An Implementation Scheme for XML Transformation Languages through Derivation of Stream Processors

Keisuke Nakano

Department of Mathematical Engineering and Information Physics,
University of Tokyo
ksk@mist.i.u-tokyo.ac.jp

Abstract. We propose a new implementation scheme for XML transformation languages through derivation of stream processors. Most of XML transformation languages are implemented as tree manipulation, where input XML trees are completely stored in memory. It leads to inefficient memory usage in particular when we apply a facile transformation to large-sized inputs. In contrast, XML stream processing can minimize memory usage and execution time since it begins to output the transformation result before reading the whole input. However, it is much harder to program XML stream processors than to specify tree manipulations because stream processing frequently requires 'stateful programming'. This paper proposes an implementation scheme for XML transformation languages, in which we can define an XML transformation as tree manipulation and also we can obtain an XML stream processor automatically. The implementation scheme employs a framework of a composition of attribute grammars.

1 Introduction

In recent years, various languages specialized for XML transformation have been proposed[25, 28, 10, 3]. Since XML documents have tree structure, these languages support various functions of pattern matching for paths in order to access particular nodes. These node accessing methods are generally implemented as *tree manipulation* that requires the whole tree structure of an input XML document to be stored in memory. This implementation might be inefficient in memory usage in particular when a facile transformation, such as tag renaming and element filtering, is applied to large-sized XML documents because it does not require all information about the tree structure.

XML stream processing has been employed as one of solutions to reduce memory usage[22, 5]. XML stream processing completes a transformation by storing no tree structure of XML documents in memory. While taking advantage in memory usage, XML stream processing has a problem that programs are quite complicated because XML stream processing is defined by 'stateful program' in the sense that programmers need to consider what to memorize on reading a start tag, what to output on reading an end tag, and so on. It imposes a burden on programmers and causes error-prone unreadable programs.

To release programmers from the bother of stream processing, two kinds of approaches have been proposed. The first approach is to add various primitive combinators or functions for stream processing[11, 27]. Though it helps us to make stream

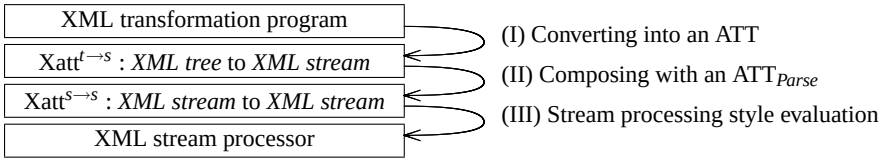


Fig. 1. Derivation of stream processing programs

processing programs, we still need to write programs with the prospect that the input XML is not a tree but a stream. The second approach is to give a mechanism deriving a stream processing program from a specification of a tree manipulation or node accessing[1, 17, 19, 4, 15]. We can write the program in tree manipulation style and need not to bear in mind that the input XML is a stream. However, the derivation mechanism can be applied to only a restricted class of XML transformations.

This paper basically takes the second approach, provided that the derivation method can deal with a wider class of XML transformations than before. Our derivation mechanism is based on a framework of *attributed tree transducer*(ATT)[6] which is a formal computational model for *attribute grammar*(AG)[12]. The mechanism is summarized by dividing it into three steps (Fig.1). In the first step (I), a given XML transformation is converted into an ATT specialized for XML transformation, called Xatt$^{t\to s}$. The Xatt$^{t\to s}$ represents a transformation from XML trees to XML streams. This paper roughly shows how to convert an XML transformation into Xatt$^{t\to s}$ through two kinds of practical XML transformation: XPath querying[24] and partial modification. The former returns the collection of all elements specified by a given XPath expression. It is useful in the case that the input XML is regarded as a database. The latter returns an XML with a similar structure to the input XML where several fragments in the input are replaced with different ones. We can write a variety of XML transformations by combination of these basic transformations, though they are only a subset of the class of XML transformation specified by Xatt$^{t\to s}$. However, it does not show the limitation of this approach. This paper claims that we can derive an XML stream processor from every XML transformation program defined by Xatt$^{t\to s}$ which can deal with a wide class of XML transformation as well as non-circular AG. If one wants to implement an XML transformation language intended for stream processing, all he has to do is to give a conversion method into Xatt$^{t\to s}$ for the language. The stream processor is derived from an Xatt$^{t\to s}$ by the following two steps.

In the second step (II), we can obtain an Xatt$^{s\to s}$, which represents a transformation from XML streams to XML streams, from an Xatt$^{t\to s}$. We employ a *descriptional composition*[8], which is a composition method for AGs. If two AGs are given, where one represents a transformation from a language L_A to a language L_B and another represents a transformation from a language L_B to a language L_C, then we can obtain a single AG from L_A to L_C by descriptional composition. The obtained AG does not create any expressions in L_B even as intermediate result. The method can also be applied to ATT[14, 16]. In our framework, we use this method to compose a given Xatt$^{t\to s}$ with the other ATT representing a parsing function for XML and obtain an Xatt$^{s\to s}$. When more appropriate, the original descriptional composition cannot be applied to the pars-

ing ATT because of the restriction of the method. For this reason, we use an extended descriptional composition[16] which can deal with ATTs with stack devices required in the parsing ATT.

In the third step (III), from the $\mathrm{Xatt}^{s\to s}$ derived in the previous step, we can obtain an XML stream processor by specifying evaluation order for attributes in stream processing style. The XML stream processor can be derived by dividing the computation by $\mathrm{Xatt}^{s\to s}$ into the computation for each input symbol. This generation method is similar to Nakano and Nishimura's method in [17, 19]. However, they failed to deal with a large class of ATTs because they sticked to derivation of finite state transition machines with dependency analysis.

This paper also shows benchmark results to illustrate effectiveness of our framework. The author has implemented an XML transformation language XTiSP (an abbreviation for XML Transformation language intended for Stream Processing) based on our framework. Using XTiSP, we can program an XML transformation as tree manipulation with XPath and obtain the corresponding XML stream processor for free. We compare an implementation of XTiSP with two XSLT processors, Xalan[26] and SAXON[23].

Related work

The issue of automatic generation of stream processors has been studied for various languages. Most of these studies, however, targeted for only query languages such as XPath[1, 4, 9] and a subset of XQuery[15]. These querying languages have few expressibility to specify XML transformation. For example, they could not define the structure-preserved transformation, such as renaming a tag name a into b. Our framework allows not only querying functions but also ability to specify such transformation.

Our framework is based on the previous work[17, 19]. They succeeds in deriving a stream processing program from the specification of XML tree transformation defined by an ATT. However, in their framework, a set of ATTs we can deal with and input XMLs for them are quite restricted because of the weak applicability of AG composition method[8]. Our framework solves this problem by using the extended composition method[16].

Additionally, there are several work using a framework of attribute grammars for XML transformation. Whereas we use binary representation for XML trees, [18, 2] improve a traditional attribute grammar for unranked representation. However, their attribute grammars do not have a framework of descriptional composition we require. Although [13] employs an attribute grammar for XML stream processing, they uses only L-attribute grammars. In our framework, attribute grammars for XML stream processing are non-circular, which is wider class than L-attribute, and they are automatically derived from tree manipulation programs.

Outline

The rest of this paper is comprised of seven sections, including introduction. Section 2 gives basic notations and a brief introduction of attributed tree transducers. In Section 3, we show how to specify XML transformation by using $\mathrm{Xatt}^{t\to s}$. Section 4 presents a

composition method of Xatt$^{t\to s}$ and an XML parser. In Section 5, we mention how to obtain an XML stream processor from Xatt$^{s\to s}$. Then Section 6 shows several benchmark results to illustrate effectiveness of our framework. Finally we conclude this paper in Section 7.

2 Preliminaries

2.1 Basic Notions

The empty set is denoted by $\emptyset$. The set of natural numbers including 0 by $\mathbb{N}$ and the set of natural numbers excluding 0 by $\mathbb{N}_+$. For every $n \in \mathbb{N}$, the set $\{1,\ldots,n\}$ is denoted by $[n]$. In particular, $[0] = \emptyset$. We denote a set of finite strings over a set P of symbols by P^*. A null string is denoted by ε.

A *ranked alphabet* Σ is a finite set in which every symbol is associated with a non-negative integer called *rank*. We denote the rank of a symbol σ by $rank(\sigma)$. We may write $\sigma^{(n)}$ to indicate that $rank(\sigma) = n$. Let Σ be a ranked alphabet and A be a set of variables disjoint with Σ. The set of Σ-*labeled trees indexed by A*, denoted by $T_\Sigma(A)$ (or T_Σ, if A is empty), is the smallest superset T of A such that $\sigma(t_1,\cdots,t_n) \in T$ for every $\sigma^{(n)} \in \Sigma$ and $t_1,\cdots,t_n \in T$.

We denote by $t[x := s]$ the *substitution* of occurrences of a variable x by s. Let $t, s_1,\ldots,s_n,u_1,\ldots,u_n$ be trees in $T_\Sigma(X)$ such that every u_i for $i \in [n]_+$ is a subtree of t, provided that u_i is not a subtree of u_j for any i and j with $i \neq j$. The tree $t[u_1,\ldots,u_n := s_1,\ldots,s_n]$, or $t[u_i := s_i]_{i\in[n]_+}$ is obtained from t by simultaneously *replacing* every subtree at the points of occurrences of $u_1,\ldots,u_n$ by the trees $s_1,\ldots,s_n$, respectively. If $\rho = [u_1,\ldots := s_1,\ldots]$, then we may write $\rho(t)$ for $t[u_1,\ldots := s_1,\ldots]$.

The prefix-closed set of all *paths* of t, denoted by $path(t)(\subseteq \mathbb{N}_+^*)$, is defined by $path(\sigma(t_1,\cdots,t_k)) = \{\varepsilon\} \cup \{iw \mid i \in [k], w \in path(t_i)\}$ if $\sigma^{(k)} \in \Sigma$. We write $t|_w$ for a subtree of a tree t at a path $w \in path(t)$. Every path $w \in path(t)$ refers to a corresponding label of t, denoted by $label(t,w)$, which is defined by $label(\sigma(t_1,\cdots,t_n),\varepsilon) = \sigma$ and $label(\sigma(t_1,\cdots,t_n),iw) = label(t_i,w)$ for every $i \in [n]$ and $w \in path(t_i)$.

A *reduction system* is a system $(A,\Rightarrow)$ where A is a set and $\Rightarrow$ is a binary relation over A. We write $a_1 \Rightarrow^n a_{n+1}$ if $a_i \Rightarrow a_{i+1}(i \in [n])$ for some $a_1,\cdots,a_{n+1} \in A$. In particular, $a \Rightarrow^0 a$. $a \in A$ is *irreducible* with respect to $\Rightarrow$ if there is no $c \in A$ such that $b \Rightarrow c(\neq b)$. If b is irreducible where $a \Rightarrow^i b$ for some $i \in \mathbb{N}$ and there is no pair of irreducible term $b'(\neq b)$ and $i' \in \mathbb{N}$ such that $a \Rightarrow^{i'} b'$, we say b is a *normal form* of a and write $nf(\Rightarrow, a)$ for b.

2.2 Attributed Tree Transducers

We give a brief introduction of *attributed tree transducer*(ATT) which has been introduced by Fülöp[6] as a formal computational model of attribute grammar(AG). See [7] for detail. The ATT M is defined by a tuple $M = (Syn, Inh, \Sigma, \Delta, s_{in}, \sharp, R)$, where

- Syn is a set of *synthesized attributes*. Inh is a set of *inherited attributes*. We denote $\{s(\pi k) \mid s \in Syn, k \in [n]\}$ and $\{i(\pi) \mid i \in Inh\}$ by $\Pi_{syn}(n)$ and Π_{inh}, respectively.
- Σ and Δ are ranked alphabets, called the *input and output alphabet*, respectively.

– s_{in} is the initial (synthesized) attribute and $\sharp$ is the root symbol with rank 1. These are used for specifying the computation result.
– R is a set of attribute rules such that $R = \cup_{\sigma \in \Sigma \uplus \{\sharp\}} R^\sigma$ with finite sets R^σ of σ-rules satisfying the following conditions:
  - For every $s \in Syn$ and $\sigma \in \Sigma$, R^σ contains exactly one attribute rule of the form of $s(\pi) \to \eta$ where $\eta \in T_\Sigma(\Pi_{syn}(rank(\sigma)) \cup \Pi_{inh})$.
  - For every $i \in Inh$ and $\sigma \in \Sigma$, R^σ contains exactly one attribute rule of the form of $i(\pi k) \to \eta$ where $k \in [rank(\sigma)]$ and $\eta \in T_\Sigma(\Pi_{syn}(rank(\sigma)) \cup \Pi_{inh})$.
  - $R^\sharp$ contains exactly one attribute rule of the form of $s_{in}(\pi) \to \eta$ where $\eta \in T_\Sigma(\Pi_{syn}(1))$.
  - For every $i \in Inh$, $R^\sharp$ contains exactly one attribute rule of the form of $i(\pi 1) \to \eta$ where $\eta \in T_\Sigma(\Pi_{syn}(1))$.

where we use $\pi, \pi 1, \pi 2, \cdots$ for *path variables*.

The computation by M for input tree $t \in T_\Sigma$ is defined by a reduction system $(U, \Rightarrow_{M,\sharp(t)})$ where $U = T_\Delta(\{a(w) \mid a \in Syn \cup Inh, w \in path(t)\})$ and $\Rightarrow_{M,t}$ is defined by

– $s(w) \Rightarrow_{M,t} \eta[\pi := w]$ where $s \in Syn$, $s(\pi) \to \eta \in R^\sigma$ and $\sigma = label(t,w)$.
– $i(wk) \Rightarrow_{M,t} \eta[\pi := w]$ where $i \in Inh$, $s(\pi k) \to \eta \in R^\sigma$ and $\sigma = label(t,w)$.
– $\delta(\ldots, \eta_k, \ldots) \Rightarrow_{M,t} \delta(\ldots, \eta'_k, \ldots)$ where $\delta \in \Delta$ and $\eta_k \Rightarrow_{M,t} \eta'_k$.

where $[\pi := w]$ with $w \in \mathbb{N}^*$ stands for a substitution $[i(\pi), s(\pi 1), s(\pi 2), \cdots := i(w), s(w1), s(w2), \cdots]_{i \in Inh, s \in Syn}$. The *transformation result by M for the input tree t* is defined by $nf(\Rightarrow_{M,\sharp(t)}, s_{in}(\varepsilon))(\in T_\Delta)$.

3 Attributed Tree Transducers for XML transformation

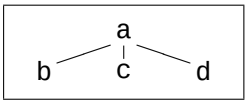
Our framework requires XML transformations to be defined by specialized ATTs in order to obtain the corresponding XML stream processor. In this section, we first introduce a model of XML trees and XML streams as annotated trees. Next we extend ATTs to ones that deal with annotated trees, called *attributed tree transducers for XML*(Xatt$^{t \to s}$). Then we show several examples of Xatt$^{t \to s}$ which represent basic XML transformations.

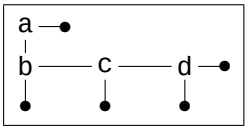
For simplicity, we deal with XML documents with no character data and no attribute. Our framework can be easily extended with them and the actual implementation supports them. In the rest of paper, a bare word 'attribute' is not used for XML attributes but for synthesized/inherited attributes in ATTs.

3.1 XML Trees and XML Streams

We use a binary representation for XML trees in which each left branch points to the leftmost child and each right branch to the first sibling node to follow in the tree

structure of XML. For example, consider a fragment of XML <a><b/><c/><d/>. It has tree structure as shown in the upper-right figure. In binary representation, the XML tree is figured as shown in the lower-right one where a black bullet stands for a leaf which means the end of siblings. Every tree in binary representation is defined by a tree over $\Sigma_{tree} = \{\mathsf{N}^{(2)}, \mathsf{L}^{(0)}\}$ where N is a binary symbol annotated with a label and L is a nullary symbol corresponding a leaf. For instance, we write $\mathsf{N_a}(\mathsf{N_b}(\mathsf{L}, \mathsf{N_c}(\mathsf{L}, \mathsf{N_d}(\mathsf{L}, \mathsf{L}))), \mathsf{L})$ for the XML tree above. The set of trees in binary representation is denoted by $T_{\Sigma_{tree}}$.





XML streams are defined by monadic trees over $\Sigma_{stream} = \{\mathsf{S}^{(1)}, \mathsf{E}^{(1)}, \mathsf{Z}^{(0)}\}$ where S and E are unary symbols corresponding a start tag and an end tag, respectively, each of them is annotated with a label as well as N, and Z is a nullary symbol standing for the end of stream. For instance, we write $\mathsf{S_a}(\mathsf{S_b}(\mathsf{E_b}(\mathsf{S_c}(\mathsf{E_c}(\mathsf{E_a}(\mathsf{Z}))))))$ for a fragment of XML <a><b/><c/><a>. The set of streams is denoted by $T_{\Sigma_{stream}}$.

3.2 Extension of Attributed Tree Transducers with annotation

Xatt$^{t\to s}$ deals with annotated trees such as XML trees and XML streams as ATT deals with trees. An Xatt$^{t\to s}$ is defined by a tuple of $M = (Syn, Inh, \Sigma, \Delta, R)$[1] in similar way to an ATT, where $\Sigma = \Sigma_{tree}$ and $\Delta = \Sigma_{stream}$ since an Xatt$^{t\to s}$ represents a transformation from XML trees to XML streams. The major difference from ATT is that R is defined by $\cup_{\sigma\in\Sigma_{tree}} R^{\sigma_x}$ with finite sets R^{σ_x} of σ_x-rules in which we can use the annotation x in a right-hand side of an attribute rule. For example, R^{N_x} may contain an attribute rule $s(\pi) \to \mathsf{S}_x(s(\pi 1))$. We show a simple example of Xatt$^{t\to s}$ which represents an identity XML transformation that takes an XML tree and returns the corresponding XML stream. The Xatt$^{t\to s}$ $M_{id} = (Syn, Inh, \Sigma, \Delta, R)$ is defined by

- $Syn = \{s\}$ and $Inh = \{i\}$
- $R = R^{\sharp} \cup R^{\mathsf{N}_x} \cup R^{\mathsf{L}}$ where
  - $R^{\sharp} = \{s_{in}(\pi) \to s(\pi 1),\ i(\pi 1) \to \mathsf{Z}\}$
  - $R^{\mathsf{N}_x} = \{s(\pi) \to \mathsf{S}_x(s(\pi 1)),\ i(\pi 1) \to \mathsf{E}_x(s(\pi 2)),\ i(\pi 2) \to i(\pi)\}$
  - $R^{\mathsf{L}} = \{s(\pi) \to i(\pi)\}$.

Letting $t = \mathsf{N_a}(\mathsf{N_b}(\mathsf{L}, \mathsf{N_a}(\mathsf{L}, \mathsf{L})), \mathsf{L})$ be a given input XML tree and $\Rightarrow$ stand for $\Rightarrow_{M,\sharp(t)}$, the transformation result of M_{id} is computed by

$$
\begin{aligned}
& s_{in}(\varepsilon) \Rightarrow s(1) \Rightarrow \mathsf{S_a}(s(11)) \Rightarrow \mathsf{S_a}(\mathsf{S_b}(s(111))) \Rightarrow \mathsf{S_a}(\mathsf{S_b}(i(111))) \\
\Rightarrow\ & \mathsf{S_a}(\mathsf{S_b}(\mathsf{E_b}(s(112)))) \Rightarrow \mathsf{S_a}(\mathsf{S_b}(\mathsf{E_b}(\mathsf{S_a}(s(1121))))) \Rightarrow \mathsf{S_a}(\mathsf{S_b}(\mathsf{E_b}(\mathsf{S_a}(i(1121)))))) \\
\Rightarrow\ & \mathsf{S_a}(\mathsf{S_b}(\mathsf{E_b}(\mathsf{S_a}(\mathsf{E_a}(s(1122)))))) \Rightarrow \mathsf{S_a}(\mathsf{S_b}(\mathsf{E_b}(\mathsf{S_a}(\mathsf{E_a}(i(1122)))))) \\
\Rightarrow\ & \mathsf{S_a}(\mathsf{S_b}(\mathsf{E_b}(\mathsf{S_a}(\mathsf{E_a}(i(112)))))) \Rightarrow \mathsf{S_a}(\mathsf{S_b}(\mathsf{E_b}(\mathsf{S_a}(\mathsf{E_a}(i(11)))))) \\
\Rightarrow\ & \mathsf{S_a}(\mathsf{S_b}(\mathsf{E_b}(\mathsf{S_a}(\mathsf{E_a}(\mathsf{E_a}(s(12))))))) \Rightarrow \mathsf{S_a}(\mathsf{S_b}(\mathsf{E_b}(\mathsf{S_a}(\mathsf{E_a}(\mathsf{E_a}(i(12))))))) \\
\Rightarrow\ & \mathsf{S_a}(\mathsf{S_b}(\mathsf{E_b}(\mathsf{S_a}(\mathsf{E_a}(\mathsf{E_a}(i(1))))))) \Rightarrow \mathsf{S_a}(\mathsf{S_b}(\mathsf{E_b}(\mathsf{S_a}(\mathsf{E_a}(\mathsf{E_a}(\mathsf{Z}))))))).
\end{aligned}
$$

[1] The initial attribute s_{in} and the root symbol $\sharp$ are omitted for simplicity.

Fig.2 visualizes the above computation. It shows the input XML tree t annotated with attributes and their dependencies. For example, $i(1121)$ is computed by $\mathsf{E_a}(s(1122))$ because of the attribute rule $(i(\pi 1) \to \mathsf{E}_x(s(\pi 2))) \in R^{\mathsf{N}_x}$ with $\pi = 112$ and $x = \mathsf{a}$ from $\mathit{label}(\sharp(t), 112) = \mathsf{N_a}$.

The Xatt$^{t\to s}$ M_{id} transforms from XML trees into the corresponding XML stream. Using two attributes s and i, we can make an evaluation in depth-first right-to-left order. Note that we do not directly use this Xatt$^{t\to s}$ for stream processing. Since we use an ATT obtained by the composition of the original Xatt$^{t\to s}$ and parsing transformation, the above evaluation does not mean that stream processing cannot start to output any result before it reads the complete input.

Xatt$^{t\to s}$ can deal with symbols other than output symbols in Σ_{stream} in right-hand sides of the attribute rules. Let us consider an Xatt$^{t\to s}$ M'_{id} has the same definition as M_{id} except that R^{N_x} is a set of the following rules:

$$s(\pi) \to \mathsf{Cat}(\mathsf{S}_x, s(\pi 1)), \quad i(\pi 1) \to \mathsf{Cat}(\mathsf{E}_x, s(\pi 2)), \quad i(\pi 2) \to i(\pi).$$

The computation by M'_{id} is done in similar to that by M_{id}. For example, if M_{id} outputs $\mathsf{S_a}(\mathsf{S_b}(\mathsf{E_b}(\ldots)))$ for an input, M'_{id} outputs $\mathsf{Cat}(\mathsf{S_a}, \mathsf{Cat}(\mathsf{S_b}, \mathsf{Cat}(\mathsf{E_b}, \ldots)))$ for the same input. The symbols S_x and E_x are used as nullary symbols and the binary symbol Cat means a concatenation of outputs and In the rest of paper we use these symbols instead of unary symbols S_x and E_x for output symbols. It will be helpful for us to obtain XML stream processors.

Furthermore, our framework allows the output alphabet of Xatt$^{t\to s}$ to include the other symbols such as Str_x, Eq_x, And, Or and If: the nullary symbol Str_x means a string value x; the unary symbol Eq_x means a boolean value representing whether the argument matches a string value x or not; the binary symbols And and Or represent boolean operations in usual way; the 3-ary symbol If means a conditional branch by the first argument. These symbols are useful for us to define a wide variety of XML transformations.

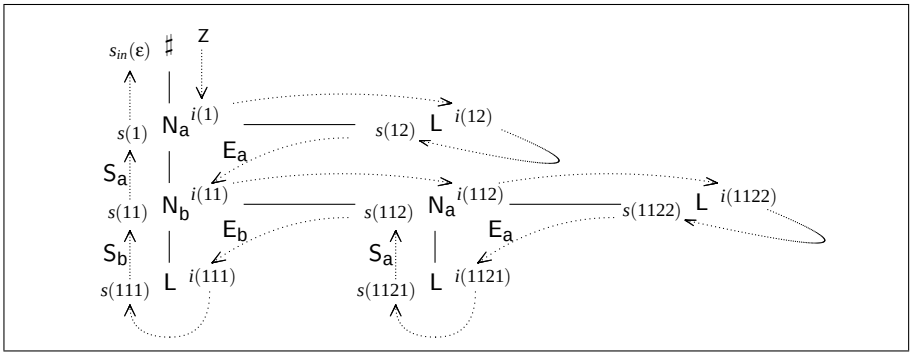


Fig. 2. The computation by M_{id} for the input $\mathsf{N_a}(\mathsf{N_b}(\mathsf{L}, \mathsf{N_a}(\mathsf{L}, \mathsf{L})), \mathsf{L})$

3.3 Conversion from Practical XML Transformations to Xatt$^{t\to s}$

This section gives practical examples and several hints for converting XML transformation into Xatt$^{t\to s}$. A basic idea of the conversion is that we give a number of pairs of synthesized attribute s and inherited attributes i whose attribute rules for N_x have the following form:

$$s(\pi) \to \mathsf{If}(e_{test}, e_{s1}, e_{s2}), \qquad i(\pi 1) \to \mathsf{If}(e_{test}, e_{i1}, e_{i2}), \qquad i(\pi 2) \to i(\pi) \quad (1)$$

where e_{test}, e_{s1}, e_{s2}, e_{i1} and e_{i2} are right-hand side expressions for Xatt$^{t\to s}$. By adjusting these expressions we can convert a variety of XML transformations into Xatt$^{t\to s}$. For example, let us consider an Xatt$^{t\to s}$ M_{fil} which has the same definition as M'_{id} except that R^{N_x} has attribute rules of the form (1) where

$$e_{test} = \mathsf{Eq}_x(\mathsf{Str}_\mathsf{b}),\ e_{s1} = s(\pi 2),\ e_{s2} = \mathsf{Cat}(\mathsf{S}_x, s(\pi 1)),\ e_{i1} = e_{i2} = \mathsf{Cat}(\mathsf{E}_x(s(\pi 2)))^2.$$

The Xatt$^{t\to s}$ M_{fil} gives a transformation which outputs the input tree with no b element, which can be figured out as follows. If $x \neq \mathsf{b}$, then R^{N_x} of M_{fil} equals to that of M'_{id} because we take the second branch $\mathsf{Cat}(\mathsf{S}_x, s(\pi 1))$ for If. Suppose that $x = \mathsf{b}$. Since we take the first branch $s(\pi 2)$, the dotted arrow which points to $s(11)$ in Fig.2 comes from $s(112)$ instead of $s(111)$. It implies that all nodes under the first branch of N_b is skipped in the evaluation. Therefore we get the transformation result which is the input tree with no b element. In the following, we show how to convert XML transformations to Xatt$^{t\to s}$ for two practical examples, *XPath querying* and *partial modification*. Each transformation is defined by a pair of synthesized and inherited attributes. A number of XML transformation programs can be regarded as the combination of these basic transformations.

XPath Querying and its extension XPath[24] is a syntax for defining parts of an XML tree by using paths on the XML tree. For example, `/child::a/descendant::b [child::c]` designates `b` elements each of which is descendant of an `a` node and has a `c` node as its child. XPath querying collects all elements satisfying the XPath expression.

A part of XPath querying can be converted into Xatt$^{t\to s}$. We consider querying by absolute XPath expressions in which only forward axes are used, *i.e.* , the expression cannot contain backward axes such as `parent` and `ancestor`. Note that this restriction is not too excessive. All backward axes in absolute XPath can be removed by the method introduced by Olteanu et al.[21].

XPath querying is converted into Xatt$^{t\to s}$ M_{que} which contains conditional branches of the form (1) where

$$e_{s1} = \mathsf{Cat}(\mathsf{S}_x, s(\pi 1)), \qquad e_{s2} = s(\pi 1), \qquad e_{i1} = \mathsf{Cat}(\mathsf{E}_x, s(\pi 2)), \qquad e_{i2} = s(\pi 2).$$

In the computation by M_{que}, the node is copied if e_{test} is evaluated to true at the node. Otherwise, no copy is created for the node. Thus we need to specify how to give e_{test} for each XPath expression in order to convert the XPath querying into Xatt$^{t\to s}$[3].

[2] The conditional branch If is useless for $i(\pi 1)$-rule because $e_{i1} = e_{i2}$.

[3] This conversion assumes that the node whose ancestor is queried is not queried by a given XPath expression. In order to query such nested nodes, the other conversion is required.

The conversion is defined by associating all subexpressions of a given XPath expression. with a synthesized or inherited attribute in an Xatt$^{t\to s}$. Subexpressions in bracketed qualifiers (also called predicates) in the XPath are associated with synthesized attributes. The other subexpressions are related with inherited attributes. Consider an absolute XPath expression $\overbrace{\texttt{/child}}^{u1}\underbrace{\texttt{::a/descendant}}_{u2}\texttt{::b}\underbrace{\texttt{[child::c]}}_{v1}$. We take three subexpressions as shown by curly braces each of which is associated with an attribute assigned to the brace. The attributes *u1* and *u2* are inherited and *v1* is synthesized. To complete the conversion, we use one more attribute *u3* to propagate information about whether the node is a descendant of the node satisfying the XPath expression. The following attribute rules define the relation of the attributes:

$$\begin{aligned}
R^{\sharp} &= \{u1(\pi1) \to \mathsf{True},\ u2(\pi1) \to \mathsf{False},\ u3(\pi1) \to \mathsf{False},\ \ldots\}\\
R^{\mathsf{N}_x} &= \{v1(\pi) \to \mathsf{Or}(\mathsf{Eq}_x(\mathsf{Str}_\mathsf{c}), v1(\pi2)),\ u1(\pi1) \to \mathsf{False},\ u1(\pi2) \to u1(\pi),\\
&\quad u2(\pi1) \to \mathsf{Or}(u2(\pi), \mathsf{And}(u1(\pi), \mathsf{Eq}_x(\mathsf{Str}_\mathsf{a}))),\ u2(\pi2) \to u2(\pi),\\
&\quad u3(\pi1) \to \mathsf{Or}(u3(\pi), \mathsf{And}(u2(\pi), \mathsf{And}(\mathsf{Eq}_x(\mathsf{Str}_\mathsf{b}), v1(\pi1)))),\\
&\quad u3(\pi2) \to,\ \mathsf{Or}(u3(\pi), \mathsf{And}(u2(\pi), \mathsf{And}(\mathsf{Eq}_x(\mathsf{Str}_\mathsf{b}), v1(\pi1)))),\ \ldots\}\\
R^{\mathsf{L}} &= \{v1(\pi) \to \mathsf{False},\ \ldots\}.
\end{aligned}$$

where several attribute rules are omitted. We use the same attribute rules as R^{L} and $R^{\sharp}$ of M_{id} for attributes s_{in}, s and i. The value of an attribute *u1* represents whether the node is the child of the root; The value of *u2* represents whether the node is the descendant of an a node which is the child of the root; The value of *v1* represents whether either the node or one of the following sibling node is a c node. An Xatt$^{t\to s}$ representing the intended XPath querying is defined by M_{que} with $e_{test} = \mathsf{Or}(u3(\pi), \mathsf{And}(u2(\pi), \mathsf{And}(\mathsf{Eq}_x(\mathsf{Str}_\mathsf{b}), v1(\pi1))))$. The expression e_{test}, which equals to the right-hand side of the attribute rules for *u3*, represents whether either the node or one of its ancestors satisfies the XPath.

Partial Modification Whereas XPath querying leaves the designate elements and strips their contexts, partial modification leaves the context of the the designate elements and replaces the elements with the other elements. The partial modification is converted into an Xatt$^{t\to s}$ in similar way to XPath querying. Let us consider an Xatt$^{t\to s}$ M_{mod} which has the same definition as M'_{id} except that R^{N_x} is a set of the following rules:

$$e_{s1} = \mathsf{Cat}(\mathsf{S}_x, \mathsf{Cat}(\mathsf{E}_x, s(\pi2))), \quad e_{s2} = \mathsf{Cat}(\mathsf{S}_x, s(\pi1)), \quad e_{i1} = e_{i2} = \mathsf{E}_x(s(\pi2)).$$

where e_{test} is a certain expression for specifying the designate node. The element is replaced with a no-child element if e_{test} is evaluated to true at the node. Otherwise, the node does not change since the attribute rules equals to that of M'_{id}.

Let us see the other example of partial modification. Consider an Xatt$^{t\to s}$ M'_{mod} which has the same definition as M_{mod} except that the first two rules of R^{N_x} is as follows:

$$e_{s1} = \mathsf{Cat}(\mathsf{S}_\mathsf{a}, s(\pi1)),\ e_{s2} = \mathsf{Cat}(\mathsf{S}_x, s(\pi1)),\ e_{i1} = \mathsf{Cat}(\mathsf{E}_\mathsf{a}, s(\pi2)),\ e_{i2} = \mathsf{Cat}(\mathsf{E}_x, s(\pi2)).$$

The name of the node changes into a if e_{test} is evaluated to true at the node. Otherwise, the node does not changed. This procedure is applied to every node of the input XML.

Now we show an example of Xatt$^{t\to s}$ M_{JkS} that plays a role of partial modification. The attribute rules of M_{JkS} are comprised of

$$
\begin{aligned}
R^{\sharp} &= \{s_{in}(\pi) \to s(\pi 1),\ i(\pi 1) \to \mathsf{Z}\} \\
R^{\mathsf{N}_x} &= \{v1(\pi) \to \mathsf{Or}(\mathsf{Eq}_x(\mathsf{Str}_{\mathtt{k}}), v1(\pi 2)), \\
&\quad s(\pi) \to \mathsf{Cat}(\mathsf{If}(\mathsf{And}(\mathsf{Eq}_x(\mathsf{Str}_{\mathtt{J}}), v1(\pi 1)), \mathsf{S}_{\mathsf{S}}, \mathsf{S}_x), s(\pi 1)), \\
&\quad i(\pi 1) \to \mathsf{Cat}(\mathsf{If}(\mathsf{And}(\mathsf{Eq}_x(\mathsf{Str}_{\mathtt{J}}), v1(\pi 1)), \mathsf{E}_{\mathsf{S}}, \mathsf{E}_x), s(\pi 2)), \\
&\quad i(\pi 2) \to i(\pi)\} \\
R^{\mathsf{L}} &= \{s(\pi) \to i(\pi),\ v1(\pi) \to \mathsf{False}\}
\end{aligned}
$$

where we use $\mathsf{Cat}(\mathsf{If}(e_{test}, e_1, e_2), e_3)$ instead of $\mathsf{If}(e_{test}, \mathsf{Cat}(e_1, e_3), \mathsf{Cat}(e_2, e_3))$ to obtain efficient stream processors. The name of every node changes into S by M_{JkS} only if the node satisfies an XPath expression `/descendant-or-self::J[child::k]` (= `//J[k]`), *i.e.* , M_{JkS} changes the name of the J node having a k node as its child into S. For example, an XML `<h><I><k/></I><J><k/></J><J><c/></J></h>` is transformed into `<h><I><k/></I><S><k/></S><J><c/></J></h>` by M_{JkS}. It uses no inherited attribute associated with the subexpression `/descendant-or-self::` since the expression should be satisfied at any node.

Combination of basic transformations The conversion of basic transformations is easily extended by the other XML transformations. For instance, consider an XML transformation T_{comb} which returns the collection of results of partial modification only for nodes specified by an XPath. We need two pairs of synthesized and inherited attributes: one pair $\langle s_1, i_1 \rangle$ is used for a partial modification; another pair $\langle s_2, i_2 \rangle$ is used for collecting results for each nodes specified by XPath. The XML transformation T_{comb} is defined by Xatt$^{t\to s}$ with these attributes. The conversion of combination of basic transformation into Xatt$^{t\to s}$ has been automatically done in the implementation of an XML transformation language XTiSP introduced in Appendix B (see also [29]).

4 Composition with an XML Parsing Attributed Tree Transducer

This section introduces a method for obtaining an ATT which represents a transformation from XML streams to XML streams, denoted by Xatt$^{s\to s}$. An Xatt$^{t\to s}$ in the previous section represents a transformation from XML trees to XML streams. In order to derive Xatt$^{s\to s}$, we compose the Xatt$^{t\to s}$ with an XML-parsing ATT M_{parse} to synthesize a single ATT, where M_{parse} represents a transformation from XML streams to trees. The composition employs a composition method for *stack-attributed tree transducers*(SATT)[16] because the parsing ATT requires a stack device which was harmful for the original composition method [8, 14]. Since the composition method is rather involved, we introduce the method specialized for the case of the composition an Xatt$^{t\to s}$ and the parsing ATT M_{parse} whose definition is presented in Appendix A. By applying the composition method to a given Xatt$^{t\to s}$ and M_{parse}, the following composition method is obtained. Although the composition method does not deal with annotations of nodes, we can easily make extension of the method for them.

Let $M = (Syn, Inh, \Sigma_{tree}, \Delta, s_{in}, \sharp, R)$ be an Xatt$^{t\to s}$ where Δ includes primitive functions such as Cat, S_x, True, and so on. The corresponding Xatt$^{s\to s}$ is defined by $M = (Syn', Inh', \Sigma_{stream}, \Delta', s_{in}, \sharp, R')$ where

- $Syn' = \{\langle s', s\rangle \mid s' \in Syn,\ s \in \{p, l\}\}$, $Inh' = \{\langle i', s\rangle \mid i' \in Inh,\ s \in \{p, l\}\}$
- $\Delta' = \Delta \cup \{Head^{(1)}, Tail^{(1)}, Cons^{(2)}, Nil^{(0)}\}$,
- and $R' = R'^{\sharp} \cup R'^{\mathsf{S}_x} \cup R'^{\mathsf{E}_x} \cup R'^{\mathsf{Z}}$ with

$$
\begin{aligned}
R'^{\sharp} &= \{s_{in} \to \varphi[s'(\pi 1) := \langle s', p\rangle(\pi 1)]_{s'\in Syn} \mid (s_{in}(\pi) \to \varphi) \in R^{\sharp}\} \\
&\quad \cup \{\langle i', p\rangle \to \varphi[s'(\pi 1) := \langle s', p\rangle(\pi 1)]_{s'\in Syn} \mid (i'(\pi 1) \to \varphi) \in R^{\sharp}\} \\
&\quad \cup \{\langle i', l\rangle \to Nil \mid i' \in Inh\} \\
R'^{\mathsf{S}_x} &= \{\langle s', p\rangle(\pi) \to \rho(\varphi) \mid (s'(\pi) \to \varphi) \in R^{\mathsf{N}_x}, s' \in Syn\} \\
&\quad \cup \{\langle s', l\rangle(\pi) \to Tail(\langle s', l\rangle(\pi 1)) \mid s' \in Syn\} \\
&\quad \cup \{\langle i', p\rangle(\pi 1) \to \rho(\varphi) \mid (i'(\pi 1) \to \varphi) \in R^{\mathsf{N}_x}, i' \in Inh\} \\
&\quad \cup \{\langle i', l\rangle(\pi 1) \to Cons(\rho(\varphi), \langle i', l\rangle(\pi)) \mid (i'(\pi 2) \to \varphi) \in R^{\mathsf{N}_x}, i' \in Inh\} \\
R'^{\mathsf{E}_x} &= \{\langle s', p\rangle(\pi) \to \varphi[i'(\pi) := \langle i', p\rangle(\pi)]_{i'\in Inh} \mid (s'(\pi) \to \varphi) \in R^{\mathsf{L}}, s' \in Syn\} \\
&\quad \cup \{\langle s', l\rangle(\pi) \to Cons(\langle s', p\rangle(\pi 1), \langle s', l\rangle(\pi 1)) \mid s' \in Syn\} \\
&\quad \cup \{\langle i', p\rangle(\pi 1) \to Head(\langle i', l\rangle(\pi)) \mid i' \in Inh\} \\
&\quad \cup \{\langle i', l\rangle(\pi 1) \to Tail(\langle i', l\rangle(\pi)) \mid i' \in Inh\} \\
R'^{\mathsf{Z}} &= \{\langle s', p\rangle(\pi) \to \varphi[i'(\pi) := \langle i', p\rangle(\pi)]_{i'\in Inh} \mid (s'(\pi) \to \varphi) \in R^{\mathsf{L}}, s' \in Syn\} \\
&\quad \cup \{\langle s', l\rangle(\pi) \to Nil \mid s' \in Syn\}
\end{aligned}
$$

where $\rho = [s'(\pi 1), s'(\pi 2), i'(\pi) := \langle s', p\rangle(\pi 1), Head(\langle s', l\rangle(\pi 1)), \langle i', p\rangle(\pi)]_{s'\in Syn, i'\in Inh}$.

The added output symbols *Head*,*Tail*,*Cons* and *Nil* are caused by a stack operator in M_{parse}. Each of them has a meaning with respect to stack operation: $Head(e)$ represents the top-most element of a stack e; $Tail(e)$ represents a stack e whose top-most element is removed; $Cons(e_1, e_2)$ represents a stack obtained by pushing a value e_1 to a stack e_2; *Nil* represents an empty stack.

For example, we obtain Xatt$^{s\to s}$ M'_{JkS} from the definition of the Xatt$^{t\to s}$ M_{JkS} by the above method. The attribute rules of M'_{JkS} are comprised of

$$
\begin{aligned}
R'^{\sharp} &= \{s_{in}(\pi) \to \langle s, p\rangle(\pi 1),\ \langle i, p\rangle(\pi 1) \to \mathsf{Z},\ \langle i, l\rangle(\pi 1) \to Nil\}, \\
R'^{\mathsf{S}_x} &= \{\langle s, p\rangle(\pi) \to \mathsf{Cat}(\mathsf{If}(\mathsf{And}(\mathsf{Eq}_x(\mathsf{Str}_{\mathsf{J}}), \langle v1, p\rangle(\pi 1)), \mathsf{S}_{\mathsf{S}}, \mathsf{S}_x), \langle s, p\rangle(\pi 1)), \\
&\quad \langle s, l\rangle(\pi) \to Tail(\langle s, l\rangle(\pi 1)), \\
&\quad \langle i, p\rangle(\pi 1) \to \mathsf{Cat}(\mathsf{If}(\mathsf{And}(\mathsf{Eq}_x(\mathsf{Str}_{\mathsf{J}}), \langle v1, p\rangle(\pi 1)), \mathsf{E}_{\mathsf{S}}, \mathsf{E}_x), Head(\langle s, l\rangle(\pi 1))), \\
&\quad \langle i, l\rangle(\pi 1) \to Cons(\langle i, p\rangle(\pi), \langle i, l\rangle(\pi)), \\
&\quad \langle v1, p\rangle(\pi) \to \mathsf{Or}(\mathsf{Eq}_x(\mathsf{Str}_{\mathsf{k}}), Head(\langle v1, l\rangle(\pi 1))),\ \langle v1, l\rangle(\pi) \to Tail(\langle v1, l\rangle(\pi 1))\}, \\
R'^{\mathsf{E}_x} &= \{\langle s, p\rangle(\pi) \to \langle i, p\rangle(\pi),\ \langle s, l\rangle(\pi) \to Cons(\langle s, p\rangle(\pi 1), \langle s, l\rangle(\pi 1)), \\
&\quad \langle i, p\rangle(\pi 1) \to Head(\langle i, l\rangle(\pi)),\ \langle i, l\rangle(\pi 1) \to Tail(\langle i, l\rangle(\pi)), \\
&\quad \langle v1, p\rangle(\pi) \to \mathsf{False},\ \langle v1, l\rangle(\pi) \to Cons(\langle v1, p\rangle(\pi 1), \langle v1, l\rangle(\pi 1))\}, \\
R'^{\mathsf{Z}} &= \{\langle s, p\rangle(\pi) \to \langle i, p\rangle(\pi),\ \langle s, l\rangle(\pi) \to Nil,\ \langle v1, p\rangle(\pi) \to \mathsf{False},\ \langle v1, l\rangle(\pi) \to Nil\}.
\end{aligned}
$$

5 Attribute Evaluation in Stream Processing Style

We show a method deriving an XML stream processor from an Xatt$^{s\to s}$. The main idea of the method is that attribute values the Xatt$^{s\to s}$ are evaluated in stream processing style.

In the attribute evaluation for Xatt$^{s\to s}$, we use extra rules for primitive functions as well as the reduction rules specified by the definition of ATT. The definition of Xatt$^{t\to s}$ allows to use primitive functions such as If, And and Cat. At the previous stage, they have been regarded as constructor symbols since the composition method for ATTs cannot be applied to primitive functions destroying tree structures. The attribute evaluation deals with them as meaningful functions to obtain the transformation result. For instance, we have the following rules:

$$\begin{array}{lll} \mathsf{If}(\mathsf{True},e_1,e_2)\Rightarrow e_1, & \mathsf{If}(\mathsf{False},e_1,e_2)\Rightarrow e_2, & \mathsf{Eq}_x(\mathsf{Str}_x)\Rightarrow \mathsf{True}, \\ \mathsf{And}(\mathsf{True},e_1)\Rightarrow e_1, & \mathsf{And}(\mathsf{False},e_1)\Rightarrow \mathsf{False}, & \cdots \end{array}$$

It is obvious that these rules do not conflict with the reduction rules defined by ATT.

We use a running example of the Xatt$^{s\to s}$ M'_{JkS} to show a method for deriving an XML stream processor. Suppose that an input for M'_{JkS} is $t=\mathsf{S_J}(\mathsf{S_k}(\mathsf{E_k}(\mathsf{S_l}(\mathsf{E_l}(\mathsf{E_J}(\mathsf{Z}))))))$ corresponding to an XML <J><k/><l/></J>. Let $\Rightarrow$ stand for $\Rightarrow_{M'_{JkS},\sharp(t)}$ in the rest of this section. From the definition by ATT, we obtain the transformation result r by $r=nf(\Rightarrow,s_{in}(\varepsilon))$. An XML stream processor for M'_{JkS} computes the normal form by integrating a partial result for each input symbol, $\mathsf{S_J}$, $\mathsf{S_k}$, $\mathsf{E_k}$, $\mathsf{S_l}$, $\mathsf{E_l}$, $\mathsf{E_J}$ and Z.

Before any input symbol is read, we find that r is computed as $nf(\Rightarrow,\langle s,p\rangle(1))$. since we have $s_{in}(\varepsilon)\Rightarrow\langle s,p\rangle(1)$ from the attribute rule of $R'^{\sharp}$ in M'_{JkS}. We cannot progress the computation until the first symbol is read. Additionally, in preparation for the following computation, we evaluate two attribute values $\langle i,p\rangle(1)$ and $\langle i,l\rangle(1)$ which may be needed when the next symbol is read. These values are computed into Z and *Nil*, respectively, by the attribute rules of $R'^{\sharp}$.

When an input symbol $\mathsf{S_J}$ is read, we find that r is computed as $nf(\Rightarrow,\langle s,p\rangle(1))$. since we have

$$\begin{array}{rl} \langle s,p\rangle(1) & \Rightarrow \mathsf{Cat}(\mathsf{If}(\mathsf{And}(\mathsf{Eq_J}(\mathsf{Str_J}),\langle v1,p\rangle(11)),\mathsf{S_S},\mathsf{S_J}),\langle s,p\rangle(11)) \\ & \Rightarrow \mathsf{Cat}(\mathsf{If}(\mathsf{And}(\mathsf{True},\langle v1,p\rangle(11)),\mathsf{S_S},\mathsf{S_J}),\langle s,p\rangle(11)) \\ & \Rightarrow \mathsf{Cat}(\mathsf{If}(\langle v1,p\rangle(11),\mathsf{S_S},\mathsf{S_J}),\langle s,p\rangle(11)) \end{array}$$

from the attribute rule of $R'^{\sharp}$ in M'_{JkS}, $\mathsf{Eq}_x(\mathsf{Str}_x)\Rightarrow\mathsf{True}$ and $\mathsf{And}(\mathsf{True},e_1)\Rightarrow e_1$. We cannot progress the computation until the next symbol is read for computing the value of $\langle s,p\rangle(11)$ and $\langle v1,p\rangle(11)$. Additionally, in preparation for the following computation, we evaluate two attribute values $\langle i,p\rangle(11)$ and $\langle i,l\rangle(11)$ which may be needed when the next symbol is read. These values are computed as follows:

$$\begin{array}{rl} \langle i,p\rangle(11) & \Rightarrow^{*} \mathsf{Cat}(\mathsf{If}(\langle v1,p\rangle(11),\mathsf{E_S},\mathsf{E_J}),\mathit{Head}(\langle s,l\rangle(11))) \\ \langle i,l\rangle(11) & \Rightarrow \mathit{Cons}(\langle i,p\rangle(1),\langle i,l\rangle(1)) = \mathit{Cons}(\mathsf{Z},\mathit{Nil}) \end{array}$$

where we use the values of $\langle i,p\rangle(1)$ and $\langle i,l\rangle(1)$ that is prepared by the last step. The attribute value of $\langle i,p\rangle(11)$ is just partially computed since it requires the following input symbol to know the values of $\langle v1,p\rangle(11)$ and $\langle s,l\rangle(11)$.

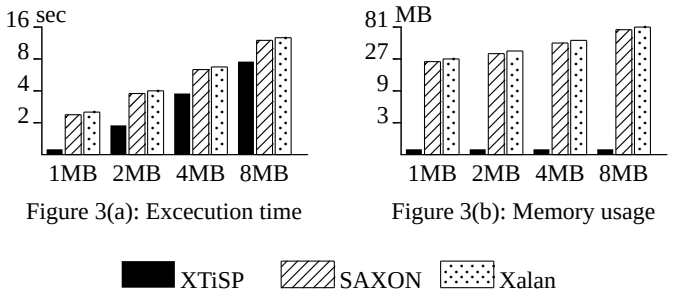


Figure 3(a): Excecution time Figure 3(b): Memory usage

Fig. 3. Benchmark results

When the next input symbol $\mathsf{S_k}$ is read, we find that r is computed as $nf(\Rightarrow, \langle s,p\rangle(1))$. since we have

$$
\begin{aligned}
& \mathsf{Cat}(\mathsf{If}(\langle v1,p\rangle(11), \mathsf{S_S}, \mathsf{S_J}), \langle s,p\rangle(11)) \\
\Rightarrow\ & \mathsf{Cat}(\mathsf{If}(\mathsf{Or}(\mathsf{Eq_k}(\mathsf{Str_k}), \mathit{Head}(\langle v1,l\rangle(111))), \mathsf{S_S}, \mathsf{S_J}), \langle s,p\rangle(11)) \\
\Rightarrow\ & \mathsf{Cat}(\mathsf{If}(\mathsf{Or}(\mathsf{True}, \mathit{Head}(\langle v1,l\rangle(111))), \mathsf{S_S}, \mathsf{S_J}), \langle s,p\rangle(11)) \\
\Rightarrow\ & \mathsf{Cat}(\mathsf{If}(\mathsf{True}, \mathsf{S_S}, \mathsf{S_J}), \langle s,p\rangle(11)) \ \Rightarrow\ \mathsf{Cat}(\mathsf{S_S}, \langle s,p\rangle(11)) \\
\Rightarrow\ & \mathsf{Cat}(\mathsf{S_S}, \mathsf{Cat}(\mathsf{If}(\mathsf{And}(\mathsf{Eq_k}(\mathsf{Str_J}), \langle v1,p\rangle(111)), \mathsf{S_S}, \mathsf{S_k}), \langle s,p\rangle(111))) \\
\Rightarrow\ & \mathsf{Cat}(\mathsf{S_S}, \mathsf{Cat}(\mathsf{If}(\mathsf{And}(\mathsf{False}, \langle v1,p\rangle(111)), \mathsf{S_S}, \mathsf{S_k}), \langle s,p\rangle(111))) \\
\Rightarrow\ & \mathsf{Cat}(\mathsf{S_S}, \mathsf{Cat}(\mathsf{If}(\mathsf{False}, \mathsf{S_S}, \mathsf{S_k}), \langle s,p\rangle(111))) \\
\Rightarrow\ & \mathsf{Cat}(\mathsf{S_S}, \mathsf{Cat}(\mathsf{S_k}, \langle s,p\rangle(111))).
\end{aligned}
$$

Note that two Cat applications in $\mathsf{Cat}(\mathsf{S_S}, \mathsf{Cat}(\mathsf{S_k}, \ldots))$ will not be modified by the following computation. Therefore the XML stream processor can output the string $\mathsf{S_S}$ and $\mathsf{S_k}$, corresponding <S><k>, and the rest of result is computed by $\langle s,p\rangle(111)$. The output is a desirable behavior for XML stream processing. The transformation by M_{JkS} replaces every J element into S only when the node has a k element as its child. Thought the transformation cannot return any symbol even if an input symbol <J> is read, the symbol <S> and its children are output once an input symbol <k> is found at the child position. If no <k> element is found at the child position of a J element, then the J element are output without changing the element name.

The XML stream processor computes the transformation result by repeating the similar procedure to above for the following input. Letting *#Inh* be the number of inherited attributes of an $\mathrm{Xatt}^{s\to s}$ M, the stream processor computes the fixed number $\#Inh+1$ of values for each input symbol: one is used for the transformation result to be output afterward, the others may be needed at the next computation as values of inherited attributes.

6 Experimental Results

We have implemented our framework as an XML transformation language XTiSP[29], in which we can use two kinds of primitive transformations: XPath iteration and par-

tial modification. See Appendix B for summary of XTiSP. The implemented system takes an XTiSP program and returns a stream processing program written in Objective Caml[20]. The system itself is also written in Objective Caml.

We compared a program in XTiSP, which is converted into M_{JkS}, with the corresponding program written in XSLT[25]. We used Xalan[26] and SAXON[23] as XSLT processors. The comparison was done for an input XML generated randomly such that each tag name is `I`, `J` or `k`. The experiments were conducted on a PC (PowerMacintosh G5/Dual 2GHz, 1GB memory). We measured execution time and memory usage for several inputs whose size are 1MB, 2MB, 4MB and 8MB. Fig.3(a) and Fig.3(b) show the comparison results. Our implementation is much faster and much more memory-efficient than the others. We also tried more complicated examples of XML transformation such as XML database into XHTML and then our implementation won definitely.

However we cannot always benefit from automatic derivation of stream processors. As an extreme example, Xatt$^{t\to s}$ can define a mirror transformation which reverses the order of child elements at every node. In this case, the program cannot output anything except for the start tag of the root until the end tag of the last child of the root is read whose next tag is the end tag of the root, that is the last token event of the input. This kind of transformation is not appropriate to stream processing. Although we have no way to find whether a given XML transformation is appropriate or not, we can easily add a mechanism to measure growth of stacks required by stream processing.

7 Conclusion

We have shown an implementation scheme for XML transformation language intended for stream processing. If one wants to implement an XML transformation language, all he has to do is to give a conversion method into Xatt$^{t\to s}$ for the language. The programmer can obtain an efficient stream processing program without writing stateful programs like SAX.

Additionally, we have implemented an XML transformation language XTiSP, which has its encoding method into Xatt$^{t\to s}$, and have compared with other XML tree transformation languages to confirm effectiveness of our system. XTiSP works much faster than the other implementations. because that it can output until the whole input is read.

Recently the author is addressing automatic generation from programs written in XSLT into Xatt$^{t\to s}$. If we automatically obtain Xatt$^{t\to s}$ from XSLT programs, we can also obtain the corresponding XML stream processor for XSLT.

Acknowledgment

This work is partially supported by the *Comprehensive Development of e-Society Foundation Software* program of the Ministry of Education, Culture, Sports, Science and Technology, Japan. The author also thanks anonymous reviewers for their comments.

References

1. M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *International Conference on Very Large Databases*, 2000.

2. M. Benedikt, C. Y. Chan, W. Fan, R. Rastogi, S. Zheng, and A. Zhou. Dtd-directed publishing with attribute translation grammars. In *International Conference on Very Large Databases*, 2002.
3. V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *International Conference on Functional Programming*. ACM Press, 2003.
4. Y. Diao, P. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and scalable filtering of XML documents. In *International Conference on Data Engineering*, 2002.
5. Expat XML parser. `http://expat.sourceforge.net`.
6. Z. Fülöp. On attributed tree transducers. *Acta Cybernetica*, 5:261–280, 1980.
7. Z. Fülöp and H. Vogler. *Syntax-directed semantics—Formal models based on tree transducers*. Monographs in Theoretical Computer Science. Springer-Verlag, 1998.
8. H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *Symposium on Compiler Construction*, SIGPLAN Notices, 1984.
9. T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *International Conference of Database Theory*, volume 2572 of *LNCS*, 2003.
10. H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003.
11. O. Kiselyov. A better XML parser through functional programming. In *Practical Aspects of Declarative Languages*, volume 2257 of *LNCS*, 2002.
12. D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2), 1968.
13. C. Koch and S. Scherzinger. Attribute grammars for scalable query processing on XML streams. In *International Workshop on Database Programming Languages*, 2003.
14. A. Kühnemann. *Berechnungsstärken von Teilklassen primitiv-rekursiver Programmschemata*. PhD thesis, Technical University of Dresden, 1997. Shaker Verlag, Aachen.
15. B. Ludäscher, P. Mukhopadhayn, and Y. Papakonstantinou. A transducer-based XML query processor. In *International Conference on Very Large Databases*, 2002.
16. K. Nakano. Composing stack-attributed tree transducers. Technical Report METR–2004–01, Department of Mathematical Informatics, University of Tokyo, Japan, 2004.
17. K. Nakano and S. Nishimura. Deriving event-based document transformers from tree-based specifications. In *Workshop on Language Descriptions, Tools and Applications*, volume 44-2 of *Electronic Notes in Theoretical Computer Science*, 2001.
18. F. Neven. Extensions of attribute grammars for structured document queries. In *International Workshop on Database Programming Languages*, volume 1949 of *LNCS*, 1999.
19. S. Nishimura and K. Nakano. XML stream transformer generation through program composition and dependency analysis. *Science of Computer Programming*. To appear.
20. The Caml language homepage. `http://caml.inria.fr/`.
21. D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *EDBT Workshop on XML Data Management*, volume 2490 of *LNCS*, 2002.
22. SAX: The simple API for XML. `http://www.saxproject.org/`.
23. SAXON: The XSLT and XQuery processor. `http://saxon.sourceforge.net/`.
24. XML path language (XPath). `http://www.w3c.org/TR/xpath/`.
25. XSL transformations (XSLT). `http://www.w3c.org/TR/xslt/`.
26. Xalan-Java homepage. `http://xml.apache.org/xalan-j/`.
27. XP++: XML processing plus plus. `http://www.alphaworks.ibm.com/tech/xmlprocessingplusplus`.
28. XQuery 1.0: An XML query language. `http://www.w3.org/TR/xquery/`.
29. XTiSP: An implementation framework of XML transformation languages intended for stream processing. `http://xtisp.psdlab.org/`.

A Stack-Attributed Tree Transducers M_{parse}

Stack-attributed tree transducer(SATT) is an extension of attributed tree transducer(ATT). See [16] for detail. The major difference from ATT is that SATT can deal with a stack device for attribute values. Roughly speaking, SATT can simulate an ATT with an arbitrary number of attributes. In SATT, attributes are divided into two kinds of attributes, stack attributes and output attributes. Only stack attributes have a stack value which can be operated by *Head*, *Tail* and *Cons*. An XML parsing transformation M_{parse} is an example of SATT. An SATT M_{parse} is defined by

$$M_{parse} = (Syn, Inh, StSyn, StInh, \Sigma_{stream}, \Sigma_{tree}, s_{in}, \sharp, R)$$

where $Syn = \{p\}$, $Inh = \emptyset$, $StSyn = \{l\}$, $StInh = \emptyset$ and $R = R^{\sharp} \cup R^{\mathsf{S}_x} \cup R^{\mathsf{E}_x} \cup R^{\mathsf{Z}}$ with

$$R^{\sharp} = \{s_{in}(\pi) \to p(\pi 1)\},\ R^{\mathsf{S}_x} = \{p(\pi) \to \mathsf{N}_x(p(\pi 1), Head(l(\pi 1))),\ l(\pi) \to Tail(l(\pi 1))\}$$
$$R^{\mathsf{E}_x} = \{p(\pi) \to \mathsf{L},\ l(\pi) \to Cons(p(\pi 1), l(\pi 1))\},\ R^{\mathsf{Z}} = \{p(\pi) \to \mathsf{L},\ l(\pi) \to Nil\}$$

B Summary of XTiSP

We summarize our language XTiSP[29]. A program written in XTiSP specifies a transformation from an XML to an XML. A simplified syntax of XTiSP is defined by

$$\begin{aligned} exp = {} & f\texttt{(}exp\texttt{,}\cdots\texttt{,}exp\texttt{)} \mid \texttt{<}exp\texttt{>[}exp\texttt{]} \mid exp\texttt{;}exp \mid xpath \\ & \mid \texttt{if } exp \texttt{ then } exp \texttt{ else } exp \texttt{ endif} \\ & \mid \texttt{invite } xpath \texttt{ do } exp \texttt{ done} \\ & \mid \texttt{visit } xpath \texttt{ do } exp \texttt{ done} \end{aligned}$$

A symbol f represents a function which takes a fixed number of strings or boolean values and returns a string or a boolean value. An expression `<`*e1*`>[`*e2*`]` is used for constructing an element whose tag name and children are given by evaluation of *e1* and *e2*, respectively. An expression *e1*`;`*e2* returns a concatenation of evaluation results for *e1* and *e2*. A symbol *xpath* stands for an XPath expression which returns a collection of elements satisfying the XPath. An `if` clause represents a conditional branch in a usual way. `invite` and `visit` are used for XPath-based iteration mentioned below. An expression between `do` and `done` is called *iteration body*.

The transformation by XTiSP is defined by changing the *current node*. Initially, the current node is the root node of an input XML. When an iteration body for `invite`/`visit` is evaluated, the current node is changed into a node specified by XPath assigned with the iteration. An `invite` iteration returns a collection of results returned by evaluating the iteration body where the current node is changed into every node satisfying the XPath. An `visit` iteration returns a subtree of the input XML whose root is the current node, provided that every node v satisfying the XPath is replaced with the result returned by the iteration body where the current body is changed into v.

All XTiSP programs can be converted into $\mathrm{Xatt}^{t \to s}$. Roughly speaking, an XTiSP program is converted by associating each subexpression of the program, in particular `invite`/`visit` iteration, with a pair of synthesized and inherited attributes.