View metadata, citation and similar papers at core.ac.uk

brought t

provided by Elsevier -

ELSEVIER

Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 54 (2005) 257–290

Science of Computer Programming

www.elsevier.com/locate/scico

XML stream transformer generation through program composition and dependency analysis☆

Susumu Nishimura[a,*], Keisuke Nakano[b]

[a] *Department of Mathematics, Faculty of Science, Kyoto University, Kyoto 606-8502, Japan*
[b] *Department of Mathematical Engineering and Information Physics, University of Tokyo, Tokyo 113-8656, Japan*

Received 12 May 2003; received in revised form 27 January 2004; accepted 12 July 2004

Available online 13 August 2004

Abstract

XML stream transformation, which sequentially processes the input XML data on the fly, makes it possible to process large sized data within a limited amount of memory. Though being efficient in memory-use, stream transformation requires stateful programming, which is error-prone and hard to manage.

This paper proposes a scheme for generating XML stream transformers. Given an attribute grammar definition of transformation over an XML tree structure, we systematically derive a stream transformer in two steps. First, an attribute grammar definition of the XML stream transformation is inferred by applying a program composition method. Second, a finite state transition machine is constructed through a dependency analysis. Due to the closure property of the program composition method, our scheme also allows modular construction of XML stream transformers.

We have implemented a prototype XML stream transformer generator, called *altSAX*. The experimental results show that the generated transformers are efficient in memory consumption as well as in execution time.
© 2004 Elsevier B.V. All rights reserved.

☆ This work was conducted while the authors were at Research Institute for Mathematical Sciences, Kyoto University. Some of the preliminary results of this paper have appeared in the proceedings of LDTA'2001 Workshop on Language Descriptions, Tools and Applications [19].

* Corresponding author.
E-mail addresses: susumu@math.kyoto-u.ac.jp (S. Nishimura), ksk@mist.i.u-tokyo.ac.jp (K. Nakano).

0167-6423/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.07.001

1. Introduction

XML (Extensible Markup Language) is the W3C's standard [9] for representing tree-structured information in the conventional text format. In essence, an XML document is a human-readable text file whose content is delimited by nested *markup* tags, e.g., `<a> ... </a>`. It provides a simple yet flexible means to express tree-structured data. Due to this flexibility, XML is now widely accepted as the standard document format for structured data representation.

The standardization of XML has increased the opportunities of *XML transformations*. Suppose one possesses a list of items in an XML file and would like to view a selected set of the items on a Web browser. This can be realized by a pair of XML transformations, *filtering* and *view generation*: the former draws a subset of relevant items from the source data set and the latter converts the result of the former into XHTML, the rendering language for Internet Web browsers. The XML transformation technology is also vital for building Internet-based information systems: two different systems can communicate with each other by transforming their internal data representation into a common XML representation and vice versa.

This paper focuses on automatic generation of XML *stream transformations*, where the syntactic elements in the input XML stream, i.e., markup tags, character data (the embedded strings interleaved by markup tags), etc., are processed on the fly. XML stream transformation has a strong advantage in its memory efficiency. It allows progressive processing [16], i.e., partial results of a transformation may be output before the entire input is read. The memory allocated for partially output data would be reclaimed, and we could exploit the chance of a minimized memory use. (We notice that stream transformations do not necessarily improve the memory efficiency. Certain transformations are never improved by any transformation strategy. We will discuss this issue in Section 5.1.) Most of the current XML transformation systems and tools adopt the *tree transformation* model, which manipulates *XML trees*, the in-memory tree-structured XML representation. The tree transformation is memory inefficient in general, since the entire input must be loaded onto the memory before the transformation. As a counterpart to DOM (Document Object Model [7], the W3C standard of general purpose document manipulation API), a stream-based API called SAX (Simple API for XML) [24] has been developed and is widely used in those applications where memory efficiency is a major concern.

In contrast to its memory efficiency, stream transformation requires a translation of the intended tree-to-tree transformation into a corresponding imperative procedure that sequentially processes the input XML symbols. In the translation, programmers must be responsible not only for analyzing the structure of the input XML and but also for outputting the transformation result in a well-formed XML format. The translation is usually carried out by implementing a *state transition machine*, which controls the behavior of the sequential transformation by a set of transition rules over states. However, managing such a state transition system is a complicated task and thus could be a source of errors. It is also difficult to maintain such a state transition system: even a slight modification in the original transformation may lead to a substantial change in the state transition system.

The aim of this paper is to provide a convenient tool that automates the translation of XML tree transformation programs into stream transformation programs. For this,

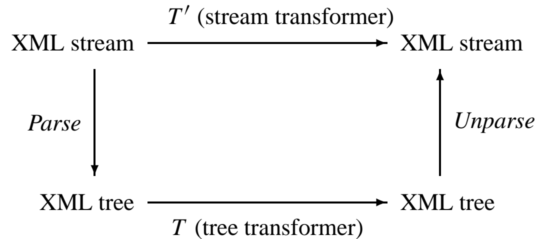


Fig. 1. A diagram for XML transformation.

we propose an algorithm that infers the state transition system, where each state transition is accompanied with a program code computing a partial output result. Based on this algorithm, we developed a prototype program generation system *altSAX*. The user can specify XML operations in terms of tree transformation, which are easier to understand and maintain. They can later be translated, by *altSAX*, to memory efficient stream transformations. This would provide a great help in developing XML stream transformation programs.

This paper describes the principles and techniques which constitute our program generator *altSAX*.

The fundamental idea behind our stream transformer generation scheme is the use of a program composition method drawn from the study of attribute grammars [11]. The diagram in Fig. 1 illustrates the idea. A complete XML tree transformation task comprises three subtransformations, each of which is characterized by a mathematical function: *Parse* parses the entire input to build an XML tree, T transforms the structure of the XML tree, and finally *Unparse* outputs the result of transformation as an XML stream. Synthesizing an XML stream transformer is nothing but fusing $\mathit{Unparse} \circ T \circ \mathit{Parse}$ into one single function (the function T' in the figure).

The program generation scheme based on the program composition method has a notable advantage that it is possible to compose two or more successive tree transformations into a single stream transformation program: suppose one has a series of tree transformations $T_1, \ldots, T_n$ to be successively applied to the input. If $T_1, \ldots, T_n$ meet a certain condition (given in Section 2.5), one can fuse $\mathit{Unparse} \circ T_n \circ \cdots \circ T_1 \circ \mathit{Parse}$ into one single stream transformation program by repeatedly applying the composition method. It allows XML transformations to be developed in a modular way.

The idea of using the program composition technique for stream transformer generation has been exploited by the authors [19]. Weakness in the language defining the transformations, however, discouraged us from putting it into practice. In the present paper, we consider a more realistic language. Most importantly, it can express *conditional transformations*, where the result of a transformation may depend on the contents in the source XML, e.g., tag names and character strings.

There is another proposal of a system for SGML document transformation generation, called SIMON [10]. However, SIMON concerns composition of tree transformations only and does not consider stream transformer generation.

Applying a program composition method solely does not provide a complete solution to the present problem, however. The result of program composition is merely a specification

of a transformation and does not serve as a stream transformation program. We need further elaboration to derive the state transition system underlying the transformation where an appropriate computation task is associated with each state transition. The main technical contribution of the present paper is to give an algorithm that systematically constructs such a state transition machine from the specification derived by the program composition method.

Our algorithm constructing the state transition machine is based on a dependency graph analysis, which differentiates our approach from existing ones. There are several proposals for streamed XML processing [2,6,25,16,12,22]. Most of these proposals are specialized to certain limited applications such as XML validation, filtering, etc. Our program generation scheme focuses on general purpose XML transformations. Also, they are all based on automata information induced from *schema information* or *node addressing pattern expressions*. A *schema information* confines XML documents to a restricted set of documents: it specifies the set of markup tag names, the markup occurrence order by regular expressions, and so on. *Node addressing pattern expressions* such as XPath [32] provide a means to address subtrees in the input XML by regular expression-like patterns.

The proposal by Ludäscher et al. [16] is aimed at general purpose streamed XML queries written in a subset of XQuery [33] language. We would argue that the class of transformations expressed by their method seems very close to ours but our method has an advantage in its simplicity. Their method (and other automata-based ones as well) can be quite complicated, particularly in the process of coupling each state transition with a computation task. This is because the automata are induced independently to the intended transformation and hence a rather complex analysis would be required to relate each state transition with an appropriate computation task. In contrast, our scheme analyzes against a program code and relates a computation task with each transition in a natural way.

In principle, our algorithm does not require any automata information for generating stream transformers. However, our experience says that the automata information is indispensable to obtain feasible XML stream transformation programs. Without automata information, the algorithm tends to produce a large number of states even when it is applied to transformations of a reasonable complexity. We therefore assume that the source XML comes with schema information, from which we can induce a corresponding automaton. We will show that a remarkable reduction in the number of states is achieved by taking a product of the generated state transition machine and the induced automaton.

The stream transformer generation scheme presented in this paper covers a large class of common XML transformations. It can deal with a certain set of basic transformations such as tag renaming, filtering, etc. Furthermore, our scheme also allows these basic transformations to be composed together. This makes it possible to develop XML stream transformation programs in a modular way. For example, we can construct a stream transformer that generates a stylized presentation of certain selected data items from the source XML just by defining two transformations, one for selecting relevant data items and the other for putting them in the presentation structure.

There are still some weaknesses in our stream transformation scheme, however. They mainly arise from the limitations of the current technology of attribute grammar composition. The most considerable weakness is that the input XML must not exceed a

presupposed maximum nesting depth. This indicates that a family of XML documents whose nesting depth is not known in advance, e.g., XHTML, cannot be the input to the transformation. Nevertheless we argue that, even with this bounded depth limitation, our scheme covers a significant class of XML transformations, say, those transformations which create a different presentation of the input data items. At the end of this paper, we will discuss the limitations of the present scheme and suggests possible cures, leaving definite solutions to future investigations.

The rest of this paper is organized as follows: Section 2 introduces attribute grammars as our transformation definition language and presents how XML stream transformation definitions are derived by using a composition method for attribute grammars. Section 3 introduces our transformer generation scheme. We first exhibit the fundamental idea of our transformer generation scheme, which is based on a dependency graph analysis technique, and then we refine the analysis technique to put it to work for practical XML transformations. Section 4 reports on a prototype implementation of our XML stream transformer generation tool. Finally Section 5 concludes the paper with several remarks on the limitations of our scheme and suggests possible solutions to them.

2. Stream transformer synthesis by attribute grammar composition

This section introduces *attribute grammars* (or AGs, for short) [15] as our transformation definition language and shows how a composition method for AGs is applied to derive the definition of XML stream transformation from a given specification of XML tree transformation.

Throughout the paper, the bare word "attribute" is reserved for AGs to avoid confusion. We always write "XML attributes" to refer to so-called attributes in XML.

2.1. The data model

XML trees are usually represented by *unranked trees*, where each tree node has an arbitrary (finite) number of child nodes. For example, an XML fragment

```
<a><b>text</b><c></c><d></d></a>
```

is represented by an unranked tree in Fig. 2(a). However, in the present paper, we use a different representation because AGs cannot describe computation over unranked trees.

We model XML trees by *binary trees*. A binary node in the tree, called an *element node*, corresponds to a pair of matching markup tags; a leaf node is either an *empty node*

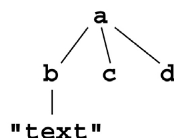


Fig. 2(a). Unranked tree representation.

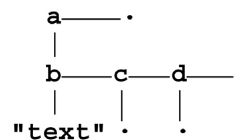


Fig. 2(b). Binary tree representation.

Fig. 2. Two representations for XML tree.

(represented by a dot in the figure) or a *content node*, which corresponds to a string in the XML document. An element node is attached with its tag name and a content node with its string data. Each left branch (right branch, resp.) expresses the parent–child relationship (the sibling relationship, resp.) For example, the binary tree representation of the above XML fragment is displayed in Fig. 2(b).

For the sake of simplicity, we omit XML attributes from the present paper. We can easily include XML attributes in the representation just by attaching XML attribute data to element nodes.

The binary tree representation of XML trees is modeled by the grammar specified by the following set of production rules.

$$S \rightarrow T, \quad T \rightarrow \texttt{Node}\ \$\text{tag}\ T\ T,$$
$$T \rightarrow \texttt{Content}\ \$\text{cdata}, \quad T \rightarrow \texttt{Empty},$$

where S is a start non-terminal symbol, T is a non-terminal representing binary trees, `Node`, `Content`, and `Empty` are terminals representing constructors of element nodes, content nodes, and empty nodes, resp. $tag and $cdata are terminals representing tag names and data strings, resp. The present paper does not consider so-called *mixed contents*, i.e., string data interleaved by markups, say, `<a>This <b>is a</b> mixed content</a>`. This is just for the sake of simplicity, and we can allow mixed contents by using an alternate `Content` production rule $T \rightarrow \texttt{Content}\ \$\text{cdata}\ T$.

The above XML fragment is expressed in the binary tree syntax as follows.

```
Node "a" (Node "b" (Content "text")
          (Node "c" Empty (Node "d" Empty Empty))) Empty
```

XML streams are represented by a list-like structure, as defined by the following set of production rules.

$$S \rightarrow E, \quad E \rightarrow \texttt{Begin}\ \$\text{tag}\ E, \quad E \rightarrow \texttt{End}\ \$\text{tag}\ E,$$
$$E \rightarrow \texttt{PCDATA}\ \$\text{cdata}\ E, \quad E \rightarrow \texttt{Nil}.$$

There are four terminals of list constructors, `Begin` of begin tags, `End` of end tags, `PCDATA` of data strings, and `Nil` of the end-of-list mark. The above XML fragment is expressed in the XML stream syntax as follows.

```
Begin "a" (Begin "b" (PCDATA "text" (End "b"
          (Begin "c" (End "c" (Begin "d" (End "d" (End "a" Nil)))))))
```

2.2. The attribute grammars

In this paper, we adopt attribute grammars (AGs) as a language for defining XML transformations.

An AG defines a syntax-directed computation over derivation trees of a context-free grammar, where every node is associated with a fixed set of values, called *attributes*. Each attribute is distinguished by a unique name and we write $N.a$ to denote the value of attribute a on a node N.

Let us explain how attributes are calculated through a simple example given in Fig. 3. The figure gives an AG definition for *Unparse*, which transforms the input XML tree

$S \rightarrow T$:
  $S.result = T.unparse$;
  $T.acc =$ `Nil`;

$T \rightarrow$ `Node` \$tag T_1 T_2:
  $T.unparse =$ `Begin` \$tag $T_1.unparse$;
  $T_1.acc =$ `End` \$tag $T_2.unparse$;
  $T_2.acc = T.acc$;

$T \rightarrow$ `Empty`:
  $T.unparse = T.acc$;

$T \rightarrow$ `Content` \$cdata:
  $T.unparse =$ `PCDATA` \$cdata $T.acc$;

Fig. 3. An unparsing transformation *Unparse* in AG.

(a derivation tree of the context-free language of binary trees defined in Section 2.1) into the corresponding stream representation.

Attributes on a node of an input tree are calculated according to *semantic rules* associated with the production rule used for the derivation at that node. For example, consider a node $N =$ `Node` "a" N_1 N_2. It matches the production $T \rightarrow$ `Node` \$tag T_1 T_2 where N, N_1, and N_2 match the non-terminal symbols T, T_1, and T_2, resp., while the tag name string "a" does the terminal symbol \$tag. By the first semantic rule of this production, the attribute $N.unparse$ is assigned the value `Begin` \$tag $N_1.unparse$ where the value of attribute $N_1.unparse$ is computed recursively for the subtree rooted at N_1. The second semantic rule $T_1.acc =$ `End` \$tag $T_2.unparse$ sets the attribute $N_1.acc$ to the value `End` "a" $N_2.unparse$ where the value of $N_2.unparse$ is also recursively computed. The third rule simply passes the attribute $N.acc$ to the attribute *acc* of the child node N_2.

The AG *Unparse* computes attributes on every node according to the semantic rules associated with that node. The attribute computation over an input tree `Node` "a" (`Content` "text") `Empty` by *Unparse* is visualized in Fig. 4. The edges of the derivation tree are depicted with thick lines and the nodes are annotated with corresponding non-terminal/terminal symbols; the arrows indicate the flow of computation among attributes. The overall result of attribute computation is returned through the attribute *result*, a special attribute distinguished from others. In this paper, we only consider AGs for transformations between languages, in particular, between those of XML binary trees and XML streams, defined in Section 2.1.

Throughout the paper, we consider only *well-formed* AGs [15,1]: an AG is well-formed if its computation over any derivation tree does not induce cyclic dependency among attributes.[1]

Attributes are classified into either *synthesized* or *inherited* ones. An attribute is called synthesized if its value on a node of a derivation tree is defined in terms of attributes on its child nodes; an attribute on a node is called inherited if it is defined in terms of attributes of its parent and/or sibling nodes. In the present *Unparse* example, *result* and *unparse* are synthesized attributes and *acc* is an inherited attribute. Inherited attributes

[1] The usual definition of well-formedness requires every attribute to be defined. In this paper, all attributes are defined either explicitly or implicitly: we assume every attribute missing explicit definition is assigned a special value representing errors.

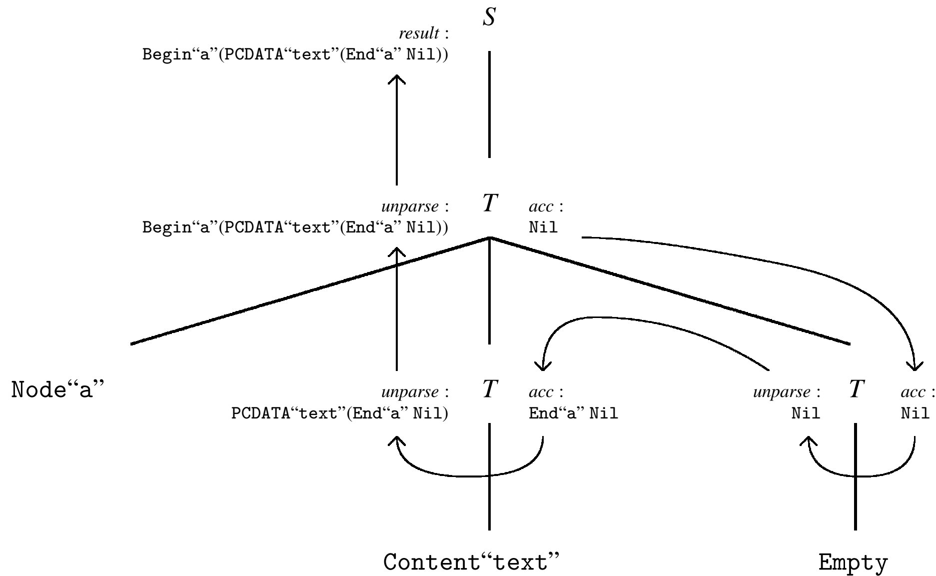


Fig. 4. Attribute evaluation by *Unparse*.

are useful for expressing context-dependent information. In the *Unparse* example, the inherited attribute *acc* attached on a node denotes the XML stream to follow that node.

There is another orthogonal classification of attributes, *syntactic/semantic* attributes. An attribute is called syntactic, if its value ranges over the language of transformation target; otherwise, it is called semantic. In the *Unparse* example, all attributes are syntactic. We will give an example of the use of semantic attributes in the following subsection.

In order to express conditional transformations, we allow a production rule to have *conditional semantic rules*, which select different sets of semantic rules depending on conditional tests. Conditional semantic rules are written in the following syntax.

IF *condition* THEN
  $SemanticRules_1$
ELSE
  $SemanticRules_2$
ENDIF.

The semantic rules $SemanticRules_1$ are selected if the expression *condition* evaluates to true; otherwise the other branch $SemanticRules_2$ is selected. The conditionals can be nested.

We note that a set of semantic rules may not define all the attributes. (That is, some attributes may not have a corresponding rule.) We assume that, when the semantic rule for an attribute a is missing, it is implicitly assigned a special value *undef*, representing errors. In other words, the attribute has an implicit rule $E.a = undef$. We also assume

that the value of an attribute whose definition refers to another undefined attribute is also undefined (i.e., it is given the special value *undef*).

2.3. Writing an XML tree transformation in AG

Let us show how we can define XML tree transformations through an example. Fig. 5(a) gives a sample input XML data, which is a (supposedly very long) list of stock quotes where each stock entry consists of a reference name `symbol` and its price information. The schema information for the stock quotes data is defined in Fig. 5(b), using W3C's Document Type Definition language (DTD) [9]. Each line `<!ELEMENT` *tag model*`>` in the DTD defines: (i) a markup name *tag* and (ii) the children of *tag*, occurring in the order specified by a regular expression *model*. The special notation `#PCDATA` stands for any string. For example, the DTD in Fig. 5(b) specifies that a `stock_quotes` element has zero or more `stock_quote` elements as its children, while a `stock_quote` element has four elements `symbol`, `price`, `change`, and `volume`, occurring in this order. (For a complete semantics of DTD, see [9].)

Fig. 6 gives an XML tree transformation **FILT**, a filtering transformation selecting only those stock items that gained more than 1.0 rise in price (i.e., the numeric value under the `change` element is bigger than 1.0) and more than 10^7 volume of trades.

In the definition of **FILT**, the attribute *xml* stands for the transformation result of the subtree rooted at that node. Whether a subtree is filtered or not is determined by the value of the *cond* attribute at every `stock_quote` node. The *cond* attribute on a tree node is set to

```
<?xml version="1.0"?>
 <stock_quotes>
   <stock_quote>
     <symbol>RIMS</symbol>
     <price>32.23</price>
     <change>2.17</change>
     <volume>13993600</volume>
   </stock_quote>
     and many more...
 </stock_quotes>
```

Fig. 5(a). A sample data.

```
<!ELEMENT stock_quotes (stock_quote)*>
<!ELEMENT stock_quote
          (symbol,price,change,volume)>
<!ELEMENT symbol (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT change (#PCDATA)>
<!ELEMENT volume (#PCDATA)>
```

Fig. 5(b). Schema in DTD syntax.

Fig. 5. Stock quotes list XML.

```
S  →  T:
    S.result = T.xml;

T  →  Node $tag T1 T2:
    IF ( $tag = "stock_quote" ) THEN
        IF ( T1.cond ) THEN
            T.xml = Node $tag T1.xml T2.xml;
        ELSE
            T.xml = T2.xml;
        ENDIF
    ELSE IF ( $tag = "change" ) THEN
        T.cond = T1.data > 1.0 & T2.cond;
        T.xml = Node $tag T1.xml T2.xml;
    ELSE IF ( $tag = "volume" ) THEN
        T.cond = T1.data > 10000000 & T2.cond;
        T.xml = Node $tag T1.xml T2.xml;
    ELSE
        T.cond = T2.cond;
        T.xml = Node $tag T1.xml T2.xml;
    ENDIF

T  →  Content $cdata:
    T.data = to_number($cdata);
    T.xml = Content $cdata;

T  →  Empty:
    T.cond = true;
    T.xml = Empty;
```

Fig. 6. A filtering transformation **FILT**.

false if and only if any of its subtrees does not meet the filtering condition, i.e., if it contains a subtree that is either `<change>`x`</change>` with $x \leq 1.0$ or `<volume>`x`</volume>` with $x \leq 10^7$. The *cond* attribute is computed by setting it to false as soon as a node that does not meet the filtering condition is encountered. (The rules for the conditional branches for `change` and `volume` nodes are responsible for this task.) The semantic rule of `Empty` production sets the initial value of the *cond* attribute to *true*. The string data under `change` and `volume` nodes are converted to numeric data by a built-in function `to_number`[2] and passes the numeric value to the parent node through the attribute *data*. (The semantic rules for the `Content` production do this.)

2.4. Parsing and unparsing transformations

In order to synthesize a stream transformation from a user-defined tree transformation, we need to combine the tree transformation with parsing and unparsing transformations. Users only need to define the tree transformation, leaving the definitions of parsing and unparsing to the transformer generation system.

[2] The value returned by `to_number` is undefined if its argument is not in a valid string representation of numeric data. This causes no problem in the present transformation because any element nodes with non-numeric data string (i.e., the `symbol` elements) never refer to the result of numeric conversion.

$$
\begin{array}{l}
S \;\rightarrow\; E:\\
\quad S.result \;=\; E.parse;\\[1em]
E \;\rightarrow\; \texttt{Begin}\ \$tag\ E_1:\\
\quad E.parse \;=\; \texttt{Node}\ \$tag\ E_1.parse\ E_1.stk_1;\\
\quad E.stk_i \;=\; E_1.stk_{i+1}; \quad (\, i = 1, \ldots, d-1\,)\\[1em]
E \;\rightarrow\; \texttt{End}\ \$tag\ E_1:\\
\quad E.parse \;=\; \texttt{Empty};\\
\quad E.stk_1 \;=\; E_1.parse;\\
\quad E.stk_{i+1} \;=\; E_1.stk_i; \quad (\, i = 1, \ldots, d-1\,)\\[1em]
E \;\rightarrow\; \texttt{PCDATA}\ \$cdata\ E_1:\\
\quad E.parse \;=\; \texttt{Content}\ \$cdata;\\
\quad E.stk_i \;=\; E_1.stk_i; \quad (\, i = 1, \ldots, d\,)\\[1em]
E \;\rightarrow\; \texttt{Nil}:\\
\quad E.parse \;=\; \texttt{Empty};
\end{array}
$$

Fig. 7. A parsing transformation *Parse* in AG.

We have already seen the definition of *Unparse* in Fig. 3. The definition of the parsing transformation *Parse* is technically more involved. The source of complication is that, since XML is defined as a context-free language, we need a stack device to parse it. Unfortunately the AG composition method, which will be discussed in Section 2.5, is not capable of dealing with stack or similar devices.

In the present paper we circumvent this problem by assuming that the input XML has a bounded nesting depth. With this assumption, we can define the parsing transformation as in Fig. 7, by simulating a stack of finite depth d by d attributes $stk_1, \ldots, stk_d$, where d is the maximum nesting depth of the input (with the outermost nest level being 1).

Note that the definition of *Parse* leaves the matching of begin tags and end tags unchecked: it assumes that all the markup tags of the input are balanced. Though we can give a more intricate variation of *Parse* that strictly checks the balance, we adopted the present definition for the sake of simplicity. This simplification does not cause a problem in practice, since a stream-based XML parser frontend[3] such as SAX is responsible for checking the tag balance.

2.5. The AG composition method

Ganzinger and Giegerich [11] have proposed a method, called *descriptional composition*, for composing multiple AGs into a single one. Their method provides a completely mechanized composition scheme, which is driven by a set of rewriting rules. Basing on their method, Boyland and Graham [4] have proposed an extension for dealing with semantic attributes and conditionals. A summary of the descriptional composition algorithm is given in Appendix A.

[3] This is a jargon in the XML community. The word “stream-based XML parser frontend” stands for a *lexer*, which produces a stream of XML syntactic elements from a bare input XML.

AG composition can be applied to a limited class of AGs satisfying the so-called *SAMODUR (syntactic at most once dynamic use requirement)* condition [4], which is an extension of the *single-use* condition proposed in [11]. An AG is SAMODUR if every syntactic attribute is referenced at most once in every different set of semantic rules selected by conditionals. That is, the same syntactic attribute may be referenced more than once from different conditional branches but not from the same branch. It is even allowed that an attribute is not referenced at all. On the other hand, semantic attributes can be referenced an arbitrary number of times.

In what follows, we assume that all AG definitions meet the SAMODUR property. That is true for all the examples (*Unparse*, *Parse*, and **FILT**) presented so far.

The next theorem is vital for the synthesis of XML stream transformations.

Theorem 2.1 (*Closure property, Boyland and Graham [4]*). *Two SAMODUR AGs can always be composed by the descriptional composition. The result of the composition is also a SAMODUR AG.*

The closure property is necessary for the transformer synthesis procedure, since an XML stream transformation is obtained as the result of repeated compositions of three transformations, *Parse*, the user-defined tree transformation, and *Unparse*. In addition, we can greatly benefit from this property, as it further allows us to compose two or more successive user-defined tree transformations. Again, thanks to the closure property, we can obtain a synthesized XML stream transformation for the transformation $Unparse \circ T_n \circ \cdots \circ T_1 \circ Parse$, if all the XML tree transformations $T_1, \ldots, T_n$ are SAMODUR.

The present paper does not give any further detailed account for the AG composition. The detail of the AG composition is a separate topic, and readers should be able to follow the subsequent discussions just by acknowledging the facts above. Interested readers are deferred to the literature given in Appendix A.

3. The transformer generation scheme

In the previous section, we have discussed how to synthesize an XML stream transformer using an AG composition. The synthesized result, however, is just a specification and is not directly executable as an XML stream transformer. This section presents a scheme for converting the synthesized AG specification into an executable stream transformer.

3.1. Transformer generation by attribute dependency analysis

The intuition behind our scheme is that an XML stream transformer is a finite state transition machine where each transition is accompanied with a computation task and an output action. The essence of our scheme lies in an algorithm that constructs such a finite state transition machine. The algorithm is comprised of two phases. First, it constructs a state transition machine by tracing how the attribute dependency (i.e., the define-use dependency among attributes) changes for every possible input, where each state is represented by a different pattern of attribute dependency. Second, it calculates

```
S → T:
  S.result = T.xml;

T → Node $tag T1 T2:
  T.xml = Node "a" (Node "b" T1.xml Empty) T2.xml;

T → Empty:
  T.xml = Empty;
```

Fig. 8(a). A simple transformation.

```
S → E:
  S.result = E.s1;
  E.h1 = Nil;

E → Begin $tag E1:
  E.s1 = Begin "a" (Begin "b" E1.s1);
  E.s2 = E1.s3;
  E1.h1 = End "b" (End "a" E1.s2);
  E1.h2 = E.h1;
  E1.h3 = E.h2;

E → End $tag E1:
  E.s1 = E.h1;
  E.s2 = E1.s1;
  E.s3 = E1.s2;
  E1.h1 = E.h2;
  E1.h2 = E.h3;

E → Nil:
  E.s1 = E.h1;
```

Fig. 8(b). The synthesized transformation.

what to compute (and what to output) for each transition, using a special representation of attribute values.

The two phases are explained in the following sections. As an example, we consider the tree transformation in Fig. 8(a), which recursively replaces every markup <*tag*> ... </*tag*> with <a> ... . Fig. 8(b) gives the result of the program composition for $Unparse \circ T \circ Parse$. To simplify the presentation, we assumed the maximum nesting depth of the input is 2 and omitted the production rules for data strings and their associated semantic rules.

The explanation below is a summary of an algorithm proposed in our previous work [19], which only applies to the cases where the transformation definition does not involve any conditionals or semantic attributes. We first explain our algorithm in this simplified setting in order to highlight our fundamental idea in the finite state transition machine construction.

3.1.1. *Generating state set and transition rule*

Our program generation scheme first constructs a finite state transition machine. As a device for attribute dependency analysis, it utilizes *attribute dependency graphs*.

Each state of the state transition machine is identified by the pattern of an attribute dependency graph. The patterns of attribute dependency graphs are derived from the semantic rules of each production rule. For example, the attribute dependency graphs for the synthesized AG of Fig. 8(b) are given in Fig. 9. For each production rule $E \rightarrow \cdots E_1$, attributes annotating E are placed on the upper bar, and those annotating E_1 on the lower bar. Synthesized attributes are placed on the left side and inherited attributes on the right side of each bar. Each attribute a on a bar is marked by a black triangle if it is defined

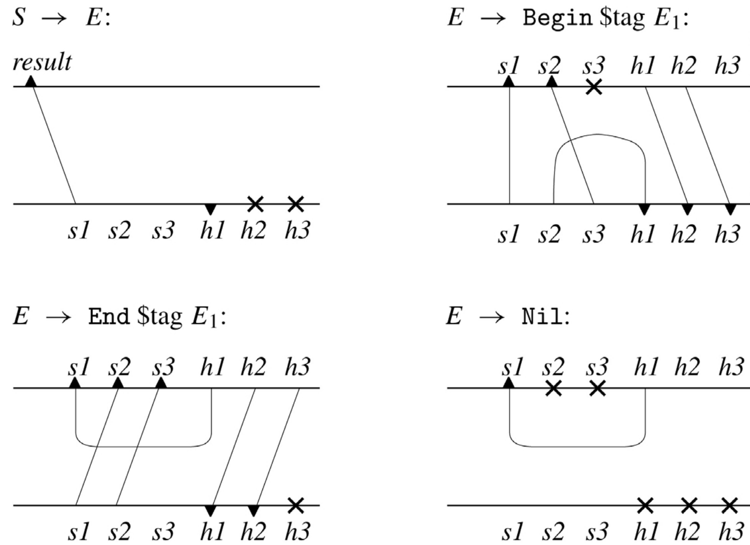
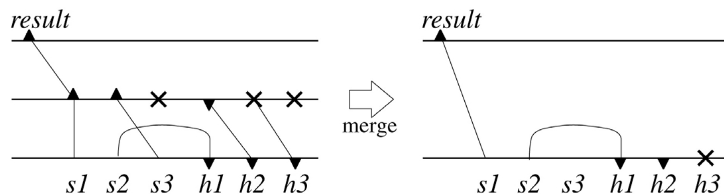


Fig. 9. Attribute dependency graphs.

(i.e., if there is a rule defining the value of *a*); otherwise, it is marked by a cross. Note that the inherited attributes on the upper bar and the synthesized attributes on the lower bar are never marked, because they are not defined but just referenced by other attributes in the semantic rules.

Each edge in a graph represents a define-use relationship for a pair of attributes. If a defined attribute has a dependency (i.e., it refers to another attribute value), an edge to the referred attribute is drawn. If an attribute is defined but has no dependency (i.e., if the attribute has no reference to other attributes and is assigned an XML symbol stream terminated by `Nil`), no edge is drawn.

The construction of a finite transition machine enumerates all the states (i.e., the dependency patterns), beginning with the graph of the start production as the initial state. It examines all the possible inputs at every state to find new states. For example, suppose a begin tag is read from the input at the initial state. To deliver the new state, we paste the graph of the `Begin` production under that of the start production (the left figure below) and *merge* them into a single graph (the right figure below), by erasing the middle bar and taking the transitive closure of dependency edges. Any attribute that transitively depends on an undefined attribute (e.g., the attribute *h3* in the figure) is marked undefined.



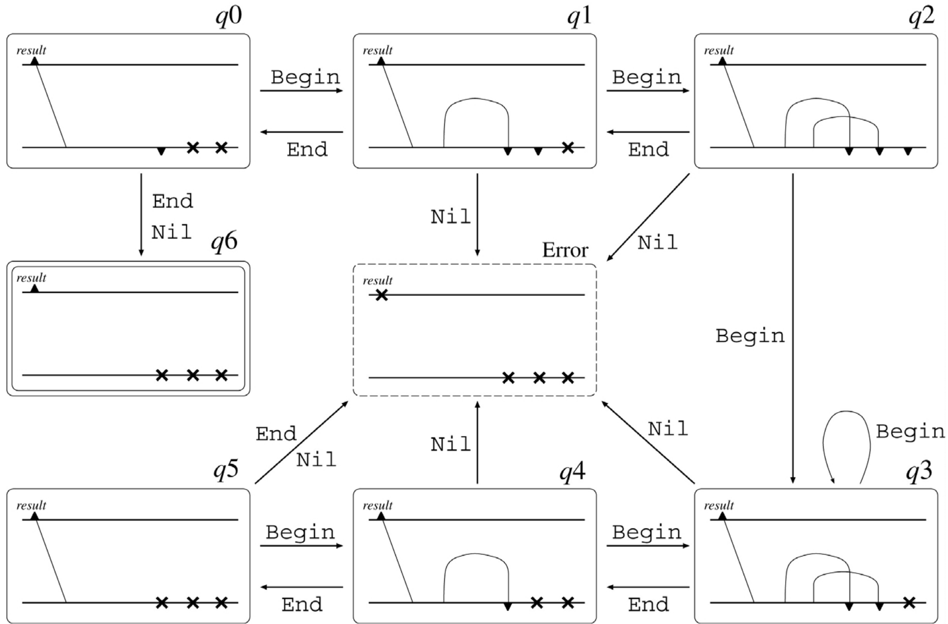
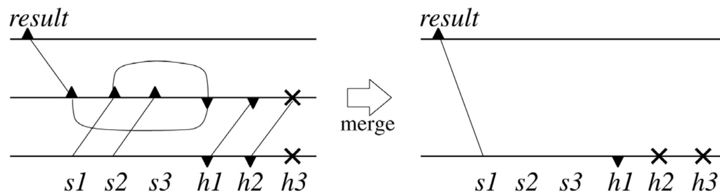


Fig. 10. Finite state transition machine.

Suppose an end tag comes next. Then, similar to the above, the graph of End is pasted and then merged.



Here notice that the resulting graph has the same pattern as that of the start production. This indicates that reading a begin tag and then an end tag at the initial state brings the transition machine back to the initial state.

Repeating the above process for every possible input until no new state is added by the process, we obtain the state transition diagram in Fig. 10, where $q0$ is the initial state, and the final and error states are designated by double-lined and dashed-lined ovals, resp. The state $q6$ is a final state, since the attribute *result* is defined and has no dependency, which indicates that the transformation is finished. The state Error is an error state, since the attribute *result* is undefined, which indicates an error during transformation.

We note that the construction of such a finite state machine always terminates, since the number of different dependency patterns is finite. However, this is not the case for the algorithm that deals with conditionals and semantic attributes, as we will discuss in Section 3.3.

3.1.2. Calculating computation tasks

After constructing the finite state transition machine, we calculate the computation tasks and output actions associated with each state transition.

In the evaluation of attribute values against an XML stream, some attribute values may be determined only after more inputs are read. For example, in the state transition from $q1$ to $q2$, the value of the attribute *h1* cannot be fully determined, since the semantic rule $E_1.h1$ = `End` "`b`" (`End` "`a`" $E_1.s2$) defined for the production `Begin` refers to the attribute $E_1.s2$, whose value is yet to be determined.

Though we cannot give a fixed value to $E_1.h1$, we can instead assign it the "rest of the computation", which expresses how to compute the attribute when the referenced attribute value is fixed. We use a special *compositional representation* to express the "rest of the computation". The compositional representation R is a composition of *constructors* C, as defined by the following grammar:

$$R ::= X \mid C \mid R \circ R$$
$$C ::= \texttt{Begin}\ tag\ [] \mid \texttt{End}\ tag\ [] \mid \texttt{PCDATA}\ cdata\ [] \mid \texttt{Nil} \mid [[]$$

where X stands for a variable that ranges over the set of compositional representations, *tag* and *cdata* represent a constant string for tag name and data string, resp., and [] is a *hole*, a placeholder to be eventually filled with an expression. Each constructor C is intended to represent a function, written $\lambda x.C[x]$ in λ-notation [3], where $C[x]$ stands for the expression obtained by filling the hole [] with x. (`Nil` is also interpreted as a function $\lambda x.$`Nil`, which takes an argument that is never referenced.) For the sake of readability, we write `Begin` *tag* for `Begin` *tag* [], `End` *tag* for `End` *tag* [], `PCDATA` *cdata* for `PCDATA` *cdata* [], and `Id` for []. The constructors can be arbitrarily combined using the function composition operator $_ \circ _$. For example, in the transition from $q1$ to $q2$, the attribute *h1* receives a compositional representation `End` "`b`" $\circ$ `End` "`a`".

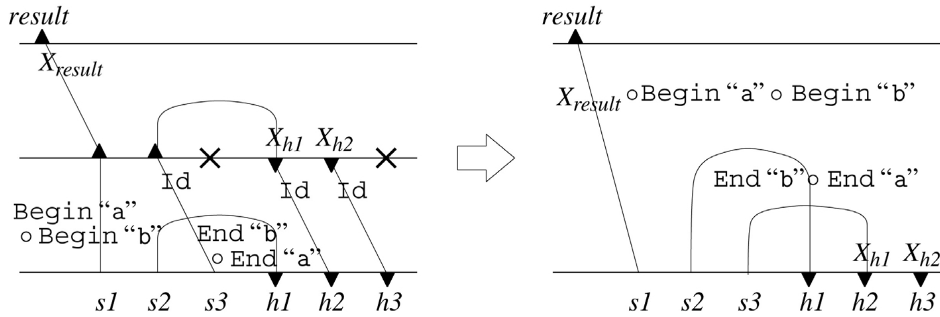
Function composition is associative (i.e., $(R_1 \circ R_2) \circ R_3 = R_1 \circ (R_2 \circ R_3)$) and has `Id` as a unit element (i.e., $\texttt{Id} \circ R = R \circ \texttt{Id} = R$). We may use these algebraic laws to simplify expressions. In particular we would conventionally write a compositional representation as $C_1 \circ \cdots \circ C_n$, omitting parentheses.

We can alternatively express the compositional representation in a *concatenation list* reading the function composition operator as a list concatenation operator, `Id` as an empty list, and other constructors as corresponding XML symbols. More formally, the concatenation list representation $\lceil R \rceil$ for a compositional representation R is defined by

$$\lceil \texttt{Id} \rceil = [], \quad \lceil X \rceil = X, \quad \lceil C \rceil = [C], \quad \lceil C_1 \circ C_2 \rceil = C_1 @ C_2,$$

where [] is an empty list, $[C]$ is a singleton list of a constructor C, and _@_ is a list concatenation operator. The concatenation list representation provides a conventional means to implement the compositional representation in a non-functional programming language.

Let us explain by an example how the computation task associated with each transition is calculated. The figure below visualizes a process of graph merger for the state transition from $q1$ to $q2$, where we annotate each dependency edge with a compositional representation.



The attributes in the graph for the `Begin` production (the lower graph on the left) are attached with concrete compositional representations derived from the semantic rules. In contrast, each attribute a in the graph of the state $q1$ (the upper graph on the left) is attached a variable X_a. This is because the assigned compositional representations in a particular state may vary depending on the transition path along which the state is reached. The varying representation assigned to each attribute is thus represented by a variable.

The new representation assigned in the next state (in this case $q2$) is calculated along the transitive closure path of edges. For example, to calculate the compositional representation of *h2* in $q2$, we compose the representations assigned to the graph edges along the transitive closure path and obtain the new representation $\mathtt{Id} \circ X_{h1} \circ \mathtt{Id} = X_{h1}$.

According to the analyses above, we can generate a complete XML stream transformation program. Fig. 11 shows a pseudo program code of the generated transformer. In the program, the state transitions are expressed by procedure calls to $q0, \ldots, q5$, corresponding to the states in Fig. 10. Each procedure receives the compositional representations of attributes *h1*, *h2*, *h3* (through the arguments X_{h1}, X_{h2}, X_{h3}, resp.) from the procedure corresponding to the previous state. The *MAIN* procedure initiates transformation by a call to the procedure $q0$, with compositional representations derived from the semantic rules of start production being the arguments of the procedure call. The symbol *undef* in the program denotes a special *undefined* value. The **error** statement aborts program execution and the **exit** statement terminates execution successfully.

In order to minimize the memory usage, the generated program writes the accumulated partial result of the transformation to the *result* attribute for every transition. The **output** statement does this task: it receives a compositional representation $C_1 \circ \cdots \circ C_n$ and writes a sequence of corresponding XML symbols for $C_1, \ldots, C_n$ in this order. Since the **output** statement flushes the partial transformation result for each transition, we can assume the variable X_{result} is always assigned `Id` in every state.

3.2. Dealing with data values and conditionals

The program generation scheme we have explained so far cannot compute over data values embedded in the input XML, i.e., tag names and character data strings. This limitation rules out many practical XML transformations, e.g., the filtering transformation given in Fig. 6: it needs to inspect the embedded data values in order to judge the relevance of data items.

```
procedure MAIN()
    call q0(Nil, undef, undef)

procedure q0(Xh1, Xh2, Xh3)
    case next_XML_symbol of
      Begin $tag  ⇒  output Begin "a" ∘ Begin "b";
                                  call q1(End "b" ∘ End "a", Xh1, undef);
      End $tag    ⇒  output Xh1;  exit;
      Nil         ⇒  output Xh1;  exit;

procedure q1(Xh1, Xh2, Xh3)
    case next_XML_symbol of
      Begin $tag  ⇒  output Begin "a" ∘ Begin "b";
                                  call q2(End "b" ∘ End "a", Xh1, Xh2);
      End $tag    ⇒  output Xh1;  call q0(Xh2, Xh3, undef);
      Nil         ⇒               error;

procedure q2(Xh1, Xh2, Xh3)
    case next_XML_symbol of
      Begin $tag  ⇒  output Begin "a" ∘ Begin "b";
                                  call q3(End "b" ∘ End "a", Xh1, undef);
      End $tag    ⇒  output Xh1;  call q1(Xh2, Xh3, undef);
      Nil         ⇒               error;

procedure q3(Xh1, Xh2, Xh3)
    case next_XML_symbol of
      Begin $tag  ⇒  output Begin "a" ∘ Begin "b";
                                  call q3(End "b" ∘ End "a", Xh1, undef);
      End $tag    ⇒  output Xh1;  call q4(Xh2, undef, undef);
      Nil         ⇒               error;

procedure q4(Xh1, Xh2, Xh3)
    case next_XML_symbol of
      Begin $tag  ⇒  output Begin "a" ∘ Begin "b";
                                  call q3(End "b" ∘ End "a", Xh1, undef);
      End $tag    ⇒  output Xh1;  call q5(undef, undef, undef);
      Nil         ⇒               error;

procedure q5(Xh1, Xh2, Xh3)
    case next_XML_symbol of
      Begin $tag  ⇒  output Begin "a" ∘ Begin "b";
                                  call q4(End "b" ∘ End "a", undef, undef);
      End $tag    ⇒               error;
      Nil         ⇒               error;
```

Fig. 11. The generated XML transformer.

For a practical application of our program generation scheme, AGs must be reinforced with enriched features, namely conditionals and semantic attributes introduced in Section 2.3. However, it is not straightforward how the program generation scheme can be made compatible with those new features. They would disrupt our program generation scheme in the following ways.

- When conditionals are involved in computation, the attribute dependency pattern cannot be always determined uniquely for every input symbol, since a different set of semantic rules (and hence a different dependency pattern) may be selected depending on the run-time value of conditional predicates. We need a fundamental refinement on the strategy for the construction of a finite state transition machine, whose states are identified by dependency patterns.
- In the presence of semantic attributes, the dependency may be non-linear. That is, computation for an attribute may depend on two or more other attributes (e.g., given a semantic rule $E.a = \texttt{Begin}\ E_1.t\ E_1.b$, the attribute a depends on two attributes t and b). The "rest of the computation" of a semantic attribute with non-linear dependency cannot be expressed in the form of linear compositional representation defined in Section 3.1.

We propose a remedy to these problems by refining the state enumeration process in the finite state transition machine construction. Instead of looking into just one next input symbol, we examine the state reached by a sequence of input symbols, expecting that reading several more input symbols (called *lookahead* symbols) would provide enough information to resolve the problems mentioned above.

The refined state enumeration process is driven by a scheme called *delayed graph merging*, in which graph merging is delayed until enough information is provided by the lookahead symbols. In the following, we explain how the problems mentioned above are resolved using a delayed graph merging scheme.

3.2.1. Delayed graph merging for conditionals

Let us first show, through an example, how the dependency pattern is inferred when conditionals are involved.

Suppose we obtain an AG specification of an XML stream transformation, as shown below. (Note that this is a non-sensical example taken for explanatory purposes only.)

```
S → E :
  S.result = E.xml;

E → Begin $tag E1 :
  IF ( E1.s ) THEN
      E.xml = Begin "a" E1.xml;
      E.s = ($tag = "a");
  ELSE
      E.xml = Nil;
      E.s = ($tag = "a");
  ENDIF
```

A state search path for the example transformation that would be examined by the refined state enumeration procedure is illustrated in Fig. 12. The graph Fig. 12(a) represents the initial state. Suppose the first symbol is a begin tag. This yields a sequence of two graphs (Fig. 12(b)), whose merger is delayed until the value of the attribute $E_1.s$, referenced by the conditional predicate, is fixed. In the figure, an attribute referenced by a conditional predicate is marked by an asterisk, and an attribute whose dependency is not yet determined is marked by a white triangle. Every piece of graph that has an attribute

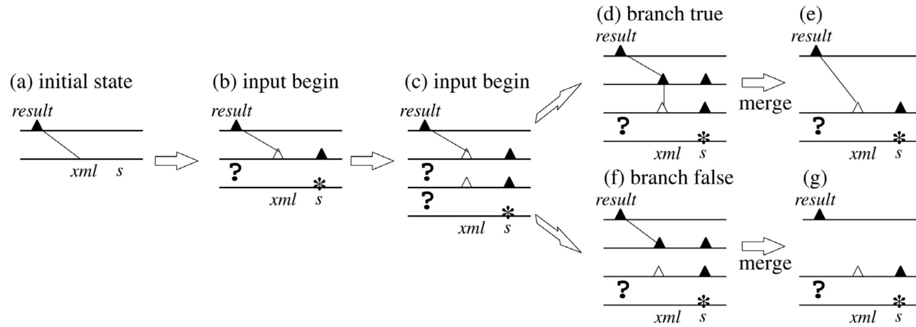


Fig. 12. Delayed graph merging process for conditional.

with undetermined dependency is marked by a question mark and is left unmerged. Note that the attribute s on the middle bar in Fig. 12(b) has a fixed dependency independent of the conditional predicate and is marked by a black triangle.

Now suppose the second input is again a begin tag. We then obtain the graph sequence in Fig. 12(c), where the conditional construct for the previous input is ready to select a conditional branch. There are two possible dependency patterns to select, namely one for the true branch and the other for the false branch (Fig. 12(d) and (f), resp.). At this moment, the procedure merges any consecutive sequence of graphs which has a fixed dependency. In both cases, the two uppermost graphs in the sequence are merged, and the new states Fig. 12(e) and (g) are generated.

According to this extension, each state is now represented by a sequence of graphs, where graphs with fixed dependency are identified by the dependency pattern, while the rest of the graphs are identified by the production rule from which they are derived. For example, Fig. 12(b) and (e) represent the same state because both of the upper graphs have the same pattern and the lower ones are derived from the `Begin` production.

Note that Fig. 12(c), (d), and (f) are intermediate representations and are not counted as new states. The program generator generates a program which, when it reads a begin tag at the state (b), evaluates the predicate expression $E_1.s$ and makes a conditional transition to either (e) or (g) depending on the value of the predicate expression.

3.2.2. Delayed graph merging for semantic attributes

In this paper, we circumvent the problem of a non-linear dependency as follows: when there is a graph with a non-linear dependency, the state enumeration procedure leaves the graph and continues analysis for lookahead symbols, expecting the non-linear dependency to eventually converge to a linear (or null) dependency. The delayed graph merging is utilized for this task.

Let us show how the delayed graph merging resolves non-linear dependency. Consider the following example of XML stream transformation. (This is an example taken for explanatory purposes, too.)

$$S \rightarrow E:$$
$$S.result = E.xml;$$

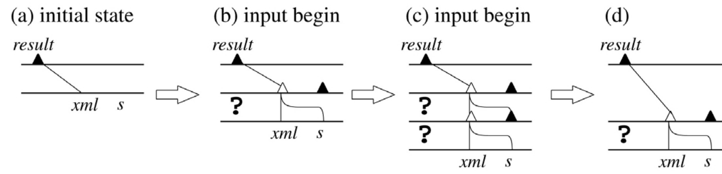


Fig. 13. Delayed graph merging process for non-linear dependency.

$E \rightarrow$ `Begin` \$tag E_1:
  $E.xml$ = `Begin` $E_1.s$ $E_1.xml$;
  $E.s$ = \$tag;

A state search path for the example transformation is illustrated in Fig. 13. The graph in Fig. 13(a) represents the initial state. Suppose the first symbol is a begin tag. This yields a sequence of two graphs (Fig. 13(b)), whose merging is delayed because the value of attribute $E.xml$ depends on two different attributes, namely $E_1.xml$ and $E_1.s$. In the figure, an attribute that has non-linear dependency is marked by a white triangle. Every graph that has non-linear dependency is marked by a question mark, which indicates merging this graph with others is postponed until a merger of any consecutive sequence of graphs makes every non-linear dependency converge to a linear (or null) dependency.

Now suppose the second input is again a begin tag. We then obtain the graph sequence of Fig. 13(c). Here we can see that the merger of the upper two graphs resolves the non-linear dependency in the middle graph. Hence we merge them to obtain a sequence of graphs of Fig. 13(d) as a new state.

Here we notice that the refined state set enumeration process may not terminate. When there is a state search path along which not enough information is provided by any finite number of lookaheads (i.e., when references from a conditional predicate never resolve and/or some non-linear dependency never converges to a linear or null dependency), an everlasting sequence of graphs waiting for the dependency resolution will be produced.

After the construction of the finite state transition machine, the computation task associated with each state transition is calculated. Since the graphs merged by the delayed graph merging scheme involves dependency edges with at most one single dependency, we can utilize the compositional representation $C_1 \circ \cdots \circ C_k$ $(k \geq 1)$ introduced in Section 3.1.2, with the following extension to the set of constructors:

$$C ::= \cdots \mid E \quad (E \text{ has at most one occurrence of } [\,])$$
$$E := [\,] \mid p(E_1, \ldots, E_n) \quad (n \geq 0)$$

where p stands for built-in functions and constants (a constant is a built-in function with parity 0) such as arithmetics, boolean operations, etc. We do not allow the holes to appear in the tag name and data string positions of constructors `Begin` *tag* [], `End` *tag* [], and `PCDATA` *cdata* []. The computation associated with each state transition is calculated similarly to that in Section 3.1.2.

The compositional representation for XML symbol constructors is dealt with in the same way as described in Section 3.1.2, during the execution of the transformation

program: the compositional representation assigned for the special attribute *result* is output for every transition; for other attributes, it is composed with other representations before it is passed to the next state. Constant folding is applied to a compositional representation $C_1 \circ \cdots \circ C_n$, if C_i's are extended constructors of the form $p(E_1, \ldots, E_n)$ and C_n has no hole (i.e., C_n represents a constant value). For example, the representation $2*[] \circ 1+3*[] \circ 5$ is folded into 32, which is obtained by calculating $2 * (1 + 3 * 5)$.

Note that the scheme we proposed above is just one of the possible solutions. There could be other different ways to execute the AG specification. For example, one may just leave the execution of the obtained AG entirely to the general lazy evaluation scheme as Johnsson did [14], which also minimizes the memory usage. However, our method has a strong advantage over lazy evaluation. As we will discuss in Section 3.3, we can exploit more opportunities of optimizations through static analysis into the obtained AG specification. Another reason why we choose not to rely on the general lazy evaluation scheme is that it is available only in a limited variety of programming languages such as Haskell [27]. Our scheme can be implemented in a wide range of conventional languages such as Java, on which we implemented our prototype program generation system.

A more moderate improvement to the present scheme would be to allow non-linear dependency. It would make the state enumeration procedure more likely to terminate. On the other hand, the compositional representation must be refined so that it can express the "rest of the computation" that depends on two or more values and the generated program must be responsible to manage it. This would greatly complicate the program generation phase. For the sake of simplicity, we adopted the conservative approach that enforces the linearity of dependency in the current implementation. The non-termination problem can be compensated by a refined state enumeration process discussed below.

3.3. State space compaction by schema information

The program generation scheme discussed so far tends to generate a large (and often infinite) number of states. The reason for the state explosion is that the state enumeration algorithm counts all the dependency patterns reached by arbitrary inputs as states. This causes the resulting finite state transition machine to include many irrelevant states, which are never reached by any *well-typed* inputs, i.e., the inputs that respect the given schema information.

In the rest of this section, we show how it is possible to reduce the state space (and sometimes to turn an infinite state space into a finite one) by pruning irrelevant states according to the schema information.

In order to utilize the schema information for state space compaction, we need a finite automaton that accepts the set of all XML streams which are confined to the given schema. We assume that the schema information is given by a DTD [9]. It is easy to convert a DTD into a finite automaton if the DTD is non-recursive, but it is not possible if the given DTD is recursive (i.e., if it is allowed to have nested tags of the same name, say, `<a> ... <a> ... </a> ... </a>`) [25]. This is because recursive DTDs allow an arbitrary number of nestings and in that case no finite automaton can check the balance of begin tags and end tags. However, when the input document has a bounded nesting depth (and this is what we

$$\gamma(d, \texttt{EMPTY}) = \varepsilon$$

$$\gamma(d, tag) = \begin{cases} \texttt{Begin}\ tag;\ \gamma(d-1, r);\ \texttt{End}\ tag & \text{if } d > 0, \texttt{<!ELEMENT}\ tag\ r\texttt{>} \text{ is in the given DTD, and } \gamma(d-1, r) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$$\gamma(d, \texttt{\#PCDATA}) = \texttt{PCDATA}$$

$$\gamma(d, r_1; r_2) = \begin{cases} \emptyset & \text{if } \gamma(d, r_1) = \emptyset \text{ or } \gamma(d, r_2) = \emptyset \\ \gamma(d, r_1); \gamma(d, r_2) & \text{otherwise} \end{cases}$$

$$\gamma(d, r_1 + r_2) = \begin{cases} \emptyset & \text{if } \gamma(d, r_1) = \emptyset \text{ and } \gamma(d, r_2) = \emptyset \\ \gamma(d, r_1) & \text{if } \gamma(d, r_2) = \emptyset \\ \gamma(d, r_2) & \text{if } \gamma(d, r_1) = \emptyset \\ \gamma(d, r_1) + \gamma(d, r_2) & \text{otherwise} \end{cases}$$

$$\gamma(d, r*) = \begin{cases} \varepsilon & \text{if } \gamma(d, r) = \emptyset \\ \gamma(d, r)* & \text{otherwise} \end{cases}$$

Fig. 14. Conversion of DTD to regular expression.

have assumed), we can construct a finite automaton for documents with a bounded nesting depth as follows.

First we convert the given DTD into a regular expression. The set of regular expressions is defined by the following BNF:

$$r ::= \emptyset \mid \varepsilon \mid a \mid r; r \mid r + r \mid r*,$$

where $\emptyset$ represents the empty set of strings (i.e., no string is matched), ε the null string, a the string that contains the single alphabet symbol, $r_1; r_2$ concatenation of two strings, $r_1 + r_2$ alteration of either of the two strings, and $r*$ zero or more repetition of strings. The alphabet of the resulting regular expression is `Begin` $tag_1, \ldots,$ `Begin` tag_n, `End` $tag_1, \ldots,$ `End` tag_n, `PCDATA`, `Nil`.

The regular expression to obtain is `Begin` $tag;\gamma(d-1, tag)$;`End` tag;`Nil`, where the function γ is defined in Fig. 14, d is the maximum nesting depth, and tag is the root tag name specified in the DTD. The idea behind this conversion is that any input beyond the presupposed maximum nesting d is ignored and is replaced by $\emptyset$, which represents an "out of nesting depth" error.

Once a regular expression is obtained, the standard algorithm for converting a regular expression into a deterministic finite automaton applies [13]. Let $M = (Q, \mathcal{A}, \delta, q_0, F)$ be the obtained automaton, where Q is the finite set of states, $\mathcal{A} = \{$`Begin` $tag_1, \ldots,$ `Begin` tag_n, `End` $tag_1, \ldots,$ `End` tag_n, `PCDATA`, `Nil`$\}$ be the alphabet, $\delta : Q \times \mathcal{A} \to Q$ be the state transition function, $q_0 \in Q$ be the initial state, and $F \subseteq Q$ be the set of final states.

Let $M' = (Q', \mathcal{A}', \delta', q_0', F')$ be the automaton of the finite state machine generated by the attribute dependency analysis. (M' is very similar to M, but we notice that it has a different alphabet $\mathcal{A}' = \{$`Begin`, `End`, `PCDATA`, `Nil`$\}$.) We can reflect the information

drawn from the given DTD to the finite state machine M' by taking a *product* of the two automata M and M' as below.

The product automaton $M \times M'$ is an automaton $(Q'', \mathcal{A}'', \delta'', (q_0, q_0'), F'')$ satisfying

$$
\begin{array}{rcl}
Q'' & \subseteq & Q \times Q' \\
\mathcal{A}'' & = & \mathcal{A} \\
\delta''((q, q'), a) & = & \begin{cases} (p, p') & \text{if, for some } tag \in \{tag_1, \ldots, tag_n\}, \\ & \quad a = \texttt{Begin}\, tag,\ p = \delta(q, \texttt{Begin}\, tag), \\ & \quad \text{and } p' = \delta'(q', \texttt{Begin}) \\ & \qquad \text{or} \\ & \quad a = \texttt{End}\, tag,\ p = \delta(q, \texttt{End}\, tag), \\ & \quad \text{and } p' = \delta'(q', \texttt{End}) \\ (\delta(q, a), \delta'(q', a)) & \text{if } a = \texttt{PCDATA} \text{ or } a = \texttt{Nil} \\ \text{not defined} & \text{otherwise} \end{cases} \\
F'' & = & (F \times F') \cap Q''
\end{array}
$$

where Q'' is the minimum state set reached by the transition function δ''.

Taking product has a significant effect in the reduction of the number of states in practice, since in most XML transformations the number of states of the DTD automaton is much less than that of the finite state transition machine obtained by the attribute dependency analysis.

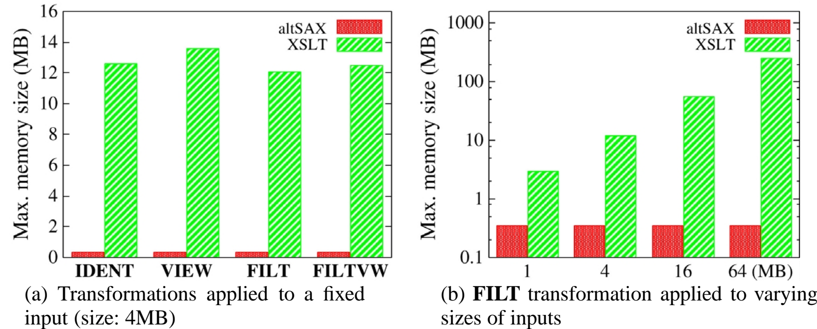
In addition to the reduction in number of states, we can exploit another chance of optimization. The product of two automata provides refined information on the input: it provides the tag name information for each begin tag input. This information can be utilized to detect some run-time behavior statically. For example, suppose a transition incurred by an input `Begin` *tag* is accompanied by a computation that involves conditionals, say, `IF` (\$tag = “a”) `THEN ... ELSE ... ENDIF`. Since the tag name referred to by \$tag is known from the automaton transition, we can replace the conditional with the appropriate branch, discarding the other. This not only slightly improves the execution time, but also contributes to further reduction in the number of states, since we can omit those states which are reached by the irrelevant branch.

4. Implementation and experimental results

We have implemented the program generation scheme discussed in the previous section in a prototype XML stream transformer generator *altSAX*.

The generator takes an XML tree transformation definition (written in the style of AG) and a DTD that confines the input XML. The generator infers the maximum nesting depth from the DTD if it is non-recursive; otherwise the user must supply the maximum nesting depth. The generator produces an XML stream transformation program written in Java (while the generator itself is written in Objective Caml [26]). We adopt a Xerces SAX interface [31] as the frontend.

Table 1
altSAX vs. XSLT (memory usage)



(a) Transformations applied to a fixed input (size: 4MB)

(b) **FILT** transformation applied to varying sizes of inputs

We evaluated several XML stream transformation programs generated by the tool, with comparison to the Xalan XSLT processor [30], one of the most popular XML transformers based on the tree transformation model. The experiments were conducted on a PC (Pentium III 600 MHz, 256 MB memory) running Sun Solaris 8. Transformations are applied to XML files of different sizes that represent a database of stock quotes, whose DTD is given in Fig. 5(b).

We measured memory usage and execution time for the following four transformations.

IDENT Just copying the input to the output.
VIEW Generating an XHTML view of stock quotes (see Appendix B for the definition).
FILT Filtering stock data (which has been defined in Fig. 6).
FILTVW Creating an XHTML view of selected stock information by applying **FILT** followed by **VIEW**.

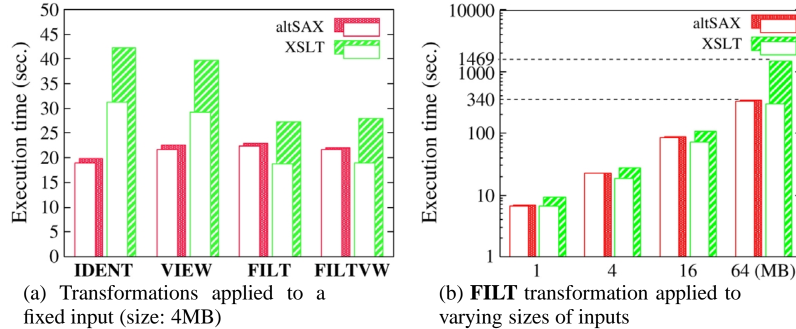
Using *altSAX*, we can generate the **FILTVW** transformation program just as the composition of **FILT** and **VIEW** transformations. We compared this composed **FILTVW** transformation program with an XSLT program that does the same transformation as the composed transformation program does.

Table 1 shows the required maximum memory sizes (which are estimated from the garbage collection messages) of the four transformations.

Table 1(a) compares maximum memory sizes required by the different transformations. This result shows that programs generated by *altSAX* is far more memory-efficient than XSLT transformations. Table 1(b) shows how the maximum memory size varies for increasing sizes of inputs. (Note that we use a logarithmic scale for the y-axis. We show the results for the **FILT** transformations only. The results for other transformations are very similar.) While the amount of memory required by XSLT transformations is proportional to the input size, only a constant size of memory is required by the generated programs. These results together show that transformation programs generated by *altSAX* achieve reasonable memory-efficiency and they are robust to the increasing sizes of inputs.

The reduced strain on the memory subsystem improves the execution time too. Table 2(a) compares the execution times of the four different transformations for a fixed input. The generated programs are faster than XSLT transformations in elapsed time,

Table 2
altSAX vs. XSLT (execution time)



(a) Transformations applied to a fixed input (size: 4MB)

(b) **FILT** transformation applied to varying sizes of inputs

We measured both elapsed and cpu times. The former is shown by shaded bars and the latter by solid bars.

though a bit slower in cpu time for some transformations.[4] Table 2(b) shows the execution times of **FILT** transformation applied to increasing sizes of inputs. (Note that we use a logarithmic scale for the y-axis.) We can observe that the elapsed time of the XSLT transformation rapidly increases for the inputs of larger sizes, compared to the cpu time. This indicates that loading the entire input puts a strong strain on the memory subsystem, in particular when the size of the input is very large, and degrades the performance of transformation significantly. In contrast, generated programs execute in shorter (elapsed) time for larger inputs, because they put less strain on the system. They can finish the entire transformation process with small overheads, as indicated by the small differences between elapsed and cpu times.

5. Concluding remarks

We have proposed a scheme that automatically translates XML tree transformation, specified as attribute grammars, into an executable XML stream transformer. Generated XML stream transformers are more efficient in memory use compared to conventional tree transformers since the input is processed on the fly. We have implemented the proposed scheme in a prototype XML stream transformer generator *altSAX*. Using *altSAX*, we can easily develop stream transformations just by giving the specification of the intended tree transformation.

The novelty of our scheme is in the use of attribute grammar composition and a static analysis on the composition result for generating an executable program. In particular, the closure property of the composition enables us to develop stream transformations in a modular way, i.e., we may combine simple transformations to obtain a more involved one.

As we have mentioned in Section 1, the current scheme for stream transformer generation has a few limitations which would cause problems when it is applied to a

[4] The generated programs are not very efficient in cpu time because of their interpretive overheads. Current implementation generates intermediate code for interpretation, which is a major bottleneck of execution. This should be improved by further optimization.

$$
\begin{array}{l}
S \;\rightarrow\; T:\\
\quad S.result \;=\; T.rev\\
\quad T.acc \;=\; \texttt{Empty}\\[1em]
T \;\rightarrow\; \texttt{Node}\ \$\mathrm{tag}\ T_1\ T_2:\\
\quad T.rev \;=\; T_2.rev\\
\quad T_1.acc \;=\; \texttt{Empty}\\
\quad T_2.acc \;=\; \texttt{Node}\ \$\mathrm{tag}\ T_1.rev\ T.acc\\[1em]
T \;\rightarrow\; \texttt{Empty}:\\
\quad T.rev \;=\; T.acc
\end{array}
$$

Fig. 15. Reversal transformation (the rule for `Content` is omitted).

certain class of practical transformation tasks. We conclude this article by mentioning these limitations and suggesting possible remedies to them.

5.1. Inherently memory-inefficient transformations

There is a class of *inherently memory-inefficient* transformations. Fig. 15 gives such an instance of transformation that reverses the order of occurrences of markups at every nesting level. The generated transformation program would consume as much memory as the tree transformation would do, as it needs to keep the whole input XML symbols until it reaches the very last input symbol.

There is no solution to this problem: any program must load the entire input on the memory to conduct this transformation.

5.2. Limited expressiveness of transformations

We have chosen AGs as the transformation specification language, because they provide a firm basis for program composition. The composition method makes it possible to automate XML stream transformer generation and also to combine two or more transformations in a modular way.

However, the class of AGs which we consider in this paper cannot express certain transformations, because of the limited computational power of AGs that arises from the following factors. First, AGs must satisfy the linearity condition (SAMODUR), as required by AG composition. Second, the attribute evaluation of an AG takes at most linear time in the size of the input, counting the computation of a semantic rule as one unit of time [8]. Therefore, the transformations written in AGs are only able to increase the size of the input by a constant factor, and so are those expressed by the transformation definition language considered in this paper. The class of transformations with a linear increase in size covers many popular XML stream transformations, e.g., those transformations experimented in Section 4. However, it would be desirable to be able to apply our scheme to transformations of higher complexity, say, sorting transformations with $O(n \log n)$ complexity.

Recent advances in the study of program composition techniques for more general transformation frameworks may contribute to improve this deficiency in the expressiveness of our current transformation definition language. Voigtländer proposed a composition method for macro tree transducers [29], which are an abstraction of transformation

systems expressed by primitive recursive program schemes. He also showed that his composition method can be adopted for composing a class of first-order fragment of functional programs [28]. The first author of the present paper also proposed a composition method for functional programs [20,21] by interpreting attribute grammar composition in a functional setting. These refined methods allow compositions under a more relaxed condition than the AG composition: they can compose two transformations T_1 followed by T_2, if T_1 is context-linear (linear in the use of inherited attributes, in terms of attribute grammars) and T_2 is recursion-linear (linear in the use of synthesized attributes). Being not recursion-linear, T_1 can recurse on the same node repeatedly; thus T_1 is able to express computation of higher complexity than AGs.

We notice that, however, there is no closure property under the relaxed condition (i.e., the result is neither context-linear nor recursion-linear in general). Hence the modular development of transformation programs will be disabled. Furthermore, our program generation scheme based on attribute dependency analysis would not immediately apply to the results obtained by the above refined methods, since it crucially exploits the linearity condition of AGs. It would be interesting to devise a program generation scheme for those AGs which do not meet the linearity condition.

5.3. Bounded nesting depth of the input

Our transformer generator can only deal with transformations whose input is presupposed to have a finite bound on the nesting depth. It is very desirable for XML transformers to be able to process inputs of arbitrary nesting depth, since this includes a significant family of XML languages such as XHTML.

There are two problems in dealing with XML inputs which have no presupposed bound on the nesting depth. The fundamental problem is that the AG composition does not allow the use of stack or similar data structures. Recently the second author proposed a composition method for *stack-attributed tree transducers*, which are a formal counterpart of attribute grammars that allows stack-like data structures as attribute values [17]. His method enables us to write the parsing transformation for arbitrary nested inputs and to compose it with other transformations. (It is also proved that the closure property holds for the composition method.) Based on the refined composition method, he designed an XML transformation language, called XTiSP [18,34]. An XTiSP program is translated into an attribute grammar, which is then turned into an XML stream transformer. However, the graph-based analysis in Section 3.1.1 does not apply, because stack values allow attributes to depend on two or more other attributes. This problem is not solved even by the delayed graph merging in Section 3.2, since the number of dependent attributes would vary dynamically as the length of the stack grows and shrinks. XTiSP circumvents the problem by postponing the graph merging process to the execution time of the XML transformation.

Another problem caused by the bounded nesting depth is the non-regularity of XML. We were able to substantially reduce the state space by taking a product of the produced state transition machine and the automaton induced from the given schema information (Section 3.3). However, the language of XML with no nesting bound forms a context-free language, which does not have the automata counterpart. Segoufin and Viannu's automata construction method for XML validation [25] may solve this problem, where their method

can be used to obtain an automaton that approximates the language specified by an arbitrary schema.

5.4. Flexible addressing mechanism

AGs are often too primitive to specify complicated transformation tasks. This is due to their limited ability in addressing nodes in the tree structure. In AGs, different nodes can only communicate via the least common ancestor node, and therefore the communication between remote distant nodes is often expressed in a very awkward way. In contrast, popular XML transformation languages provide an advanced addressing mechanism such as XPath [32].

It would be very useful if our transformation definition language allows such a flexible addressing mechanism. XTiSP by the second author makes it possible to translate a restricted class of XPath expressions into AGs. Olteanu et al. [23] proposed a translation scheme from absolute XPath expressions into those without reverse references (i.e., references to ancestor/preceding sibling nodes). These may add an extra flexibility in specifying complicated transformations in AGs.

Acknowledgments

The authors are grateful to Shin-Cheng Mu for his kind help and advice on the manuscript. We are also grateful to anonymous reviewers for their valuable comments.

Appendix A. The attribute grammar composition algorithm

The descriptional composition algorithm by Ganzinger and Giegerich [11] for composing two AGs is presented. This short section presents a summary of the algorithm through a simple example. For a formal description of the algorithm, readers are referred to the literature [11,4,5].

As an example, we consider the fusion of *Unparse* $\circ$ *Parse*, where the definitions of *Parse* and *Unparse* transformations are given in Fig. 3 and Fig. 7, resp. For the sake of simplicity, we assume the maximum nesting depth of the input is 2.

Following the presentation in [5], we present the composition process in three steps: *projection*, *symbolic evaluation*, and *renaming*. The *projection* step derives an intermediate representation of the composed AG. Next, the *symbolic evaluation* step eliminates the intermediate data structure, namely the XML tree produced between the two transformations *Unparse* and *Parse* in the present example. Finally, the *renaming* step turns the intermediate representation into an AG representation.

Projection. The first step is *projection*, which derives an intermediate representation of the synthesized AG, where the intermediate data constructions between the two AGs are not eliminated yet.

To obtain the intermediate representation, we create new set of semantic rules by projecting every attribute of *Unparse* over each semantic rule from *Parse*. That is, for every rule $N.a = e$ in *Parse* and every synthesized (inherited, resp.) attribute b of *Unparse*,

except for the special attribute *result*, we have a new rule $(N.a).b = e.b$ ($e.b = (N.a).b$, resp.). For example, projecting every attribute of *Unparse* over a semantic rule $E.parse =$ `Node` \$tag $E_1.parse\ E_1.stk_I$ from the production rule $E \rightarrow$ `Begin` \$tag E_1 of *Parse*, we obtain the following new semantic rules:

$$(*)\quad \begin{array}{l} (E.parse).unparse \;=\; \underline{(\texttt{Node}\ \$\text{tag}\ E_1.parse\ E_1.stk_I)}.unparse \\ \underline{(\texttt{Node}\ \$\text{tag}\ E_1.parse\ E_1.stk_I)}.acc \;=\; (E.parse).acc \end{array}$$

The start production is treated differently from the others. First, the semantic rule $S.result = e$ for the start production of *Parse* is removed from the semantic rules and the projection algorithm is applied to the rest of the semantic rules as is done for the other production rules. Then, the semantic rules for the start production $S \rightarrow T$ of *Unparse* are added to the semantic rules of *Parse* and every reference to an attribute a of *Unparse* of the form $T.a$ is replaced by $e.a$. When applied to the running example, the projection yields the following new semantic rules for the start production $S \rightarrow E$.

$$\begin{array}{l} S.result = \underline{(E.parse).unparse} \\ \underline{(E.parse).acc} = \texttt{Nil} \end{array}$$

The underlined expressions originate from $T.unparse$ and $T.acc$, resp., in semantic rules for the start production of *Unparse*. In this particular case, there are no projected rules from *Parse*, since no attribute other than *result* is defined in the semantic rules for the start production of *Parse*.

Symbolic Evaluation. The second step is *symbolic evaluation* [5], which eliminates expressions of the form $(\texttt{C}\ e_1\ \cdots\ e_k).a$, which stands for attribution on the intermediate data.

In the above projected semantic rules $(*)$, we can find that there are occurrences of the same expression of intermediate data construction (i.e., the underlined ones). Considering an instance of the semantic rules for the `Node` production, where

$$T \;=\; \texttt{Node}\ \$\text{tag}\ E_1.parse\ E_1.stk_I, \quad T_1 \;=\; E_1.parse, \text{ and } \quad T_2 \;=\; E_1.stk_I,$$

we obtain the following semantic rules corresponding to the intermediate data construction.

$$(**)\quad \begin{array}{l} \underline{(\texttt{Node}\ \$\text{tag}\ E_1.parse\ E_1.stk_I).unparse} \;=\; \texttt{Begin}\ \$\text{tag}\ ((E_1.parse).unparse) \\ (E_1.parse).acc \;=\; \texttt{End}\ \$\text{tag}\ ((E_1.stk_I).unparse) \\ (E_1.stk_I).acc \;=\; \underline{(\texttt{Node}\ \$\text{tag}\ E_1.parse\ E_1.stk_I).acc} \end{array}$$

Merging $(*)$ and $(**)$ and dismissing the underlined expressions by the transitivity of equality, we can cancel the intermediate data construction and obtain the following new set of semantic rules for the `Begin` production rule.

$$\begin{array}{l} E \;\rightarrow\; \texttt{Begin}\ \$\text{tag}\ E_1: \\ \quad (E.parse).unparse \;=\; \texttt{Begin}\ \$\text{tag}\ ((E_1.parse).unparse) \\ \quad (E_1.parse).acc \;=\; \texttt{End}\ \$\text{tag}\ ((E_1.stk_I).unparse) \\ \quad (E_1.stk_I).acc \;=\; (E.parse).acc \end{array}$$

Repeating the above rewriting process, we can eliminate all the expressions of intermediate data construction.

$S \rightarrow E$:
$S.result = E.parse_unparse$;
$E.parse_acc = \mathtt{Nil}$;

$E \rightarrow \mathtt{Begin}$ \$tag E_1:
$E.parse_unparse = \mathtt{Begin}$ \$tag $E_1.parse_unparse$;
$E.stk_1_unparse = E_1.stk_2_unparse$;
$E_1.parse_acc = \mathtt{End}$ \$tag $E_1.stk_1_unparse$;
$E_1.stk_1_acc = E.parse_acc$;
$E_1.stk_2_acc = E.stk_1_acc$;

$E \rightarrow \mathtt{End}$ \$tag E_1:
$E.parse_unparse = E.parse_acc$;
$E.stk_1_unparse = E_1.parse_unparse$;
$E.stk_2_unparse = E_1.stk_1_unparse$;
$E_1.parse_acc = E.stk_1_acc$;
$E_1.stk_1_acc = E.stk_2_acc$;

$E \rightarrow \mathtt{PCDATA}$ \$cdata E_1:
$E.parse_unparse = \mathtt{PCDATA}$ \$cdata $E.parse_acc$;
$E.stk_1_unparse = E_1.skt_1_unparse$;
$E.stk_2_unparse = E_1.stk_2_unparse$;
$E_1.parse_acc = E.parse_acc$;
$E_1.stk_1_acc = E.stk_1_acc$;
$E_1.stk_2_acc = E.stk_2_acc$;

$E \rightarrow \mathtt{Nil}$:
$E.parse_unparse = E.parse_acc$;

Fig. A.1. The result of composition of *Unparse* ∘ *Parse*.

Renaming. The final step of the composition algorithm is *renaming*. This step rewrites every successive attribution of the form $(T.a).b$ to a single attribute $T.a_b$, where the successive attributions by a and b are replaced with a single concatenated attribution a_b. For example, the above result of the symbolic evaluation rewrites to the following:

$E \rightarrow \mathtt{Begin}$ \$tag E_1:
$E.parse_unparse = \mathtt{Begin}$ \$tag $E_1.parse_unparse$
$E_1.parse_acc = \mathtt{End}$ \$tag $E_1.stk_1_unparse$
$E_1.stk_1_acc = E.parse_acc$

Fig. A.1 shows the result of composition of *Unparse* ∘ *Parse*. It is easy to check that this composition result works as an identity transformation for the input of bounded nesting depth 2.

Appendix B. VIEW: An XHTML view generator

The definition of **VIEW** transformation is given in Fig. B.1. **VIEW** generates an XHTML view of a stock quotes list. When it is applied to the stock quotes list in Fig. 5(a), it generates an XHTML table as shown below.

```
<?xml version="1.0">
<html>
<head>
  <title>Virtual Stock Ticker</title>
</head>
<body>
  <h1>Virtual Stock Ticker</h1>
  <table>
    <tr>
      <th>Symbol</th> <th>Price</th> <th>Change</th> <th>Volume</th>
    </tr>
    <tr>
```

```
S → T:
     S.result = Node “html”
                  (Node “head”
                     (Node “title” (Content “VisualStockTicker”) Empty)
                  (Node “body”
                     (Node “h1” (Content “VisualStockTicker”) Empty
                     (Node “table”
                        (Node “tr”
                           (Node “th” (Content “Symbol”)
                           (Node “th” (Content “Price”)
                           (Node “th” (Content “Change”)
                           (Node “th” (Content “Volume”) Empty))))
                        T.html ) Empty)) Empty)) Empty;

 T → Node $tag T1 T2:
     IF ( $tag = “stock_quotes” ) THEN
         T.html = T1.html;
     ELSE IF ( $tag = “stock_quote” ) THEN
         T.html = Node “tr” T1.html T2.html;
     ELSE IF ( $tag = “symbol” || $tag = “price”
             || $tag = “change” || $tag = “volume” ) THEN
         T.html = Node “td” T1.html T2.html;
     ELSE
         T.html = T2.html;
     ENDIF

 T → Content $cdata:
     T.html = Content $cdata;

 T → Empty:
     T.html = Empty;
```

Fig. B.1. Definition of **VIEW** transformation.

```
      <td>RIMS</td> <td>32.23</td> <td>2.17</td> <td>13993600</td>
    </tr>
```

and many more...

```
  </table>
</body>
</html>
```

References

[1] A.V. Aho, R. Sethi, J.D. Ullman, Compilers—Principles, Techniques, and Tools, Addison-Wesley, 1986.

[2] M. Altinel, M.J. Franklin, Efficient filtering of XML documents for selective dissemination of information, in: International Conference on Very Large Databases, 2000, pp. 53–64.

[3] H. Barendregt, The Lambda Calculus: Its Syntax and Semantics, 2nd edition, North-Holland, 1984.

[4] J. Boyland, S.L. Graham, Composing tree attributions, in: Proceedings of the 21th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM Press, 1994, pp. 375–388.

[5] L. Correnson, E. Duris, D. Parigot, G. Roussel, Declarative program transformation: A deforestation case-study, in: Principles and Practice of Declarative Programming, LNCS, vol. 1702, Springer Verlag, 1999, pp. 360–377.

[6] Y. Diao, P. Fischer, M.J. Franklin, R. To, YFilter: Efficient and scalable filtering of XML documents, in: Proc. of 18th International Conference on Data Engineering, ICDE 2002, 2002, pp. 341–342.

[7] Document object model (DOM), URL http://www.w3.org/DOM/.

[8] J. Engelfriet, Attribute grammars: attribute evaluation methods, in: B. Lorho (Ed.), Methods and Tools for Compiler Construction, Cambridge University Press, 1984, pp. 103–138.

[9] Extensible markup language (XML), URL http://www.w3c.org/XML/.

[10] A. Feng, T. Wakayama, SIMON: A grammar-based transformation system for structured documents, Electronic Publishing 6 (4) (1993) 361–372.

[11] H. Ganzinger, R. Giegerich, Attribute coupled grammars, in: Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction, SIGPLAN Notices, vol. 19, 1984, pp. 157–170.

[12] T.J. Green, G. Miklau, M. Onizuka, D. Suciu, Processing XML streams with deterministic automata, in: Proc. of Database Theory ICDT 2003, LNCS, no. 2572, 2003, pp. 173–189.

[13] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to automata theory, languages, and computation, 2nd edition, Addison-Wesley, 2001.

[14] T. Johnsson, Attribute grammars as a functional programming paradigm, in: G. Kahn (Ed.), Proceedings of the Conference on Functional Programming Languages and Computer Architecture, LNCS, vol. 274, Springer Verlag, 1987, pp. 154–173.

[15] D.E. Knuth, Semantics of context-free languages, Mathematical Systems Theory 2 (2) (1968) 127–145.

[16] B. Ludäscher, P. Mukhopadhayn, Y. Papakonstantinou, A transducer-based XML query processor, in: Proceedings of 28th International Conference on Very Large Data Bases, 2002, pp. 227–238.

[17] K. Nakano, Composing stack-attributed tree transducers, Tech. Rep. METR–2004–01, Department of Mathematical Informatics, University of Tokyo, Japan 2004.

[18] K. Nakano, XTiSP: XML transformation languages intended for stream processing, URL http://xtisp.psdlab.org/.

[19] K. Nakano, S. Nishimura, Deriving event-based document transformers from tree-based specifications, in: Workshop on Language Descriptions, Tools and Applications, LDTA'2001, Electronic Notes in Theoretical Computer Science, vol. 44, Elsevier Science, 2001.

[20] S. Nishimura, Correctness of a higher-order removal transformation through a relational reasoning, in: Programming Language and Systems, First Asian Symposium, APLAS 2003 Proceedings, LNCS, vol. 2895, Springer Verlag, 2003, pp. 358–375. A full version is available as a preprint Kyoto-Math 2003-06 from http://www.math.kyoto-u.ac.jp/~susumu/papers/aplas03-long.ps.gz.

[21] S. Nishimura, Fusion with stacks and accumulating parameters, in: Proc. of ACM SIGPLAN 2004 Symposium on Partial Evaluation and Program Manipulation, PEPM'04, ACM Press, 2004 (in press).
[22] D. Olteanu, T. Kiesling, F. Bry, An evaluation of regular path expressions with qualifiers against XML streams, in: Proc. of 19th International Conference on Data Engineering, ICDE 2003, 2003, pp. 702–704.
[23] D. Olteanu, H. Meuss, T. Furche, F. Bry, XPath: Looking forward, in: Proc. of the EDBT Workshop on XML Data Management, XMLDM, LNCS, vol. 2490, Springer Verlag, 2002, pp. 109–127.
[24] SAX: The simple API for XML, URL http://www.saxproject.org/.
[25] L. Segoufin, V. Viannu, Validating streaming XML documents, in: Principles of Database Systems: PODS2002, ACM Press, 2002, pp. 53–64.
[26] The Caml language homepage, URL http://caml.inria.fr/.
[27] The Haskell home page, URL http://www.haskell.org/.
[28] J. Voigtländer, Using circular programs to deforest in accumulating parameters, in: K. Asai, W.-N. Chin (Eds.), ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, ACM Press, 2002, pp. 126–137.
[29] J. Voigtländer, Composition of restricted macro tree transducers, Master's Thesis, Dresden University of Technology, Germany, March, 2001.
[30] Xalan-Java homepage, URL http://xml.apache.org/xalan-j/.
[31] Xerces2-Java homepage, URL http://xml.apache.org/xerces2-j/.
[32] XML path language (XPath), URL http://www.w3c.org/TR/xpath/.
[33] XQuery 1.0: An XML query language, URL http://www.w3.org/TR/xquery/.
[34] K. Nakano, An implementation scheme for XML transformation languages through derivation of stream processors, in: Proc. of 2nd Asian Symposium on Programming Languages and Systems, APLAS 2004, 2004 (in press).