# Xemantics: a Rewriting Calculus-Based Semantics of XSLT

Claude Kirchner, Zhebin Qian, Preet Kamal Singh, Jürgen Stuber

# Xemantics: a Rewriting Calculus-Based Semantics of XSLT

Claude KIRCHNER LORIA & INRIA
Zhebin QIAN LORIA & UHP
Preet Kamal SINGH LORIA & IIT-Delhi
Jürgen STUBER LORIA & INRIA

email: {ckirchne,qian,singh,stuber}@loria.fr
WWW: http://www.loria.fr/~{ckirchne,qian,singh,stuber}

December 5, 2001

**Abstract**

XSLT is a rule-based language defined by the W3C for the transformation of XML-documents into XML-documents. XML-documents are essentially labelled rooted ordered trees or equivalently terms without arity restrictions. This motivates us to use term rewriting, a well-studied paradigm for the transformation of terms, to define and implement XSLT transformation. On the one hand term rewriting is conceptually simple and may be used as a formal definition of XSLT, on the other hand it can be used as a programming language and executed efficiently. We define a core fragment of XSLT in the ELAN language, developed in the PROTHEO group over recent years. The code generated by the ELAN compiler is efficient enough to use this definition directly to transform nontrivial documents.
**Keywords**: XSLT, ELAN, rule-based language, term rewriting, rewriting calculus, rewrite strategies.

## 1 Introduction

XSLT is a rule based transformation language proposed by the W3C (World Wide Web Consortium) for the transformation of XML documents into XML documents. One of its original purposes was to transform datas in XML format into a human-readable document to be displayed or printed (XSLT stands for eXtensible Stylesheet Language Transformations), and it is now also becoming a standard tool to transform data between various XML formats. As XML is rapidly becoming a standard format for data storage and exchange, rule based languages like XSLT are set to become an important part of tomorrows computing infrastructure. For instance, XSLT begins to be extensively used in areas as diverse as automatic generation of web pages [13], documentation generation [20], data processing and presentation on a variety of devices in health care [12] and knowledge representations [8, 16].

However, currently XSLT is described informally, with a normative part of about 70 pages in plain English [6], and it uses the definitions of XML [4], Namespaces in XML [3] and XPath [7] which are also informal. Although XSLT is well-designed, the lack of a formal semantics poses several problems. From previous experiences with formal specification it is likely that an informal specification is incomplete or inconsistent in certain respects, or that it does not completely resolve the ambiguities inherent in natural language. All of these problems may lead to differences in the behavior of implementations, which may cause for instance portability problems in practice. It would also be desirable to have a formal specification to be able to prove the correctness of both implementations of the language and of specific transformations written in XSLT, in particular if XSLT is to be used in security-critical applications. Finally, a well-written formal specifications is much more concise, for example the formal specification

for an older version of XPath given by Wadler [18] fits on one page compared to ten pages for the informal one. This conciseness makes it much easier to use.

From the point of view of theoretical computer science XML-documents are just terms without arity restrictions. Terms are well-known concepts in semantics, logic and in particular in equational reasoning, where equations can be oriented into *rewrite rules* that transform terms into equal terms [1, 14]. Because XSLT is a rule based language, it is natural to use the rewrite rule based technology developed over the last thirty years to give it, as least in part, a formal semantics. Up to now there is very little work on formal semantics for XSLT and we are only aware of Wadler's papers on the semantics of XPath [18, 19]. On the other hand, there are pletty of alternative XML processing systems some of them having a formal semantics like Xduce [11].

Our goal in this work is therefore to propose a formal semantics for the core of XSLT, using first a formalism of judgments and proofs, and then the ELAN language and system. ELAN [15, 2] is a general rule based programming language whose formal semantics is based on the rewriting calculus [5] and which can be compiled for efficient execution.

ELAN supports the definition of context free languages, which we use to give a concise partial definition of the XML syntax. However the built-in lexical analyzer imposes certain restrictions, for instance we need to put quotes around text to parse it as strings. To abstract away from these syntactic considerations we transform the XML syntax into an internal abstract syntax that uses the standard prefix functional notation for terms, with a list of attributes and their values added. We then describe XSLT using these abstract terms. Using ELAN we can thus provide both a formal and executable specification of core XSLT .

The paper is organized as follows: In Section 2 we informally describe XSLT and term rewriting and highlight both their similarities and differences. In Section 3 we give a formal semantics of XSLT based on inference rules. In Section 4, 5 and 6 we present the definitions for XML, XPath and XSLT in ELAN.

## 2 A comparison of term rewriting and XSLT

In term rewriting the application of a rule replaces a matching subterm by another subterm. This is iterated until no more replacements are possible and a *normal form*, the result, is reached. Note that here the output created by a rule application in general serves as the input of following rule applications. Furthermore, rewriting and matching determines in a single step both the rules that are applicable and the positions where they can be applied. It remains to choose among the possible rule applications. For example, the approach commonly used for equational reasoning is to do this nondeterministically, and to ensure the *confluence property* [1] by a suitable completion of the term rewriting system, which implies that the result of all terminating computations is the same. Thirty years of research have led to a well-developed arsenal of techniques for achieving and/or proving confluence, termination and a variety of other properties of term rewriting systems [1, 14]. Another way to choose the rule applications is to specify a strategy, such as for example (leftmost-)innermost or outermost. In ELAN leftmost-innermost is used for so-called unlabeled rules, but it is also possible to define custom strategies for labeled rules. Strategies specify which rule applications are to be tried in which order for a whole computation. It contains among others operators for composition, repetition and don't-care and don't-know nondeterminism. [2] From the expressivity capabilities of rewriting, it is easy to encode a Turing machine as a term rewriting system (even with only one rule [10]), which shows Turing-completeness of this programming paradigm.

[1] Confluence means that any computations leading to different (intermediate) results can be joined later.

[2] This is similar to the tactic languages used in interactive theorem provers for higher-order logic or type theory.

There are various extensions of term rewriting systems, for instance by matching modulo some equational theory like associativity and commutativity (this is also supported by ELAN). This allows to treat common mathematical theories that satisfy these axioms (numbers, sets, multisets) and also allows an easy expression of algorithms for constraint solving, as for example two equations that need to react in some way can be picked out by the matching algorithm. As we will see, to help with XPath matching, it will be useful to match modulo some restricted form of *context matching*, where context variables can be used to match the path of arbitrary length in patterns such as $a//b$. Finding suitable restrictions of context matching is currently ongoing research.

In contrast to rewriting, XSLT creates in a single pass a target tree (an output term) from a source tree (an input term). Each application of a template rule creates a fragment of the output that extends the partial tree towards the leaves. Later rule applications do not use this fragment in any way, they neither read nor modify it. The source tree is also not modified, it is only read, for example by the XPath matching algorithm.

In XSLT, matching is used in two different ways. It is used to determine whether a template rule is applicable at a certain node (position) in the source tree, and it is used by the `apply-templates` instruction inside a template to determine a new list of nodes in the input term to transform. The first form of matching is restricted to the path from the root to the node where the rule has to be applied, while the second may use paths that traverse the source tree in arbitrary directions (the axes in XPath). The template rules can be applied to the selected nodes in parallel, since the computations for two nodes do not influence each other. For each node a rule is chosen according to a preference that is determined by the import hierarchy, explicit priorities and implicit priorities from the form of the path to match.

XSLT has variables and parameters, which can be bound to a value only once, like variables in functional or logic programming languages. By using templates that do recursive calls without generating any output, XSLT can compute arbitrary computable functions, i.e. it is also Turing complete.

This informal comparison shows that there are substantial differences in the core mechanisms of term rewriting and XSLT which make it non straitforward to transfer results from term rewriting to XSLT. Furthermore, the kinds of matching commonly available in term rewriting implementation (e.g. associativity, commutativity and more generaly algebraic equational axioms) are currently not able to directly simulate XPath matching. Consequently the encoding of XSLT in rewriting calculus is a deep one as we could have expected at first sight that a shallow encoding would have been sufficient, and indeed more interesting.

## 3 A formal semantics for a core fragment of XSLT

We define a formal semantics for a core subset of XSLT by inference rules.

### 3.1 Scope

We formalize the core of XSLT [6] by inference rules. The formalization covers the following aspects:

- Data model with root, element, attribute and text nodes.[3]
- Template rule matching[4]
- Selection of nodes in the source document by `xsl:apply-templates`[5]

[3] We do not cover processing instructions and comments.

[4] We don't handle mode, name and priority attributes.

[5] We do not cover the `mode` attribute.

- `xsl:value-of` to insert text into the output
- Construction of the output tree
- `xsl:attribute`, `xsl:element`
- `xsl:for-each`, `xsl:if`
- Sorting, on an abstract level

When some feature is not formalized completely we will note this explicitly.

We do not define the semantics of XPath in this section. Instead we reuse the second one given by Philip Wadler in [19]. Wadler formally defines a function $\mathcal{S}^a[\![e]\!]n$ that computes the value of an XPath expression $e$ at the node $n$ of an implicitly given source document.

The formal semantics of this section shows only how sorting is used, but abstracts away from the concrete sort orderings and how to specify them.

### 3.2 Data model

We consider XML documents independently of any DTD, which correspond to untyped terms with unrestricted arity. Equivalently, these terms may be considered as ordered trees with a root, where nodes are labeled by operators. Nodes are labeled with a node type, either root, element, attribute or text. Element and attribute nodes are additionally labeled with a name, text nodes with a string[6]. The labeling is restricted as follows:

- The root of the tree must be the only node labeled root.
- Text nodes have no children.
- There are no adjacent sibling text nodes and no empty text nodes.[7]
- Attributes have a single text node as their child.
- Attribute siblings have distinct names.
- Attribute children are to the right of all other children.

We call a term that satisfies that restriction an *XML term*. We write XML terms as

$$\mathit{operator}(\mathit{child}_1, \ldots, \mathit{child}_n)$$

where operator is either `/` for the root node, *name* for an element node, *@name* for an attribute node, or `"`*text*`"` for a text node. We assume that we have some method to denote nodes in the tree. We let *root_node* denote the root node. Other nodes will be obtained via XPath expressions.

A *template* is a term that may contain element nodes from the XSLT and the target namespaces. Lists are separated by commas. Where necessary we will put parentheses around lists, and in particular we write $()$ for the empty list.

Let $r$ be a list of XML terms. Then *merge-adjacent*$(r)$ is the list of XML terms in $r$ with adjacent text nodes merged. Let $s$ be a string value. Then *remove-empty*$(s)$ is a list consisting of the single text

[6] We assume that the namespace is part of the name. We use this only in templates to distinguish instructions in the `xsl` namespace from literal result elements in other namespaces.

[7] This is an artifact of the markup nature of XML.

node $s$ if $s$ is nonempty, and the empty list otherwise. *string*$(v)$ converts an XPath value $v$ to a string as specified in Section 4.2 of the XPath Recommendation [7].

A *sorting specification* is a list of XML terms of the form `sort`$(as)$ where $as$ are the attributes describing the sort order. For a node-set $S$ and a sorting specification $so$ we let *sort*$(so, S)$ be the list of the nodes in $S$ in the specified order. For an empty sort specification the set is put in document order, that is, the order obtained by a preorder traversal of the term.

### 3.3 Stylesheets

A stylesheet consists of a set of template rules. A template rule consists of an XPath pattern for the source DTD and a template. The following built-in rules are always present[8]:

```
<xsl:template match="*|/">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>
```

A template rule with pattern $p$ is said to *match* at the node $n$ in the source tree if there is some node $n'$ that is either $n$ itself or an ancestor of $n$ such that $n \in \mathcal{S}^a[\![p]\!]n'$.

### 3.4 Judgments

Let $S$ be the stylesheet, $s$ an XML term (the source tree), $l$ a list of nodes in $s$ (the current node list), $i$, $1 \leq i \leq$ *length*$(l)$ (the index of the current node $l_i$ in $l$), $t$ a template, and $r$ a list of XML terms (the result tree).

$s \vdash_S r$ the result of applying the stylesheet $S$ to the source tree $s$ is $r$

$s, l \vdash_S r$ the result of applying the stylesheet $S$ in the context $s, l$ is $r$

$s, l, i \vdash_S r$ the result of applying the stylesheet $S$ in the context $s, l, i$ is $r$

$s, l, i, t \vdash_S r$ the template $t$ is transformed into the result $r$ in the context $s, l, i$

$\mathcal{S}^a[\![e]\!]l_i = v$ the result of the XPath expression $e$ w.r.t. the node $l_i$ is the value $v$ (This is the notation used by Wadler [19] where the source document remains implicit.)

Note that by the definition of these judgments ill-formed XML terms are not allowed to occur. For instance, if $S$ is a stylesheet that tries to add two attributes with the same name then we cannot prove any judgment $s \vdash_S r$.

### 3.5 Rules

The rules describe deductions, such that the judgment below the line (the conclusion) holds if the judgments above the line (the premises) hold. To apply an XSLT stylesheet one starts at the bottom of the Start rule and recursively considers the premises until a complete tree (a proof) is obtained.

[8] We don't handle modes, processing instructions and comments, hence the corresponding rules are absent.

The first three rules describe the core mechanism of XSLT up to the selection of a template $t$ to be processed.

*Start*
$$\frac{s, \textit{root\_node} \vdash_S r}{s \vdash_S \texttt{/}(r)}$$

*Traverse*
$$\frac{s, l, 1 \vdash_S r_1 \quad \ldots \quad s, l, n \vdash_S r_n}{s, l \vdash_S \textit{merge-adjacent}(r_1, \ldots, r_n)}$$

where $l$ has the length $n$.

*Apply Template Rule*
$$\frac{s, l, i, t \vdash_S r}{s, l, i \vdash_S r}$$

where <xsl:template match=$p$>$t$</xsl:template> is a template rule in $S$ such that $p$ matches at $l_i$. The rule is selected according to import precedence and priority. It is an error if the selection of the rule is ambiguous.

The following rules copy literal content from the template to the result.

*Literal List*
$$\frac{s, l, i, t_1 \vdash_S r_1 \quad \ldots \quad s, l, i, t_k \vdash_S r_k}{s, l, i, (t_1, \ldots, t_k) \vdash_S \textit{merge-adjacent}(r_1, \ldots, r_k)}$$

where $n \geq 0$.

*Literal Element*
$$\frac{s, l, i, t \vdash_S r}{s, l, i, \textit{op}(t) \vdash_S \textit{op}(r)}$$

where $op$ is an element with a name not in the XSL name space.

*Literal Attribute*
$$\frac{s, l, i, t \vdash_S r}{s, l, i, \texttt{@}\textit{op}(t) \vdash_S \texttt{@}\textit{op}(r)}$$

where $op$ is an attribute with a name not in the XSL name space.

*Literal Text*
$$\frac{}{s, l, i, \texttt{"}\textit{text}\texttt{"} \vdash_S \texttt{"}\textit{text}\texttt{"}}$$

where "*text*" is a text node.

This final set of rules evaluates XSLT-instructions in a template.

*Eval* `apply-templates`
$$\frac{\mathcal{S}^a[\![p]\!] l_i = S \qquad s, \textit{sort}(so, S) \vdash_S r}{s, l, i, \texttt{xsl:apply-templates}(\texttt{@select}(p), so) \vdash_S r}$$

where $so$ is a sorting specification that may be empty. If no select attribute is given we assume it is `node()`, which selects all children of the current node, as is the default.

*Eval* `for-each`
$$\frac{\mathcal{S}^a[\![p]\!] l_i = S \qquad s, l_1, 1, t \vdash_S r_1 \quad \ldots \quad s, l_n, n, t \vdash_S r_n}{s, l, i, \texttt{xsl:for-each}(\texttt{@select}(p), so, t) \vdash_S \textit{merge-adjacent}(r_1, \ldots, r_n)}$$

where $sort(so, S) = l_1, \ldots, l_n$ and $n \geq 0$.

*Eval* `if`$_1$

$$\frac{\mathcal{S}^a[\![e]\!]l_i = v \qquad s, l, i, t \vdash_S r}{s, l, i, \texttt{xsl:if}(\texttt{@test}(p), t) \vdash_S r}$$

if $v = \text{true}$.

*Eval* `if`$_2$

$$\frac{\mathcal{S}^a[\![e]\!]l_i = v \qquad s, l, i, t \vdash_S r}{s, l, i, \texttt{xsl:if}(\texttt{@test}(p), t) \vdash_S ()}$$

if $v \neq \text{true}$.

*Eval* `value-of`

$$\frac{\mathcal{S}^a[\![e]\!]l_i = v}{s, l, i, \texttt{xsl:value-of}(\texttt{@select}(e)) \vdash_S \textit{remove-empty}(\textit{string}(v))}$$

*Eval* `element`

$$\frac{\mathcal{S}^a[\![e]\!]l_i = v \quad s, l, i, t \vdash_S r}{s, l, i, \texttt{xsl:element}(\texttt{@name}(e), t) \vdash_S v(r)}$$

*Eval* `attribute`

$$\frac{\mathcal{S}^a[\![e]\!]l_i = v \quad s, l, i, t \vdash_S r}{s, l, i, \texttt{xsl:attribute}(\texttt{@name}(e), t) \vdash_S \texttt{@}v(r)}$$

where $r$ is a text node.

This set of rules captures formally our understanding of the informal definition of XSLT.

What does this semantics tell us about XSLT? One obvious feature is that the input is never modified. With respect to the output, once a fragment of the target tree is generated it is never again read or modified. The rules only copy the results obtained in the premises into the final result (except for the mangling of text nodes by *merge-adjacent* and *remove-empty*).

Another interesting point is that the computations need not be carried out sequentially. Both computations at different nodes and at different instances of `apply-templates` are independent of each other and can be carried out in parallel. Or one might be only interested in a certain subtree, and only carry out the computation that leads to it, discarding the rest. In the inferences we see this in the premises of the *Traverse* and *List Context* rules, which are independent of each other.[9]

## 4 Parsing and representing XML documents in ELAN

For the representation of XSLT in ELAN we have to begin by first representing XML documents, which we will do in this section, and we have to give a definition of XPath, which will be done in the next section. After that we will be able to define our core subset of XSLT transformations.

### 4.1 The XML document data model and ELAN terms

How can an XML document be expressed as an ELAN term? From the point of view of the data model, any XML document is a tree consisting of various kinds of nodes. We considered four types of nodes:

[9] This property is preserved even if variables are added, as these can be bound to a value only once, and their scope is limited to the subtree below the binding.

root nodes, element nodes, attribute nodes and text nodes.[10] In the following we give for each type of node their name (if available), their attributes, their representation in ELAN and their string-value.

Nodes in a document are linearly ordered by the *document order*, which is obtained by a pre-order traversal of the tree beginning with the root node, with attribute nodes of an element before its children.[11] It is called document order because in the XML representation the start tags of element nodes occur in this order.

### 4.1.1 The root node

The root node can be seen as an element node that is implicit in the XML document. There is only one root node in the document. It is explicitly accessible in XPath by an absolute XPath pattern beginning with '/'. Internally we represent it by `root_node`.

Normally XML requires that the root node has exactly one child, but this restriction is sometimes relaxed, for example for the output of an XSLT transformation, so we don't make it.

The string-value of the root node is the concatenation of the string-values of all the text node descendants of the root node in document order.

### 4.1.2 Element nodes

In XML documents, a pair of tags like $<tagname\ldots>\ldots</tagname>$ correspond to an element node. The name of the element is the tag name, which is required to be a qualified name. A qualified name (QName) contains a single colon, separating the name into a namespace prefix and a local part. The syntax of prefixes and local names are described by the nonterminal NCName in [3]. The prefix part is mapped to the URI reference for a declared namespace, which specifies a collection of universal names. If a default namespace is declared, then the default namespace is considered to apply to the element in the scope where it is declared, in case the element has no namespace prefix. If no default namespace has been specified, then the elements without prefix in the scope are considered to be in no namespace. Such a namespace scheme is also applicable to the attribute names. (Full details are given in *Namespaces in XML* [3]).

To specify *QName* in ELAN, we translate the grammar rules given in the XML recommendation in a straightforward way. First we declare the nonterminals *Prefix*, *LocalPart*, *QName* as sorts:[12]

```
sort    ... Prefix LocalPart QName ...;
end
```

The grammar rules themselves become operator declarations, which contain the terminal symbols. For *QName* we get

```
operator
...
  @                 :(LocalPart) QName;
  @':'@             :(Prefix LocalPart) QName;
  @                 :(QName) EleName;
```

[10] Namespace nodes, processing instruction nodes and comment nodes are not included.

[11] In XML only element or text nodes are called children of an element node, but not attributes. Nevertheless the element is called the parent of its attributes. This will become important in XPath expressions.

[12] Sorts in ELAN correspond to the types of programming languages.

```
  a                    :Prefix;
  b                    :LocalPart;
...
end
```

where @ is the place-holder for an operator argument, which is then specified by the corresponding *sort* between the brackets '(' and ')' after *':'*. As above, the first operator means that a *LocalPart* instance can be directly treated as a *QName* instance. The second operator means that a *Prefix* and a *LocalPart* connected by a *':'* character also form a *QName*. The third one means that a *QName* is also an *EleName*. (More details are given in *Namespaces in XML* [15].) The constant declarations for 'a' and 'b' are examples that need to be replaced by the actual names used in an application. Then *a:b* is a typical qualified XML name, and it is a term of sort *QName* in ELAN. However, the lexical analyzer of ELAN imposes certain restrictions on *Prefix* and *LocalPart* that are not present in XML. E.g., *a-b* is a qualified *LocalPart* name in XML, but it is not a correct constant name in ELAN.

To distinguish source document, target document and stylesheet we use sorts with the prefix *'S'* for the source document, sorts with the prefix *'T'* for the target document and sorts with the prefix *'XSL'* for the XSL stylesheet. After the sorts we define some common variables. We will use the first example from the XML recommendation to illustrate this.

- For the source document, *SEleName* (without namespace) can be defined as

```
@                    :(SLocalPart) SQName;
@                    :(SQName) SEleName;
root                 :SLocalPart;
doc                  :SLocalPart;
chapter              :SLocalPart;

...
```

  However, the names can also be directly declared as follows, (which means the same thing)

```
root                 :SEleName;
doc                  :SEleName;
chapter              :SEleName;
...
```

- Similarly, for the target document, *TEleName* can be defined as (maybe with a default namespace)

```
root                 :TEleName;
html                 :TEleName;
body                 :TEleName;

...
```

- For the XSLT stylesheet, if needed, *XSLEleName* can be defined as (with the *xsl* namespace prefix)

```
xsl':'@             :(XSLLocalPart) XSLQName;
XSLQName            :XSLEleName;
template            :XSLLocalPart;
apply-templates     :XSLLocalPart;

...
```

To do this more concisely in ELAN we can use the pre-processor. We first list all the names of the same type in the specification file as

```
stagids     root.doc.title.chapter.section.para.note.emph.nil
```

and use a FOR EACH directive for the preprocessor in the operator declarations

```
FOR EACH Id:identifier SUCH THAT Id:=(listExtract)elem(stagids):
{Id:SEleName;}
```

This will insert a constant declaration for each of the names.

In the following, we have put the tag names of sort *EleName* into the list `stagids` in the specification file. For attribute node names we use the list `sattrids`, and the respective names for the target document are in `ttagids` and `tattrids`.

An element (identified by the sort *Ele*) has a name, and may have attribute nodes or child nodes. The child nodes of an element node may be other element nodes or text nodes, which are represented by strings.

```
@           :(Ele)  Child;
@           :(string) Child;
```

Common variables are

```
str         :string;
pL,cL       :list[Child];
```

`string` is a built-in data-type in ELAN. The lexical analyzer of ELAN requires that string constants are enclosed in double quotes, with internal quotes and backslashes escaped by a preceding backslash.

- If the element has no attributes and no child nodes then it is represented in XML as

  ```
  <elename/>
  ```

  In ELAN it is represented in functional notation by

  ```
  @()                 :(EleName) Ele;
  ```

- An element having no attributes, but some children is written in XML as

  ```
  <elename> ... </elename>
  ```

  The number of children between the pair of tags is arbitrary. Such an operator of variable arity can not be directly represented in ELAN, but it can be easily simulated by putting the arguments in a list. The functional notation thus becomes

```
@(@)                :(EleName list[Child]) Ele;
```

When nothing occurs between the tags, the ELAN representation should be

```
elename(nil)
```

The string-value of an element node is the concatenation of the string-values of all text node descendants of the element node in document order.

### 4.1.3 Attribute nodes

An element node can also have several attribute nodes. An attribute node contains an attribute name and a string value, where the string-value implies the attribute value. In our data model, an attribute node is associated with its parent element, but it is different from its parent node's children.

```
@=@                 :(AttrName string) Attr;
```

Common variables are

```
at                  :Attr;
aL,aL1              :list[Attr];
```

In XML, if the element has attributes then the attribute declarations follow the *elename* in the start tag.

```
<elename a1=str1 a2=.../>
<elename b1=str1 b2=...> ... </elename>
```

The number of attribute nodes is also variable, so we use a list representation here, too. We can now represent elements with attributes by

```
@(@)                :(EleName list[Attr]) Ele;
@(@,@)              :(EleName list[Attr] list[Child]) Ele;
```

All the elements are converted to the standard form, in which if their attributes or children are absent, then we put the empty list *nil* in that place.

```
[]t() => t(nil, nil)
end

[]t(pL) => t(nil, pL)
end

[]t(aL) => t(aL, nil)
end
```

Some pre-processing is required to ensure some properties that cannot be ensured on the syntactical level. It is assumed that an element node does not have the same attribute defined more than once in its attribute list. Also, we cannot ensure that the value of the *id* attribute for all the elements is unique, which is required by XML.

### 4.1.4 Text nodes

A text node is represented by a string, which is the node's value. The text node has no node name. Such a node can only be a leaf node of the tree, as it has neither any attributes nor any children.

## 4.2 Parsing XML documents as ELAN terms

With the following declarations we can directly parse XML and convert it to functional representation.

```
<@/>          :(EleName) XEle;
<@> @ </@>    :(EleName XML EleName) XEle;
<@ @> @ </@>:(EleName AttrS XML EleName) XEle;
@             :(string) XEle;

@             :(Attr) AttrS;
@ @           :(Attr AttrS) AttrS;

@             :(XEle) XML;
@ @           :(XEle XML) XML;

t,t1, tg      :EleNme;
at            :Attr;
xe            :XEle;
x             :XML;
al            :AttrS;
```

In the above definition, the two *EleName* instances for opening and closing tag should have the same name. However, this cannot be expressed by an operator definition. Instead it is implicit in the rule for the transformation to functional representation given below, which can only be applied if the tags have the same name.

The *P()* function is the main parsing function, which converts the whole XML document in to a tree recursively. But we call the *Prs()* function (which uses the *P()* function) to return the pointer to the root node of that tree only. The pointer to a node is defined by the sort *Node*.

```
Prs(@)              :(XML) Node;
P(@)                :(XML) list[Child];
P(@)                :(AttrS) list[Attr];

[] Prs(x) => root_node(P(x))
end

[] P(xe x) => P(xe) @ P(x)
end

[] P(<tg/>) => tg().nil
end

[] P(<tg> x </tg>) => tg(P(x)).nil
end
```

```
[] P(<tg al> x </tg>) => tg(P(al),P(x)).nil
end

[] P(str) => str.nil
end

[] P(at al) => P(at) @ P(al)
end

[] P(at) => at.nil
end
```

This technique can be applied to the parsing of the source XML document and the XSLT stylesheet. The reverse procedure can be applied to the result term obtained after transformation, outputting it as a XML document. This function is called as *I()* function, and the rules for it are almost the inverse of the parsing rules.

```
e          :XEle;
c          :Child;
T          :list[Child];
l          :XML;

I(@)       :(list[Child]) XML;
I(@)       :(Child) XEle;
I(@)       :(list[Attr]) AttrS;

[] I(c.nil) => I(c)
end

[] I(c.T) => e I(T)
        where e:= () I(c)
end

[] I(tg(nil,nil)) => <tg/>
end

[] I(tg(al,nil)) => <tg I(al)/>
end

[] I(tg(nil,T)) => <tg> I(T) </tg>
end

[] I(tg(al,T)) => <tg I(al)> I(T) </tg>
end

[] I(str) => str
```

```
    end

    [] I(at.nil) => at
    end

    [] I(at.al) => at I(al)
    end
```

#### 4.2.1 Examples

In the example D.1 of the Appendix of *XSLT Specification*, the source document appears as

```
<doc>
    <title>"Document Title"</title>
    <chapter>
        <title>"Chapter Title"</title>
        <section>
            <title>"Section Title"</title>
            <para>"This is a test."</para>
            <note>"This is a note."</note>
        </section>
        <section>
            <title>"Another Section Title"</title>
            <para>
                "This is "
                <emph> "another" </emph>
                " test."
            </para>
            <note> "This is another note." </note>
        </section>
    </chapter>
</doc>
```

which can be parsed to the following ELAN term:

```
root(doc(title("Document Title".nil).chapter(title
("Chapter Title".nil).section(title("Section Title
".nil).para("This is a test.".nil).note("This is a
 note".nil).nil).section(title("Another Section Ti
tle".nil).para("This is".emph("another".nil)."test
.".nil).note("This is another note.".nil).nil).nil
).nil).nil)
```

### 4.3 Node Representation

In the previous section, we discussed how an XML tree can be mapped to a term in ELAN and vice versa. It remains to represent references to nodes in the tree. If we would use the functional representation together with the standard notion of position from term rewriting (a list of integers that tell to which

child to descend) we would have to descend from the root node to access any node in the tree, which would make operations in XPath expensive. Thus we chose a special representation for element nodes that carry their parent and their following siblings as a context. Note that we can also access preceding siblings via the parent.

1. The root node is represented by the operator *root_node*:

```
root_node(@)        :(list[SChild]) EleNode;
```

2. The representation of an element node contains its parent node, the element itself and its following sibling nodes. Children and attributes can be accessed in the element.

```
@/@.@               :(EleNode Sele list[Schild]) EleNode;
@                   :(EleNode) Node;

pn                  :EleNode;
```

3. If the node is an attribute node, then we only need its parent node and itself to identify it. The attribute nodes do not have any children.

4. If the node is a text node, then we can use the same scheme as for an element node. That is, its parent node representation plus itself and the following sibling node. As in the case of an attribute node, a text node cannot be the parent of any other node. As a result, for both these cases, no step can follow these representations.

```
@                   :(Sattr) EndStep;
@.@                 :(string list[SChild]) EndStep;
@/@                 :(EleNode EndStep) Node;

es                  :EndStep;
n                   :Node;
nL                  :list[Node];
```

This node representation both identifies the nodes and provides cheap access to all closely related nodes.

## 5 Evaluating XPath Expressions in ELAN

### 5.1 Prerequisite: Some list operations

We first define some common operations for lists.

- The *in* function tests whether an element occurs in a list of elements.

```
@ in @              :(Element list[Element]) bool;
```

- The function *-()* removes the elements of the second list from the first list.

```
-(@,@)              :(list[Element] Element) list[Element];
-(@,@)              :(list[Element] list[Element]) list[Element];
```

- The standard library function *concat()* concatenates two lists. It has the infi x operator '@' as alias.

```
concat(@,@)          :(list[Element] list[Element]) list[Element];
```

In our applications, if we have defi ned some operation on an element with a result of type *List* as

```
op(@)       :(Element) list[Result];
```

then that operator is often extended to a list of elements as

```
op(@)       :(list[Element]) list[Result];
```

The extension is defi ned by the following two rules, where *e* is an element and *eL* is a list of elements:

```
[] op(nil) => nil
end

[] op(e.eL) => op(e) @ op(eL)
end
```

### 5.2 Location paths

A location path in XPath is used to select one or more nodes in a tree by exploring the tree step by step. It can be an absolute path or a relative path. For an absolute path the exploration starts at the root node, while for a relative path it starts at some given node. We may view an absolute path as a relative path that is given the root node as the start node.

```
@                   :(RelLocPath) LocPath;
@                   :(AbsLocPath) LocPath;

/                   :AbsLocPath;
/@                  :(RelLocPath) AbsLocPath ;
@                   :(AbbrAbsLocPath) AbsLocPath;

patt                :LocPath;
path,rp,rp1,rp2     :RelLocPath;
st,st1,st2          :Step;
```

To achieve a uniform presentation we transform absolute paths into relative paths.

```
[] /rp => root + rp
end

[] / => root
end
```

The above also uses a function + that concatenates location paths. It is defi ned on as follows:

```
@+@                 :(LocPath LocPath) LocPath assocLeft;

[] rp + st => rp/st
end

[] rp1 + rp2/st => rp1 + rp2 + st
end
```

A location path consists of one or more *steps*. Each step contains an axis specifier, a node test, and optionally one or more predicates. The axis can indicate (or imply) a principal node type, e.g. attribute or element, and the direction of tree-exploring, e.g. ancestor or child. When the node belongs to range of the indicated axis, the node test defines a test on the node, for example for a certain node name. The predicate imposes further conditions on the selected nodes, e.g. indicating the proximity position in a node list.

Here is an example of location path with unabbreviated steps `/child::doc/child::chapter[position()=` which selects the second *section* of the fifth *chapter* of the *doc* element which is a child of the *root* node.

```
@                   :(Step) RelLocPath;
@/@                 :(RelLocPath Step) RelLocPath;
@                   :(AbbrRelLocPath) RelLocPath;

@                   :(NodeTest) Step;
@@                  :(AxisSpecifier NodeTest) Step;
@@                  :(Step Predicate) Step;
@                   :(AbbrStep) Step;

@':'':'             :(AxisName) AxisSpecifier;
@                   :(AbbrAxisSpecifier) AxisSpecifier;

ancestor            :AxisName;
ancestor-or-self    :AxisName;
attribute           :AxisName;
child               :AxisName;
descendant          :AxisName;
descendant-or-self  :AxisName;
following           :AxisName;
following-sibling   :AxisName;
parent              :AxisName;
preceding           :AxisName;
preceding-sibling   :AxisName;
self                :AxisName;

subnode             :AxisName;
subnode-or-self     :AxisName;

@                   :(NameTest) NodeTest;
@()                 :(NodeType) NodeTest;
```

```
[@]                 :(PredicateExpr) Predicate;
@                   :(Expr) PredicateExpr;

*                   :NameTest;
@                   :(QName) NameTest;

text                :NodeType;
node                :NodeType;

@                   :(int) Expr;
@                   :(bool) Expr;

@                   :(SEleName) QName;
@                   :(SAttrName) QName;
```

Xpath also provides some reserved wild-card node test names as follows :

- *node()* is a node test which matches any node of any type
- *text()* is a node test which matches any text node
- * is always true for any node of the principal node type

Common variables are:

```
a,a1                :AxisName;
nt,nt1              :NodeTest;
```

XPath also defi nes abbreviations for certain common expressions. Note that we write `//` as `/ /`, because in ELAN `//` starts a comment. `@` and `.` also need to be escaped in declarations, but not in rules.

```
/ /@              :(RelLocPath) AbbrAbsLocPath;
@/ /@             :(RelLocPath Step) AbbrRelLocPath;

'.'                  :AbbrStep;
'.''.'               :AbbrStep;
'@'                  :AbbrAxisSpecifier;
```

They get transformed into unabbreviated expressions by the following rules:

```
[] / /rp => descendant-or-self::node() + rp
end

[] rp/ /st => rp/descendant-or-self::node()/st
end

[] . => self::node()
end
```

```
[] .. => parent::node()
end

[] root => self::root
end

[] nt => child::nt /* except root node*/
end

[] @nt => attribute::nt
end
```

### 5.3 Operations on Location Paths

#### 5.3.1 Axis relations between nodes

We now defi ne operations for all the axes of XPath, and for their application to nodes.

- The root for all the nodes.

```
[] root(root_node(cL)) => root_node(cL).nil
end

[] root(pn/se.cL) => root(pn)
end

[] root(pn/es) => root(pn)
end
```

- Relation operation defi ned over a list of nodes

```
@(@)                :(AxisName Node) list[Node];
@(@)                :(AxisName list[Node]) list[Node];

[] a(nil) => nil
end

[] a(n.nL) => a(n)@a(nL)
end
```

- Self, parent and ancestor relations. These are basic relations which are directly supported by our node representation.

```
[] self(n) => n.nil
end
```

```
[] parent(root_node(cL)) => nil
end

[] parent(pn/t(aL,pL).cL) => self(pn)
end

[] parent(pn/es) => self(pn)
end

[] ancestor(root_node(cL)) => nil
end

[] ancestor(pn) => ancestor(parent(pn))@parent(pn)
end

[] ancestor(pn/es) =>ancestor(pn)@self(pn)
end

[] ancestor-or-self(n) => ancestor(n)@self(n)
end
```

- Attribute, child and descendant relationships.

```
[] attribute(root_node(pL)) => nil
end

[] attribute(pn/str.cL) => nil
end

[] attribute(pn/t(aL,pL).cL) => getattribute(pn/t(aL,pL).cL,aL)
end
```

The *getattribute()* function takes the representation of the node and its attribute list. Then it appends each element of the attribute list to the representation of the node to get representations for its attribute nodes as follows:

```
getattribute(@,@)   :(Node list[SAttr]) list[Node];

[] getattribute(pn,nil) => nil
end

[] getattribute(pn,at.aL1) => pn/at.getattribute(pn,aL1)
end
```

For the child relation we use the *getchild()* function, which is analogous to the *getattribute()* function.

```
[] child(pn/str.cL) => nil
end

[] child(root_node(pL)) => getchild(root_node(pL),pL)
end

[] child(pn/t(aL,pL).cL) => getchild(pn/t(aL,pL).cL,pL)
end

getchild(@,@)       :(Node list[SChild]) list[Node];

[] getchild(pn,nil) => nil
end

[] getchild(pn,str.cL) => pn/str.cL.getchild(pn,cL)
end

[] getchild(pn,t(aL,pL).cL) => pn/t(aL,pL).cL.getchild(pn,cL)
end
```

The descendant relation can then be defi ned using the child relation.

```
[] descendant(pn) => descendant-or-self(child(pn))
end

[] descendant(pn/es) => nil
end

[] descendant-or-self(n) => self(n)@descendant(n)
end
```

- Some sibling relationships. Sibling relationships are very complex, in the sense that we need to process the contents of some steps to get the result.

```
[] following-sibling(root_node(cL)) => nil
end

[] following-sibling(pn/str.nil) => nil
end

[] following-sibling(pn/t(aL,pL).nil) => nil
end

[] following-sibling(pn/str.t(aL,pL).cL) =>
                  pn/t(aL,pL).cL
                  .following-sibling(pn/t(aL,pL).cL)
```

```
end

[] following-sibling(pn/str.str1.cL) =>
                  pn/str1.cL
                  .following-sibling(pn/str1.cL)
end

[] following-sibling(pn/t(aL,pL).t1(aL1,pL1).cL) =>
                  pn/t1(aL1,pL1).cL
                  .following-sibling(pn/t1(aL1,pL1).cL)
end

[] following-sibling(pn/t(aL,pL).str.cL) =>
                  pn/str.cL
                  .following-sibling(pn/str.cL)
end
```

The preceding-sibling relation uses the *-()* function. We find all the children of its parent node, then subtract the current node and the following-siblings of the current node from it to get the preceding-siblings.

```
[] preceding-sibling(root_node(cL)) => nil
end

[] preceding-sibling(pn/el.cL) =>
       -(child(pn), pn/el.cL . following-sibling(pn/el.cL))
end

[] preceding-sibling(pn/str.cL) =>
       -(child(pn), pn/str.cL . following-sibling(pn/str.cL))
end

[] preceding-sibling(pn/at) => nil
end
```

- Some extended relations

  The *following* axis contains the nodes after the context node in document order, excluding descendants of the context node itself and excluding attribute nodes (and namespace nodes). Analogously the *preceding* axis contains nodes before the context node.

```
[] following(n) =>
       descendant-or-self(following-sibling(ancestor-or-self(n)))
end

[] preceding(n) =>
       descendant-or-self(preceding-sibling(ancestor-or-self(n)))
end
```

The *subnode* axis returns all the descendant nodes and all the attribute nodes of itself and its descendants. We use it internally as the inverse of the *ancestor* axis.

```
[] subnode(pn) => attribute(pn) @ subnode-or-self(child(pn))
end

[] subnode(pn/es) => nil
end

[] subnode-or-self(n) => self(n) @ subnode(n)
end
```

### 5.3.2 Some functions for XSLT

The XPath functions will be called by XSLT transformation rules. The interfaces between them are the two functions *Match()* and *Select()*.

The function *Match()* checks whether the current node matches the pattern given in a template rule.

```
Match(@,@)  :(Node LocPath) bool;
```

The *Select()* function will produce a node list, starting from the current node (or node list) according to a path expression.

```
Select(@,@)         :(Node LocPath) list[Node];
Select(@)           :(Node) list[Node];
```

## 5.4 Implementation of the *Match()* function

The *Match()* function is equivalent to the match function of Wadler [18].

```
[] Match(n,path) => n in Select(descendant-or-self(root(n)),path)
end
```

Now, we can express all the operations on patterns in XSLT by the *select()* function.

## 5.5 Implementation of the *Select()* function

We start by defi ning a function *select()* that computes a list of matching nodes that is not necessarily in document order. After that we will defi ne *Select()* in terms of *select()*.

### 5.5.1 Single-step path without Predicate

For *select()* in case of a step with axis *self*, we only need to select those nodes from the current list which match the given node test.

```
[] select(n, self::node()) => self(n)
end

[] select(pn/str.cL, self::text()) => self(pn/str.cL)
end
```

```
[] select(n, self::text()) => nil
end

[] select(root_node(cL), self::root) => self(root_node(cL))
end

[] select(n, self::root) => nil
end

[] select(root_node(cL), self::t) => nil
end

[] select(pn/t(aL,pL).cL, self::t) => self(pn/t(aL,pL).cL)
end

[] select(n, self::t) => nil
end

[] select(pn/an=str, self::an) => self(pn/an=str)
end

[] select(pn/an=str, self::an1) => nil
end

[] select(nil, self::nt) => nil
end

[] select(n.nL,self::nt) =>
           select(n,self::nt)@select(nL,self::nt)
end

[] select(n,.) => self(n)
end
```

For a single-step location path with any axis specifi er other than *self*, it can be transformed to a pattern with axis *self*.

```
[] select(n,a::nt) =>
               select(a(n),self::nt)
end

[] select(n,..) => parent(n)
end
```

For example,

```
[] select(n,ancestor::nt) =>
                select(ancestor(n), self::nt)
end
```

### 5.5.2 Single-step path with predicate

If the step has some additional predicates then we use them to filter the node list.

```
[] select(n,a::nt[expr]) =>
              pred(select(n,a::nt),expr)
end
```

The *pred()* function can remove the elements which do not satisfy *expr* from the list returned by the *select()* function. For example,

```
i           :int;
pred(@,@)   :(list[Node] Expr) list[Node];

[] pred(nL,position()=i) => self(i-th elem(nL))
end
```

### 5.5.3 Multi-step path

We process multi-step paths by applying *select()* step-by-step, where the each step is applied to the node list resulting from the previous step.

```
[] select(n,path/st) => select(select(n,path),st)
end
```

## 5.6 Ordering the selected nodes

The node list returned by *select()* is in general not in document order. In the case of a single step, the functions have been defined so that the list is in the right order, but the list resulting from applying *select()* to a multi-step path or to a list of nodes will in general not be in document order, so we have to reorder the resulting node list.

```
Select(@,@)         :(Node LocPath) list[Node];

[] Select(n,path/st) => reorder(nL,n1L)
                    where nL:= () subnode-or-self(root(n))
                    where n1L:= () select(n,path/st)
end

[] Select(n,st) => select(n,st)
end
```

We define a *reorder()* function to reorder the list of nodes. The method of re-ordering nodes is that each node of the whole source tree is tested in order whether it is in the result list. If the node is in the list, it is put into a new list and removed from the old list.

```
reorder(@,@)        :(list[Node] list[Node]) list[Node];
reorder(@,@)        :(Node list[Node]) list[Node];

[] reorder(nil,nL) => nil
end

[] reorder(nL,nil) => nil
end

[] reorder(n.nL,n1L) => n . reorder(nL,-(n1L,n))
                    if n in n1L
end

[] reorder(n.nL,n1L) => reorder(nL,n1L)
end
```

### 5.7 A high-efficiency *Match()* method

In practice, the *Match()* function defined below is of quite low efficiency.

```
[] Match(n,path) => n in select(descendant-or-self(root(n)),path)
end
```

Here, we are trying to find all the nodes in the source tree which satisfy the given pattern, and then to check whether the given node lies in the result node list or not. However, this way we also compute many other, unrelated nodes, which are not needed to test whether a pattern matches the node. It will be more efficient to go backwards from the node that we want to test. Based on this idea, a high-efficiency *Match()* is presented. To implement this we start from the current node and invoke the *select()* function based on the reversed path. The reverse path when applied on the current node gives that set of nodes, which would give the current node on application of the original path. To test for a match it suffices to check whether this set is non-empty.

```
isEmpty(@)          :(list[Node]) bool;

[] Match(n,path) => not isEmpty(select(n,rev(path)))
end

[] isEmpty(n.nL) => false
end

[] isEmpty(nil) => true
end

reverse(@)          :(LocPath) LocPath;
rev(@)              :(LocPath) LocPath;
last(@)             :(LocPath) LocPath;
first(@)            :(LocPath) LocPath;
```

```
[] rev(st) => first(st) + last(st)
end

[] rev(rp) => first(rp) + reverse(rp) +  last(rp)
end

[] reverse(rp/st1/st2) => reverse(st1/st2) + reverse(rp/st1)
end

[] reverse(attribute::nt/parent::nt1) => attribute::nt
end

[] reverse(a::nt/a1::nt1) => R(a1)::nt
end

[] first(rp/a::nt) => self::nt
end

[] first(a::nt) => self::nt
end

[] last(rp/st) => last(rp)
end

[] last(a::nt) => R(a)::node()
end
```

The above function uses a *R()* function which defi nes the reverse axis for each axis.

```
R(@)                :(AxisName) AxisName;

[] R(ancestor) => subnode
end

[] R(ancestor-or-self) => subnode-or-self
end

[] R(attribute) => parent
end

[] R(child) => parent
end

[] R(parent) => child
end

[] R(following) => preceding
```

```
end

[] R(preceding) => following
end

[] R(following-sibling) => preceding-sibling
end

[] R(preceding-sibling) => following-sibling
end

[] R(descendant) => ancestor
end

[] R(descendant-or-self) => ancestor-or-self
end

[] R(self) => self
end
```

Note that the reverse axis for *parent* can be both *child* and *attribute*. To obtain a complete definition we have chosen it as *child*. For matching in XSLT this is not relevant, as the XPath expressions used for matching are restricted to axes that go downward in the tree.

## 6 Defining XSLT in ELAN

### 6.1 XSLT Stylesheets

An XSLT stylesheet is a well-formed XML document, so the XML data model is also applicable to it. It will form a tree with nodes being the XSL elements (with names in the XSL namespace), target tree elements, XSL attributes and text nodes. For example, a fragment of an XSLT stylesheet appears as the following.

```
<xsl:stylesheet>
  <xsl:strip-space elements=''doc chapter''/>

  <xsl:template match=''doc''>
    <html>
      <head>
        <xsl:value-of select=''title''>
      </head>
    </html>
    <body>
      <xsl:apply-templates/>
    </body>
  </xsl:template>
```

```
  ...
<xsl:stylesheet>
```

where *xsl:template*, *xsl:value-of* and *xsl:apply-templates* are elements in the XSLT namespace, and other nodes such as *html*, *head* and *body* are result tree elements in the target XML namespace.

### 6.1.1 Top-level Elements

An XSLT stylesheet always begins with the element *xsl:stylesheet* (or an equivalent literal result element as a simplified syntax). The *xsl:stylesheet* element may contain the following types of elements as its children:

- xsl:import
- xsl:include
- xsl:strip-space
- xsl:preserve-space
- xsl:output
- xsl:key
- xsl:decimal-format
- xsl:namespace-alias
- xsl:attribute-set
- xsl:variable
- xsl:param
- xsl:template

An element occurring as a child of an *xsl:stylesheet* element is called a *top-level element*. Of these top-level elements, except *xsl:template*, others are not directly involved in transformation. They are used either to set the transformation environment or to optimize the stylesheet structure (e.g. modularization). Only *xsl:template* elements represent the template rules for the transformation of the source tree.

```
<xsl':'stylesheet
        version=@ xmlns':'xsl=@ xmlns=@>
        @
</xsl':'stylesheet>
                :(string string string TLEleS) XSL;
@               :(TLEle) TLEleS;
@ @             :(TLEle TLEleS) TLEleS;

tle             :TLEle;
tles            :TLEleS;
```

Here XSL is the sort for XSLT stylesheets and TLEle is the sort for top-level elements, such as *xsl:template*.

The current node is processed by finding all the template rules with patterns that match the node, and choosing the best amongst them. The chosen rule's template is then instantiated with the node as the current node and with the list of source nodes as the current node list. A template typically contains instructions that select a new list of source nodes for further processing. The process is continued recursively until no new node list is selected for processing. The transformation always begins with the root node, which corresponds to the source XML term.

## 6.2 XSLT Template Rules

### 6.2.1 xsl:template element

A template rule is specified with the *xsl:template* element. The match attribute is a pattern that identifies the source node or nodes to which the rule applies. The child of the *xsl:template* element is the template that is to be instantiated when the template rule is applied. In XSLT, a template rule appears as

```
<xsl:template
    match= pattern>
    <!--Content:(xsl:param*, template -->
</xsl:template>
```

In ELAN, we define

```
<xsl':'template match=@> @ </xsl':'template>    :(LocPath Tmpl) TLEle;
@             :(TmplEle) Tmpl;
@ @           :(TmplEle Tmpl) Tmpl;

<@/>          :(TEleName) TmplEle;
<@> </@>      :(TEleName TEleName) TmplEle;
<@ @/>        :(TEleName TAttrS) TmplEle;
<@ @> </@>    :(TEleName TAttrS TEleName) TmplEle;
<@ @/>        :(TEleName list[TAttr]) TmplEle;

<@> @ </@>    :(TEleName Tmpl TEleName) TmplEle;
<@ @> @ </@>  :(TEleName TAttrS Tmpl TEleName) TmplEle;
<@ @> @ </@>  :(TEleName list[TAttr] Tmpl TEleName) TmplEle;

@             :(string) TmplEle;
@             :(Ins) TmplEle;

tm            :Tmpl;
tme           :TmplEle;
```

We consider a rule to consist of the XPath pattern for matching and the template. The template may contain result elements, text nodes and XSLT instructions or some text nodes. Formally we let the template have the sort *list[TChild]* in the internal representation for a rule.

```
<@,@>         :(Cond list[TChild]) Rule;
```

```
@            :(LocPath) Cond;

rL           :list[Rule];
```

So an *xsl:template* can be transformed to a *Rule*, and a stylesheet to a list of *Rules*. The template is transformed to the result elements. The order of XSLT rules is reversed, and the first rule matched is always triggered. The method for rule selection specified in the XSLT Recommendation may be emulated by choosing a suitable ordering of the template rules.

```
Tr(@)        :(XSL) list[Rule];
Tr(@)        :(TLEleS) list[Rule];

[] Tr(<xsl:stylesheet version="1.0" xmlns:xsl=str xmlns=str1> tles
            </xsl:stylesheet>) => Tr(tles)
end

[] Tr(tle tles) => Tr(tles) @ Tr(tle)
end

[] Tr(<xsl:template match=patt> tm </xsl:template>) => <patt,Tr(tm)>.nil
end

Tr(@)        :(Tmpl) list[TChild];

[] Tr(tme tm) => Tr(tme) @ Tr(tm)
end
```

*xsl:apply-templates* is a typical XSLT instruction appearing in the template.

### 6.2.2 xsl:apply-templates element

```
<xsl:apply-templates
    select = node-set-expr>
    <!-- Content:(xsl:sort|xsl:with-param)* -->
</xsl:template>
```

Usually *xsl:apply-templates* is used in the template for recursively processing more source nodes. However, in general the *select* attribute can be used to process nodes selected by an arbitrary XPath expression, and the default case of selecting the children may be modeled by assuming a suitable default value of the select attribute. The selected set of nodes is processed in document order, unless a sorting specification is present. The set of nodes is evaluated using the *Select()* function in our XPath module.

```
<xsl':'apply-templates  select=@ /> :(LocPath) Ins;

sel(@)        :(LocPath) TChild;

[] Tr(<xsl:apply-templates select=patt />) => sel(patt).nil
```

```
end

<xsl':'apply-templates/>            :Ins;

[] <xsl:apply-templates /> => <xsl:apply-templates select=node()/>
end
```

### 6.3 Result Tree Creating Instructions

#### 6.3.1 Literal Result Elements

In a template, an element in the stylesheet (that does not belong to the XSLT namespace) creates an element node in the result tree. The content of the element is also a template, whose evaluation gives the content of the created element node. The created element node will have attribute nodes corresponding to the attributes on the element in the stylesheet.

```
taL      :list[TAttr];

[] <tg> tm </tg> => <tg nil> tm </tg>
end

[] <tg atS> tm </tg> => <tg P(atS)> tm </tg>
end

[] <tg /> => <tg nil/>
end

[] <tg> </tg> => <tg nil/>
end

[] <tg atS /> => <tg P(atS) />
end

[] <tg atS> </tg> => <tg P(atS) />
end

[] Tr(<tg taL/>) => tg(taL,nil).nil
end

[] Tr(<tg taL> tm </tg>) => tg(taL,Tr(tm)).nil
end
```

#### 6.3.2 xsl:element element

```
<xsl:element
  name ={qname}>
  <!-- Content: template -->
</xsl:element>
```

The *xsl:element* element creates an element in the result tree. The element name is specified by the *name* attribute.

```
<xsl':'element name=@> @ </xsl':'element>    :(TEleName Tmpl) Ins;

[] <xsl:element name=tg> tm </xsl:element> => <tg nil> tm </tg>
end
```

### 6.3.3 xsl:attribute element

```
<xsl:attribute
  name ={qname}>
  <!-- Content: template -->
</xsl:attribute>
```

The *xsl:attribute* element can be used to add attributes to result elements, whether created by literal result elements or by an instruction such as *xsl:element*. The attribute name is specified by the *name* attribute.

```
<xsl':'attribute name=@> @ </xsl':'attribute>    :(TAttrName Tmpl) Ins;

[] Tr(<xsl:attribute name=an> str </xsl:attribute>) => an=str.nil
end
```

In fact, the transformation result of an *Ins* is a *TChild*. The attribute is not a child node, but to simplify the processing we add it temporarily as a child, and put it into the attribute list in a second step:

```
@            :(TAttr) TChild;

[] tg(atL,at.Tl) => tg(at.atL,Tl)
end
```

### 6.3.4 xsl:text element

```
<xsl:text>
  <!-- Content: #PCDATA -->
</xsl:text>
```

The *xsl:text* element creates a text node in the result tree, with the value of the created text node being the string value of the *xsl:text* element.

```
<xsl':'text> @ </xsl':'text>    :(string) Ins;

[] <xsl:text> str </xsl:text> => str
end
```

An *xsl:text* element is equivalent to a string, the representation of a text node.

```
[] Tr(str) => str.nil
end
```

### 6.3.5 xsl:value-of element

```
<xsl:value-of select = string-expr/>
```

The *xsl:value-of* element creates a text node in the result tree from the values of some nodes in the source tree, which result from selecting the nodes by the expression in the *select* attribute.

```
<xsl':'value-of select=@ />   :(LocPath) Ins;
<xsl':'value-of select=@> </xsl':'value-of>
                              :(LocPath) Ins;

[] <xsl:value-of select=rp> </xsl:value-of> =>
                      <xsl:value-of select=rp/>
end

val(@)                        :(LocPath) TChild;

[] Tr(<xsl:value-of select=patt />) => val(patt).nil
end
```

## 6.4 Repetition Instruction

### 6.4.1 xsl:for-each

```
<xls:for-each
  select = node-set-expression>
  <!--Content: (xsl:sort*, template) -->
</xsl:for-each>
```

The *xsl:for-each* instruction contains a template which is instantiated for each node selected by the expression in the *select* attribute. The *select* attribute is required.

```
<xsl':'for-each select=@> @ </xsl':'for-each>
                 :(LocPath Tmpl) Ins;
sel(@,@)         :(LocPath list[TChild]) TChild;

[] Tr(<xsl:for-each select=patt> tm </xsl:for-each>) =>
               sel(patt,Tr(tm)).nil
end
```

Here, *sel()* is overloaded, with the second argument being the template to be applied to the selected nodes.

## 6.5 Conditional Processing

### 6.5.1 xsl:if

```
<xsl:if>
  test = boolean-expression
  <!-- Content:template -->
</xsl:if>
```

The *xsl:if* element has a *test* attribute, which specifies a boolean expression. Its content is a template. If the expression is evaluated and the result is true, then the content template is instantiated. Otherwise, nothing is created.

```
@            :(bool) BExpr;
be           :BExpr;

ifcond(@,@)  :(BExpr list[TChild]) TChild;
<xsl':'if test=@> @ </xsl':'if>    :(BExpr Tmpl) Ins;

[] Tr(<xsl:if test=be> tm </xsl:if>) => ifcond(be,Tr(tm)).nil
end
```

*BExpr* can be a boolean expression.

### 6.6 Sorting

#### 6.6.1 xsl:sort

```
<xsl:sort  select= string-expression
           data-type={''text''|''number''|qname-but-not-ncname}
           order=''ascending''|''descending''/>
```

Sorting is specified by adding *xsl:sort* elements as children of an *xsl:apply-templates* or *xsl:for-each* element. Instead of processing the selected nodes in document order, they are sorted according to the specified sort keys and then processed in that order.

*xsl:sort* has a *select* attribute whose value is an expression. For each node to be processed, the expression is evaluated with that node as the current node and with the complete list of nodes being processed in unsorted order as the current node list. The resulting object is converted to a string as if by a call to the *string()* function. This string is used as the sort key for that node. The default value of the select attribute is ., which will cause the string-value of the current node to be used as the sort key. The following optional attributes of *xsl:sort* control how the list of nodes is to be sorted.

```
<xsl':'sort @ />                :(SortAttrS) Ins;
<xsl':'sort @> </xsl':'sort>    :(SortAttrS) Ins;

select       :SortAttrName;
data-type    :SortAttrName;
order        :SortAttrName;
@            :(string) SortAttrVal;
@            :(LocPath) SortAttrVal;
@=@          :(SortAttrName SortAttrVal) SortAttr;
@            :(SortAttr) SortAttrS;
@ @          :(SortAttr SortAttrS) SortAttrS;

sas          :SortAttrS;
sa           :SortAttr;
sn,n1        :SortAttrName;
```

```
val,v1       :SortAttrVal;

[] <xsl:sort sas> </xsl:sort> => <xsl:sort sas/>
end
```

- *order* specifies whether the strings should be sorted in ascending or descending order. The default is ascending.
- *data-type* specifies the data of the strings. *text* specifies that the sort keys should be sorted lexicographically. Similarly, *number* specifies that the sort keys should be converted to numbers and then sorted according to their numeric values.

```
sort(@,@,@)    :(SortAttrVal SortAttrVal SortAttrVal) TChild;

[] Tr(<xsl:sort sas/>) => sort(v1,v2,v3).nil
            where v1:= () get(select,node(),sas)
            where v2:= () get(data-type,"number",sas)
            where v3:= () get(order,"descending",sas)
end
```

The *get()* function extracts each attribute value of the *xsl:sort* element. If an attribute is not present, then we set its default value there. The second parameter of *get()* function is the default value for that attribute.

```
get(@,@,@)    :(SortAttrName SortAttrVal SortAttrS) SortAttrVal;

[] get(sn,val,sn=v1 sas) => v1
end

[] get(sn,val,sn=v1) => v1
end

[] get(sn,val,n1=v1) => val
end

[] get(sn,val,n1=v1 sas) => get(sn,val,sas)
end
```

## 7 An XSLT (subset) Processor in ELAN

### 7.1 An integrated framework of XSLT processing

For the users, an XSLT stylesheet is applied on the source XML document, and a result XML document is produced.

```
app @ to @       :(XSL SXML) TXML;
app(@,@)         :(XSL SXML) TXML     alias app @ to @:;
```

```
xsl             :XSL;
xml             :SXML;

[] app xsl to xml => I(app Tr(xsl) to Prs(xml))
end
```

*Tr(xsl)* is the *Rule* list transformed from the stylesheet. It is applied to the ELAN term parsed from the source XML document. The *I()* function transforms the result ELAN term back to that in XML format.

### 7.1.1 *Rule* Selection

The *Rule* list will be applied to the current node or the current node list. When the *Rule* list functions on a node list, the nodes in the list are processed one-by-one.

When the current node matches the *Rule*'s pattern, the corresponding template will be instantiated.

```
app @ to @      :(list[Rule] list[Node]) list[TChild];
app @ to @      :(list[Rule] Node) list[TChild];

[] app rL to n.nL => app rL to n @ app rL to nL
end

[] app rL to nil => nil
end

app @,@ to @    :(list[Rule] list[Rule] Node) list[TChild];

[] app rL to n => app rL rL to n
end

[] app rL,<patt,Tl>.rL1 to n => inst rL,Tl with n
        if match(n,patt)
end

[] app rL,<patt,Tl>.rL1 to n => app rL,rL1 to n
end
```

If no explicit *Rule* can be triggered, the implicit built-in *Rule* should be triggered. The first *Rule* is for text nodes, the second *Rule* for attribute nodes, and the third one for element nodes.

```
san    :SAttrName;

[] app rL nil to pn/str.cL => str.nil
end

[] app rL nil to pn/san=str => str.nil
end

[] app rL,nil to pn => inst rL,sel(child::node()).nil with pn
end
```

### 7.1.2 Instantiation of templates

When the proper *Rule* is triggered, then its corresponding template is instantiated using the current node (or node list).

```
inst @,@ with @    :(list[Rule] list[TChild] Node) list[TChild];
inst @,@ with @    :(list[Rule] list[TChild] list[Node]) list[TChild];

[] inst rL,Tl with n.nL => inst rL,Tl with n @ inst rL,Tl with nL
end

[] inst rL,Tl with nil => nil
end
```

- Literal result elements

```
[] inst rL,tg(taL,tl1).Tl with n => tg(taL,tl2).inst rL,Tl with n
        where tl2:=() inst rL,tl1 with n
end
```

- Selecting the new node list (for further processing)

```
[] inst rL,sel(patt).Tl with n => app rL to nL @ inst rL,Tl with n
        where nL:=() Select(n,patt)
end
```

- Attribute nodes

```
 [] inst rL,tat.Tl with n => tat.inst rL,Tl with n
end
```

- Text nodes

```
[] inst rL,str.Tl with n => str.inst rL,Tl with n
end

[] inst rL,val(patt).Tl with n => valueof(n,patt).inst rL,Tl with n
end
```

The pattern expression in the *valueof()* function is evaluated and the resulting node list is converted to a string by a call to the *String()* function. The *String()* function converts the node list to a string by returning the string-value of the node in the node list that is the first in document order. If the node list is empty, a null string is returned. The *stringval()* function returns the string-value of the node according to the data model description.

```
valueof(@,@)    :(Node LocPath) string;
String(@)       :(list[Node]) string;
```

```
stringval(@)    :(Node) string;
stringval(@)    :(list[Node]) string;

[] valueof(n,path) => String(Select(n,path))
end

[] String(nil) => ""
end

[] String(n.nL) => stringval(n)
end

[] stringval(pn/str.cL) => str
end

[] stringval(pn/at=str) => str
end

[] stringval(pn) => stringval(select(pn,descendant::text()))
end

[] stringval(nil) => ""
end

[] stringval(n.nL) => stringval(n)+stringval(nL)
end
```

- For-each Processing

  For the node list specified by *Select()*, the specified template is instantiated.

```
[] inst rL,sel(patt,tl1).Tl with n => inst rL,tl1 with nL
                                        @inst rL,Tl with n
        where nL:=() Select(n,patt)
end
```

- Conditional Instruction Processing

  If the condition in the instruction is satisfied, then the corresponding template will be applied. Otherwise, it will be ignored.

```
[] inst rL,ifcond(be,tl1).Tl with n => inst rL,tl1 with n
                                         @inst rL,Tl with n
        if tst(n,be)
end

[] inst rL,ifcond(be,tl1).Tl with n => inst rL,Tl with n
end
```

```
tst(@,@)            :(Node BExpr) bool;
```

Here we are using the *tst()* function, which takes the current node and a boolean expression, and then tests that expression for truth. The boolean expression is an expression formed over integers, booleans and patterns. For the patterns, we need to select some nodes depending on the pattern and starting from the current node, which is just a path, and then find the value of the first node in that node-set according to the data model. This is done by calling the function *valueof()*.

- Sorting of the node list

```
datatype    :string;

[] inst rL,sort(patt,datatype,"descending").Tl with nL =>
      inst rL,Tl with extract(ssrt)
          where ssrt:=(dSort) make(nL,patt,datatype)
end

[] inst rL,sort(patt,datatype,"ascending").Tl with nL =>
      inst rL,Tl with extract(ssrt)
          where ssrt:=(aSort) make(nL,patt,datatype)
end

<@,@>         :(Node int) NSrt;
<@,@>         :(Node string) NSrt;
make(@,@,@)   :(list[Node] LocPath string) list[NSrt];
make(@,@,@)   :(Node LocPath string) NSrt;
extract(@)    :(list[NSrt]) list[Node];
```

We define two sorting strategies *aSort* and *dSort* for ascending and descending order, respectively. Sorting is applied to a list whose elements are pairs of nodes and the values of the nodes (string type or numeric type according to the *datatype* attribute). The *make()* function forms such a list. The *extract()* function extracts the node list from the sorted list of pairs. The following are the strategies and labelled rules for sorting.

```
stratop global
  aSort    :<list[NSrt]> bs;
  dSort    :<list[NSrt]> bs;
end

strategies for list[NSrt]
implicit
  [] dSort => normalize(ddSort)
  end

  [] aSort => normalize(aaSort)
```

```
  end
end

rules for list[NSrt]
  x, y        :Node;
  x1, y1      :Node;
  nsL         :list[NSrt];
  i,j         :int;
  s1,s2       :string;
local
  [ddSort] <x,i>.<y,j>.nsL => <y,j>.<x,i>.nsL
              choose  try
                  if j>i
              end
  end

  [ddSort] <x,s1>.<y,s2>.nsL => <y,s2>.<x,s1>.nsL
               choose  try
                  if strcmp(s2,s1) >0
               end
  end

  [aaSort] <x,i>.<y,j>.nsL => <y,j>.<x,i>.nsL
               choose  try
                   if j<i
               end
  end

  [aaSort] <x,s1>.<y,s2>.nsL => <y,s2>.<x,s1>.nsL
               choose  try
                   if strcmp(s2,s1) <0
               end
  end
end
```

### 7.2 Examples

- Document Example D.1 in the Appendix of XSLT specifi cation.

```
app
<xsl:stylesheet
     version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns="http://www.w3.org/TR/xhtml1/strict">
<xsl:template match=doc>
   <html>
     <head>
```

```
        <title>
        <xsl:value-of select=title />
        </title>
    </head>
    <body>
        <xsl:apply-templates/>
       </body>
   </html>
</xsl:template>

<xsl:template match=doc/title>
   <h1>
    <xsl:apply-templates/>
   </h1>
</xsl:template>

<xsl:template match=chapter/title>
   <h2>
    <xsl:apply-templates/>
   </h2>
</xsl:template>

<xsl:template match=section/title>
   <h3>
    <xsl:apply-templates/>
   </h3>
</xsl:template>

<xsl:template match=para>
   <p>
    <xsl:apply-templates/>
   </p>
</xsl:template>

<xsl:template match=note>
   <p class="note">
    <b>
        "NOTE:"
    </b>
    <xsl:apply-templates/>
   </p>
</xsl:template>

<xsl:template match=emph>
   <em>
    <xsl:apply-templates/>
```

```
   </em>
</xsl:template>
</xsl:stylesheet>
to
<doc>
    <title>"Document Title"</title>
    <chapter>
        <title>"Chapter Title"</title>
        <section>
            <title>"Section Title"</title>
            <para>"This is a test."</para>
            <note>"This is a note."</note>
        </section>
        <section>
            <title>"Another Section Title"</title>
            <para>
                "This is "
                <emph> "another" </emph>
                " test."
            </para>
            <note> "This is another note." </note>
        </section>
    </chapter>
</doc>
 end
```

Then we got the result document.

```
<html><head><title>"Document Title"</title></head><body>
<h1>"Document Title"</h1><h2>"Chapter Title"</h2><h3>"Se
ction Title"</h3><p>"This is a test."</p><p class="note"
><b>"NOTE:"</b>"This is a note."</p><h3>"Another Section
 Title"</h3><p>"This is "<em>"another"</em>" test."</p><
p class="note"><b>"NOTE:"</b>"This is another note."</p>
</body></html>
```

- Data Example(HTML output) D.2 in the appendix of the Specifi cation

```
app
<xsl:stylesheet
     version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
     xmlns="http://www.w3.org/TR/xhtml1/strict">
<xsl:template match=root>
   <html lang="en">
     <head>
```

```
        <title>
        "Sales Results By Division"
        </title>
    </head>
    <body>
        <table border="1">
        <tr>
            <th>"Division"</th>
            <th>"Revenue"</th>
            <th>"Growth"</th>
            <th>"Bonus"</th>
        </tr>
        <xsl:for-each select=sales/division>
            <xsl:sort data-type="number" select=revenue order="descending"/>
            <tr>
            <td>
                <em>
                <xsl:value-of select=@id />
                </em>
            </td>
            <td>
                <xsl:value-of select=revenue />
            </td>
            <td>
                <xsl:if test= growth<0>
                <xsl:attribute name=style>
                    <xsl:text> "Red Color" </xsl:text>
                </xsl:attribute>
                </xsl:if>
                <xsl:value-of select=growth />
            </td>
            <td>
                <xsl:value-of select=bonus />
            </td>
            </tr>
        </xsl:for-each>
        </table>
       </body>
   </html>
</xsl:template>
</xsl:stylesheet>
to
<sales>
    <division id="North">
        <revenue>"10"</revenue>
        <growth>"9"</growth>
```

```
            <bonus>"7"</bonus>
        </division>
        <division id="South">
            <revenue>"4"</revenue>
            <growth>"3"</growth>
            <bonus>"4"</bonus>
        </division>
        <division id="West">
            <revenue>"6"</revenue>
            <growth>"-1.5"</growth>
            <bonus>"2"</bonus>
        </division>
    </sales>
    end
```

Then we got the result document.

```
<html lang="en"><head><title>"Sales Results By Division"
</title></head><body><table border="1"><tr><th>"Division
"</th><th>"Revenue"</th><th>"Growth"</th><th>"Bonus"</th
></tr><tr><td><em>"North"</em></td><td>"10"</td><td>"9"<
/td><td>"7"</td></tr><tr><td><em>"West"</em></td><td>"6"
</td><td style="Red Color">"-1.5"</td><td>"2"</td></tr><
tr><td><em>"South"</em></td><td>"4"</td><td>"3"</td><td>
"4"</td></tr></table></body></html>
```

## 8 Experiments

To get an idea about the performance of the XSLT processor written in ELAN we have conducted some experiments. In the first experiment we use the catalogue of all TEX packages that is freely available [9]. We use a simple stylesheet to build an HTML-document that contains an ordered list of the names of the packages (the syntax is already modified for ELAN):

```
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match=catalogue>
  <html>
    <body>
      <ol>
        <xsl:apply-templates/>
      </ol>
    </body>
  </html>
</xsl:template>

<xsl:template match=entry>
```

```
  <li>
    <xsl:value-of select=./about/name />
  </li>
</xsl:template>
</xsl:stylesheet>
```

The full catalogue contains 1385 entries. We applied this stylesheet to samples of 100 to 300 entries taken from the beginning of the catalogue, using the compiled ELAN program.[13] The results are shown in Figure 1.

| # entries | time/s | # rewrites | # rewrites/s |
|---|---|---|---|
| 100 | 7.7 | 7742018 | 1002852 |
| 200 | 28.6 | 29369008 | 1028686 |
| 300 | 64.1 | 63545979 | 992130 |

Table 1: Performance on TEX catalogue samples (PII 366MHz)

As another example we tried a stylesheet from the DHYDRO project. This stylesheet transforms a log in XML of users accessing the DHYDRO system into HTML for display. We removed those parts of the stylesheet that are not covered by our implementation (for example statistics that involve counting), which left only the core of listing the logins with their data (e.g. username, date, request, reply, processing time). We got two sample logfiles with 88 and 149 entries to test it. Here parsing took most of the time (a few seconds), while the time for rewriting was very small ($< 1$s).

From these examples we see that even our rather naive implementation of XSLT in ELAN is already quite fast, and that the main limitation is the Earley parser, which imposes several hard limits on the size of inputs.

## 9 Conclusion

In this paper we have given both a formal semantics of XSLT by proof rules and an implementation in ELAN. Due to the simplicity of ELAN that is based on term rewriting this implementation is still quite concise and may serve both as a formal definition of XSLT and XPath and as an implementation that can actually execute a transformation. The formal specification, though incomplete, provides a simple definition of the core mechanism of XSLT that can serve a variety of purposes, from the verification of XSLT stylesheets and aspects of XSLT processors that fall into its scope to further theoretical study. This may for instance be some kind of type discipline (see below), or features of XSLT and XPath may motivate certain extensions for term rewriting system, such as context matching.

When we started this work we intended to give a shallow embedding of XSLT into ELAN, hoping that the features of XSLT can be mapped directly to those of ELAN, e.g. obtaining node selection from a combination of term matching with the strategies of ELAN. However, during this work it became clear that this was infeasible, as there is a large gap between XPath and term matching and XPath matching had to be implemented explicitly, i.e. we needed to use a deep embedding. Similarly, the difference between the iterated replacement used in term rewriting and the target tree construction in XSLT also needed to be programmed explicitly. Essentially, we used ELAN as a functional language and were not able to make use of its special capabilities.

[13] We had to increase a constant in the Earley parser for context-free languages that is used in ELAN, in order to be able to use inputs with 20 or more entries. At 400 entries we hit another hard limit in the number of string constants.

A part of the reason that using strategies was too difficult is the nonlocal data driven nature of XSLT, where one needs to know the structure of the source document to foresee what may happen during matching, e.g. to characterize the set of nodes where the transformation may continue. Due to the untyped nature of XSLT, i.e. the source and target trees are not required to be well-formed with respect to a DTD or an XML Schema, and due to the lack of type information in XSLT this is nontrivial to achieve. We conjecture that it is possible to derive types automatically with respect to source and target DTDs by using a suitable notion of automaton, however this is beyond the scope of this work.

With respect to performance we note that even though our program follows the specification quite closely in most areas, its performance is quite good, the most severe limitation being fixed limits and memory consumption of the parser. Since we essentially programmed an interpreter for XSLT and didn't map features of XSLT directly to ELAN, we can expect that the performance for XML transformations by rewriting in ELAN is about a magnitude faster, and that it should be practicable to use ELAN in applications if the Earley parser can be bypassed, for instance by using a different parser with the ATerms library [17].

## Acknowledgments

We thank Nadia Viscogliosi for providing us with sample data from the DHYDRO project, and Pierre-Etienne Moreau for his invaluable help in all things concerning the ELAN compiler.

## References

[1] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, Cambridge, UK, 1998.

[2] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. In Claude Kirchner and Hélène Kirchner, editors, *Proc. 2nd Int. Workshop on Rewriting Logic and Applications*, volume 15, Pont-à-Mousson (France), September 1998. Electronic Notes in Theoretical Computer Science. `http://www.elsevier.nl/locate/entcs/volume15.html`.

[3] Tim Bray, Dave Hollander, and Andrew Layman, editors. *Namespaces in XML*. W3C, 14 January 1999. `http://www.w3.org/TR/1999/REC-xml-names-19990114`.

[4] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler, editors. *Extensible Markup Language (XML) 1.0 (Second Edition)*. W3C, 16 October 2000. `http://www.w3.org/TR/2000/REC-xml-20001006`.

[5] Horatiu Cirstea and Claude Kirchner. The rewriting calculus — Part I *and* II. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9(3):427–498, May 2001.

[6] James Clark, editor. *XSL Transformation (XSLT) Version 1.0*. W3C, 16 November 1999. `http://www.w3.org/TR/1999/REC-xslt-19991116`.

[7] James Clark and Steve DeRose, editors. *XML Path Language (XPath) Version 1.0*. W3C, 16 November 1999. `http://www.w3.org/TR/1999/REC-xslt-19991116`.

[8] Stephen Cranefield. Networked knowledge representation and exchange using UML and RDF. *Journal of Digital Information*, 1(8), 2001. http://jodi.ecs.soton.ac.uk/Articles/v01/i08/Cranefield/.

[9] The TeX catalogue online, CTAN edition. http://www.ctan.org/tex-archive/help/Catalogue/ca October 10 2001.

[10] M. Dauchet. Simulation of Turing machines by a left-linear rewrite rule. In N. Dershowitz, editor, *Proceedings 3rd Conference on Rewriting Techniques and Applications, Chapel Hill (N.C., USA)*, volume 355 of *Lecture Notes in Computer Science*, pages 109–120. Springer-Verlag, April 1989.

[11] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language (preliminary report). In *International Workshop on the Web and Databases (WebDB)*, May 2000.

[12] Benjamin Jung, Egil P. Andersen, and Jane Grimson. SynExML as a vehicle for electronic patient records. A status report from the SynEx project. In *Proc. XML Europe*, 2000. http://www.gca.org/papers/xmleurope2000/papers/s32-02.html.

[13] Pramod Kankure. Produce dynamic web pages with Java and XSLT. Two approaches for building an easily portable solution. In IBM developerWorks April 2001. http://www-106.ibm.com/developerworks/xml/library/x-dynweb/.

[14] Claude Kirchner and Hélène Kirchner. Rewriting, solving, proving. A preliminary version of a book available at www.loria.fr/~ckirchne/rsp.ps.gz, 1999.

[15] Protheo Team. The ELAN home page. WWW Page, 2001. http://elan.loria.fr.

[16] C. M. Sperberg-McQueen and Claus Huitfeldt. Practical extraction of meaning from markup using XSLT. An abstract submitted to ACH/ALLC 2001, 3 December 2000. http://www.nyu.edu/its/humanities/ach_allc2001/papers/sperberg-mcqueen/.

[17] M. Van den Brand, H.A.. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software-Practice and Experience*, 30:259–291, 2000.

[18] Philip Wadler. A formal semantics of patterns in XSLT, December 1999. http://cm.bell-labs.com/cm/cs/who/wadler/topics/xml.html.

[19] Philip Wadler. Two semantics for XPath, 4 January 2000. http://cm.bell-labs.com/cm/cs/who/wadler/topics/xml.html.

[20] Xsldoc - the XSLT API documentation generator. http://www.xsldoc.org/, 2001.

## Contents