSL output and the other templates will be translated into Xpath expressions. Where input elements have a fixed position within the document tree they will be accessed through absolute paths that all start at the input document root. Otherwise the XPath expression that identifies them will start with a double slash ('//'), indicating they may be found anywhere in the document tree.

When select templates are translated into XPath expressions, the element to which the 'htl:select' element refers has to become the current element. That means that its ancestors, as far as they appear in the template, should be part of the path, and the descendants should appear in the additional conditions to be fulfilled by the element from the input document. So the example template of the "filter specification" section would be translated to the XPath expression:

'/doc/house[price>50000 and price<100000 and picture and street and city]'

when used by a 'htl:select' element with a 'root' context attribute, with a 'htl:id' attribute of value 'houseID'.

Due to the different roles that templates may play they should be interpreted differently in different contexts. Therefore the translation to XSL has different mode versions for the templates associated with certain elements. So an element within a template used for selection will be treated differently depending on whether it occurs inside or outside the element used in the selection (the context element). All in all there are three modes: a default mode in which analysis of the output specification starts and two for the analysis of templates used in the specifications of selections.

As remarked before, switching output insertion points within a XSLT transformation is not possible. And as XSL templates can not be located inside other XSL templates a translated HTL document does not contain separate templates. So all operations are performed within one single template, in which context switches are performed by xsl:for-each and xsl:if or xsl:choose elements only. This results in a monolithic and rather unreadable output, something like a C program without functions, but as the result needs only to be read by XSL transformation engines, this is not a serious problem.

So, using XSLT as an intermediate in the HTL transformations has some advantages and some disadvantages. The main advantages are that it allows us to use an existing, proven technology. The main disadvantages are that it requires an additional translation step for each HTL transformation and it

is not possible to use all features of XSLT, like template switching and, consequently, it is not possible to use template modes in the translation result to allow different interpretations for HTL templates.

6 Conclusion

In the course on our research for the HERA [Houben, 2000] project we felt the need for a general purpose transformation tool for XML documents. The transformations envisaged ranged from retrieval of specified data from a database, via testing of documents for compliance to specified criteria (transformation into go / no go decisions) to conversion of documents to browser or printer pages with a specified layout. In a practical system there should also be possibilities to adapt transformations to specific user profiles or platform requirements. Such adaptations can be considered as transformations on transformation specifications. It would be very useful if the transformation tool would be able to do these adaptations as well as the primary transformations. This called for some special requirements to the transformation tool. Our transformation specification language, HTL, is designed to satisfy these requirements as well as those for easy and clear specification of transformations of other documents.

The method of specifying transformations of XML documents described in this report describes these documents in a way that directly reflects their structure. The specifications are based on the required structure of the output documents, allowing full freedom in defining their layout, independent of the contents of the input documents. As the format of the specification closely resembles the format of the documents to be used and produced the complexity of transforming of the specifications themselves, e.g. to adapt them to specific needs, is comparable to the transformation of the documents proper. Because the actual transformations may be done by one of the many available XSLT engines they can be performed rather efficiently.

7 References

[ZapThink, LLC] "The XML.org XML Standards Report", issued by ZapThink, LLC http://www.zaplink.com/, 2000 - *

[Houben 2000] G.J. Houben, HERA: Automatically Generating Hypermedia Front-Ends for Ad Hoc Data from Heterogeneous and Legacy Information systems. Proc. Engineering Federated Information Systems, pp. 81-88, 2000

[W3C XSL Working Group 1999a] W3C XSL Working Group, XSL Transformations (XSLT) Version 1.1. http://www.w3.org/TR/xslt11, 1999

[W3C XSL Working Group 1999b] W3C XSL Working Group, W3C XML Linking Working Group, XML Path Language (XPath) Version 1.0. http://www.w3.org/TR/xpath, 1999

[Jelliffe, 2001] Rick Jelliffe, The Schematron, Academia Sinica Computer Centre, http://www.asc.net/xml/resource/schematron/schematron.html, 2001

[Chamberlin et al. 2001] Don Chamberlin, Daniela Florescu, Jonathan Robie, Jérôme Siméon, Mugur Stefanescu, XQuery: A Query Language for XML, http://www.w3.org/TR/xquery/, 2001

[Apparao et al., 1998] Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Ian Jacobs, Arnaud le Hors, Gavin Nicol, Jonathan Robie, Robert Sutor, Chris Wilson, Lauren Wood, “Document Object Model Level 1”, http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/, 1998

[Xalan 2000] Xalan Apache Project, Xalan-Java version 1.2.2 http://xml.apache.org/xalan/index.html, 2000

[Clark et al. 2001] James Clark, Murata Makoto, “Relax NG Specification”, http://www.oasis-open.org/committees/relax-ng/spec-20010811.html, 2001

Appendix A. HTL syntax

Formal description of HTL syntax in relax-ng [Clark 2001].

```
<grammar xmlns="http://relaxng.org/ns/structure/0.9">
  <start>
    <ref name="transform"/>
  </start>

  <define name="transform">
    <element name="htl:transform">
      <interleave>
        <ref name="output">
        <zeroOrMore>
          <ref name="template"/>
        </zeroOrMore>
      </interleave>
    </element>
  </define>

  <define name="output">
    <element name="htl:output">
      <oneOrMore>
        <ref name="outputElement"/>
      </oneOrMore>
    </element>
  </define>

  <define name="outputElement">
    <choice>
      <ref name="param"/>
      <ref name="outputAttribute"/>
      <ref name="select"/>
      <ref name="reject"/>
      <ref name="value-of"/>
      <ref name="name-of"/>
      <ref name="copy"/>
      <ref name="outputPattern"/>
    </choice>
  </define>
```

```
<define name="outputPattern">
  <element>
    <anyName>
      <except>
        <name>htl:*</name>
      </except>
    </anyName>
    <zeroOrMore>
      <attribute>
        <anyName>
          <except>
            <name>htl:*</name>
          </except>
        </anyName>
      </attribute>
    </zeroOrMore>
    <zeroOrMore>
      <ref name="outputElement"/>
    </zeroOrMore>
  </element>
</define>

<define name="template">
  <element name="htl:template">
    <optional>
      <attribute name="htl:id"/>
    </optional>
    <optional>
      <attribute name=”mode”>
        <choice>
          <value>conditional</value>
          <value>negate</value>
          <value>reject</value>
        </choice>
      </attribute>
    </optional>
    <oneOrMore>
      <ref name="templateElement"/>
    </oneOrMore>
  </element>
</define>

<define name="templateElement">
  <choice>
    <ref name="param"/>
    <ref name="templAtribute"/>
    <ref name="from"/>
    <ref name="to"/>
    <ref name="operator"/>
    <ref name="descendant"/>
    <ref name="any"/>
    <ref name="namedSubtemplate"/>
  </choice>
</define>
```

```
<define name="descendant">
  <element name="htl:descendant">
    <optional>
      <attribute name="htl:id"/>
    </optional>
    <ref name="templatePattern"/>
  </element>
</define>

<define name="any">
  <element name="htl:any">
    <optional>
      <attribute name="htl:id"/>
    </optional>
    <ref name="templatePattern"/>
  </element>
</define>

<define name="namedSubTemplate">
  <element>
    <anyName>
      <except>
        <name>htl:*</name>
      </except>
    </anyName>
    <optional>
      <attribute name="htl:id"/>
    </optional>
    <zeroOrMore>
      <attribute>
        <anyName>
          <except>
            <name>htl:*</name>
          </except>
        </anyName>
      </attribute>
    </zeroOrMore>
    <ref name="templatePattern"/>
  </element>
</define>

<define name="templatePattern">
  <interleave>
    <optional>
      <ref name="expression"/>
    </optional>
    <zeroOrMore>
      <ref name="templateElement"/>
    </zeroOrMore>
  </interleave>
</define>

<define name="param">
  <element name="htl:param">
    <choice>
      <attribute name="name"><text/></attribute>
      <attribute name="idref"><text/></attribute>
    </choice>
    <empty/>
  </element>
</define>
```

```
<define name="outputAttribute">
  <element name="htl:attribute">
    <attribute name="name"/>
    <ref name="expression"/>
  </element>
</define>

<define name="templAttribute">
  <element name="htl:attribute">
    <attribute name="name"/>
    <oneOrMore>
      <choice>
        <ref name="from"/>
        <ref name="to"/>
        <ref name="expression"/>
      </choice>
    </oneOrMore>
  </element>
</define>

<define name="expression">
  <choice>
    <text/>
    <ref name="operator"/>
  </choice>
</define>

<define name="select">
  <element name="htl:select">
    <optional>
      <ref name="context"/>
    </optional>
    <choice>
      <attribute name="idref"/>
      <ref name="template"/>
      <group>
        <attribute name="idref"/>
        <ref name="template"/>
      </group>
    </choice>
    <oneOrMore>
      <ref name="outputElement"/>
    </oneOrMore>
  </element>
</define>
```

```
 <define name="reject">
   <element name="htl:reject">
     <optional>
       <ref name="context"/>
     </optional>
     <choice>
       <attribute name="idref"/>
       <ref name="template"/>
       <group>
         <attribute name="idref"/>
         <ref name="template"/>
       </group>
     </choice>
     <oneOrMore>
       <ref name="outputElement"/>
     </oneOrMore>
   </element>
 </define>

<define name="value-of">
   <element name="htl:value-of">
     <optional>
       <ref name="context"/>
     </optional>
     <choice>
       <attribute name="idref"/>
       <ref name="template"/>
       <group>
         <attribute name="idref"/>
         <ref name="template"/>
       </group>
     </choice>
   </element>
 </define>

 <define name="name-of">
   <element name="htl:name-of">
     <optional>
       <ref name="context"/>
     </optional>
     <choice>
       <attribute name="idref"/>
       <ref name="template"/>
       <group>
         <attribute name="idref"/>
         <ref name="template"/>
       </group>
     </choice>
   </element>
 </define>

 <define name="context">
   <attribute name="context">
     <choice>
       <value>self</value>
       <value>root</value>
       <value>parent</value>
       <value>any</value>
     </choice>
   </attribute>
 </define>
```

```
  <define name="copy">
    <element name="htl:copy">
    </element>
  </define>

  <define name="from">
    <element name="htl:from">
      <choice>
        <text/>
        <ref name="operator"/>
      </choice>
    </element>
  </define>

  <define name="to">
    <element name="htl:to">
      <choice>
        <text/>
        <ref name="operator"/>
      </choice>
    </element>
  </define>

  <define name="operator">
    <element name="htl:operator">
      <attribute name="op"/>
      <optional>
        <attribute name="op1"/>
      </optional>
    </element>
  </define>

</grammar>
```

Appendix B. HTL semantics

The semantics of HTL are defined through XSL. For each HTL construct the equivalent XSL construct is given.

```
Types:
      string
      attribute
      attribset
      node                    // element
      nodelist                // list of nodes

Predicates:
      child(node, node)       // “second node is child of first”
      attrib(node, attribute) // “attribute is attribute of element”
      empty(nodelist)         // “nodelist contains no elements”
      empty(attribset)        // “attribset contains no elements”

Functions:
      string name(attribute)  // name of attribute
      string value(attribute) // attribute value
      string name(node)       // name of node
      string namespace(node)  // namespace of node element
      attribset attribs(node) // attribute set of node
      nodelist children(node) // sequence of children
      nodelist list(node, nodelist) // nodelist with node added
      node extract(node, node)// node with specified descendant
                              // removed

Variables:
      node root               // root node
      nodelist null           // empty list

Definitions:

// Names followed by a colon are (unique) definition names
// As far as there are corresponding syntax definitions, these have
// the same name
// For each definition: Expressions over the line are preconditions,
// expressions below the line define the relevant semantics
// (expressed in XSLT)
// Except for variables with a value defined in the preconditions,
// variable declarations assume universal quantification over the
// type in question.
// Using these definitions, the semantics of a HTL document can be
// expressed as a translation to XSLT (function xsl), which uses
// functions serial, descendant, xpath and xexpr, also defined here.

serial1a:   node n
      empty(children(n))
      ------------------------------------------------
      serial(n) = '<' name(n) serial(attribs(n)) '/>'

serial1b:   node n
      !empty(children(n))
      ------------------------------------------------
      serial(n) = '<' name(n) serial(attribs(n)) '>'
            serial(children(n)) '</' name(n) '>'
```

```
serial2a:   node n; nodelist nl
      nl = list(n, null)
      ----------------------
      serial(nl) = serial(n)

serial2b:   node n; nodelist nl1, nl2
      nl1 = list(n, nl2)      // list constructor
      -----------------------------------
      serial(nl1) = serial(n) serial(nl2)

serial3:    attribute a
      --------------------------------
      serial(a) = name(a) '=' value(a)

serial4:    attribute a; attribset as1, as2
      a in as1
      as2 = as1 \ a
      ---------------------------------------
      serial(as1) = serial(a) serial(as2)
      or serial(as1) = serial(as2) serial(a)

serial5:    attribset as1, as2, as3
      as1 = union(as2, as3) and empty(intersection(as2, as3))
      -------------------------------------------------------
      serial(as1) = serial(as2) serial(as3)
      or serial(as1) = serial(as3) serial(as2)

operator:   string a, operator; node n1, n2;
      operator = '<htl:operator op =' a '>'
            serial(n1), serial(n2) '</htl:operator>'
      ---------------------------------------------------------
      xpath(operator) = xpath(serial(n1)) a xpath(serial(n2))
      xexpr(operator) = xpath(operator)

from: string ll, from
      from = '<htl:from>' ll '</htl:from>'
      ------------------------------------
      xpath(from) = '[ . &gt; ' ll ']'
      xexpr(from) = xpath(from)

to:   string ul, to
      to = '<htl:to>' ul '</htl:to>'
      ------------------------------
      xpath(to) = '[ . &lt;= ' ul ']'
      xexpr(to) = xpath(to)

descfunc: node n1, n2, n3
      ---------------------------------------------
      descendant(n1, n2) = child(n1, n2)
            or (descendant(n1, n3) and child(n3, n2))

descendant: string desc, patt
      desc = '<htl:descendant>' patt '</htl:descendant>'
      --------------------------------------------------
      xpath(desc) = '//' xpath(patt)
      xexpr(desc) = '[ .//' xpath(patt) ']'
```

```
any1: string any1, patt
      any1 = '<htl:any>' patt '<htl:any>'
      ----------------------------------
      xpath(any1) = '*/' xpath(patt)
      xexpr(any1) = '[ ./' xpath(patt) ']'

templAttribute:   string s, attr1
      attr1 = '<htl:attribute name="' s '"/>'
      ---------------------------------------
      xpath(attr1) = '/@' s
      xexpr(attr1) = '[@' s ']'
      xsl(attr1) = '<xsl:attribute name="' s '"/>'

outputAttribute:  string s1, s2, attr2
      attr2 = '<htl:attribute name="' s1 '">' s2 '</htl:attribute>'
      ------------------------------------------------------------
      xexpr(attr2) = '[@' s1 '=' s2 ']'
      xsl(attr2) = '<xsl:attribute name="' s1 '">'
            s2 '</xsl:attribute>'

any2: string any2; node n; nodelist nl
      namespace(n) != 'htl'
      any2 = '<' name(n) '>' serial(nl) '</' name(n) '>'
      -------------------------------------------------
      xpath(any2) = name(n) '/' xpath(serial(nl))
      xexpr(any2) = '[' name(n) '[' xexpr(serial(nl)) ']]'

any3: string s, any3; node n
      namespace(n) != 'htl'
      A(nl in nodelist: s != serial(nl))
      any3 = '<' name(n) '>' s '</' name(n) '>'
      -----------------------------------------
      xpath(any3) = name(n) '[ . = ' s ']'
      xexpr(any3) = '[' name(n) '=' s ']'

template1: node n; attribute a
      name(n) = ‘htl:template’ and attribute(a, n)
      not (name(a) = ‘mode’ and value(a) = ‘negate’)
      ------------------------------------------------------
      xpath(serial(n)) = xpath(serial(children(n))

template2: node n; attribute a
      name(n) = ‘htl:template’ and attribute(a, n)
      name(a) = ‘mode’ and value(a) = ‘negate’
      ------------------------------------------------------
      xpath(serial(n)) = ‘not(‘ xpath(serial(children(n)) ‘)’

param:      string s1, s2, param; node n
      serial(n) = '<htl:template htl:id="' s1 ' ">'
            s2 '</htl:template>'
      child(root, n)
      param = '<htl:param idref="' s1 '"/>'
      -------------------------------------
      xexpr(param) = xexpr(s2)
      xpath(param) = xpath(s2)
      xsl(param) = xsl(s2)

copy: ----------------------------------
      xsl('<htl:copy />') = '<xsl:copy />'
```

```
any4: string s1, s2, s3, s4; node n1, n2, n3; nodelist nl
      serial(n1) = '<' name(n1) 'idref="' s1 '">'
            serial(nl) '</' name(n1) '>'
      n2 in nl and (n3 = n2 or descendant(n2, n3))
      serial(n3) = '<' name(n3) 'htl:id="' s1 '">'
            s2 '</' name(n3) '>'
      ------------------------------------------------
      xpath(serial(n1)) = xpath(serial(extract(n2, n3)))
            xexpr(serial(n3))

name-of:    string s1, s2, s3
      s2 = '<htl:name-of idref="' s1 '">' s3 '</htl:name-of>'
      ------------------------------------------------ // viz. any4
      xsl(s2) = '<xsl:value-of select="name(' xpath(s2) ')"/>'

value-of:   string s1, s2, s3
      s2 = '<htl:value-of idref="' s1 '">' s3 '</htl:value-of>'
      ------------------------------------------------ // viz. any4
      xsl(s2) = '<xsl:value-of select="' xpath(s2) '"/>'

any5: string s1, s2, s3, s4; node n1, n2, n3; nodelist nl
      serial(n1) = '<htl:template>' serial(nl) '</htl:template>'
      child(root, n1) and n2 in nl and (
            n3 = n2 or descendant(n2, n3))
      serial(n3) = '<' name(n3) 'htl:id="' s1 '">'
            s2 '</' name(n3) '>'
      ------------------------------------------------
      xpath('<' s3 'idref="' s1 '">' s4 '</' s3 '>')
      = xpath(serial(extract(n2, n3))) xexpr(serial(n3))

select:     string s1, s2, s3
      s2 = '<htl:select idref="' s1 '">' s3 '</htl:select>'
      ------------------------------------------------ // viz. any5
      xsl(s2) = '<xsl:for-each select="' xpath(s2) '>'
            xsl(s3) '</xsl:for-each>'

reject:     string s1, s2, s3
      s2 = '<htl:reject idref="' s1 '">' s3 '</htl:reject>'
      ------------------------------------------------ // viz. any5
      xsl(s2) = '<xsl:for-each select="not(' xpath(s2) ')>'
            xsl(s3) '</xsl:for-each>'

outputPattern:    string s; node n
      namespace(n) != ‘htl’
      ------------------------------------------------------------
      xsl(‘<’ name(n) serial(attribs(n)) ‘>’ s ‘</’ name(n) ‘>’)
      = ‘<’ name(n) serial(attribs(n)) ‘>’ xsl(s) ‘</’ name(n) ‘>’

output:     string s
      ------------------------------------------------------
      xsl(‘<htl:output>’ s ‘</htl:output>’)
      = ‘<?xml version=”1.0”?>
           <xsl:stylesheet version=”1.0”
             xmlns:xsl=”http://www.w3.org/1999/XSL/Transform”>
           <xsl:output method=”xml”/>’
            xsl(s)
        ‘</xsl:stylesheet>’
```