A Formal Model for an Expressive Fragment of XSLT

Geert Jan Bex[1], Sebastian Maneth[2*], and 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 aim of this paper is two-fold. First, we want to show that the recent extension of XSL with variables and passing of data values between template rules has increased its expressiveness beyond that of most other current XML query languages. Second, in an attempt to increase the understanding of this already wide-spread but not so transparent language, we provide an essential and powerful fragment with a formal syntax and a precise semantics.

1 Introduction

XSL [6] is a current W3C [8] proposal for an XML extensible stylesheet language. Its original primary role was to allow users to write transformations of XML to HTML, thus describing the presentation of XML documents. Nowadays, many people use XSL as their basic tool for XML to XML transformations which renders XSL into an XML query language. It has been noted by the database community [9, 1], though, that the transformations XSL can express are rather limited. For instance, XSL does not have joins or skolem-functions (and, hence cannot do sophisticated grouping of output data). In other words, XSL lacks the most basic property any query language should have: it is not relationally complete. However, as the language is still under development, some features have changed over time. Recently various extensions were added to the language [7]. The most apparent ones being the addition of *variables* and *parameter passing* between template rules. We show that these additions, together with the use of modes (which are actually states as used in finite state machines and which were already defined in earlier versions of XSL) render XSL into a powerful query language. Indeed, XSL 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 all other current XML query languages.

* The work of this author was supported by the EC TMR Network GETGRATS.

** Research Assistant of the Fund for Scientific Research, Flanders. Work partly performed while visiting the University of California San Diego. Contact author of this submission.

Actually together with the addition of the new features, XSL was split into two parts: XSL Transformations (XSLT) [7] and XPath [5]. The latter contains the description of XSL's associated pattern language, while the former defines the real transformation language. To emphasize that we are focusing on the transformation part, with the new features, we refer to XSL by XSLT in the rest of this paper.

The main source for the definition of XSLT is its specification [7] which is a bit difficult to read, especially when one only wants an impression of how the language works or what it is capable of. To remedy this, we define an abstract formal model of XSLT incorporating most of its features, but all of those which are necessary to simulate, say, XML-QL. The purpose of this model is two-fold:

(i) the clean and formal semantics provides the necessary mathematical model for studying properties of XSLT;
(ii) our formal model abstracts away from the actual syntax of XSLT and emphasizes on its features in such a way that the interested reader can get a feeling of the language and what it is capable of.

Next, we use this model to gain some insight in XSLT. First, we obtain that XSLT can compute all unary monadic second-order (MSO) structural properties. In brief, MSO is first-order logic (FO) extended with set quantification and is an expressive and versatile logic: on trees, for instance, it captures many robust formalisms, like regular tree languages [25], query automata [20], finite-valued attribute grammars [22, 18], ... Moreover, MSO can take the inherent order of children of vertices into account, a desirable property for XML pattern languages [13, 24]. By structural patterns we mean MSO without joins, that is, we cannot check whether the values of two attributes are the same (see Section 4 for details). In fact, Neven and Schwentick [19] showed that, already w.r.t. structural patterns, MSO is more expressive than FO extended with various kinds of regular path expressions. Thus, as most current XML query languages are based on FO extended with regular path expressions, this already indicates that XSLT cannot be simulated by, say, XML-QL.

Next, we show that XSLT can simulate most existing XML query languages. To study decidability of type checking, Milo, Suciu, and Vianu [16] defined the k-pebble tree-transducer as a formalism capturing the expressiveness of all existing XML query languages, including XML-QL [9], XQL [23], Lorel [2], StruQL [14], UnQL [4] and the previous version of XSL. Their model does not take value equation into account (needed for joins, for instance) but can easily be modified to do so. We obtain that XSLT can simulate this model, thereby establishing XSLT to be more expressive than all of the above. For more concrete simulations, we refer the interested reader to [3], were we show how the XML-QL queries in [9] can be expressed in actual XSLT.

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). The idea is that we want to use variables as a look-ahead or to fetch data values occurring 'far' from the current node. The resulting language is, hence, not Turing complete and, hopefully, more efficient to evaluate. The study of the properties of our formal model, however, is beyond the scope of the present paper. The most important fact is that the defined language is more expressive than, say, XML-QL, as opposed to the previous version of XSL [9, 1].[1]

The rest of the 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 our formal model. Finally, in Section 4 we obtain our expressibility results.

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). The computation on a document t starts at its root in the starting mode[2] and proceeds roughly as follows. When the computation arrives at a node, say u, in a certain mode, say q, the program tries to find a template rule with mode q and whose matching pattern matches u.[3] 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 selects lists of nodes for further processing (we refer to patterns that select nodes for further processing as *selecting patterns*). Each of these selected nodes are then processed independently as before. Finally, the documents that are constructed by these subprocesses are inserted in at the positions where the subprocesses were generated.

To illustrate the new features of XSLT we use the DTD in Figure 1. It describes an organization as a sequence of groups together with a list of top managers. Each group has an id, consists of a manager and a list of other groups, or just consists of a list of employees. 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`.

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 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 at the root in mode start. This rule selects each top manager (in mode `selecttopmgr`). In particular, the pattern `/organization/topmgr/employee` is matched against the current node which

[1] In previous work we defined a formal model for the version of XSL not incorporating data values [15].

[2] Actually, modes are optional, but for convenience we assume every template has a one and that there is a start mode.

[3] 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.

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

Fig. 1. A DTD describing an organization.

is labeled by `organization` and then selects all `employee` children of all `topmgr` children (the symbol / means 'child of'). For each selected employee (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 symbol // means 'descendant of', the expression between the brackets '[...]' is a filter on group elements). In this latter selection the XSLT program passes the ID e_1 along as a parameter. Next, for each employee e_2 selected by the latter selection, the program outputs an element `pair` with attribute values e_1 and e_2 for the attributes `topmgrID` and `employeeID`, respectively.

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

(*i*) Modes enables 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`, this element was selected in.

(*ii*) Variables can be used for two purposes. The most apparent one, which is illustrated by the above query, is that they allow 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 a binary tree, representing a Boolean circuit, to its truth value. Essentially, the use of variables allows for a bottom-up computation. The restriction to binary trees is just for expository purposes. In fact, it can be shown that XSLT can evaluate any bottom-up tree automaton over unranked trees [17]. 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.

(*iii*) Passing of data values to other template rules can be crucial for performing joins if the items that have to be joined are 'far' apart. Moreover, when

```
<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>
```

Fig. 2. An XSLT program computing the query of Section 2.

```
<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>
```

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

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

Fig. 4. The output of the XSLT program of Figure 2 on the document of Figure 3.

node IDs are present in the XML document,[4] we can use this mechanism to place ‘pebbles’ on the input document which enables us to do complicated grouping operations.

It are exactly these three features which 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>
```

Fig. 5. The fragment of an XSLT program evaluating tree-structured Boolean circuits that takes care of or-nodes.

In the next section, we give an abstract formal syntax for XSLT. First of all, we restrict matching patterns to test only the label of the current node (as is already the case in Figure 2). This is no restriction, as Theorem 8 shows that we can test many properties of the current node in the body of the template rule. Further, we divide a template rule into two parts: the *variable definition part* and the *construction part*. Variables can only be assigned data values. In particular, a variable can be defined as the value of some attribute of the current node or by an XSLT apply-templates statement that will return exactly one data value. We will refer to such special templates as *selection template rules*. In the construction part of the template rule, the actual output is defined relative to some conditions on the values of the variables, the parameters, and the attribute values of the current node, and possibly whether the current node is the root, a leaf, or the first or last child of its parent.

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

3 A formal Model for XSLT

3.1 Trees and forests

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

The set of Σ-forests, denoted by $\mathcal{F}_\Sigma$, is inductively defined as follows: every $\sigma \in \Sigma$ is a Σ-forest; if $\sigma \in \Sigma$ and $f \in \mathcal{F}_\Sigma$ then $\sigma(f)$ is a Σ-forest; if $f_1, \ldots, f_n \in \mathcal{F}_\Sigma$ then $f_1 \cdots f_n$ is a Σ-forest. A *Σ-tree* is a Σ-forest of the form $\sigma(f)$. We denote the set of all Σ-trees by $\mathcal{T}_\Sigma$. Note that there is no a priori bound on the number of children of a node in a forest. In the following, whenever we say tree or forest, we always mean Σ-tree and Σ-forest.

The reason we consider forests is that even when we use XSL for tree to tree transformations only, we sometimes need to specify template rules that construct forests.

For every forest $f \in \mathcal{F}_\Sigma$, the *set of nodes of* f, denoted by 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 $\text{Nodes}(f) = \{\varepsilon\} \cup \{iu \mid i \in \{1, \ldots, n\}, u \in \text{Nodes}(t_i)\}$. If $f = t_1 \cdots t_n$, then $\text{Nodes}(f) = \{iu \mid i \in \{1, \ldots, n\}, u \in \text{Nodes}(t_i)\}$. Thus, for a tree the node ε 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.

Next, we add XML attributes to the above defined attributes. To this end, for the rest of the paper, we fix a finite set of attributes A. An *attributed forest with domain* S is a pair $(f, (\lambda_a^f)_{a \in A})$ with $f \in \mathcal{F}_\Sigma$ and where for each $a \in A$, $\lambda_a^f : \text{Nodes}(f) \mapsto S$ is a function assigning a value in S to each node of f. The set of all attributed forests with domain S, is denoted by $\mathcal{F}_\Sigma^S$. 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 refer to those defined by 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 convenient, not a necessary restriction. In an analogous way one can define the set of attributed trees, denoted by $\mathcal{T}_\Sigma^{\mathbf{D}}$. For a set B, $\mathcal{F}_\Sigma^{\mathbf{D}}(B)$ denotes the set of attributed forests f over $\Sigma \cup B$ such that symbols of B may only appear at the leaves of f.

In our formal model we abstract away from a particular selection pattern language. Recall that XSLT uses the pattern language described in XPath [5] (see [26], 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 like select parent, left sibling, right sibling, or first child. Actually, the proof of Theorem 9 indicates that such local selections only are enough to simulate all 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

$(\mathcal{T}_{\Sigma}^{\mathbf{D}} \times (X \mapsto \mathbf{D})) \mapsto (\mathbb{N}^* \mapsto 2^{\mathbb{N}^*})$ 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 be a tree, and γ be 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.

3.2 Syntax

Definition 1. An *XSLT program* is a tuple $P = (\Sigma, \Delta, M^c, M^s, \text{start}, R)$, where

- Σ is an alphabet of *input symbols*;
- Δ is an alphabet of *output symbols*;
- M^c and M^s are finite sets of *construction* and *selection modes*;
- start $\in M^c$ is the *start* mode; and
- R is a finite set of *construction* and *selection template rules* (to be defined below).

As mentioned at the end of section 2, we distinguish between two types of template rules: *constructing* and *selecting* ones. The former are used to create output. So, the result of applying these is a forest. The latter are used to fetch data values. So, each one returns exactly one domain element. The mode will determine the nature of the template: constructing or selecting.

Definition 2. A *construction apply-templates-expression* r (at-expression for short) is of the form $q(p, \bar{z})$, where $q \in M^c$, p is a pattern and $\bar{z}$ is a possibly empty sequence of variables in $\mathcal{X}$ and domain elements in $\mathbf{D}$. A *selection at-expression* is defined as a constructing one with the restriction that p only selects single nodes, that is, for every tree t, each assignment of variables γ, and each node u of t, $p(t, \gamma)(u)$ is a singleton set. We denote the set of construction (selection) at-expression by $\mathcal{AT}^{c}$ ($\mathcal{AT}^{s}$).

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

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

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 will produce output.

Definition 3. An *attribute expression* is an expression of the form $a(.)$ where a is an attribute. An *atomic test* is one of the following: (i) an expression of the form $x = y$ where x and y are attribute expressions, variables, or domain elements; or, (ii) an expression of the form root, leaf, first-child, or last-child. Finally, a *test* is a Boolean combination of atomic tests.

During a computation the expressions $a(.)$ will evaluate to the value of the attribute a of the current node. Further, root, leaf, first-child, last-child evaluate to true whenever the current node is the root, a leaf, the first or the last child of its parent. Selection template rules are defined next. Recall that they output one domain element.

Definition 4. *Let $q \in M^c$ and $\sigma \in \Sigma$. A (q, σ)-selection template rule* is of the form

template $\sigma(q, x_1, \ldots, x_n)$
    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 $n, m \geq 0$, $k \geq 1$; and,

– all x's and y's are variables (the former are parameters while the latter are local variables);
– each $r_i \in \mathcal{AT}^s$ or is an attribute expression; further, if $r_i \in \mathcal{AT}^s$ then every variable occurring in it is among $y_1, \ldots, y_{i-1}, x_1, \ldots, x_n$;
– all c_i are tests containing only variables in $X := \{x_1, \ldots, x_n, y_1, \ldots, y_m\}$; and
– every z_i is a domain element, a variable in X, or a selecting at-expression with the restriction that all variables occurring in it should belong to X.

Definition 5. Let $q \in M^c$ and $\sigma \in \Sigma$. A (q, σ)-*construction template rule* is of the same form as a selection rule only now each z_i is a forest in $\mathcal{F}_\Sigma^{\mathbf{D} \cup X}(\mathcal{AT}^c)$ (recall that these are forests where attributes take values in $D \cup \mathcal{X}$ and where leaves may be labeled with constructing at-expression) with the restriction that each variable occurring in an at-expression occurring at a leaf of z_i should be in X.

To keep the model total and deterministic we require the existence of exactly one (q, σ)-template rule for each mode and each σ. Further, to ensure that an XSLT program generates tree to tree translations, we require that each z_i in a (start, σ)-construction template rule is a tree (rather than a forest).

Example 6. We illustrate the above 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 rule, respectively. In the second template rule display(p_2,varID) is the tree consisting of one node labeled with display(p_2,varID); further, ε denotes the empty tree. In the last rule `pair`[`topmgr`$\to$varID; `employeeID`$\to$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.

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

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

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

3.3 Semantics

To define the semantics we need the following. 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 $\bar{x} \backslash \bar{d}$ we denote the function that maps each x_i to d_i.

We next define the semantics of an XSLT program P on a tree t. Thereto, we need the following concept. A *local configuration* is an element of Nodes$(t) \times (M^c \cup M^s) \times \mathbf{D}^*$. Intuitively, $\theta := (u, q, d_1, \ldots, d_n)$ means that the program has selected node u in mode q with values $\bar{d}$ as parameters. For ease of presentation, we define the result of P on θ, denoted by $P^t(\theta)$, in a direct and procedural way. The latter has the advantage over the usual definition, in terms of rather complicated but formally more correct rewrite relations, that it is more transparent. The drawback is that it does not deal with the border case when XSLT programs get into infinite cycles. However, it should be clear that $P^t(\theta)$ is undefined whenever one of the generated subprocesses computes forever. We defer the formal semantics in terms of rewrite relations to the full version of the paper. We distinguish between two cases. In both of these, let the label of u be σ.

- Suppose q is a selection mode. Then $P^t(\theta) \in \mathbf{D}$. Let the (q, σ)-template rule be of the form as specified in Definition 4, where each r_i is $q_i(p_i, \bar{x}'_i)$ or the attribute expression $a_i(.)$.
  Intuitively, this template is evaluated as follows. First, the values of the variables $y_1, \ldots, y_n$, 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 output then is determined by z_i where c_i is the first test that evaluates to true.

Suppose the variables $y_1, \ldots, y_m$ get assigned the domain values $e_1, \ldots, e_m$, respectively. That is,

- if r_i is an at-expression, then $e_i := P^t(v_i, q_i, \bar{x}'_i[\gamma_i])$, where γ_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 $e_i := \lambda^t_{a_i}(u)$.

Next, suppose 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 iff u is the root, a leaf, the first or the last child of its parent, respectively. To ensure that the translation is total, we require that at least one such c_i exists.[5] Then, z_i determines the output value in the following way. If z_i is a constant, a variable, or an attribute expression then $P^t(\theta)$ equals the corresponding value. If z_i is a selecting at-expression $q'(p, \bar{w})$, then $P^t(\theta) := P^t(v, q', \bar{w}[\bar{x}\backslash\bar{d}, \bar{y}\backslash\bar{e}])$ where v is the node selected by p, that is, $p(t, [\bar{x}\backslash\bar{d}, \bar{y}\backslash\bar{e}])(u) := \{v\}$.

– Suppose q is a construction mode. Then $P^t(\theta) \in \mathcal{F}^{\mathbf{D}}_{\Sigma}$. Let the (q, σ)-template rule be of the form as specified in Definition 5. Suppose the variables $y_1, \ldots, y_m$ get assigned the values $e_1, \ldots, e_m$, as described above, and c_i is the first condition that evaluates to true. Then z_i determines the output value in the following way. Recall that z_i is forest in $\mathcal{F}^{\mathbf{D}\cup\mathcal{X}}_{\Sigma}(\mathcal{AT}^c)$, that is, a forest where attributes take values in $\mathbf{D} \cup \mathcal{X}$ and where leaves may be labeled with constructing at-expression. Now, $P^t(\theta)$ 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

$$P^t(u_1, q', \bar{w}[\bar{x}\backslash\bar{d}, \bar{y}\backslash\bar{e}]) \cdots P^t(u_\ell, q', \bar{w}[\bar{x}\backslash\bar{d}, \bar{y}\backslash\bar{e}]),$$

    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 document order). Recall that each $P^t(u_i, q', \bar{w}[\bar{x}\backslash\bar{d}, \bar{y}\backslash\bar{e}])$ returns an attributed forest.

The *initial* local configuration is defined as $\theta_{\text{start}} := (\varepsilon, \text{start})$.

Definition 7. *The* result *of an XSLT program P on a tree t, denoted by $P(t)$, is defined as $P^t(\theta_{start})$.*

3.4 Some remarks

We conclude this section with some remarks. First, we note that XSLT does not make the explicit distinction between constructing and selecting template rules,

[5] Obviously, one could also add an ‘otherwise’ construct rather than having this condition.

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 show in the next section that we have captured a powerful fragment capable of simulating most existing XML query languages and even more.

4 Expressiveness

We next show that XSLT is capable of computing very expressive structural patterns. Thereto, we first say how we view attributed trees as logical structures (in the sense of mathematical logic [10]) over the binary relation symbols E and $<$, and the unary relation symbols $(O_\sigma)_{\sigma\in\Sigma}$. The domain of t, viewed as a structure, equals the set of nodes of t, i.e., $\mathrm{Nodes}(t)$. E is the edge relation and equals the set of pairs $(v, v\cdot i)$ for every $v, v\cdot i \in \mathrm{Nodes}(t)$. The relation $<$ specifies the ordering of the children of a node, and equals the set of pairs $(v\cdot i, v\cdot j)$, where $i<j$ and $v\cdot j \in \mathrm{Nodes}(t)$. For each σ, O_σ is the set of nodes that are labeled with a σ. The logic MSO* is MSO over the above vocabulary (with MSO defined in the usual way, see, e.g., [10]) extended with atomic formulas of the form $a(x)=d$, where a is an attribute and $d \in \mathbf{D}$. Denote the latter atomic formula by φ. Its semantics is then defined as follows $t \models \varphi[u]$ iff $\lambda_a^t(u) = d$, that is, the attribute a of u has value d. Note that we do not allow atomic formulas of the form $a(x) = b(y)$, so we do not allow joins.

Clearly, MSO* can define all XPath matching patterns (they are even FO* definable). The next theorem says that XSLT is capable of expressing all unary MSO* patterns. In particular, this means that one does not need matching patterns in templates. That is, XSLT actually allows to specify rules like

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

where p is a pattern rather than just a label. It means that a rule can only be applied on nodes that satisfy p. The next theorem implies that one can test for p in the body of the template rule and, hence, does not need matching patterns.

Theorem 8. *Let $\varphi(x)$ be an MSO* formula. There exists an XSLT program P and a mode q_φ such that $P^t(u, q_\varphi) = \mathbf{true}$ iff $t \models \varphi[u]$.*

Proof. Due to space limitations we can only give a brief hint of the proof. We refer the interested reader to [17]. We start with some terminology. The subtree of t rooted at v is denoted by t_v. The *envelope of t at v*, denoted by $\overline{t_v}$, is the tree obtained from t by deleting the subtree rooted at v but keeping v. Note that t_v and $\overline{t_v}$ have v in common. Let the quantifier-depth of φ be k and denote by $\tau_k(t,v)$ all the unary MSO* formulas $\varphi(x)$ up to quantifier-depth k that hold in t with x interpreted by v. It is well known that $\tau_k(t,v)$ is finite for all t and v (for fixed k). Further, whether $t \models \varphi[v]$ depends only on $\tau_k(\overline{t_v}, v)$ and $\tau_k(t_v, v)$.[6]

[6] This would not be the case if atomic formulas of the form $a(x) = b(y)$ were allowed.

Moreover, $\tau_k(\overline{t_v}, v)$ and $\tau_k(t_v, v)$ can be computed by unranked tree automata working on $(\overline{t_v}, v)$ and t_v, respectively [20]. Here, $(\overline{t_v}, v)$ is the tree $\overline{t_v}$ with v as a distinguished node. Hence, it remains to show how XSLT programs can compute bottom-up unranked tree automata. In brief, this can be done by using variables as look-ahead like the XSLT program in Figure 5.

To study decidability of type checking, Milo, Suciu, and Vianu [16] 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.[7] We next 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: move-to-parent, move-to-first-child, move-to-last-child, move-to-left-sibling, move-to-right-sibling, remain-and-change-state, place-new-pebble, and pick-current-pebble. There are two kinds of output transitions. A binary output outputs a Σ-symbol σ, possibly with attributes defined as 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 σ. Both branches inherit the positions of all pebbles on the input and do not communicate with each other, that is, each branch moves the pebbles independently of the other. In a nullary output, the node being output is a leaf of the output tree, again possibly with attributes, and the computation halts.

It should be clear that, apart from the pebbles, the above described model is extremely close to XSLT: XSLT is equiped with modes (states), can do local movements and the simple output transitions. Under the assumption that each node has a unique id, XSLT can also simulate pebbles.[8] 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 which implies that XSLT can simulate most other current XML query languages, like for instance, XML-QL [16] . We refer the interested

[7] When proving properties of XML transformations, restricting to binary trees is usually sufficient as unranked ones can be encoded into ranked ones; of course, this is not the case when one tries to define a formal model for XSLT which works directly on unranked trees.

[8] Actually, this assumption is not necessary as XSLT is equiped 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. So there is no need to store the node id's; they can be computed on demand.

reader to [3] were we show how the XML-QL queries in [9] can be expressed in actual XSLT.

Theorem 9. *XSLT can simulate k-pebble deterministic tree-transducers with equality tests on data values.*

We point out that non-deterministic tree-transducers can be simulated by giving a non-deterministic semantics to XSLT in the obvious way.

Although by the above discussion k-pebble transducers seem extremely close to our formal model of XSLT, it is not clear whether the former can simulate the latter. Indeed, Theorem 8 shows that XSLT can check any MSO property of the root of the tree. We do not know whether deterministic k-pebble transducers can do this. In fact, this problem is closely related to a seemingly difficult open problem in formal language theory: can tree-walking automata (with or without pebbles) define all MSO definable tree languages [11, 12, 21]. On the other hand, Theorem 8 does hold for *non-deterministic* k-pebble transducers. Furthermore, it is not clear whether all deterministic transformations they can express can be simulated by XSLT. Regardless of the above, it is known that query languages based on first-order logic extended with regular path expressions, like XML-QL, are weaker than MSO [19], hence, by Theorem 8 and Theorem 9, weaker than XSLT.

Acknowledgements

The last author thanks Bertram Ludäscher and Victor Vianu for encouraging and helpful discussions.

References

1. S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web : From Relations to Semistructured Data and XML*. Morgan Kaufmann, 1999.
2. 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.
3. G. J. Bex. Examples of translations from XML-QL to XSLT. http://www.luc.ac.be/~gjb/xml-ql2xslt.html.
4. 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.
5. J. Clark. XML Path Language (XPath). http://www.w3.org/TR/xpath.
6. J. Clark and S. Deach. Extensible stylesheet language (XSL). http://www.w3.org/TR/WD-xsl.
7. James Clark. XSL transformations version 1.0. http://www.w3.org/TR/WD-xslt, august 1999.
8. World Wide Web Consortium. Extensible Markup Language (XML). http://www.w3.org/XML/.

9. A. Deutsch, M. fernandez, D. Florescu, A. Levy, D. Maier, and D. Suciu. Querying XML data. *Data Engineering Bulletin*, 22(3):10–18, 1999.
10. H.-D. Ebbinghaus and J. Flum. *Finite Model Theory.* Springer, 1995.
11. J. Engelfriet and H. J. Hoogeboom. Tree-walking pebble automata. In J. Karhumäki, H. Maurer, G. Paun, and G.Rozenberg, editors, *Jewels are forever, contributions to Theoretical Computer Science in honor of Arto Salomaa*, pages 72–83. Springer-Verlag, 1999.
12. J. Engelfriet, H.J. Hoogeboom, and J.-P. van Best. Trips on trees. *Acta Cybernetica*, 14:51–64, 1999.
13. M. Fernandez, J. Siméon, and P. Wadler. XML query languages: Experiences and exemplars. http://www-db.research.bell-labs.com/user/simeon/xquery.html, 1999.
14. 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.
15. S. Maneth and F. Neven. Structured document transformations based on XSL. To appear in the proceedings of the Seventh International Workshop on Database Programming Languages, *Lecture Notes in Computer Science*, 1999.
16. T. Milo, D. Suciu, and V. Vianu. Type checking for XML transformers. To appear in PODS'2000.
17. F. Neven. *Design and Analysis of Query Languages for Structured Documents — A Formal and Logical Approach.* Doctor's thesis, Limburgs Universitair Centrum (LUC), 1999.
18. F. Neven. Extensions of attribute grammars for structured document queries. To appear in the proceedings of the Seventh International Workshop on Database Programming Languages, *Lecture Notes in Computer Science*, 1999.
19. F. Neven and T. Schwentick. Expressive and efficient pattern languages for tree-structured data. To appear in PODS'2000.
20. 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.
21. F. Neven and T. Schwentick. On the power of tree-walking-automata. Submitted, 2000.
22. 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.
23. J. Robie. The design of XQL. http://www.texcel.no/whitepapers/xql-design.html, 1999.
24. D. Suciu. Semistructured data and XML. In *Proceedings of the 5th International Conference on Foundations of Data Organization and Algorithms*, 1998.
25. W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, chapter 7. Springer, 1997.
26. P. Wadler. A formal semantics of patterns in XSLT. Markup Technologies, 1999.