# A Formal Model for an Expressive Fragment of XSLT*

Geert Jan Bex[1] Sebastian Maneth[2] Frank Neven[1,†]

[1]Limburgs Universitair Centrum,
Universitaire Campus,
Dept. WNI, Infolab,
B-3590 Diepenbeek, Belgium,
E-mail: {gjb,frank.neven}@luc.ac.be

[2]Leiden University,
LIACS,
PO Box 9512, 2300 RA Leiden,
The Netherlands
E-mail: maneth@liacs.nl

**Abstract**

The extension of the XSL (eXtensible Style sheet Language) by variables and passing of data values between template rules has generated a powerful XML query language: XSLT (eXtensible Style sheet Language Transformations). An informal introduction to XSTL is given, on the bases of which a formal model of a fragment of XSLT is defined. This formal model is in the spirit of tree transducers, and its semantics is defined by rewrite relations. It is shown that the expressive power of the fragment is already beyond that of most other XML query languages. Finally, important properties such as termination and closure under composition are considered.

## 1 Introduction

XSLT [Cla99b] is the W3C [Con] recommendation for an XML style sheet language. The original primary role of XSLT was to allow users to write transformations of XML to HTML, thus describing the presentation of XML documents. Nowadays, many people use XSLT as their basic tool for XML to XML transformations which renders XSLT into an XML query language.

The definition of XSLT has been quite unstable before it became a recommendation in November 1999. Before that time, it has been noted by the database community [DFF+99a, ABS00], that the transformations expressible in earlier versions of XSLT were rather limited. For instance, XSLT did not have joins or skolem-functions (and, hence could not do sophisticated grouping of output data). In other words, XSLT lacked the most basic property any query language should have: it was not relationally complete. We show in this paper that current XSLT is much more powerful than its previous versions. Indeed, XSLT not only becomes relationally complete, but it can do explicit grouping (with or without skolem functions), it can simulate regular path expressions, and it can simulate most other current XML query languages.

The main source for the definition of XSLT is its specification [Cla99b] which is a long technical document that is rather difficult to read, especially if one only intends to

*A preliminary version of this paper was presented at the 1st International Conference on Computational Logic, London, July, 2000.

†Post-doctoral researcher of the Fund for Scientific Research, Flanders

get an impression of how the language works and what it is capable of. To remedy this, we define an abstract formal model $\text{XSLT}_0$ of XSLT incorporating most of its features, but all of those which are necessary to simulate, say, the core of XML-QL. The purpose of this model is two-fold:

(*i*) its clean and formal semantics provides the necessary mathematical model for studying properties of XSLT;

(*ii*) the formal model abstracts from the actual syntax of XSLT and emphasizes its transformational features in such a way that the interested reader can quickly get a feeling of the language and its capabilities.

We want to emphasize that we do *not* provide a model for all of XSLT. For instance, we excluded for-loops, and variables can only be instantiated by data values (not by result tree fragments or node-sets). The resulting language is not Turing complete as all data values are seen as atomic entities. [1] The most important observation is that the defined language is much more expressive than the initial versions of XSLT. [2]

We use our model to gain some insight into XSLT. We start by defining a valid abstraction $\text{QL}_0$ of the core of XML-QL and show that it can be simulated in $\text{XSLT}_0$. Next, we consider the $k$-pebble tree transducers of Milo, Suciu, and Vianu [MSV00]. These were defined, in the context of type checking, to capture the expressiveness of most current XML query languages including XML-QL [DFF$^+$99a], XQL [Rob99], Lorel [AQM$^+$97], StruQL [FFK$^+$98], UnQL [BDHD96], and the previous version of XSLT. Their model does not take value equations into account (needed for joins, for instance), but can easily be modified to do so. We emphasize that this is just informal evidence for the expressiveness of $\text{XSLT}_0$. Readers not familiar with $k$-pebble transducers can safely ignore this comparison.

Next, we obtain that $\text{XSLT}_0$ can compute all unary monadic second-order (MSO) structural patterns. In brief, MSO logic is first-order (FO) logic extended by set quantification. It is an expressive and versatile logic: on trees, for instance, it captures many robust formalisms, like regular tree languages [Tho97a], query automata [NS99], finite-valued attribute grammars [NVdB98, Nev99b], etc. By structural patterns we mean MSO without joins, that is, it cannot be checked whether the values of two attributes are the same (see Section 4 for details). In fact, Neven and Schwentick [NS00] showed that, already with respect to structural patterns, MSO logic is more expressive than FO logic extended by various kinds of regular path expressions. Thus, as most current XML query languages are based on FO logic extended by regular path expressions, this already indicates that XSLT cannot be simulated by, say, XML-QL.

On the negative side, we show that the termination problem undecidable is undecidable. In fact, deciding termination for XSLT without data values is complete for EXPTIME [MN99]. Further, we show that XSLT programs are not closed under composition.

The remainder of this paper is structured as follows. In Section 2 we introduce the important features of XSLT by means of two examples. In Section 3 we define the formal

[1] If data values were considered as strings and operations like substring or concatenation were allowed, then the resulting language would be Turing complete.

[2] In previous work we defined a formal model for the version of XSLT not incorporating data values [MN99].

model $\text{XSLT}_0$. In Section 4, we study some properties of this model. Finally, we present some conclusions in Section 5.

The interested reader is referred to [Bex00], were it is shown how the XML-QL queries of [DFF+99a] can be expressed in actual XSLT.

## 2 XSLT by Example

A basic XSLT program is a collection of *template rules*, where each such rule consists of

- a *matching pattern*,
- a *mode* (which indicates the (finite) state the computation is in), and
- a *template* (see, for example, the program in Figure 2).

Input and output documents of an XSLT program $P$ are unranked trees, with the notational convention that a left bracket is represented by `<xxx>` (where `xxx` is a label) and the corresponding right bracket is represented by `</xxx>`; thus, `<a>b</a>` represents the tree $a(b)$. The computation of $P$ on an input document $t$ starts at the root node of $t$ in the start mode[3] and proceeds roughly as follows. Whenever the computation is at a node $u$ in a certain mode $q$, the program tries to find a template rule with mode $q$ whose matching pattern matches $u$.[4] If it finds such a template rule, the program executes the corresponding template. The latter usually instructs XSLT to produce some XML output, and at various positions in this XML output to select lists of nodes for further processing (we refer to patterns that select nodes for further processing as *selecting patterns*). Each of these selected nodes is then processed independently in the same way as described before. Finally, the documents that are constructed by these subprocesses are inserted at the positions where the subprocesses were initiated.

```
<!DOCTYPE organization [
    <!ELEMENT organization  group+ topmgr>
    <!ELEMENT topmgr       employee+>
    <!ELEMENT group         (mgr group+) | employee+>
    <!ELEMENT mgr           employee>
    <!ELEMENT employee      EMPTY>
    <!ATTLIST employee      id  ID  #REQUIRED>
    <!ATTLIST group         id  ID  #REQUIRED>
]>
```

Figure 1: A DTD describing an organization.

To illustrate the features of XSLT we consider input documents conforming to the document type definition (DTD) shown in Figure 1. This DTD describes an organization as a sequence of groups, together with a list of top managers. A group consists of a

[3] Actually, the use of modes is optional, but for convenience we assume that every template has a mode, and that there is a start mode.

[4] Usually, and in all our examples, such a matching pattern only refers to the label of the current node. In fact, we show in Section 4 that such patterns suffice.

manager and a list of groups, or it just consists of a list of employees. Each group and each employee has an ID associated with it. For simplicity, we identify employees by their ID. The XSLT program in Figure 2 computes pairs $(e_1, e_2)$ of employees, where $e_1$ is a top manager different from `Bill` and is a direct or indirect manager of $e_2$. Pairs are encoded simply by a `pair` element with attributes `topmgrID` and `employeeID` (cf. Figure 4).

```
<xsl:template match="organization" mode="start">
    <result>
        <xsl:apply-templates select="/organization/topmgr/employee"
                    mode="selecttopmgr"/>
    </result>
</xsl:template>

<xsl:template match="employee" mode="selecttopmgr">
    <xsl:variable name="varID">
        <xsl:value-of select="@id"/>
    </xsl:variable>
    <xsl:if test="$varID != 'Bill'">
        <xsl:apply-templates mode="display"
             select="//group[mgr/employee[@id=$varID]]/group//employee">
             <xsl:with-param name="varID" select="$varID"/>
        </xsl:apply-templates>
   </xsl:if>
</xsl:template>

<xsl:template match="employee" mode="display">
    <xsl:param name="varID"/>
    <pair>
        <xsl:attribute name="topmgrID">
            <xsl:value-of select="$varID"/>
        </xsl:attribute>
        <xsl:attribute name="employeeID">
            <xsl:value-of select="@id"/>
        </xsl:attribute>
    </pair>
</xsl:template>
```

Figure 2: An XSLT program computing the query of Section 2.

Note that patterns in XSLT are XPath expressions [Cla99a] (for a formal semantics see [Wad99]). In brief, the leading `/` selects the root of the document. Root here does not correspond to the top element of the XML document but to an extra (invisible) node above it. We will abstract from this in our formal model. Successive symbols `/` mean 'child of', and the symbol `//` means 'descendand of'.

On the face of it, the program just makes a join between the list of top managers and the group managers, that is, the ones occurring in the top manager list and the

```
<organization>
    <group id="HR">
        <mgr><employee id="Bill"/></mgr>
        <group id="HR-prod">
            <mgr><employee id="Edna"/></mgr>
            <group id="HR-prod-empl">
                <employee id="Kate"/>
                <employee id="Ronald"/>
            </group>
        </group>
        <group id="HR-QA">
            <mgr><employee id="John"/></mgr>
            <group id="HR-QA-empl">
                <employee id="Jane"/>
                <employee id="Jake"/>
            </group>
        </group>
    </group>
    <topmgr>
        <employee id="Bill"/>
        <employee id="John"/>
    </topmgr>
</organization>
```

Figure 3: An XML document conforming to the DTD of Figure 1.

```
<result>
    <pair topmgrID="John" employeeID="Jane"/>
    <pair topmgrID="John" employeeID="Jake"/>
</result>
```

Figure 4: The output of the XSLT program shown in Figure 2 taking as input the document shown in Figure 3.

ones occurring as a manager of a group. However, it does so in a rather direct and procedural way. In brief, the XSLT program starts by applying the first template rule in mode start, at the root of the input tree. [5] If the node is labeled by `organization`, then each top manager is selected in mode `selecttopmgr`. This is done by the pattern `/organization/topmgr/employee` which selects all `employee` children of all `topmgr` children of the `organization`-labeled root node. For each employee selected in mode `selecttopmgr`, the second template rule is applied, which stores the employee's ID, say $e_1$, in the variable `varID` and verifies, by using the latter, whether $e_1$ is different from Bill. If so, it selects all the descendants of the group manager who have an ID $e_1$ (in mode `display`). In particular, the selection pattern in the second template rule says 'select all employees that are descendants of a group that itself is a child of a group whose manager has the same ID as the one stored in the variable `varID`'. The expression between the brackets '[...]' is a filter on group elements that makes sure that only groups that have a manager with ID `varID`. In this selection the ID $e_1$ is passed along as parameter. Finally, the third template rule is applied to each employee $e_2$ selected by the second rule and outputs an element `pair` with attribute values $e_1$ and $e_2$ for the attributes `topmgrID` and `employeeID`, respectively.

The program in Figure 2 is not the most compact way to specify the desired query in XSLT, but it nicely illustrates three important features of XSLT: modes, variables, and passing of data values. Let us discuss these briefly:

($i$) Modes enable XSLT to act differently upon arrival at the same element types. For instance, as described above, when our program arrives at an employee element, its action depends on the actual mode, `select` or `display`, in which this element was selected.

($ii$) Variables can be used for two purposes. The most apparent one is the one illustrated by the above query, namely, to perform joins between data values. A less apparent application is to use them as a 'look-ahead'. In Figure 5 we give a fragment of an XSLT program evaluating the truth value of a binary tree which represents a Boolean circuit. Essentially, the use of variables allows for a bottom-up computation. In brief, when arriving at an `or`-labeled node, the program returns the correct truth value based upon the truth values of the first and second subtree. The restriction to binary trees is just for expository purposes. In fact, we show in Section 4 how to simulate arbitrary bottom-up tree automata over unranked trees.

[5] Actually, a real XSLT processor does not require an XSLT program to start in a specific mode. We do require this, as it makes our model more uniform.

(*iii*) Passing of data values to other template rules can be crucial for performing joins if the items to be joined do not occur at the same node. Moreover, when node IDs are present in the XML document,[6] we can use this mechanism to place 'pebbles' on the input document which enables us to do complicated grouping operations.

A fourth feature, not illustrated by the example in Figure 2, is that XPath can also select siblings and ancestors. Exactly these four features render XSLT into a quite powerful transformation language.

```
<xsl:template match="or">
  <xsl:variable name="arg1">
    <xsl:apply-templates select="./*[1]"/>
  </xsl:variable>
  <xsl:variable name="arg2">
    <xsl:apply-templates select="./*[2]"/>
  </xsl:variable>
  <xsl:choose>
    <xsl:when test="($arg1 = 'false') and ($arg2 = 'false')">
      <xsl:value-of select="'false'"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="'true'"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Figure 5: The fragment for `or`-nodes of an XSLT program evaluating tree-structured Boolean circuits.

If we considered XSLT from a language theoretic point of view, then we could say that XSLT corresponds to *tree-walking tree transducers with registers* (working over unranked trees). In fact, the distinction between ranked and unranked trees can be dropped if the transducer model is capable of realizing the encoding of unranked trees by ranked ones, and the corresponding decoding. For example, it can be shown (cf. the proof of Theorem 18 in [MN99]), that the MSO transducer of [Cou94] can realize the standard encoding of unranked trees (see, e.g., [PQ68]) and the corresponding decoding. For similar reasons the authors of [MSV00] have drawn their attention to the ranked case only. Indeed, modes and the possibility to select all neighbours of the current node allows to explore the whole input tree as a tree-walking automaton; the output facility renders it into a transducer; and, the variables together with parameter passing can be seen as registers for data values. This connection will become clearer in Section 4.1.

# 3 A Formal Model for XSLT

In this section the formal model $\mathrm{XSLT}_0$ of XSLT is introduced. In the first subsection, unranked trees are defined. They form the data model of $\mathrm{XSLT}_0$: an $\mathrm{XSLT}_0$ program

[6] If not, XSLT is capable of generating them itself (see Section 4).

realizes a transformation from unranked trees to unranked trees. More precisely, such trees are attributed in the sense that nodes may have attributes associated with them, which take values from some domain. This provides a convenient way to represent XML documents. Nodes of an attributed tree can be selected by a pattern, which we define in a most general form; roughly speaking a pattern is a function which, given a tree and one of its nodes, returns a set of nodes.

The second subsection defines the syntax of $\mathrm{XSLT}_0$ programs and shows how the XSLT program of Figure 2 can be represented as an $\mathrm{XSLT}_0$ program. Finally, the third subsection defines the semantics of $\mathrm{XSLT}_0$ programs, based on rewriting. Also, it gives two examples of $\mathrm{XSLT}_0$ programs, which are used in Section 4 to show that $\mathrm{XSLT}_0$ programs are not closed under composition.

### 3.1 Trees, Forests, and Patterns

We start with the necessary definitions regarding trees and forests over an alphabet (that is, a finite set) $\Sigma$ (the symbols in $\Sigma$ correspond to the element names of the XML document, seen as a tree). To use these trees as adequate abstractions of actual XML documents, we extend them with attributes that take values from an infinite (recursively enumerable) domain $\mathbf{D} = \{d_1, d_2, \ldots\}$.

The set of $\Sigma$-trees, denoted by $\mathcal{T}_\Sigma$, is inductively defined as follows:

- every $\sigma \in \Sigma$ is a $\Sigma$-tree
- if $\sigma \in \Sigma$ and $f_1, \ldots, f_n \in \mathcal{T}_\Sigma$, $n \geq 1$ then $\sigma(f_1 \cdots f_n)$ is a $\Sigma$-tree.

A $\Sigma$-*forest* is a (possibly empty) sequence $f_1 \cdots f_n$, $n \geq 0$ of $\Sigma$-trees. The empty $\Sigma$-forest is denoted by $\varepsilon$. The set of all $\Sigma$-forests is denoted by $\mathcal{F}_\Sigma$. Note that there is no a priori bound on the number of children of a node in a $\Sigma$-tree; such trees are therefore *unranked*. In the following, whenever we say tree or forest, we always mean $\Sigma$-tree or $\Sigma$-forest, respectively.

The reason for considering forests is that, even if $\mathrm{XSLT}_0$ is used for tree-to-tree transformations only, we sometimes need to specify template rules which construct forests (see, e.g., Example 3.11).

For every forest $f \in \mathcal{F}_\Sigma$, the *set of nodes of* $f$, denoted by $\mathrm{Nodes}(f)$, is the subset of $\mathbb{N}^*$ inductively defined as follows:

- if $f = \sigma(t_1 \cdots t_n)$ with $\sigma \in \Sigma$, $n \geq 0$, and $t_1, \ldots, t_n \in \mathcal{T}_\Sigma$, then $\mathrm{Nodes}(f) = \{\varepsilon\} \cup \{iu \mid i \in \{1, \ldots, n\}, u \in \mathrm{Nodes}(t_i)\}$; and
- if $f = t_1 \cdots t_n$, then $\mathrm{Nodes}(f) = \{iu \mid i \in \{1, \ldots, n\}, u \in \mathrm{Nodes}(t_i)\}$.

Thus, for a tree the node $\varepsilon$ represents its root and $ui$ represents the $i$-th child of $u$. Further, for a forest the node $iu$ represents the node $u$ of the $i$-th tree in the forest. For every forest $f \in \mathcal{F}_\Sigma$ and every node $u$ of $f$, the *label* of $f$ at $u$ is denoted by $f[u]$. The *substitution* of the forest $f'$ at node $u$ in $f$ is denoted by $f[u \leftarrow f']$.

Next, we add XML attributes to the above defined forests. To this end, for the rest of the paper, we fix a finite set of attributes $A$.

**Definition 3.1** Let $\Sigma$ be an alphabet and $S$ a set. An *attributed* $\Sigma$*-forest with domain* $S$ is a pair $(f, (\lambda_a^f)_{a \in A})$, where $f \in \mathcal{F}_\Sigma$ and for each $a \in A$, $\lambda_a^f : \mathrm{Nodes}(f) \mapsto S$ is a

(partial) function assigning values in $S$ to nodes of $f$. The set of all attributed forests with domain $S$, is denoted by $\mathcal{F}_\Sigma^S$. □

The notions of label and substitution carry over from $\Sigma$-forests to attributed $\Sigma$-forests in the obvious way.

For $S$ we will usually take $\mathbf{D}$. However, to create output in template rules we will use attributed forests over $\mathbf{D} \cup \{x_1, \ldots, x_n\}$ where the variables $x_1, \ldots, x_n$ are those defined in the variable defining part of the template. Of course, in real XML documents, usually, not all element types have the same set of attributes. Obviously, this is just a convenience and not a restriction. In an analogous way one can define the set of attributed trees, denoted by $\mathcal{T}_\Sigma^S$. For a set $B$, $\mathcal{F}_\Sigma^S(B)$ denotes the set of attributed forests $f$ over $\Sigma \cup B$ such that symbols of $B$ may only appear at the leaves of $f$. Below, $B$ will be the set of apply template expressions.

In the next example we elaborate a bit on the connection between actual XML documents and attributed trees.

**Example 3.2** It should be clear that in an XML document, all text elements can be replaced by appropriate attributes. In terms of the corresponding tree this means that all nodes (including the leaves) are labeled by bracket labels (such as `beer` or `color` in this example), or by a special label T. Thus, the use of attributes to replace text elements solves the technicality of how to represent `<a>bcd</a>` as an unranked tree (either by the tree $a(bcd)$ with three leaves labeled by $b$, $c$, $d$, respectively, or by a tree with root node labeled $a$ and a single leaf labeled $bcd$).

For instance, consider the following XML document.

```
<beer name="Leffe trippel">
    <alc> 8.7 </alc>
    <description>
      <color> blonde </color>
      <taste> bitter </taste>
    </description>
</beer>
```

Figure 6 shows the attributed tree $s$ that corresponds to this document, where text elements are seen as single nodes, and the expression [name → "Leffe trippel"] means that $\lambda^s_{\text{name}}(\varepsilon) =$ "Leffe trippel" (cf. Example 3.3). Let us now represent text elements by attributes. The corresponding XML document looks as follows.

```
<beer name="Leffe trippel", alc="8.7">
    <description color="blonde", taste="bitter"/>
</beer>.
```

The latter can easily be represented by the attributed tree

$$t := \text{beer}(\text{description}),$$

where $\lambda^t_{\text{name}}(\varepsilon) =$ "Leffe trippel", $\lambda^t_{\text{alc}}(\varepsilon) =$ "8.7", $\lambda^t_{\text{color}}(1) =$ "blonde", and $\lambda^t_{\text{taste}}(1) =$ "bitter". Here, the domain $\mathbf{D}$ is assumed to contain the subset {"Leffe trippel", "8.7", "blonde", "bitter"}

In what follows we use a more convenient notation for attributed trees, similar to the original XML representation (and, already used in Figure 6). Namely, the label of a node may be followed by a list $[a_1 \to d_1, \ldots, a_n \to d_n]$ meaning that attribute $a_i$ has value $d_i$ at that node. If an attribute is not listed at a node, then it is assumed to be undefined. So, the attributed tree $t$ can be written as

$$\text{beer}[\text{name} \to \text{“Leffe trippel”}, \text{alc} \to \text{“8.7”}](\\ \text{description}[\text{color} \to \text{“blonde”}, \text{taste} \to \text{“bitter”}]).$$

As another example, consider the XML document

```
<p>This is <em>not</em> a problem.</p>
```

Here, we use the special text label T and the attribute text, to represent the document by the tree. That is,

```
<p>
    <T text="This is"/>
    <em text="not"/>
    <T text="a problem."/>
</p>
```

The latter can be readily represented by an attributed tree. Indeed, consider

$$t' = \text{p}(\text{T}[\text{text} \to \text{“This is”}]\text{em}[\text{text} \to \text{“not”}]\text{T}[\text{text} \to \text{“a problem.”}]).$$

Actually, this resembles the second choice of representing `<a>bcd</a>` mentioned above. □

### Patterns

In our formal model we abstract from a particular selection pattern language. Recall that XSLT uses the pattern language described in XPath [Cla99a] (see [Wad99], for a formal semantics). Patterns can be rather involved as illustrated by the second template rule in Figure 2, where the pattern depends on the value of the variable `varID`. In addition, patterns can also be *moving instructions* of the form parent, left sibling, right sibling, or first child. Actually, the proof of Theorem 4.2 indicates that such local selections are already enough in order to simulate many existing XML query languages.

In the following, we assume an infinite set of variables $\mathcal{X}$. We define a pattern over the variables $X \subseteq \mathcal{X}$ as a function of type

$$(\mathcal{T}_\Sigma^{\mathbf{D}} \times (X \mapsto \mathbf{D})) \mapsto (\mathbb{N}^* \mapsto 2^{\mathbb{N}^*})$$

and denote the set of all patterns over $X$ by $\mathcal{P}_\Sigma^X$. The idea is as follows. Let $p$ be a pattern, $t$ a tree, and $\gamma$ a variable assignment (for the variables in $p$). Then $p(t, \gamma)(u)$ is the set of selected nodes when the pattern is applied at node $u$.

**Example 3.3** Let $p$ be the XPath pattern

$$\texttt{beer[name[@val=}x\texttt{]]//color},$$

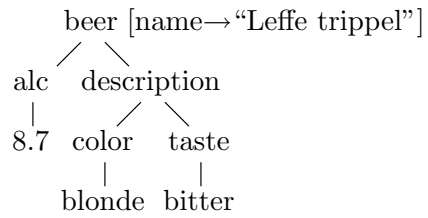



Figure 6: The attribute tree $s$.

where $x$ is a variable. Let $\gamma$ map $x$ to "Leffe trippel" and let $s$ be the attributed tree shown in Figure 6, i.e., the tree beer(alc(8.7),description(color(blonde),taste(bitter))) over the alphabet {8.7, alc, beer, bitter, blonde, color, description} with $\lambda^s_{\text{name}}(\varepsilon) =$"Leffe trippel" and $\lambda^s_{\text{name}}(v)$ undefined for all nodes $v$ of $s$ not being the root node $\varepsilon$. Recall that $s$ is a possible representation of the XML document shown in Example 3.2. Then $p(s, \gamma)(\varepsilon) = \{21\}$, because the node 21 is the only node of $s$ that is a color-labeled descendant of a beer-labeled node which has "Leffe trippel" as value of the attribute name. □

### 3.2 Syntax of XSLT$_0$

In this subsection an abstract formal syntax for (a fragment of) XSLT is defined, called XSLT$_0$. First, we restrict matching patterns to test only the label of the current node (as is already the case in Figure 2). This is not a harmful restriction, as Theorem 4.3 shows that we can test many properties of the current node in the body of a template rule. Second, we divide a template rule into two parts: the *variable definition part* and the *constructing part*. Variables can only be assigned by data values. As before, we assume an infinite (recursively enumerable) domain **D** and the only allowed operation on elements of **D** is the equality test. Further, a variable can only be defined as the value of some attribute of the current node or by an XSLT apply-templates statement which will return exactly one data value. We will refer to such special templates as *selecting template rules*. In the constructing part of the template rule, the actual output is defined relative to some conditions on the values of the variables, the parameters, the attribute values of the current node, and possibly on the fact whether the current node is the root, a leaf, or the first or last child of its parent; this output is a forest, which may contain at its leaves expressions which determine in which mode, at which nodes, and with which parameters the computation process should continue. This is done by an *apply-templates-expression* (for short, at-expression).

The distinction between a template rule that generates output (which will be referred to as a *constructing* template rule) and one that is selecting, will be realized in an XSLT$_0$ program by the modes: a mode $q$ is either *constructing* or *selecting*. Accordingly, at-expressions are either constructing or selecting depending on the type of the mode that they select. Moreover, selecting at-expressions may only select single nodes; this is guaranteed by restricting the patters that select nodes to moving instructions (recall the definition of patterns given in the discussion following Example 3.2).

- A *constructing at-expression* is of the form

$$q(p, \bar{z}),$$

where $q$ is a constructing mode, $p$ is a pattern and $\bar{z}$ is a possibly empty sequence of variables in $\mathcal{X}$ and domain elements in $\mathbf{D}$.

- A *selecting at-expression* is of the form

$$q(m, \bar{z}),$$

where $q$ is a selecting mode, $m$ is a moving instruction, and $\bar{z}$ as before. We assume $m$ can be at least of the form `to-self`, `to-first-child`, `to-last-child`, `to-left-sibling`, or `to-right-sibling`. These instructions have the obvious meaning.

We denote the set of constructing (selecting) at-expression by $\mathcal{AT}^{\mathrm{c}}$ ($\mathcal{AT}^{\mathrm{s}}$).

For instance, the apply-templates-expression in the second template of Figure 2 is a constructing one and corresponds in our model to the expression

$$\mathrm{display}(p, \mathrm{varID}),$$

with $p$ the pattern

```
//group[mgr/employee[@id=varID]]/group//employee.
```

Note that application of this pattern eventually leads to the generation of a `pair` element. So the expression is constructing in the sense that it eventually produces output.

For an attribute $a$ we call $a(.)$ an *attribute expression*. For a set $X \subset \mathcal{X}$ of variables, a *test (over X)* is a Boolean combination of expressions

- $x = y$ where $x$ and $y$ are attribute expressions, variables in $X$, or domain elements, and
- root, leaf, first-child, or last-child.

During a computation the expression $a(.)$ will evaluate to the value of the attribute $a$ of the current node. Further, root, leaf, first-child, and last-child evaluate to true whenever the current node is the root, a leaf, the first, or the last child of its parent, respectively.

**Definition 3.4** An *XSLT*$_0$ *program* is a tuple $P = (\Sigma, \Delta, M, \mathrm{start}, R)$, where

- $\Sigma$ is an alphabet of *input symbols*;
- $\Delta$ is an alphabet of *output symbols*;
- $M$ is a finite set of *modes* which is disjoint with $\Sigma \cup \Delta$, and is partitioned into the sets $M^c$ and $M^s$ of *constructing* and *selecting modes*, respectively;
- $\mathrm{start} \in M^c$ is the *start* mode; and
- $R$ is a finite set of *rules*, partitioned into the sets $R^c$ and $R^s$ of *constructing* and *selecting template rules*, respectively, which are defined as follows. For $q \in M$ and $\sigma \in \Sigma$, a $(q, \sigma)$-template rule $r$ is of the form

*template* $q(\sigma, x_1, \ldots, x_n)$
  *vardef*
    $y_1 := r_1; \ldots; y_m := r_m$
  *return*
    if $c_1$ then $z_1$; ...; if $c_k$ then $z_k$
*end*

where $n, m \geq 0$ and $k \geq 1$. The $x$'s and $y$'s are variables. Each $r_i$ is either an attribute expression or a selecting at-expression using variables among $x_1, \ldots, x_n, y_1, \ldots, y_{i-1}$. Each $c_i$ is a test over $X = \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$. In order to define what the $z_i$'s are, we distinguish whether the rule is constructing or selecting.

- If $q$ is a constructing mode, then $r$ is a *constructing apply template rule* and each $r_i$ is an attributed $\Delta$-forest with domain $\mathbf{D} \cup \mathcal{X}$ and leaves possibly labeled by at-expressions (using only variables in $X$), i.e., each $r_i$ is in $\mathcal{F}_\Delta^{\mathbf{D} \cup \mathcal{X}}(\mathcal{AT}^c)$.
- If $q$ is a selecting mode, then $r$ is a *selecting apply template rule* and each $z_i$ is a domain element, a variable in $X$, or a selecting at-expression (using only variables in $X$).

□

Intuitively, a rule of an XSLT$_0$program is evaluated as follows. First, the values of the variables $y_1$,...,$y_n$, are defined. Such a value can be an attribute value of the current node or can be defined by invoking an at-rule that will compute the desired data value. The return value then is determined by $z_i$ where $c_i$ is the first test that evaluates to true.

**Remark 3.5** To keep the model total and deterministic we require the existence of exactly one $(q, \sigma)$-template rule for every mode $q$ and every input symbol $\sigma$. Moreover, by default the empty forest will be output whenever none of the tests $c_1, \ldots, c_k$ evaluates to true (cf. Definition 3.8); i.e., we assume the presence of a return instruction

$$\text{if } c_{k+1} \text{ then } z_{k+1},$$

where $c_{k+1}$ equals true and $z_{k+1} = \varepsilon$.

To ensure that the output of an XSLT$_0$ program is a tree (rather than a forest), we require that, for every input symbol $\sigma$, each $z_i$ in the $(\text{start}, \sigma)$-constructing template rule is a tree (rather than a forest). □

**Remark 3.6** For convenience we use the keyword *self* if we want to copy the current node together with the values of all its attributes. The use of self can be simulated by first defining the variables $x_a := a(.)$ for all attributes $a \in A$, and then writing $\sigma[(a \to x_a)_{a \in A}]$, where $\sigma$ is the label of the current node. □

**Example 3.7** We illustrate the the definition of XSLT$_0$programs by translating the program in Figure 2 into our syntax. The patterns $p_1$ and $p_2$ refer to the patterns in the first and second template rules of the XSLT program in Figure 2, respectively.

In the second template rule display($p_2$,varID) is the tree consisting of one node labeled with display($p_2$,varID); recall that $\varepsilon$ denotes the empty forest. In the last rule, `pair`[`topmgr`→varID; `employeeID`→myID] denotes the tree $t$ consisting of one node where $\lambda^t_{\texttt{topmgrID}}(\varepsilon)$ := varID and $\lambda^t_{\texttt{employeeID}}(\varepsilon)$ := myID. For readability, we omitted the test 'if true then'. All modes are constructing. For clarity, we use a box to indicate at-expressions.

*template* start(organization)
    *return*
        `result`(selecttopmgr($p_1$))
*end;*

*template* selecttopmgr(employee)
    *vardef*
        varID := id(.)
    *return*
        if varID ≠ Bill then display($p_2$,varID);
        if varID = Bill then $\varepsilon$
*end;*

*template* display(employee,varID)
    *vardef*
        myID := id(.)
    *return*
        `pair`[`topmgrID`→varID; `employeeID`→myID]
*end;*

Note that the rule "if varID = Bill then $\varepsilon$" is not needed, according to Remark 3.5. □

### 3.3 Semantics of XSLT$_0$ Programs

In this section we define the semantics of an XSLT$_0$ program $P$ on an input tree $t$. This is done by defining two rewrite relations, one for the selecting and one for the constructing process. A *local configuration* is an element of

$$LC(t) := \text{Nodes}(t) \times (M^c \cup M^s) \times \mathbf{D}^*.$$

Note that here we will use a comma-separated list $d_1, \ldots, d_n$ to denote an element in $\mathbf{D}^*$. Intuitively, $\theta := (u, q, d_1, \ldots, d_n) \in LC(t)$ means that the program has selected node $u$ in mode $q$ with $d_1, \ldots, d_n$ as the values of the parameters $x_1, \ldots, x_n$, respectively. A local configuration is selecting (constructing) if its associated state is selecting (constructing). Define $LC^s(t)$ and $LC^c(t)$ as the set of selecting and constructing local configurations, respectively. In the selection process, a selecting local configuration is rewritten, until a domain element is reached (which is the value of a variable or an attribute at the current node, or it is a constant generated by a return rule).

Let $\bar{w}$ consist of a sequence of variables of $\mathcal{X}$ and domain elements. For a function $\gamma : \mathcal{X} \mapsto \mathbf{D}$, we denote by $\bar{w}[\gamma]$ the sequence of domain elements obtained from $\bar{w}$ by

replacing each occurrence of the variable $x$ in $\bar{w}$ by $\gamma(x)$. By $x_1\backslash d_1, \ldots, x_n\backslash d_n$, or $\bar{x}\backslash\bar{d}$ for short, we denote the function that maps each $x_i$ to $d_i$. For a binary relation $R$, we denote by $R^*$ its transitive closure.

**Definition 3.8** The *selecting rewrite relation induced by $P$ on $t$*, denoted by $\rightarrow_{P,t}$, is the binary relation on $LC^s(t) \cup \mathbf{D}$ defined as follows. For $\theta = (u, q, d_1, \ldots, d_n) \in LC^s(t)$ and $\alpha \in LC^s(t) \cup \mathbf{D}$,

$$\theta \rightarrow_{P,t} \alpha$$

if

- the $(q, \sigma)$-selecting template rule has $n$ local variables for $t[u] = \sigma$; further, let the $(q, \sigma)$-template rule be of the form as specified in Definition 3.4, where each $r_i$ is $q_i(p_i, \bar{x}'_i)$ or the attribute expression $a_i(.)$;
- there are $e_1, \ldots, e_m \in \mathbf{D}$ such that for every $1 \leq i \leq m$,
  - if $r_i$ is an at-expression, then $(v_i, q_i, \bar{x}'_i[\gamma_i]) \rightarrow^*_{P,t} e_i$, where $\gamma_i$ maps each $x_j$ to $d_j$, for $j = 1, \ldots, n$, and $y_j$ to $e_j$ for $j = 1, \ldots, i-1$, and $v_i$ is the node selected by $p_i$, that is, $p_i(t, \gamma_i)(u) = \{v_i\}$; or
  - if $r_i$ is an attribute expression, then $e_i := \lambda^t_{a_i}(u)$;
- $c_i$ is the first condition that evaluates to true by interpreting each $y_j$ by $e_j$, $x_j$ by $d_j$, $a(.)$ by $\lambda^t_a(u)$, and root, leaf, first-child, last-child by true if and only if $u$ is the root, a leaf, the first or the last child of its parent, respectively;
  - if $z_i$ is a constant, a variable, or an attribute expression then $\alpha \in \mathbf{D}$ and it equals the corresponding value;
  - if $z_i$ is a selecting at-expression $q'(p, \bar{w})$, then $\alpha \in LC^s(t)$ and $\alpha = (v, q', \bar{w}[\bar{x}\backslash\bar{d}, \bar{y}\backslash\bar{e}])$ where $v$ is the node selected by $p$, that is, $p(t, [\bar{x}\backslash\bar{d}, \bar{y}\backslash\bar{e}])(u) := \{v\}$.

□

In the constructing process, constructing local configurations are replaced by output forests, which possibly contain constructing local configurations at their leaves, until only output symbols are present. A rewrite step, viz. the application of a constructing template rule, may involve the replacement of a pattern by the nodes of the input tree which match the pattern. We choose to order these nodes in pre-order (of the input tree), which is the document order (of the input document).

**Definition 3.9** The *constructing rewrite relation induced by $P$ on $t$*, denoted by $\Rightarrow_{P,t}$, is the binary relation on $\mathcal{F}_\Delta(LC^c(t))$ (recall that these are the $\Delta$-forests where leaves may be labeled with constructing local configurations) defined as follows. For $\xi, \xi' \in \mathcal{F}_\Delta(LC^c(t))$,

$$\xi \Rightarrow \xi',$$

if there is a leaf node $v \in \text{Nodes}(\xi)$ with $\xi[v] = (u, q, d_1, \ldots, d_n) \in LC^c(t)$ such that

- the $(q, \sigma)$-constructing template rule has $n$ local variables for $t[u] = \sigma$; further, let the $(q, \sigma)$-template rule be of the form as specified in Definition 3.4; recall that there each $z_j$ is a forest in $\mathcal{F}^{\mathbf{D}\cup\mathcal{X}}_\Delta(\mathcal{AT}^c)$, that is, a forest where attributes take values in $\mathbf{D} \cup \mathcal{X}$ and where leaves may be labeled with constructing at-expressions;

- there are $e_1, \ldots, e_m \in \mathbf{D}$ and a $c_i$ that fulfill the same conditions as in Definition 3.8;
- $\xi' = \xi[v \leftarrow f]$, where $f$ is the forest obtained from $z_i$ by replacing
  - every occurrence of a $y_j$ and a $x_j$ as the value of some attribute by the data values $e_j$ and $d_j$, respectively;
  - every occurrence $q'(p', \bar{w})$ of an at-expression at a leave of $z_i$ by the forest

$$(u_1, q', \bar{w}[\bar{x}\backslash\bar{d}, \bar{y}\backslash\bar{e}]) \cdots (u_\ell, q', \bar{w}[\bar{x}\backslash\bar{d}, \bar{y}\backslash\bar{e}]),$$

    where $p'(t, [\bar{x}\backslash\bar{d}, \bar{y}\backslash\bar{e}])(u) = \{u_1, \ldots, u_\ell\}$ and $u_1 \prec \ldots \prec u_\ell$ (here $\prec$ denotes pre-order).

□

The *initial* local configuration of $t$ is defined as $\theta_{\text{start}}(t) := (\varepsilon, \text{start}, \varepsilon)$, i.e., it is the tree consisting of the single (root) node labeled $(\varepsilon, \text{start}, \varepsilon)$.

**Definition 3.10** The transformation *realized* by $P$, is the (partial) function $\tau_P : \mathcal{T}_\Sigma^{\mathbf{D}} \to \mathcal{T}_\Delta^{\mathbf{D}}$ with $\tau_P(t) = s$ if $t \in \mathcal{T}_\Sigma^{\mathbf{D}}$ and there is an $s \in \mathcal{T}_\Delta^{\mathbf{D}}$ such that $\theta_{\text{start}}(t) \Rightarrow_{P,t}^* s$. □

Note that the above is well-defined. By Remark 3.5 and the definition of $\to$ and $\Rightarrow$, the transformation is deterministic and generates trees only.

We illustrate the above definitions by an example. In the above we did not instantiate a concrete pattern language. In the examples we assume that it at least contains the functionality of XPath, the pattern language of XSLT. As explained before, we will assume that the following selection patterns are available: `to-self`, `to-first-child`, `to-last-child`, `to-left-sibling`, and `to-right-sibling` with the obvious meanings.

**Example 3.11** Let $P_1 = (\Sigma_1, \Delta_1, M_1^c, \text{start}, R_1)$ be the XSLT$_0$ program where $\Sigma_1 = \{a\}$, $\Delta_1 = \{b, c\}$, $M_1^c = \{\text{start}, \text{double}\}$, and $R_1$ consists of the two rules

*template* start($a$)
  *return*
    $b($double(`to-self`) double(`to-self`)$)$
*end;*

and

*template* double($a$)
  *return*
    if leaf then $c$;
    if ¬leaf then double(`to-first-child`) double(`to-first-child`)
*end;*

Note that in this example we made use of constructing a forest in a rule, namely the forest double(`to-first-child`) double(`to-first-child`), even though the translation will be a tree-to-tree translation. This program transforms a monadic tree consisting of $n$ $a$'s (that is a tree where every node has rank at most one) into the tree $b(c \cdots c)$ with $2^n$ $c$'s. A part of the derivation is illustrated in Figure 7. Essentially, every $a$ is doubled at each transition. Only when reaching a leaf, a $c$ is being output.

Let $P_2 = (\Sigma_2, \Delta_2, M_2^c, \text{start}, R_2)$ be the XSLT$_0$ program where $\Sigma_2 = \{b, c\}$, $\Delta_2 = \{a\}$, $M_2^c = \{\text{start}, \text{copy}\}$, and $R_2$ consists of the two rules

*template* start($b$)
    *return*
        copy(`to-first-child`)
*end;*

*template* copy($c$)
    *return*
        if ¬last-child then $a$( copy(`to-right-sibling`) );
        if last-child then $a$
*end;*

This program transforms a tree $b(c \cdots c)$ with $m \geq 1$ $c$'s into a monadic tree of height $m$ consisting only of $a$'s. In brief, the program jumps to the first $c$ without generating any output. Then it visits every sibling from left to right, each time generating an $a$ as output.

The above programs will be used in the proof of Theorem 4.5 to show that XSLT$_0$ is not closed under composition. □

$$
\begin{array}{lcl}
\Theta_{\text{start}(a(a))} & = & (\varepsilon, \text{start}, \varepsilon) \\
(\varepsilon, \text{start}, \varepsilon) & \Rightarrow_{P_1, a(a)} & b((\varepsilon, \text{double}, \varepsilon)(\varepsilon, \text{double}, \varepsilon)) \\
 & \Rightarrow^*_{P_1, a(a)} & b((1, \text{double}, \varepsilon)(1, \text{double}, \varepsilon)(1, \text{double}, \varepsilon)(1, \text{double}, \varepsilon)) \\
 & \Rightarrow^*_{P_1, a(a)} & b(cccc)
\end{array}
$$

Figure 7: Derivation of the program $P_1$ on input $a(a)$

**Example 3.12** Let $P = (\Sigma, \Sigma, M^c, \text{output\_subtree}, R)$ be the XSLT$_0$ program where $M^c = \{\text{output\_subtree}\}$, $\Sigma$ is an arbitrary alphabet, and $R$ contains for every $\sigma \in \Sigma$ the rule

*template* output_subtree($\sigma$)
    *return*
        if leaf then *self*
        if last-child then *self*( output_subtree(`to-first-child`) )
        if ¬last-child then

*self*( output_subtree(`to-first-child`) ) output_subtree(`to-right-sibling`)

*end;*

Recall from Remark 3.6 that the keyword *self* copies the current node together with the values of all its attributes. We invite the reader to verify that $P$ simply copies the input tree. □

We conclude this section with some remarks. We note that XSLT does not make the explicit distinction between constructing and selecting template rules, or even, between the variable definition part and the constructing part of a template rule. However, we feel that by making this explicit, programming becomes more structured. On the other hand, we did not incorporate everything XSLT has to offer. For instance, we refrained from including for-loops. Nevertheless, we indicate in the next section that we have captured a powerful fragment capable of simulating most existing XML query languages and even more. With the aid of the example in Section 2, we invite the reader to check that any program in our abstract syntax can be readily translated into actual XSLT.

# 4 Properties

In this section we show that $\text{XSLT}_0$, and hence XSLT, is quite expressive in the sense that it can simulate most other current XML query languages, and that it can express all join-free MSO patterns. The latter even implies that XSLT is strictly more expressive than other XML query languages. On the negative side we obtain that deciding termination is undecidable and that $\text{XSLT}_0$ is not closed under composition.

## 4.1 An abstract declarative query language for XML

Although there is no generally accepted transformation language for structured documents, several ones have emerged during the last years, including XML-QL [DFF+99a], XSLT [Cla99b], and XQL [Rob99], which are specifically for XML, and Lorel [AQM+97], StruQL [FFK+98], and UnQL [BDHD96], for the semi-structured data model. As argued by Fernandez, Siméon, and Wadler [FSW99], XML queries roughly consists of a *pattern* clause and a *constructor* clause.[7] The purpose of the pattern language is to identify the different parts of the document that have to be combined to obtain the output document. The constructing part, on the other hand, indicates how the selected parts should be assembled.

Such queries can, for instance, be written as

$$\texttt{WHERE}\ \varphi(x_1, \ldots, x_n),\ \texttt{CONSTRUCT}\ \text{result}(s), \qquad (*)$$

where $\varphi$ is a pattern selecting nodes and $s$ is a tree in $\mathcal{T}_\Sigma^{\mathbf{D}\cup\{\bar{x}\}}$ containing at leaves special constructs like $\text{lab}(x_i)$ and $\text{subtree}(x_i)$ indicating that at this position the label or the subtree rooted at the matched node for $x_i$ should be plugged in.

More precisely, let $\bar{u}^1 \prec \ldots \prec \bar{u}^\ell$ be the tuples of nodes of $t$ selected by $\varphi$ (again, $\prec$ denotes pre-order, extended to tuples of nodes in the obvious way). Then the output of the above query on $t$ is

$$\text{result}(s[\bar{u}^1] \cdots s[\bar{u}^\ell]),$$

[7] We want to point out that the recent XQuery proposal differs substantially from this paradigm. [Cha01, CFS01]

where $s[\bar{u}^j]$, $\bar{u}^j = u_1^j, \ldots, u_n^j$, denotes the tree obtained from $s$ by replacing each $\text{lab}(x_i)$ by $t[u_i^j]$, each $\text{subtree}(x_i)$ by $t/u_i^j$, and each $x_i$ occurring in an attribute definition of $a$ by $\lambda_a^t(u_i^j)$. If we denote the query in $(*)$ by $Q$ then we denote the result of $Q$ on $t$ by $Q(t)$.

As a pattern language, most of these XML query languages, combine the query primitives that are used in relational databases, hence SQL, with constructs that allow to navigate along paths, often by means of regular expressions. Clearly, they are all contained in first-order logic (FO) extended by vertical and horizontal regular path expressions as defined next.

First we say how we view attributed trees as logical structures (in the sense of mathematical logic [EF95]) over the binary relation symbols $E$ and $<$, and the unary relation symbols $(O_\sigma)_{\sigma\in\Sigma}$ and $(\lambda_a^t)_{a\in A}$. We denote this vocabulary by $\tau_\Sigma$. The domain of $t$, viewed as a structure, equals the set of nodes of $t$, i.e., $\text{Nodes}(t)$. $E$ is the edge relation and equals the set of pairs $(v, vi)$ for every $v, vi \in \text{Nodes}(t)$. The relation $<$ specifies the ordering of the children of a node, and equals the set of pairs $(vi, vj)$, where $i < j$ and $vi, vj \in \text{Nodes}(t)$. For each $\sigma$, $O_\sigma$ is the set of nodes that are labeled with a $\sigma$.

We next define the logic FOREG. An atomic FOREG formula is an atomic formula over $\tau_\Sigma$ or is of the form

1. $a(x) = b(y)$,
2. $a(x) = d$,
3. $[r]^{\rightarrow}(x, y)$, or
4. $[r]^{\downarrow}(x, y)$,

where $x$ and $y$ are variables, $a$ and $b$ are attributes, $d \in \mathbf{D}$, and $r$ is a regular expression over $\Sigma$. These formulas hold for nodes $u$ and $v$ in $t$, if

1. $\lambda_a^t(u) = \lambda_b^t(v)$,
2. $\lambda_a^t(u) = d$,
3. $u$ is a left sibling of $v$ and the labels on the path from $u$ to $v$ (both included) match $r$, or
4. $u$ is an ancestor of $v$ and the labels on the path from $u$ to $v$ (both included) match $r$.

A FOREG formula $\varphi$ is the usual closure of FOREG atomic formulas under conjunction, negation, and first-order quantification over nodes. Note that there is no quantification over elements from $\mathbf{D}$. If $\varphi$ holds for a tree $t$, then this is denoted by $t \models \varphi$.

We define $\text{QL}_0$ as queries of the form (*) where $\varphi$ is a FOREG formula. As explained above, $\text{QL}_0$ forms an abstraction of current XML query languages. However, we point out that we did not include skolem functions (as present in XML-QL for instance), but we did include vertical horizontal expressions.

The next theorem shows that $\text{XSLT}_0$ can express all queries definable in $\text{QL}_0$. In fact, this only holds under the assumption that each node has a unique id, that is, an attribute

with a unique value throughout the document. Strictly speaking, this assumption is not necessary as XSLT is equipped with the function `generate-id(.)` which generates a unique id for the current node. Furthermore, this id only depends on the current node, that is, when invoked for the same node several times it will return the same value. However, as we feel that the function `generate-id(.)` is quite unnatural we exclude it from our model $\mathrm{XSLT}_0$ and state the unique-id requirement explicitly.

**Theorem 4.1** *Under the assumption that each node has a unique id, for every $QL_0$ query $Q$ there is an $XSLT_0$ program $P$ such that $Q(t) = \tau_P(t)$ for every input tree $t$.*

**Proof.** Consider a $\mathrm{QL}_0$ query $Q$ of the form (*). Essentially, on a tree $t$, we have to (1) enumerate all tuples $\bar{u}^1 \prec \ldots \prec \bar{u}^\ell$ in pre-order, (2) for each $i$ check whether $t \models \varphi(\bar{u}^i)$, and (3) if so, output $s[\bar{u}^i]$.

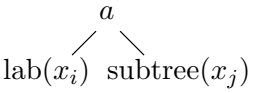
1. We first explain how to enumerate all tuples $\bar{u}^1 \prec \ldots \prec \bar{u}^\ell$ in pre-order. That is, how to enumerate assignments for the variables $x_1, \ldots, x_n$ in the pattern $\varphi$. Therefore, we use the same $n$ variables $x_1, \ldots, x_n$ in the $\mathrm{XSLT}_0$ program. The variables are then passed between template rules. The unique-id assumption allows us to remember positions in the input tree. Indeed, we simply assign node id's to the variables. Initially we assign the root node $\varepsilon$ to each variable. Next, suppose the variables are assigned id's. Let $u$ be the id of the last node in pre-order. Assume not all $x_i = u$. To compute the next tuple in pre-order, let $j \in \{1, \ldots, n\}$ be the largest number such that for each $i \leq j$, $x_i = u$. Then assign $\varepsilon$ to each $x_i$ for $i \leq j$ and assign to $x_{j+1}$ the id of the node following $x_{j+1}$ in pre-order.

2. Suppose the variables are assigned the nodes $u_1, \ldots, u_n$. We show how to test whether $t \models \varphi(u_1, \ldots, u_n)$ by induction on the structure of FOREG formulas. We can assume that $\varphi$ is in prenex normal form with the quantifier-free part in disjunctive normal form.

   Atomic formulas of the form $E(u_i, u_j)$, $u_i < u_j$, $a(u_i) = b(u_j)$, $a(u_i) = d$ are tested in a straightforward manner. Indeed, to test $E(u_i, u_j)$, for instance, the $\mathrm{XSLT}_0$ program finds $u_i$ by traversing the tree in a depth-first manner. Then it checks whether $u_j$ is a child of $u_i$. Slightly more difficult are atomic formulas of the form $[r]^{\downarrow}(u_i, u_j)$ and $[r]^{\rightarrow}(u_i, u_j)$. To test the former, for instance, an $\mathrm{XSLT}_0$ program acts as follows. First it locates $u_j$ by doing a depth-first search of the tree. Then it walks towards the root simulating the deterministic automaton for the reverse of the language defined by $r$. If the program arrives at $u_i$ in a final state it accepts, otherwise when a non-final state is reached or when $u_i$ is not an ancestor it rejects. Finally, to test a disjunction of conjunctions of atomic formulas it suffices to find a disjunct for which all atomic formulas hold. This concludes the case of quantifier-free formulas.

   Next we treat the quantifiers. Suppose we have an $\mathrm{XSLT}_0$ program to test a formula $\psi(v, \bar{u})$ which is in prenex normal form. Then $\exists x \psi(x, \bar{u})$ can be tested by subsequently assigning every node to $v$ and testing whether $\psi(v, \bar{u})$ holds. The program then accepts when at least one such $v$ exists. Testing $\forall x \psi(x, \bar{u})$ is analogous.

3. It remains to show how to output $s[\bar{u}]$ when $t \models \varphi(\bar{u})$ where $\bar{u} := u_1 \cdots u_n$. We explain the latter by means of an example. Suppose that $s$ is a tree of the form




Then the template rule for $\varphi$ should have as output the forest

$$a\Big(\boxed{\text{find\_}x_i\text{\_output\_label(}\texttt{to-self}\text{)}}\ \boxed{\text{find\_}x_j\text{\_output\_subtree(}\texttt{to-self}\text{)}}\Big)\ \boxed{\text{next\_tuple(}\texttt{to-self}\text{)}}.$$

Here, an $a$ is output. Further, the following three subprocesses are started. The at-expressions find_$x_i$_output_label(`to-self`) starts a subcomputation that finds $u_i$, that is, the node assigned to $x_i$ and outputs its label. The at-expressions find_$x_j$_output_subtree(`to-self`) starts a subcomputation that finds $u_j$, that is, the node assigned to $x_j$ and outputs the subtree rooted at $u_j$. The latter can be achieved as in Example 3.12. Finally, next_tuple(`to-self`) starts a computation taking care of the next tuple in pre-order. □

**$k$-pebble transducers.** We conclude this section by providing some indirect evidence for the expressiveness of $\text{XSLT}_0$ by comparing $\text{XSLT}_0$ with $k$-pebble tree transducers. Readers not familiar with the latter can safely skip this section.

To study decidability of type checking, Milo, Suciu, and Vianu [MSV00] defined the $k$-pebble tree transducer as a formalism capturing the expressiveness of most existing XML query languages. Such transducers transform binary trees into other binary trees.[8] Next, we informally describe such deterministic transducers with *equality tests on data values*.

The $k$-pebble deterministic tree transducer uses up to $k$ pebbles to mark certain nodes in the tree. Transitions are determined in a unique way by the current node symbol, the current state (or mode), the presence/absence of the various pebbles on the current node, and equality tests on the attribute values of the nodes the pebbles are located on. Pebbles are ordered and numbered from 1 to $k$. The machine can place pebbles on the root, move them around, and remove them (actually, the use of the pebbles is restricted by a stack discipline which ensures that the model does not become too powerful, that is, accepts non-tree-regular languages). There are move transitions and output transitions. Move transitions are of the following kind: to-parent, to-first-child, to-last-child, to-left-sibling, to-right-sibling, remain-and-change-state, place-new-pebble, and pick-current-pebble. There are two kinds of output transitions. A binary output symbol $\sigma$, possibly with attributes defined by an attribute value of a pebbled node, and spawns two computation branches that compute, independently of each other, the left and the right subtree of $\sigma$. Both branches inherit the positions of all pebbles on the input and do not communicate with each other, that is, each branch moves the

[8] When proving properties of XML transformations, restricting to binary trees is usually sufficient as unranked trees can be coded by ranked ones (which in turn can be coded by binary trees; cf. the remark at the end of Section 2.

pebbles independently of the other. A nullary output $\alpha$ generates an $\alpha$-labeled leaf and the computation halts.

It should be clear that, apart from the pebbles, the above described model is extremely close to $\mathrm{XSLT}_0$: $\mathrm{XSLT}_0$ is equipped with modes (states), can do local movements and the simple output transitions. Under the assumption that each node has a unique id, $\mathrm{XSLT}_0$ can also simulate pebbles. Indeed, we just use $k$ variables $x_1$ up to $x_k$, where at each time instance $x_i$ contains the id of the node on which the $i$-th pebble is located.

The above discussion immediately leads to the next theorem.

**Theorem 4.2** *$XSLT_0$ can simulate $k$-pebble deterministic tree transducers with equality tests on data values.*

We point out that also non-deterministic tree transducers can be simulated, by giving a non-deterministic semantics to $\mathrm{XSLT}_0$ in the straightforward way.

## 4.2 Join-free MSO

We next consider monadic second-order logic (MSO) as a pattern language. MSO is first-order logic (FO) extended by quantification over set variables. We refer the unfamiliar reader to, e.g., the books by Ebbinghaus and Flum [EF95], Immerman [Imm98], or the chapter by Thomas [Tho97a]. Join-free MSO is then defined as MSO over the above vocabulary extended by atomic formulas of the form $a(x) = d$ that are interpreted as explained above. Note that we do not allow atomic formulas of the form $a(x) = b(y)$. In other words, we do not allow joins. Again no quantification over $\mathbf{D}$ is allowed, neither first-order nor second-order.

Clearly, join-free MSO can define all structural XPath matching patterns. That is, the ones without number and string functions. The next theorem says that $\mathrm{XSLT}_0$, and hence XSLT, is capable of expressing all unary join-free MSO patterns. In particular, this means that one does not need matching patterns in templates. That is, $\mathrm{XSLT}_0$ actually allows to specify rules like

```
<xsl:template match="p" mode="q">
```

where $p$ is a matching pattern rather than just a label. It means that a rule can only be applied on nodes that satisfy $p$. Theorem 4.3 implies that one can test for $p$ in the body of the template rule and, hence, does not need matching patterns. Let **true** be an element of $\mathbf{D}$.

**Theorem 4.3** *Let $\varphi(x)$ be a join-free MSO formula. There exists an $XSLT_0$ program $P$ and a mode $q_\varphi$ such that $(u, q_\varphi) \to^*_{P,t}$ **true** if and only if $t \models \varphi(u)$.*

**Proof.** Essentially, a join-free MSO formula $\varphi$ using the constants $d_1, \ldots, d_n$ can be translated into an ordinary MSO formula $\tilde{\varphi}$ over an extended (but finite) vocabulary containing, say, the sets $\{O^a_{d_1}, \ldots, O^a_{d_n} \mid a \in A\}$. Here, for a node $u$ and a tree $t$, $t \models O^a_{d_i}(u)$ if and only if $\lambda^t_a(u) = d_i$. Hence, it suffices to proof the theorem for MSO rather than join-free MSO.

There is a well-known connection between sets of trees defined by MSO formulas and (unranked) regular tree languages. Before we state this, we recall the definition of tree automata over unranked trees.

A *deterministic bottom-up tree automaton (DBTA)* is a tuple $B = (Q, \Sigma, F, \delta)$, where $Q$ is a finite set of states, $F \subseteq Q$ is the set of final states, and $\delta$ is a function $Q \times \Sigma \to 2^{Q^*}$ such that $\delta(q, a)$ is a regular language for every $a \in \Sigma$ and $q \in Q$. To ensure determinism, we require that $\delta(q, \sigma) \cap \delta(q', \sigma) = \emptyset$ for all $q \neq q' \in Q$ and $\sigma \in \Sigma$. The semantics of $B$ on a tree $t$, denoted by $\delta^*(t)$, is defined inductively as follows:

- if $t$ consists of only one node labeled with $\sigma$ then $\delta^*(t) = q$ with $q$ such that $\varepsilon \in \delta(q, \sigma)$;
- if $t$ is of the form $\sigma(t_1, \ldots, t_n)$, then $\delta^*(t) = \{q \mid \delta^*(t_1) = q_1, \ldots, \delta^*(t_n) = q_n \text{ and } q_1 \cdots q_n \in \delta(q, \sigma)\}$.

A tree $t$ over $\Sigma$ is *accepted* by the automaton $B$ if $\delta^*(t) \in F$. A set of trees is *regular* if it is accepted by a DBTA.

Let $\varphi(x)$ be an MSO formula. It is known that the set $\{(t, u) \mid t \models \varphi[u]\}$ is a regular tree language [NS99, Nev99a, Tho97a]. Here, $(t, u)$ is the tree where a distinguishing $*$ is attached to the label of $u$ (for instance, if $t[u] = \sigma$, then the label of $u$ in $(t, u)$ is $\sigma*$).

Let $B = (Q, \Sigma, F, \delta)$ be the DBTA accepting the tree language $\{(t, u) \mid t \models \varphi(u)\}$. We next construct an $\text{XSLT}_0$ program that when started at $u$ to determine whether $t \models \varphi$, stores the id of $u$ in a special variable. Subsequently, the program walks to the root and simulates the tree automaton in a depth-first fashion. Whenever it encounters a node $v$ labeled $\sigma$ it checks whether the id of $v$ equals the id of $u$. If so, the program acts as if it reads label $\sigma*$; otherwise, it acts as if it reads $\sigma$.

We explain this in a bit more detail. Let for each $q, \sigma$, $M^{q,\sigma} = (Q^{q,\sigma}, Q, \delta^{q,\sigma}, q_0^{q,\sigma}, F^{q,\sigma})$ be the (deterministic) finite state automaton accepting $\delta(q, \sigma)$. Note that $M^{q,\sigma}$ accepts strings over $Q$. To simulate $B$ we start the following procedure at the root of $t$. We use as states pairs $(\sigma, g)$ where $\sigma \in \Sigma$ and $g$ is a function from $Q$ to $\bigcup_{q \in Q} Q^{q,\sigma}$. The idea is as follows. If a node $vi \neq \varepsilon$ is called in state $(\sigma, g)$ then $t[v] = \sigma$ and for each $q \in Q$, $\delta^{q,\sigma*}(\delta^*(t_{v1}) \cdots \delta^*(t_{v(i-1)})) = g(q)$. That is, the parent of $vi$ is labeled with $\sigma$ and when $M^{q,\sigma}$ processes the states assumed by $B$ at the left siblings of $vi$, it arrives at $ui$ in state $g(q)$. Define $g_0^\sigma$ as $g_0(q) = q_0^{q,\sigma}$ for all $q \in Q$.

The $\text{XSLT}_0$ program $P$ starts at the root where it defines a variable

$$y := \boxed{(t[\varepsilon], g_0)(\texttt{to-first-child})}.$$

That is, it selects the first child of the root in state $(t[\varepsilon], g_0)$ (we tacitly assume that $t$ consists of more than one node).

Recall that for a node $v$, $t[v]$ is the label of $v$ in $t$. As the unique id of $u$ is stored in some variable, we have the means to check whether the label of a node should actually be $\sigma*$ rather than $\sigma$. We, therefore, abuse notation and assume that $t[u] = \sigma*$.

Suppose $P$ selects $vi$ in state $(\sigma, g)$.

- First, $P$ computes $\delta^*(t_{vi})$, the state $B$ assumes at $vi$. More precisely, if $vi$ is a leaf $\delta^*(t_{vi}) := \tilde{q}$ for which $\varepsilon \in \delta(\tilde{q}, t[vi])$. Otherwise, $P$ defines a variable $y := \boxed{(t[vi], g_0)(\texttt{to-first-child})}$ that selects the first child of $vi$. Denote the value of this variable by $\tilde{q}$.
- If $vi$ is the right most child of $v$ then $P$ returns the unique $q$ for which $g(q) \in F^{q,\sigma}$. This means that, $M^{q,\sigma}$ accepts $\delta^{q,\sigma*}(\delta^*(t_{v1}) \cdots \delta^*(t_{vi}))$. Otherwise, $P$ selects its right sibling in state $(\sigma, g')$ where $g'(q) = \delta^{q,\sigma}(g(q), \tilde{q})$, for each $q \in Q$.

Finally, when the variable $y$ gets assigned a final state, $P$ knows that $t \models \varphi(u)$ and can return the value **true**. □

### 4.3 Termination

An $\mathrm{XSLT}_0$ program $P$ is said to be *terminating* when $\tau_P$ is defined everywhere. We next show that deciding termination is undecidable. Of course, one can easily write an interpreter that makes sure that the execution of any $\mathrm{XSLT}_0$ program always terminates. However, it is justified to say that a non-terminating program is ill-defined.

**Theorem 4.4** *It is undecidable whether or not an XSLT$_0$ program is terminating.*

**Proof.** It is well known that the halting problem of multi-head automata on strings is undecidable [FS73]. In brief, a multi-head automaton is a two-way deterministic finite automaton over strings with multiple heads. Movements depend on the current state and the symbols below the respective heads. Clearly, a multi-head automaton can be simulated by a one-head automaton with pebbles. Each pebble corresponds to a head. One step of a multi-head automaton can be simulated by several steps of a one-head automaton with registers. Indeed, when the automaton needs to know the symbol under a specific head it just makes a sweep of the string looking for the correct position.

□

### 4.4 Composition

Let $P_1$ be an $\mathrm{XSLT}_0$ program with input alphabet $\Sigma$ and output alphabet $\Gamma$, and let $P_2$ be an $\mathrm{XSLT}_0$ program with input alphabet $\Gamma$ and output alphabet $\Delta$. Then, the *composition of $P_1$ with $P_2$* is the (partial) function

$$\{(t, s) \in T_\Sigma \times T_\Delta \mid \text{there are } t' \in T_\Gamma, s \in T_\Delta : \tau_{P_1}(t) = t' \text{ and } \tau_{P_2}(t') = s\}.$$

In order to avoid the construction of intermediate trees (viz. the tree $t'$ above), it is, for a class of translations, a desirable property to be closed under composition. Examples of such classes of translations are the deterministic top-down tree transducers [Rou70], the MSO (graph) transducers [Cou94], or the restriction of the MSO transducers to trees [EM99]. Unfortunately, $\mathrm{XSLT}_0$ programs do not posses this property, as stated in the following theorem.

**Theorem 4.5** *XSLT$_0$ programs are not closed under composition.*

**Proof.** We have to show that there are $\mathrm{XSLT}_0$ programs $P_1$ and $P_2$ such that the composition of $P_1$ with $P_2$ cannot be realized by an $\mathrm{XSLT}_0$ program.

Let us first show that the height of an output tree generated by a terminating $\mathrm{XSLT}_0$ program is at most polynomial in the height of the input tree. Indeed, let $P = (\Sigma, \Delta, M, \mathrm{start}, R)$ be an $\mathrm{XSLT}_0$ program and let $t$ be a $\Sigma$-tree. Further, let $D$ be the number of different attribute values occurring in $t$ and issued as constants in $P$. Let $h$ be the maximum height of a forest in a constructing template rule. Let $n$ be the maximum number of parameters occurring in the head of a template rule. As the

number of local configurations is $N := |\text{Nodes}(t)| \times |Q| \times D^n$, this means that the height of the output tree is bounded by $h \times N$, which is polynomial in the size of $t$.

Consider the programs $P_1$ and $P_2$ of Example 3.11. The composition $P_1$ with $P_2$ transforms an input tree of height $n$ into a tree of height $2^n$. By the previous argument, no single $\text{XSLT}_0$ program can express this transformation. □

# 5 Conclusions

We presented a formal model for an expressive fragment of XSLT. The purpose of this paper was to show that XSLT is much more powerful than initially credited by the database community. Although the present paper shows that XSLT is a powerful transformation language, we are, however, rather hesitant to promote or support XSLT as the standard XML *query* language. Our main objection is that XSLT is much too procedural for a query language and therefore might be too difficult for the average user. On the other hand, as indicated by its widespread use, XSLT is highly adequate for the simple transformations it was intended for (recall that XSL was originally intended just for XML to HTML transformations). These simple XSLT programs are typical one pass transformations from the root to the leaves of the document. Performing joins and doing complicated grouping operations seems to require XSLT programs to traverse the input document many times in several directions, and therefore are more difficult to write, especially for users with little programming experience. As a final note we mention the following. It is common practice in the database community to restrict the power of a query language, in order to enhance the ability to optimize queries. In particular, the language XML-QL reflects this tradeoff (techniques to optimize queries in the traditional way are available) while the language XSLT does not (its implementations do not perform traditional query optimization, and there has been no proposal so far on how to perform such optimization for XSLT).

# Acknowledgements

We thank Bertram Ludäscher, Dan Suciu, and Victor Vianu for encouraging and helpful discussions. We thank one anonymous referee who's detailed comments greatly improved the presentation of the paper.

# References

[ABS00] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web : From Relations to Semistructured Data and XML.* Morgan Kaufmann, 2000.

[AQM+97] S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.

[BDHD96] P. Buneman, S. Davidson, G. G. Hillebrand, and D.Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, volume 25:2 of *SIGMOD Record*, pages 505–516. ACM Press, 1996.

[Bex00] G. J. Bex. Examples of translations from XML-QL to XSLT. http://www.luc.ac.be/~gjb/xml-ql2xslt.html, 2000.

[Cha01] D. Chamberlin et al. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery, Februari 2001.

[CFS01] D. Chamberlin, J. Robie, and D. Florescu. An XML query language for heterogeneous data sources. In D. Suciu, G. Vossen, editor, *The World Wide Web and Databases*, volume 1997 of *Lecture Notes in Computer Science*, pages 1–25. Springer 2001.

[Cla99a] J. Clark. XML Path Language (XPath). http://www.w3.org/TR/xpath, November 1999.

[Cla99b] James Clark. XSL Transformations (XSLT) Version 1.0. W3C Recommendation. http://www.w3.org/TR/xslt, November 1999.

[Con] World Wide Web Consortium. Extensible Markup Language (XML). http://www.w3.org/XML, .

[Cou94] B. Courcelle. Recognizable sets of graphs: equivalent definitions and closure properties. *Math. Struct. in Comp. Science*, 4:1–32, 1994.

[DFF+99a] A. Deutsch, M. fernandez, D. Florescu, A. Levy, D. Maier, and D. Suciu. Querying XML data. *Data Engineering Bulletin*, 22(3):10–18, 1999.

[EF95] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer, 1995.

[EM99] J. Engelfriet and S. Maneth. Macro tree transducers, attribute grammars, and MSO definable tree translations. *Inform. and Comput.*, 154:34–91, 1999.

[FFK+98] M. F. Fernandez, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. Catching the boat with strudel: Experiences with a web-site management system. In L. M. Haas and A. Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data*, pages 414–425. ACM Press, 1998.

[FSW99] M. Fernandez, J. Siméon, and P. Wadler, editors. *XML Query languages: Experiences and Exemplars*, 1999. `http://www-db.research.bell-labs.com/user/simeon/xquery.html`.

[FS73] P. Flajolet and J.-M. Steyaert. Decision Problems for Multihead Finite Automata. In *Proceedings of the second International conference on Mathematical Foundations of Computer Science*, pages 225–230, 1973.

[Imm98] N. Immerman. *Descriptive Complexity*. Springer, 1998.

[MN99] S. Maneth and F. Neven. Structured Document Transformations Based on XSL. In R. Conner, A. Mendelzon, editor, *Research Issues in Structured and Semistructured Database Programming*, volume 1949 of *Lecture Notes in Computer Science*, pages 80-98. Springer 2000.

[MSV00] T. Milo, D. Suciu, and V. Vianu. Type checking for XML transformers. In *Proceedings of the Nineteenth ACM Symposium on Principles of Database Systems*, pages 11–22. ACM Press, 2000.

[Nev99a] F. Neven. *Design and Analysis of Query Languages for Structured Documents — A Formal and Logical Approach.* Doctor's thesis, Limburgs Universitair Centrum (LUC), 1999.

[Nev99b] F. Neven. Extensions of attribute grammars for structured document queries. In R. Conner, A. Mendelzon, editor, *Research Issues in Structured and Semistructured Database Programming*, volume 1949 of *Lecture Notes in Computer Science*, pages 99-116. Springer 2000.

[NS99] F. Neven and T. Schwentick. Query automata. In *Proceedings of the Eighteenth ACM Symposium on Principles of Database Systems*, pages 205–214. ACM Press, 1999.

[NS00] F. Neven and T. Schwentick. Expressive and efficient pattern languages for tree-structured data. In *Proc. 19th Symposium on Principles of Database Systems (PODS 2000), Dallas*, pages 145–156, 2000.

[NVdB98] F. Neven and J. Van den Bussche. Expressiveness of structured document query languages based on attribute grammars. In *Proceedings of the Seventeenth ACM Symposium on Principles of Database Systems*, pages 11–17. ACM Press, 1998.

[PQ68] C. Pair and A. Quere. D'efinition et etude des bilanguages r'eguliers. *Inform. and Control*, 13:565–593, 1968.

[Rou70] W.C. Rounds. Mappings and grammars on trees. *Math. Systems Theory*, 4:257–287, 1970.

[Rob99] J. Robie. The design of XQL. `http://www.texcel.no/whitepapers/xql-design.html`, 1999.

[Tho97a] W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, chapter 7. Springer, 1997.

[Wad99] P. Wadler. A formal semantics of patterns in XSLT. Markup Technologies, 1999.