Electronic Notes in Theoretical Computer Science 44 No. 2 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume44.html 25 pages

Deriving Event-Based Document Transformers from Tree-Based Specifications

Keisuke NAKANO [1] Susumu NISHIMURA [2,3]

Research Institute for Mathematical Sciences, Kyoto University
Sakyou-ku, Kyoto 606–8502, JAPAN

Abstract

Structured documents are usually processed by tree-based document transformers, which transform the document tree representing the structure of the input document into another tree structure. *Event-based document transformers*, by contrast, recognize the input as a stream of parsing events, *i. e.* lexical tokens, and process the events one by one in an event-driven manner. Event-based document transformers have advantages that they need less memory space and that they are more tolerant of large inputs, compared to tree-based transformers, which construct the intermediate tree representation.

This paper proposes an algorithm which derives an event-based transformer from a given specification of a document transformation over a tree structure. The derivation of an event-based transformer is carried out in the framework of attribute grammars. We first obtain an attribute grammar which processes a stream of parsing events, by applying a deforestation method; We then derive an attribute evaluation scheme relevant to the event-based transformation. Using this algorithm, one can develop event-based document transformers in a more declarative style than directly programming over the stream of parsing events.

Key words: Event-based document transformation, Attribute evaluation scheme, Descriptional composition, Attribute grammars, XML

1 Introduction

Structured documents are widely used for representing a hierarchical data structure in a conventional text format. There are various structured docu-

[1] Email: ksknac@kurims.kyoto-u.ac.jp
[2] Email: nisimura@kurims.kyoto-u.ac.jp
[3] Partially supported by the Grant-in-Aid for Encouragement of Young Scientists from the Ministry of Education, Culture, Sports, Science and Technology and JSPS under grant no. 12780216.

ment formats, *e. g.*, HTML for Web presentation, LaTeX for typesetting, and most notably the XML (Extensible Markup Language) standard [1], which is intended to serve as a common data representation for seamless data exchange among multiple platforms. Due to the increasing amount of data being exchanged in structured document formats, the technology for transforming structured documents is getting more significant.

A conventional scheme for document transformation is a *tree-based* scheme, where a transformation is defined as an operation over *document trees*, which model the hierarchical structure of structured documents. A tree-based transformation first constructs a document tree of the input document on the memory, manipulates the document tree, and translates back the transformed document tree into the result document. Due to its high expressiveness for describing document transformations, the tree-based transformation scheme is widely used in practice (*e. g.*, an XML transformation language XSLT [2] and Document Object Model (DOM) [3]).

The tree-based transformations, however, have a drawback that they must once load the entire document on the memory before starting tree manipulations. This indicates that the size of documents to be processed by a tree-based transformation must fit into the actual memory size. This could be a sever problem, when the size of the input document is very large, or when the size of the memory is relatively small.

For the purpose of relaxing the strain on the memory usage, Simple API for XML (SAX) [4] has been proposed as an alternative way for programming XML document transformations. The SAX API, instead of constructing a document tree, creates a stream of *parsing events* (or lexical tokens in the terminology of the formal language theory) to notify the back-end program of what syntactic objects are encountered while it reads the input XML document. For example, when a program using SAX API reads the following document

```
<message> <body> I like sweets. </body> </message>
```

it is notified by the following parsing events: a beginning of a `message` tag, a beginning of a `body` tag, a string "`I like sweets.`", end of a `body` tag, and finally end of a `message` tag. The program processes the document in an event-driven fashion, *i. e.*, each parsing event is caught by an event handler, which is responsible for processing the event.

Here we are faced with a tension between efficiency and expressiveness. An event-based document transformer may dramatically reduce the memory usage, especially that for simple document transformations. On the other hand, event-based transformers have less expressive power than tree-based transformations. Due to the poor structure awareness, event-based transformations are harder to program than tree-based ones and therefore they are usually used only for those relatively simple transformations. In addition, the event-based transformations have another drawback that it is difficult to maintain:

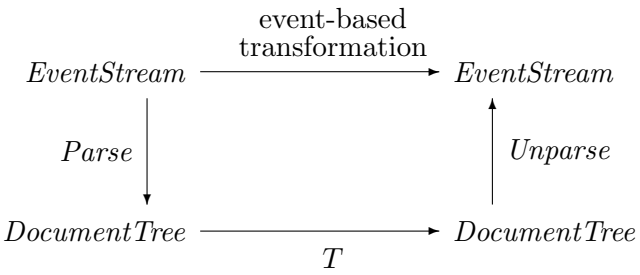


Fig. 1. A diagram for document transformation

Program codes are scattered into small actions responding to parsing events, and hence even adding a small change in the original transformation program could be very cumbersome.

This paper proposes an algorithm for automatically deriving an event-based document transformation from a specification of a tree-based transformation. It would be very useful if an event-based document transformation is obtained automatically. First of all, we obtain efficiency and expressiveness at the same time: Once we specify a document transformation in a tree-based style, we can derive an executable event-based transformation program from the specification, and the derived event-based transformation program saves not only memory space but also execution time because of the reduced memory management task. In addition, tree-based specifications, which are given in a declarative style, are much easier to maintain than event-based document transformation programs.

1.1 Deforestation

We can regard the present problem of deriving an event-based program as a *deforestation* problem [14]. Deforestation stands for general program transformation techniques, which eliminate the intermediate data construction by unifying two or more composite functions into a single function.

A tree-based transformation can be expressed by a composition of three functions, as illustrated by the diagram in Figure 1. First, a function *Parse* translates a given event stream to an intermediate tree representation. The intermediate document tree is then processed by a function T, which is responsible for the tree transformation. Finally, a function *Unparse* translates back the resulting document tree into the corresponding event stream. (In this paper, we omit the process for generating a parsing event stream from a character stream of an input document and vice versa. The standard technique for lexical analysis [6] would suffice for the process.) The tree-based transformation is represented by a composite function $\mathit{Unparse} \circ T \circ \mathit{Parse}$, and a derivation of an event-based transformation is nothing but to find an equivalent shortcut function (the upper-most arrow in the diagram), which generates no intermediate document trees.

In this paper, we apply an existing deforestation method based on the

formalism of attribute grammars to the present problem. A transformation from a term of a language L_1 to a term of another language L_2 can be defined by an attribute grammar (AG) over L_1 whose attribute values range over L_2. Ganzinger and Giegerich called such AGs attribute coupled grammars [10,11]. Let us write $G \circ F$ to denote a composition of two AGs, where the output of F is processed by G. Ganzinger and Giegerich [10] have presented an AG deforestation method, called *descriptional composition*, which derives a single deforested AG from a given composition $G \circ F$ of two attribute coupled grammars. Following Ganzinger and Giegerich, several extensions and refinements to the descriptional composition method have been studied [7,8,9].

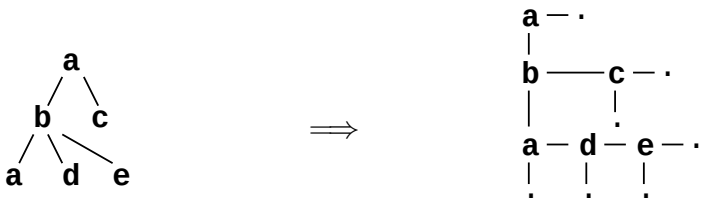
1.2 Deriving one-pass interactive event-based programs

The deforested AG, which is derived by the descriptional composition method, is not enough for solving the present problem. An AG is only a specification, and hence is not directly executable. The deforested AG must be translated into a program relevant to event-based document transformation.

An event-based document transformer should be a one-pass interactive program. A document transformer is called *one-pass* if it traverses the stream of the input parsing events only once; An event-based document transformer is called *interactive*, if it writes to an output stream simultaneously as it reads from the input stream, responding to each parsing event. To obtain such a one-pass interactive transformer, we need to find a method for deriving an appropriate attribute evaluator from the deforested AG specification.

The main difficulty in deriving such a one-pass interactive document transformation program is that an attribute value for a parsing event may depend on the attribute values of parsing events that will follow. We call such dependencies *forward dependencies*. (Even a simple document transformation results in an AG whose semantic rules contain forward dependencies. See Section 3.2.1 for details.)

In this paper, we propose an algorithm which systematically derives a one-pass interactive event-based program from an AG specification. We solve the above mentioned difficulty by separating each semantic rule into two parts: attribute dependency and value construction. The former can be computed by looking into the input event stream statically, and it turns out that dependency patterns range over a finite domain. We can therefore model an evaluation scheme by a finite state transition machine, where the state space is the set of dependency patterns and state transitions are incurred by parsing events. We need not care about forward dependencies any more, since the state transition is subject to the dependency pattern, but not to individual attribute dependencies. The remaining part of attribute evaluation, *i. e.* the value construction, is dynamically computed for each transition, by letting each attribute hold the partially determined attribute value. The finite state machine incrementally outputs the result of transformation, by writing out



(a) XML-style representation (b) binary tree representation

Fig. 2. Document tree representations

the definite part of the result for each transition. The technical contribution of this paper is to give a decidable terminating algorithm for producing such a state transition machine from an AG specification.

The rest of the paper is organized as follows: Section 2 defines the present problem in a more formal setting. In Section 3, we present our algorithm through a simple example. Section 4 discusses what document transformations are definable in our framework and discusses extension to the present work. Finally, Section 5 concludes the paper.

2 AGs for Document Transformations

2.1 The data model

The structured document format considered in the present paper is an XML-like markup language which consists of (i) an arbitrary, but fixed, finite set of tag names for markup, and (ii) named start tags and unnamed end tags. For example, the following is a structured document in our format.

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

Each <*tagname*> represents a start tag named *tagname*, and </> an end tag, which has no tag name. A document is *well-formed*, if start tags and end tags are balanced in the usual sense. (Unlike XML, tag names are omitted from end tags for simplicity. They are redundant information in a well-formed document indeed.) The *depth* of a well-formed document is the maximum level of nesting, with the outermost level being 1. The above document has the depth 3, for example.

The document format presented here is an idealized one for the subsequent theoretical study. It is missing many useful features found in real document formats, but the authors believe that it would be possible to incorporate them into the framework to be presented in the paper. We leave the topic to a future investigation (see Section 5).

$$
\begin{array}{lll}
AG & ::= & \texttt{let}\ F = \{S \to E :\ \Sigma\}\{E \to \texttt{C}\ E^* :\ \Sigma\}^+ \\
\Sigma & ::= & (occ = exp)^* \\
occ & ::= & E.a \ \mid \ S.result \\
exp & ::= & occ \ \mid \ \texttt{C}\ exp^*
\end{array}
$$

Fig. 3. Attribute grammar notation

We model document trees, the internal representation of structured documents, by binary trees. For example, the XML document tree, given in Figure 2(a), of the above example document is represented by the binary tree in Figure 2(b). In the binary tree representation, each left branch points to the leftmost child of the parent document node, and each right branch to the first sibling node to follow. If a branch has no relevant node to point to, it points to an empty node (designated by a dot in the figure). The two representations are isomorphic, and therefore transformations on one representation is convertible to those on the other representation.

The language of event streams and the language of binary document trees are formally defined as follows. Suppose $\texttt{tag}_1$, $\texttt{tag}_2$, ..., $\texttt{tag}_n$ be the fixed set of tag names. The language of event streams is specified by the set of production rules $\{S \to E,\ E \to \texttt{Begin\_tag}_1\ E,\ \cdots,\ E \to \texttt{Begin\_tag}_n\ E,\ E \to \texttt{End}\ E,\ E \to \texttt{Nil}\}$, where the non-terminal S is the start symbol. Each constructor $\texttt{Begin\_tag}_i$ corresponds to a start tag named $\texttt{tag}_i$, `End` to an end tag, and `Nil` to the end of an event stream. For example, the above example document is represented by the expression `Begin_a (Begin_b (Begin_a (End (Begin_d (End (Begin_e (End (End (Begin_c (End (End Nil))))))))))))`. Similarly, the language of document trees is specified by the set of production rules $\{S \to D,\ D \to \texttt{Node\_tag}_1\ D\ D,\ \cdots,\ D \to \texttt{Node\_tag}_n\ D\ D,\ D \to \texttt{Empty}\}$. Each constructor $\texttt{Node\_tag}_i$ corresponds to a tree node named $\texttt{tag}_i$, and `Empty` to an empty node.

2.2 Attribute grammars

Our attribute grammar (AG) notation is given in Figure 3. An AG is defined by a list of pairs $prod : \Sigma$ of a *production rule* $prod$ of the underlying grammar and its associated set Σ of *semantic rules*. In the present paper, we restrict a production rule to have the form either $S \to E$ or $E \to \texttt{C}\ E_1\ \cdots\ E_k\ (k \geq 0)$, where S is the start symbol, $E, E_1, \ldots, E_k$ are non-terminal symbols, and `C` is a data constructor. Each semantic rule in Σ has the form $occ = e$, where occ is an *attribute occurrence* $E.a$, which denotes the value of attribute a attached to the non-terminal symbol E, and e is an *expression*, which defines what value is assigned to the occurrence occ. We allow only those expressions which are either a reference to an attribute occurrence or a data construction $\texttt{C}\ e_1\ \ldots\ e_k$. We assume that a special synthesized attribute *result* is defined only for the start production $S \to E$ to designate the overall result of attribute evaluation.

Note that, in contrast to traditional AGs, we allow some attributes of a

production rule to have no corresponding semantic rule. We may explicitly write such an undefined semantic rule as $E.a = \mathit{undef}$, which reads "the value of the attribute a on the symbol E is undefined". If the value of one of expressions $e_1, \ldots e_n$ is undefined, then the value of an expression $\mathtt{C}\ e_1\ \ldots\ e_n$ is also undefined. Undefined attributes play a crucial role for defining an AG for parsing (Section 2.3).

The AGs are required to satisfy a few conditions, so that the descriptional composition method can apply to them. The original descriptional composition method, most significantly, requires AGs to respect a condition called SSUR [11]. In this paper, we relax the SSUR condition in order to allow the descriptional composition to work with undefined semantic rules [11, pp. 378].

In the rest of the paper, we assume that every AG satisfies the following conditions:

- It is *noncircular* [13] and
- satisfies *quasi-SSUR* condition.

An AG satisfies *SSUR (syntactic single use requirement)* condition [11], if every different occurrence of a *syntactic attribute* is referred to exactly once in the defining expressions of the semantic rules for every production rule. An attribute is *syntactic* if it ranges over the set of terms of a single language. An AG satisfies quasi-SSUR condition, if the *exactly once use* restriction is relaxed to *at most once use*.

2.3 AGs for parsing and unparsing

Now we define AGs for parsing and unparsing, which correspond to functions *Parse* and *Unparse*, *resp.*, in Figure 1.

To define an AG for parsing, we assume that every well-formed input document has a depth less than or equal to a fixed number $d (d > 0)$. Though this may seem restrictive, a large portion of document transformations used in practice should be covered by those transformations whose input documents have a bounded depth. For example, when XML documents are used for representing a database (it is one of most typical usages of XML!), the maximum depth of documents can be mostly determined by the *database schema* [5] accompanied with the document. In contrast, rendition languages for data presentation such as HTML and LaTeX do not have a bounded depth in general. However, a sufficiently large number would serve as a bound for practical uses. In the present paper, we do not discuss transformations for documents of an arbitrary depth, leaving it to a future investigation.

The finite-depth restriction to the input documents is a key to derive a one-pass interactive event-based program. The AG for parsing documents, which will be given later, contains undefined semantic rules to flag an error when the input document has an exceeding depth. This enables our algorithm to terminate, since the static analysis can stop up to a finite number of lookahead

$\texttt{let}\ parse^{(d)}\ =$
$S \to E$:
$S.result = E.parse$
$E'.stack\_h_1 = undef$
$\vdots$
$E'.stack\_h_d = undef$

$E \to \texttt{Begin\_tag}_1\ E'$:
$E.parse = \texttt{Node\_tag}_1\ E'.parse\ E'.stack\_s_1$
$E.stack\_s_1 = E'.stack\_s_2$
$\vdots$
$E.stack\_s_{d-1} = E'.stack\_s_d$
$E.stack\_s_d = undef$
$E'.stack\_h_1 = \texttt{Empty}$
$E'.stack\_h_2 = E.stack\_h_1$
$\vdots$
$E'.stack\_h_d = E.stack\_h_{d-1}$

$\vdots$

$E \to \texttt{Begin\_tag}_n\ E'$:
(Similarly defined as above)

$E \to \texttt{End}\ E'$:
$E.parse = E.stack\_h_1$
$E.stack\_s_1 = E'.parse$
$E.stack\_s_2 = E'.stack\_s_1$
$\vdots$
$E.stack\_s_d = E'.stack\_s_{d-1}$
$E'.stack\_h_1 = E.stack\_h_2$
$\vdots$
$E'.stack\_h_{d-1} = E.stack\_h_d$
$E'.stack\_h_d = undef$

$E \to \texttt{Nil}$:
$E.parse = \texttt{Empty}$
$E.stack\_s_1 = undef$
$\vdots$
$E.stack\_s_d = undef$

Fig. 4. AG for parsing

$\texttt{let}\ unparse\ =$
$S \to T$:
$S.result = T.unparse$
$T.acc = \texttt{Nil}$

$T \to \texttt{Node\_tag}_1\ T_1\ T_2$:
$T.unparse = \texttt{Begin\_tag}_1\ T_1.unparse$
$T_1.acc = \texttt{End}\ T_2.unparse$
$T_2.acc = T.acc$

$\vdots$

$T \to \texttt{Node\_tag}_n\ T_1\ T_2$:
(Similarly defined as above)

$T \to \texttt{Empty}$:
$T.unparse = T.acc$

Fig. 5. AG for unparsing

events into the input parsing event stream.

Figure 4 defines an AG $parse^{(d)}$, which takes a document of a depth d or less and produces a document tree. The set of synthesized attributes $stack\_s_1, \ldots, stack\_s_d$ simulates a finite stack of depth d, where $stack\_s_1$ is the stack top. Each attribute $stack\_s_i$ holds the result of parsing sibling nodes which follow the i-th outer pair of start and end tags enclosing the current point of parsing. Pushing more than d elements onto the stack causes an overflow, and the overflowed datum is lost. An exceeding depth of the input document is flagged by an undefined attribute value *undef*, which indicates a pars-

ing error. Unbalanced tags are detected in a similar manner, by the other finite stack represented by the set of inherited attributes $stack\_h_1, \ldots, stack\_h_d$. Each parsing event of a start tag pushes an empty node onto the stack, and it will be later popped and used as the end of the child nodes when the matching end tag appears.

Figure 5 defines an AG *unparse*, which generates a stream of parsing events from a given document tree. In contrast to *parse*, this AG always produces a result successfully, irrespective of the depth of the input document tree. The AG is intended to perform a depth-first traversal on a document tree. The result of the depth-first traversal is accumulated to the inherited attribute *acc*, and the accumulated result will be propagated up to the root of the document tree as the final result.

2.4 Specifying a document transformation

The problem to be solved is now formally stated as follows: Given a specification of a document transformation in a composite form $unparse \circ T \circ parse^{(d)}$, where d is the maximum depth d of the input document and T is an AG representing a tree-based transformation, derive a one-path interactive event-based document transformation program which computes the result intended by the composite specification, for every input document of a depth d or less. Note that the output document can have an arbitrary depth, irrespective of the depth of the input document.

3 The Algorithm

This section presents an algorithm for deriving a one-pass event-based document transformation program from a composite specification of tree transformation $unparse \circ T \circ parse^{(d)}$. Throughout this section, we explain our algorithm with an identity transformation T_{id}, whose trivial AG definition is given below, being the running example.

$$
\begin{array}{ll}
\texttt{let } id \ = & \\
\quad S \to T : & S.result = T.it \\
\quad T \to \texttt{Node\_A } T_1\ T_2 : & T.it = \texttt{Node\_A } T_1.it\ T_2.it \\
\quad T \to \texttt{Node\_B } T_1\ T_2 : & T.it = \texttt{Node\_B } T_1.it\ T_2.it \\
\quad T \to \texttt{Empty} : & T.it = \texttt{Empty}
\end{array}
$$

(We have assumed that there are only two different tag names, `A` and `B`, for simplicity.)

We notice that the choice of the running example is only for explanatory purpose. Even the simple example is enough for demonstrating what are the difficulties of the present problem and how we can solve the problems. Our algorithm can be applied to any transformation specified by an AG which satisfies the conditions in Section 2.2. We will discuss in Section 4 what transformations can be expressed under the restriction.

Our algorithm is comprised of three phases. First, the composite specification of a document transformation $\mathit{unparse} \circ T \circ \mathit{parse}^{(d)}$ is unified into a single deforested AG by applying the descriptional composition. Second, the resulting AG is further transformed into a finite state transition machine representing the document transformation. Finally, we obtain a one-pass interactive event-based program from the finite state transition machine. We will explain each phase in turn and observe efficiency of obtained programs by our algorithm in the rest of this section.

Let us define some notations. A function f is called a finite map, if its domain, denoted by $\mathit{dom}(f)$, is a finite set. We conventionally write $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ to denote a finite map f such that $\mathit{dom}(f) = \{x_1, \ldots, x_n\}$ and $f(x_i) = v_i$ for every i. For any finite maps f and g such that $\mathit{dom}(f) \cap \mathit{dom}(g) \neq \emptyset$, $f \uplus g$ denotes a finite map h such that $\mathit{dom}(h) = \mathit{dom}(f) \cup \mathit{dom}(g)$, $h(x) = f(x)$ if $x \in \mathit{dom}(f)$, and $h(x) = g(x)$ if $x \in \mathit{dom}(g)$.

3.1 Descriptional composition

In the first phase of the derivation, we obtain a single deforested AG, which does not produce any intermediate data, from the composite specification of a document transformation. For this purpose, we simply apply the existing descriptional composition method to the present problem. This section is not intended to be formal but to give a short summary of the descriptional composition method through the running example. For a detailed, formal definition of the algorithm, readers are deferred to [10,11,8,9].

To obtain a single deforested AG from the composite specification $\mathit{unparse} \circ T \circ \mathit{parse}^{(d)}$, we need to apply the descriptional composition twice: We first compose T with $\mathit{parse}^{(d)}$, and then *unparse* with the resulting AG. As for the running example, the composition of the identity transformation T_{id} and the AG $\mathit{parse}^{(d)}$ apparently results in $\mathit{parse}^{(d)}$.[4] Therefore, we will explain the descriptional composition by means of the composition of *unparse* and $\mathit{parse}^{(d)}$.

The descriptional composition derives a single deforested AG in three steps. Let F and G be two AGs for the composition $G \circ F$. In our running example, F is $\mathit{parse}^{(d)}$ and G is *unparse*. Throughout the rest of this section, we assume $d = 2$ for the sake of simplicity.

Projection. The first step is *projection*, which derives an intermediate representation of the composed AG, where the intermediate data constructions between the two AGs are not eliminated yet. In the intermediate representation, we temporarily write $e.a$ to denote an attribute occurrence on an arbitrary expression e.

The intermediate representation is obtained by projecting every attributes of G over each semantic rule $N.a = e$ in F so that a new semantic rule of

[4] We do not show the process of this composition, which is simpler than the one presented below.

the form $(N.a).b = e.b$ ($e.b = (N.a).b$, *resp.*) is created for each synthesized (inherited, *resp.*) attribute b of G. For example, projecting attributes of *unparse* over the semantic rule $E.parse = \texttt{Node\_A}\ E'.parse\ E'.stack\_s_1$ for the production $E \to \texttt{Begin\_A}\ E'$ of *parse*, we obtain the following new semantic rules:

$$\Sigma = \begin{array}{l} (E.parse).unparse = \underline{(\texttt{Node\_A}\ E'.parse\ E'.stack\_s_1).unparse} \\ \underline{(\texttt{Node\_A}\ E'.parse\ E'.stack\_s_1).acc} = (E.parse).acc \end{array}$$

The start production is treated differently from the others. First, the semantic rule $S.result = e_F$ for the start production of F 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 $S \to T$ of G is added to the projected semantic rules of F and every reference to an attribute a of G of the form $T.a$ is replaced by $e_F.a$, *i. e.*, the corresponding attribution on the intermediate result. [5]

If applied to the running example, the projection results in the following new semantic rules for the start production $S \to E$.

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

In this particular case, there is no projected rules from F, since no attribute other than *result* is defined in the semantic rules for the start production of F.

Symbolic evaluation. The second step is *symbolic evaluation* [8,9], which eliminates expressions of the form $(\texttt{C}\ \overline{e}).a$ derived in the previous projection step.

Consider the above projected semantic rules for the production $E \to \texttt{Begin\_A}\ E'$. They contain attributions to the same intermediate data construction (the underlined parts). Let $T \to \texttt{Node\_A}\ T_1\ T_2$ be the production rule for the construction, with $T = (\texttt{Node\_A}\ E'.parse\ E'.stack\_s_1)$, $T_1 = E'.parse$, and $T_2 = E'.stack\_s_1$. The corresponding semantic rules for this intermediate data construction are given by the following equations, according to the definition of *unparse*.

$$\Sigma' = \begin{array}{l} \underline{(\texttt{Node\_A}\ E'.parse\ E'.stack\_s_1).unparse} = \texttt{Begin\_A}\ ((E'.parse).unparse) \\ (E'.parse).acc = \texttt{End}\ ((E'.stack\_s_1).unparse) \\ (E'.stack\_s_1).acc = \underline{(\texttt{Node\_A}\ E'.parse\ E'.stack\_s_1).acc} \end{array}$$

Merging Σ and Σ' and dismissing the underlined expressions by the transitivity of the equality, we can cancel the intermediate data construction and obtain the following new set of semantic rules for the production $E \to \texttt{Begin\_A}\ E'$.

$$\begin{array}{l} (E.parse).unparse = \texttt{Begin\_A}\ ((E'.parse).unparse) \\ (E'.parse).acc = \texttt{End}\ ((E'.stack\_s_1).unparse) \\ (E'.stack\_s_1).acc = (E.parse).acc \end{array}$$

[5] This is a special case of profile symbolic evaluation described in [9].

```
let ident =
  S → E :
    S.result = E.s1
    E.h1 = Nil
    E.h2 = undef
    E.h3 = undef
    E.h4 = undef
    E.h5 = undef

  E → Begin_A E′ :
    E.s1 = Begin_A E′.s1
    E.s2 = E′.s4
    E.s3 = E′.s5
    E.s4 = undef
    E.s5 = undef
    E′.h1 = End E′.s3
    E′.h2 = E′.s2
    E′.h3 = E.h1
    E′.h4 = E.h2
    E′.h5 = E.h3

  E → Begin_B E′ :
    E.s1 = Begin_B E′.s1
    E.s2 = E′.s4
    E.s3 = E′.s5
    E.s4 = undef
    E.s5 = undef
    E′.h1 = End E′.s3
    E′.h2 = E′.s2
    E′.h3 = E.h1
    E′.h4 = E.h2
    E′.h5 = E.h3

  E → End E′ :
    E.s1 = E.h2
    E.s2 = E.h1
    E.s3 = E′.s1
    E.s4 = E′.s2
    E.s5 = E′.s3
    E′.h1 = E.h3
    E′.h2 = E.h4
    E′.h3 = E.h5
    E′.h4 = undef
    E′.h5 = undef

  E → Nil :
    E.s1 = E.h1
    E.s2 = undef
    E.s3 = undef
    E.s4 = undef
    E.s5 = undef
```

Fig. 6. Result of descriptional composition

We can eliminate all the intermediate data constructions by repeating the above rewriting process on every projected semantic rule of every production rule in F. Note that, since we allow undefined attributes, the set of projected semantic rules may not have a semantic rule $(\mathtt{C}\ e_1\ \cdots\ e_n).h = e'$ for some inherited attribute h, where the attribute h is required to be defined by the corresponding semantic rule in G. In such a case, any subexpression $(\mathtt{C}\ e_1\ \cdots\ e_n).h$ should be understood to be representing an undefined value.

Renaming. The final step of descriptional composition is *renaming*. This step rewrites any successive attributions of the form $(x.a).b$ to a single attribution $x.a\_b$, by composing the successive two attribute names into a single one.

The result of descriptional composition applied to $unparse \circ parse^{(d)}$ (after an α-conversion on the attribute names) is shown in Figure 6. Note that the quasi-SSUR condition is preserved through the composition process, *i. e.*, the resulting AG is quasi-SSUR as well.

3.2 Deriving an event-based transformation program

The second phase of our algorithm generates a finite state transition machine for a one-pass event-based attribute evaluation from the deforested AG obtained in the previous phase.

Let GF denote the deforested AG. We can observe that both the input and output of the deforested AG GF are event streams and that every at-

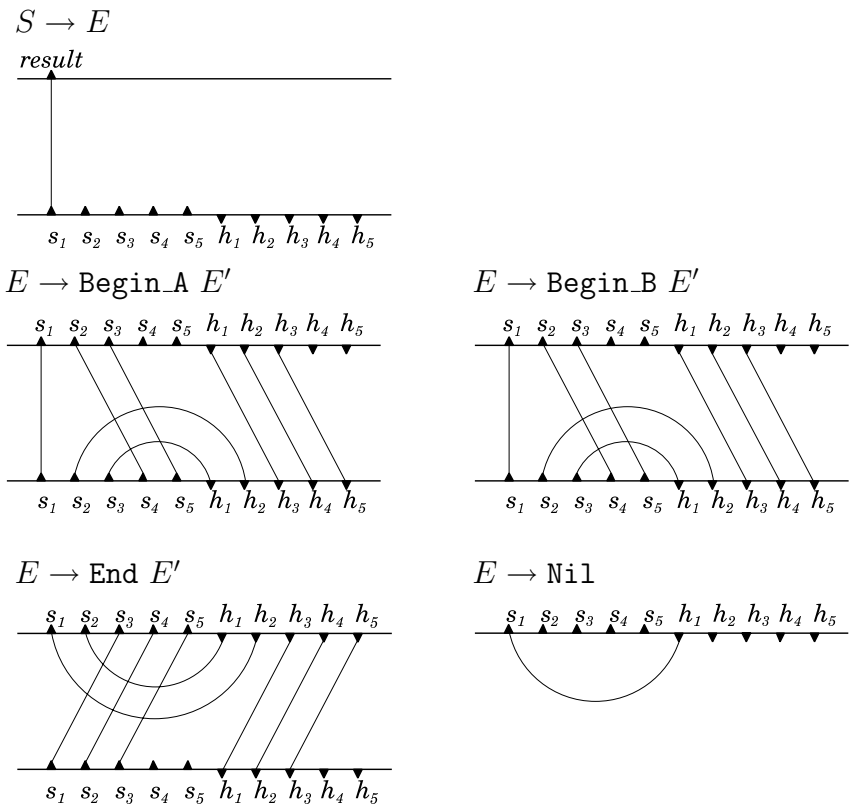


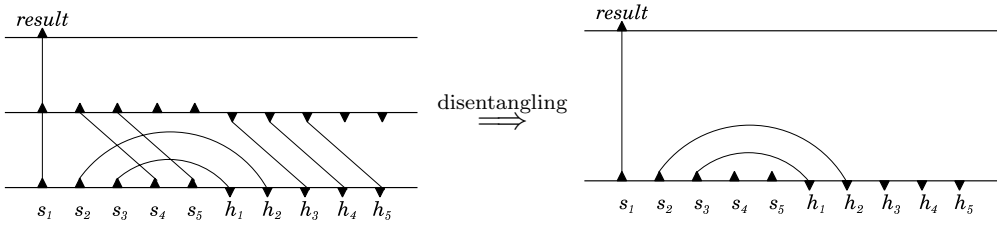
Fig. 7. Attribute dependency graphs

tribute of GF ranges over the set of event streams. In the following, we write $\Sigma_{\mathtt{C}}$ to denote the set of semantic rules of the production rule corresponding to each constructor C of event streams. Every expression defining a semantic rule is either a reference to an attribute occurrence or an event stream construction. We assume $\{syn_1, \ldots, syn_s\}$ be the set of synthesized attributes and $\{inh_1, \ldots, inh_h\}$ be the set of inherited attributes of GF.

3.2.1 A graph-based dependency analysis

To clarify what is the difficulty in generating an event-based program from an AG specification and to illustrate the key idea in our algorithm, we represent the set of semantic rules for each production rule by a graph as in Figure 7. In this graphical representation, we are concerned only with dependency between attribute occurrences, ignoring what semantic rules compute just for the moment.

Each graphical representation consists of two (or one) horizontal bars and some dependency edges. For every production rule of the form $E \to \mathtt{C}\ E'$, where E and E' are non-terminal symbols and C is a data constructor, the attribute occurrences on E is placed on the upper bar and those on E' on the lower bar. (No lower bar is present for the production rule $E \to \mathtt{Nil}$.) Each dependency edge connects two attributes, which are designated by triangle marks on the bars. We place synthesized attributes to the left and inherited



(a) $S \to E, E \to$ `Begin_A` E' (b) $S \to$ `Begin_A` E

Fig. 8. Disentangling dependency graphs

attributes to the right of each bar. The direction of data flow is indicated by the apex of each triangle mark.

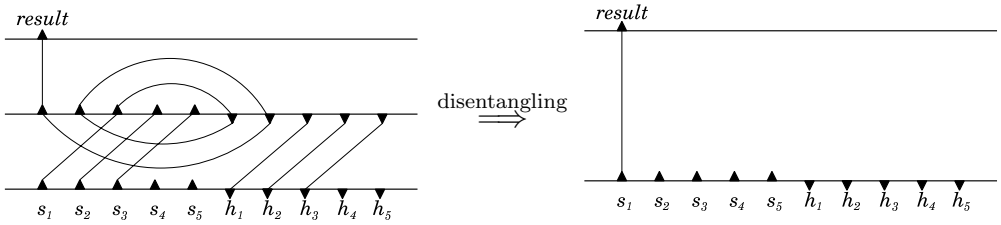
Note that, since each expression defining an attribute value is either a reference to an attribute occurrence or a data construction, each attribute occurrence is dependent on at most a single other occurrence. (An attribute occurrence *occ* has no incoming edges only if the semantic rule for the occurrence is either undefined or defined by $occ = \mathtt{C}_1(\ldots(\mathtt{C}_k\ \mathtt{Nil})\ldots)(k \geq 0)$.) Hence no attribute has more than two incoming edges. Furthermore, due to the quasi-SSUR condition, no attribute has more than two departing edges either.

We call an attribute dependency a *forward dependency*, if it departs from a synthesized attribute and comes to an inherited attribute on the same lower bar in a graph representation. For example, the dependency graph for the production rule `Begin_A` contains two forward dependency edges, from s_2 to h_2 and from s_3 to h_1, on the lower bar.

Forward dependencies are troublesome for one-pass attribute evaluation. In the present example, suppose the current event be `Begin_A`. To complete the process of the current event, a one-pass attribute evaluator would need the values of h_2 and h_1, which are to be passed for processing the next coming event. However, the values of h_2 and h_1 are dependent to s_2 and s_3, *resp.*, whose values are obtained only after the next event is processed. Therefore the one-pass evaluator gets stuck here.

In order to find a one-pass attribute evaluation strategy for the AG containing forward dependencies, our algorithm performs an analysis utilizing static lookahead into the input event stream.

The static analysis is based on a process which merges the sets of semantic rules for two successive production rules into a single one. Suppose the begging of the input stream is a parsing event `Begin_A`. The corresponding grammar productions are $S \to E$ followed by $E \to$ `Begin_A` E'. This successive productions promote two steps of attribute evaluation, whose attribute dependency is visualized in Figure 8(a). The graph is obtained by pasting the dependency graph of the latter production under that of the former one. This pasted graphs are unified into a single one, by omitting the middle bar (corresponding to the intermediate production) and then taking the transitive



(a) $S \rightarrow$ `Begin_A` $E, E \rightarrow$ `End` E' (b) $S \rightarrow$ `End` E

Fig. 9. Exploring new parsing state via disentangling

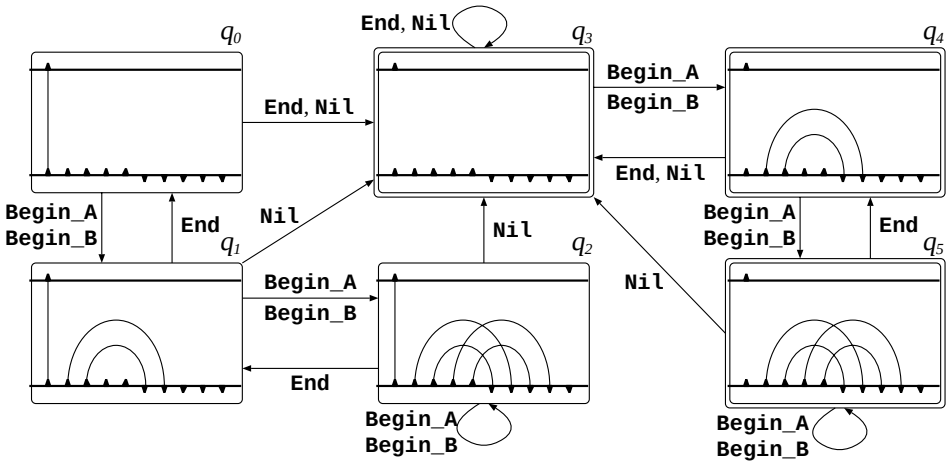


Fig. 10. A state transition diagram

closure of dependency edges (Figure 8(b)). We call this process for taking the transitive closure a *disentangling* process.

In the following, we represent a dependency graph by a finite map $\mathcal{D}$, where $\mathcal{D}(occ) = occ'$ for every occ whose attribute definition is dependent to occ'; If occ is not dependent to any occurrence, we write $\mathcal{D}(occ) = none$, where $none$ is a special symbol indicating no dependency.

3.2.2 Generating a finite state transition machine

Our algorithm constructs an attribute evaluator by regarding each attribute dependency graph generated by the disentangling process as a representation of a new state of parsing. For example, suppose that we are at the parsing state represented by the dependency graph in Figure 8(b) and that `End` be the next event to follow. To obtain the new parsing state, we apply the disentangling process again, *i. e.*, the last disentangled graph (Figure 8(b)) and the dependency graph for the production rule $E \rightarrow$ `End` E' are pasted together (Figure 9(a)), and then they are disentangled. The new parsing state is represented by the graph in Figure 9(b). Notice that the graph is isomorphic to that of the start production, hence we are at a state equivalent to the initial state of parsing.

Our algorithm constructs a parser for attribute evaluation by enumerating

all possible dependency graphs, starting from the dependency graph of the start production. The algorithm examines every possible next parsing event (namely, $\texttt{Begin\_tag}_1$, ..., $\texttt{Begin\_tag}_n$, `End`, and `Nil`) for each different dependency graph. Since there are only finite number of attributes, there are only finite varieties of dependency graphs and therefore this process must terminate. As for the running example, we obtain 6 patterns of dependency graphs as illustrated in Figure 10. Regarding each graph as a state of parsing, we can view the diagram of Figure 10 as a finite state transition machine. Starting from the initial state q_0, the machine repeatedly changes its state according to the parsing event to be read, following an arrow labeled with the event name. The machine terminates when a final state is reached; A dependency graph represents a finite state if the attribute *result* has no dependency edge. In the figure, the final states are framed by a double line.

Though the state transition rule can be statically determined by an analysis on attribute dependencies, it remains unsolved how to compute attribute values during transitions. The solution in this paper is to let each attribute carry a partially evaluated value. A partially evaluated value is expressed by a unary function, which returns the fully evaluated value when it is applied to a value of the not-yet evaluated attribute.

Since semantic rules are restricted to event stream constructions, such a functional representation of partially evaluated values can be expressed by a composition of a finite number of function constants $\texttt{Begin\_tag}_1$, ..., $\texttt{Begin\_tag}_n$, `End`, `Nil`, `Id`, and `Undef`, where the last three constants should be understood as functions $\texttt{Nil} = \lambda x.\texttt{Nil}$, $\texttt{Id} = \lambda x.x$, and $\texttt{Undef} = \lambda x.\mathit{undef}$ (*i. e.*, a function which is not defined for any input), *resp.* We can regard the function constants $\texttt{Begin\_tag}_1, \ldots$ as meta-symbols and $\circ$ as a composition operator over them, where the operator is associative and a few additional equalities $\texttt{Id} \circ f = f \circ \texttt{Id} = f$, $\texttt{Nil} \circ f = \texttt{Nil}$, and $\texttt{Undef} \circ f = f \circ \texttt{Undef} = \texttt{Undef}$ hold. We call such a representation a *symbolic representation* of partially evaluated attributes. We can assume that every semantic rule is given in the form either $\mathit{occ} = (\texttt{C}_1 \circ \cdots \circ \texttt{C}_k \circ \texttt{Nil})(\mathit{none})$, or $\mathit{occ} = (\texttt{C}_1 \circ \cdots \circ \texttt{C}_k)(\mathit{occ}')$, where $\texttt{C}_1, \ldots, \texttt{C}_k (k \geq 0)$ are neither `Nil` nor `Undef`.

The state transition machine changes the syntactic representation of partially evaluated values for every state transition, and only the partially evaluated values for inherited attributes are passed during state transitions. We use meta-variables $X_{\mathit{inh}_1}, \ldots, X_{\mathit{inh}_h}$ to denote those partially evaluated values of inherited attributes, and the meta-variables may occur in a symbolic representation to refer to the partially evaluated attributes of the inherited attributes passed from the preceding parsing state.

The definition of the finite state transition machine is given below.

Definition 3.1 (Finite State Transition Machine) *Let FRep denote the set of finite maps of the form* $\{\mathit{result} \mapsto f_0, \mathit{inh}_1 \mapsto f_1, \ldots, \mathit{inh}_h \mapsto f_h\}$*, which assigns a symbolic representation* f_i *to each attribute.*

A finite state transition machine *is a septuple* $\mathcal{M} = (A, Q, q_0, T, \delta, \Gamma, \gamma_0)$,

where A is the set of input parsing events, Q is the set of states (the set of patterns of dependency graphs), $q_0 (\in Q)$ is the initial state, $T (\subseteq Q)$ is the set of final states, δ is a finite map $Q \times A \to Q$ representing the set of state transition rules, Γ is a finite map $Q \to (A \to FRep)$ representing the set of rules for computing the symbolic representation of partially evaluated attribute values for each state transition, and $\gamma_0 (\in FRep)$ is a finite map giving the initial symbolic representation.

The state transition machine, if the current state is q and the next parsing event to be read is `C`, makes a transition to the state $\delta(q, \texttt{C})$. The symbolic representation for the new state is given by a finite map $\Gamma(q)(\texttt{C})$. As for the running example, when the machine reads a parsing event `Begin_A` at the initial state q_0, the symbolic representation is given by $\Gamma(q_0)(\texttt{Begin\_A}) = \{result \mapsto \texttt{Begin\_A}, h_1 \mapsto \texttt{End}, h_2 \mapsto \texttt{Id}, h_3 \mapsto X_{h_1}, h_4 \mapsto X_{h_2}, h_5 \mapsto X_{h_3}\}$, where the meta-variables $X_{h_1}, X_{h_2}, X_{h_3}$ designate the initial symbolic representation of the corresponding inherited attributes, in this case. During the state transitions, the finite state transition machine does not hold the value of synthesized attributes, except *result*, since *result* is the only relevant synthesized attribute to the final transformation result. The value of the other synthesized attributes is combined with the value of either *result* or an inherited attribute h_i by the disentangling procedure.

To give an algorithm for generating such a finite state transition machine, we first define a procedure $DISENT^{\mathcal{D},\texttt{C}}$ for disentangling. The superscript $\mathcal{D}$ is the attribute dependency graph representing the current parsing context and `C` is the next parsing event to be read. Let $E \to \texttt{C}\ E'$ (or $E \to \texttt{Nil}$) be the corresponding production rule and $\Sigma_{\texttt{C}}$ be the set of associated semantic rules.

Algorithm 1 (Disentangling)

procedure $DISENT^{\mathcal{D},\texttt{C}}(occ)$
  input *occ*: an attribute occurrence to be disentangled
  procedure $DISENT2(occ, f)$
  case *occ* **of**
    $S.result \Rightarrow$ **if** $\mathcal{D}(S.result) \neq none$ **then** $DISENT2(\mathcal{D}(S.result), \texttt{Id})$
      else return $(none, \texttt{Id})$
    $E.inh_j \Rightarrow$ **if** $\mathcal{D}(E.inh_j) \neq none$ **then** $DISENT2(\mathcal{D}(E.inh_j), f \circ X_{inh_j})$
      else return $(none, f \circ X_{inh_j})$
    $E.syn_k \Rightarrow$ **if** $E.syn_k = f'(occ') \in \Sigma_{\texttt{C}}$ **then**
        if $occ' \neq none$ **then** $DISENT2(occ', f \circ f')$
        else return $(none, f \circ f')$
      else return $(none, f)$
    $E'.inh_j \Rightarrow$ **if** $E'.inh_j = f'(occ') \in \Sigma_{\texttt{C}}$ **then**
        if $occ' \neq none$ **then** $DISENT2(occ', f')$
        else return $(none, f')$
      else return $(none, \texttt{Id})$

$E'.syn_k \Rightarrow$ **return** $(E.syn_k, f)$
end

return *DISENT2*$(occ, \mathtt{Id})$
end

A procedure call $DISENT^{\mathcal{D},\mathtt{C}}(occ)$ traverses the attribute dependencies and returns a pair (occ', f) of an attribute occurrence and a symbolic representation: occ' is the occurrence on which occ depends and f is the symbolic representation of the function which takes the value of occ' and computes the value of occ.

Now we are ready to present the algorithm which translates the deforested AG GF to a finite state transition machine.

Algorithm 2 (Finite state transition machine construction) *Let $\mathcal{D}_0$ and $\mathcal{F}_0$ be the attribute dependency graph and the symbolic representation for the start production $S \to E$ respectively,* i. e.*,*

$$\mathcal{D}_0(occ) = \begin{cases} occ' & (\text{if } occ = f(occ') \in \Sigma_S) \\ none & (\text{otherwise}) \end{cases}$$

$$\mathcal{F}_0(a) = \begin{cases} f & (\text{if } a = result \text{ and } S.result = f(occ) \in \Sigma_S) \\ g & (\text{if } a = inh_i \text{ and } E.inh_i = g(occ) \in \Sigma_S) \\ \mathtt{Undef} & (\text{otherwise}) \end{cases}$$

where Σ_S is the set of semantic rules for the start production.

The algorithm is defined by the following procedure MakeFSTM.

procedure *MakeFSTM*(GF)
input GF: the AG derived by descriptional composition
$A := \{\mathtt{Begin\_tag}_1, \ldots, \mathtt{Begin\_tag}_n, \mathtt{End}, \mathtt{Nil}\}$; $Q := \{\mathcal{D}_0\}$; $q_0 := \mathcal{D}_0$
$T := \emptyset$; $\delta := \{\}$; $\Gamma := \{\}$; $\gamma_0 := \mathcal{F}_0$
while there exists $\mathcal{D} \in Q$ such that $\Gamma(\mathcal{D})$ is not defined **do**
  $\Gamma_{\mathcal{D}} := \{\}$
  for each $\mathtt{C} \in A$ **do**
    $\mathcal{D}^{\mathtt{C}} := \{\}$; $\mathcal{F}^{\mathtt{C}}_{\mathcal{D}} := \{\}$
    $(occ, f) := DISENT^{\mathcal{D},\mathtt{C}}(S.result)$;
    $\mathcal{D}^{\mathtt{C}} := \mathcal{D}^{\mathtt{C}} \uplus \{S.result \mapsto occ\}$; $\mathcal{F}^{\mathtt{C}}_{\mathcal{D}} := \mathcal{F}^{\mathtt{C}}_{\mathcal{D}} \uplus \{result \mapsto f\}$;
    if $\mathtt{C} \neq \mathtt{Nil}$ **then**
      for each inh_j **do**
        $(occ, f) := DISENT^{\mathcal{D},\mathtt{C}}(E'.inh_j)$;
        $\mathcal{D}^{\mathtt{C}} := \mathcal{D}^{\mathtt{C}} \uplus \{E.inh_j \mapsto occ\}$; $\mathcal{F}^{\mathtt{C}}_{\mathcal{D}} := \mathcal{F}^{\mathtt{C}}_{\mathcal{D}} \uplus \{inh_j \mapsto f\}$
      end
    $\delta := \delta \uplus \{(\mathcal{D}, \mathtt{C}) \mapsto \mathcal{D}^{\mathtt{C}}\}$; $\Gamma_{\mathcal{D}} := \Gamma_{\mathcal{D}} \uplus \{\mathtt{C} \mapsto \mathcal{F}^{\mathtt{C}}_{\mathcal{D}}\}$; $Q := Q \cup \{\mathcal{D}^{\mathtt{C}}\}$
    if $\mathcal{D}^{\mathtt{C}}(S.result) = none$ **then** $T := T \cup \{\mathcal{D}^{\mathtt{C}}\}$
  end
  $\Gamma := \Gamma \uplus \{\mathcal{D} \mapsto \Gamma_{\mathcal{D}}\}$
end

return $\mathcal{M} := (A, Q, q_0, T, \delta, \Gamma, \gamma_0)$

The procedure *MakeFSTM* enumerates all possible attribute dependency graphs, starting from the initial dependency graph $\mathcal{D}_0$. For every dependency graph $\mathcal{D}$, it examines every parsing event `C` in turn as a possible parsing event to be read, and computes the new dependency graph $\mathcal{D}^{\mathtt{C}}$ and the new assignment $\mathcal{F}^{\mathtt{C}}_{\mathcal{D}}$ of symbolic representations. If `C` = `Nil`, since no parsing event follows, we need not compute the assignments for inherited attributes $inh_1, \ldots, inh_h$. The dependency graph $\mathcal{D}^{\mathtt{C}}$ is added to the set T of final states, if it has no dependency for the occurrence $S.result$. This process is iterated until there is no new dependency graph is generated.

A graph based analysis of attribute dependency similar to the present scheme can be found in Kastens' visit sequence construction [12], in which the attribute evaluation order is derived by an exhaustive enumeration of all possible attribute dependencies. The crucial difference between his and ours is that his scheme preserves the original production rules while ours pastes together production rules to create a new one. The attribute evaluators derived from these two schemes are very different as well. Kastens' visit sequence evaluator evaluates only a subset of attributes per a visit to a syntax node and hence it is a multi-pass evaluator. On the contrary, ours is a one-pass evaluator, which partially evaluates all attributes per a visit to a node.

3.3 Translation into an event-based program

The final phase of our algorithm translates the finite state transition machine into an event-based program.

It needs a further development to obtain a one-pass interactive event-based document transformer. One may directly map the finite state transition machine $\mathcal{M} = (A, Q, q_0, T, \delta, \Gamma, \gamma_0)$ into a functional program, which is not interactive, though. The functional program is defined by a set of mutual recursive functions $\{F_{\mathcal{D}} \mid \mathcal{D} \in Q \setminus T\}$, where each function $F_{\mathcal{D}}$ corresponds to the state $\mathcal{D} \in Q$ and partially evaluated values are expressed by a function closure. The resulting mutual recursive functional program is not executed interactively, since the result is carried around as a function closure and hence the result is obtained only at the end of parsing.

In order to obtain an interactive program, we express a partially evaluated attribute value by a concatenation list, which is either an empty list `[]`, a singleton list `[C]` (a list with only one element `C`), or a concatenation $L_1@L_2$ of two lists L_1 and L_2. A translation of a symbolic representation f into a concatenation list, denoted by $\lceil f \rceil$, is defined by the following equations.

$$\lceil \mathtt{Id} \rceil = \mathtt{[]} \quad \lceil X_{att} \rceil = X_{att} \quad \lceil \mathtt{C} \rceil = \mathtt{[C]} \quad \lceil f_1 \circ f_2 \rceil = \lceil f_1 \rceil @ \lceil f_2 \rceil$$

where `C` is either $\mathtt{Begin\_tag}_1, \ldots, \mathtt{Begin\_tag}_n, \mathtt{End}, \mathtt{Nil}$, or `Undef`. Due to the equalities that hold for the functional composition operator (Section 3.2.2), we can identify concatenation lists up to the associativity of list concatenation and the equalities $\mathtt{[Id]}@L = L@\mathtt{[Id]} = L$, $\mathtt{[Nil]}@L = \mathtt{[Nil]}$, and $\mathtt{[Undef]}@L =$

```
procedure MAIN(s)
    flush [];  call F_0(s, [Nil], [Undef], [Undef], [Undef], [Undef])
procedure F_0(s, X_h1, X_h2, X_h3, X_h4, X_h5)
    case s of
        Begin_A :: s' ⇒ flush [Begin_A];  call F_1(s', [End], [], X_h1, X_h2, X_h3)
        Begin_B :: s' ⇒ flush [Begin_B];  call F_1(s', [End], [], X_h1, X_h2, X_h3)
        End :: s'     ⇒ flush X_h2;       exit
        Nil           ⇒ flush X_h1;       exit
    end
procedure F_1(s, X_h1, X_h2, X_h3, X_h4, X_h5)
    case s of
        Begin_A :: s' ⇒ flush [Begin_A];  call F_2(s', [End], [], X_h1, X_h2, X_h3)
        Begin_B :: s' ⇒ flush [Begin_B];  call F_2(s', [End], [], X_h1, X_h2, X_h3)
        End :: s'     ⇒ flush X_h2@X_h1;
                        call F_0(s', X_h3, X_h4, X_h5, [Undef], [Undef])
        Nil           ⇒ flush [Undef];    exit
    end
procedure F_2(s, X_h1, X_h2, X_h3, X_h4, X_h5)
    case s of
        Begin_A :: s' ⇒ flush [Begin_A];  call F_2(s', [End], [], X_h1, X_h2, [Undef])
        Begin_B :: s' ⇒ flush [Begin_B];  call F_2(s', [End], [], X_h1, X_h2, [Undef])
        End :: s'     ⇒ flush X_h2@X_h1;
                        call F_1(s', X_h3, X_h4, X_h5, [Undef], [Undef])
        Nil           ⇒ flush [Undef];    exit
    end
```

Fig. 11. An interactive event-based program for identity transformation

L@[Undef] = [Undef].

An event-based interactive program, which uses the list representation of attribute values, is derived by the following translation algorithm. We assume that **flush** is a primitive command which takes a concatenation list of parsing events and write out the events to the output stream. If the argument is [Undef], the command aborts the execution with a notification of an error to the other end of the output stream. We also assume a primitive command **exit** terminates the execution successfully.

Algorithm 3 (Translation to an interactive event-based program) *Let $\mathcal{M} = (A, Q, q_0, T, \delta, \Gamma, \gamma_0)$ be a finite state transition machine. The machine $\mathcal{M}$ is translated into a program which consists of the set of procedures $\{F_{\mathcal{D}} | \mathcal{D} \in Q \setminus T\}$ and an initial procedure $MAIN$.*

For each state $\mathcal{D} \in Q \setminus T$, we derive a procedure:

```
procedure F_D(s, X_inh1, ..., X_inhh)
    case s of
        Begin_tag_1 :: s' ⇒ P_{D,Begin_tag_1}
                          ⋮
```

$\texttt{Begin\_tag}_{\mathrm{n}} :: s' \Rightarrow P_{\mathcal{D},\texttt{Begin\_tag}_{\mathrm{n}}}$
$\texttt{End} :: s' \Rightarrow P_{\mathcal{D},\texttt{End}}$
$\texttt{Nil} \Rightarrow P_{\mathcal{D},\texttt{Nil}}$
end

where s is the input stream, $\texttt{C} :: s' \Rightarrow P_{\mathcal{D},\texttt{C}}$ (or $\texttt{C} \Rightarrow P_{\mathcal{D},\texttt{C}}$ in the case $\texttt{C} = \texttt{Nil}$) is a pattern matching construct, which executes the program block $P_{\mathcal{D},\texttt{C}}$, when the next parsing event matches $\texttt{C}$ with the rest of the input stream being bound to the variable s'. Each program block $P_{\mathcal{D},\texttt{C}}$ is defined by:

$$P_{\mathcal{D},\texttt{C}} = \begin{cases} \textbf{flush } L_{result}; \textbf{ call } F_{\delta(\mathcal{D},\texttt{C})}(s', L_{inh_1}, \ldots, L_{inh_h}) & (\text{if } \delta(\mathcal{D},\texttt{C}) \notin T) \\ \textbf{flush } L_{result}; \textbf{ exit} & (\text{otherwise}) \end{cases}$$

where $L_{result} = \lceil \Gamma(\mathcal{D})(\texttt{C})(result) \rceil$ and $L_{inh_i} = \lceil \Gamma(\mathcal{D})(\texttt{C})(inh_i) \rceil$ for every i.

Finally, the initial procedure $MAIN$ is defined as follows.

procedure $MAIN(s)$
flush $\lceil \gamma_0(result) \rceil$; **call** $F_{q_0}(s, \lceil \gamma_0(inh_1) \rceil, \ldots, \lceil \gamma_0(inh_h) \rceil)$

For example, the finite state transition machine for the identity transformation (Figure 10) is translated into the program shown in Figure 11. The resulting program is a completely interactive, one-pass event-based document transformer. As it reads a parsing event, it instantly outputs a parsing event either `Begin_A`, `Begin_B`, or `End` correspondingly.

4 Definable Document Transformations and Future Extension

The proposed algorithm can automatically derive an event-based document transformation program, but it only applies to those AG specifications which satisfy the quasi-SSUR condition (Section 2.2). This section shows how several significant transformations can be defined under the restriction and suggests an extension of the present framework for relaxing the restriction.

4.1 Defining Document transformations in quasi-SSUR AGs

We show how document transformations can be defined in quasi-SSUR AGs, through a two typical classes of document transformations, (i) simple filters and (ii) context-dependent transformations.

First, simple filters such as tag renaming, elimination of unnecessary tags, replacement of particular nodes, etc. are easily defined in a quasi-SSUR AG. Our algorithm works for those simple filters and the resulting transformer does not construct any extra intermediate data. Furthermore, we can compose these simple filters together to construct a more complicated transformation. The composition of filters can be processed by a repeated application of the descriptional composition, due to the SSUR closure property [11, Soundness Theorem]. This provides a more modular way for constructing event-based transformers than directly programming with SAX API.

The other class of typical document transformations, context-dependent transformations, are more difficult to define in quasi-SSUR AGs. Consider a context-dependent transformation which eliminates every `B` node whose immediate parent is an `A` node. One might define this transformation as follows.

$$
\begin{array}{lll}
\texttt{let } \mathit{elimBunderA} = & & \\
\quad S \to T : & S.\mathit{result} = T.\mathit{val} & \\
\quad T \to \texttt{Node\_A } T_1\ T_2 : & T.\mathit{val} = \texttt{Node\_A } T_1.\mathit{elim}\ T_2.\mathit{val} & \\
 & T.\mathit{elim} = \texttt{Node\_A } T_1.\mathit{elim}\ T_2.\mathit{elim} & \\
\quad T \to \texttt{Node\_B } T_1\ T_2 : & T.\mathit{val} = \texttt{Node\_B } T_1.\mathit{elim}\ T_2.\mathit{val} & \\
 & T.\mathit{elim} = T_2.\mathit{elim} & \\
\quad T \to \texttt{Empty} : & T.\mathit{val} = \texttt{Empty} & T.\mathit{elim} = \texttt{Empty}
\end{array}
$$

The idea in this definition is to let every node compute two attributes *val* and *elim*, which hold the result of transformation for the two possible different contexts respectively, and to let parent nodes select either of them appropriately. However, this AG is not quasi-SSUR, since it has duplicated uses of $T_1.\mathit{elim}$ in the second production rule.

We can circumvent the difficulty by utilizing the assumption that the input document has a finite depth less than or equal to d. We let a set of attributes of the form $\mathit{val}_{b_k \ldots b_1} (0 \leq k \leq d-1)$ represent the result of transformation for different contexts, where the annotation $b_k \ldots b_1$ is a sequence of $\{1, 0\}$ with each b_i indicating if the i-th closer parent of the current document node is `A` or not. We can give semantic rules without violating the quasi-SSUR condition. For example, the production rule $T \to \texttt{Node\_B } T_1\ T_2$ is given a set of semantic rules: $T.\mathit{val}_{\bar{b}0} = \texttt{Node\_B } T_1.\mathit{val}_{\bar{b}00}\ T_2.\mathit{val}_{\bar{b}0}$ and $T.\mathit{val}_{\bar{b}1} = T_2.\mathit{val}_{\bar{b}1}$ for every $\{1, 0\}$-sequence $\bar{b}$ of a length less than $d-1$, and $T.\mathit{val}_{\varepsilon} = \texttt{Node\_B } T_1.\mathit{val}_0\ T_2.\mathit{val}_{\varepsilon}$. Applying our algorithm to this AG (with $d = 2$ for $\mathit{parse}^{(d)}$) produces a finite state transition machine with 28 states.

Remark. The degree of the reduction in the memory usage achieved by the derived event-based program varies, depending on each transformation. Our algorithm is not intended to achieve a reduction in the memory usage for all the transformations but only tries to *minimize* the memory usage for each transformation.

Consider the following example, which reverses the order of sibling nodes at every nesting level.

$$
\begin{array}{lll}
\texttt{let } \mathit{hrev}\ = & & \\
\quad S \to T : & S.\mathit{result} = T.\mathit{hrev}\ T.\mathit{acc} = \texttt{Empty} & \\
\quad T \to \texttt{Node\_A } T_1\ T_2 : & T.\mathit{hrev} = T_2.\mathit{hrev} & \\
 & T_1.\mathit{acc} = \texttt{Empty} & T_2.\mathit{acc} = \texttt{Node\_A } T_1.\mathit{hrev}\ T.\mathit{acc} \\
\quad T \to \texttt{Node\_B } T_1\ T_2 : & T.\mathit{hrev} = T_2.\mathit{hrev} & \\
 & T_1.\mathit{acc} = \texttt{Empty} & T_2.\mathit{acc} = \texttt{Node\_B } T_1.\mathit{hrev}\ T.\mathit{acc} \\
\quad T \to \texttt{Empty} : & T.\mathit{hrev} = T.\mathit{acc} &
\end{array}
$$

Any event-based transformer for this transformation would need to buffer all the input events until the end of the input is reached, in order to output

earlier events later. This indicates that it is inherently difficult to avoid the buffering of events this transformation. The problem resides in the transformation itself, and hence no event-based transformer would be able to achieve a reasonable reduction in the memory usage. Therefore, when our algorithm is applied to this transformation, the derived event-based transformer, instead of constructing a document tree, would need to buffer all the parsing events into a list. We thus would not gain a remarkable improvement on the the memory usage for this transformation.

4.2 Future Extension

As we have seen above, our algorithm can derive event-based document transformers for a class of simple document filters and their combinations effectively and also a context-dependent transformation. We believe that a certain class of context-dependent transformations can be likewise expressed as above in quasi-SSUR AGs. However, the method used above for expressing the context-dependent transformation would not scale up: the result of transformation is carried around by a set of attributes representing varying results for different contexts. The semantic rules must be carefully coded so that they respect the complicated context-dependency.

Though the current method is not powerful enough, the authors believe that it can be extended to support more powerful document transformations. The most crucial problem in the current framework is that tag names are encoded in a variety of constructors (*i. e.*, `Node_A`, `Node_B`, etc.) If tag names are embedded as a value in constructors (say, `(Node "A" Empty (Node "B" Empty Empty))`) and "semantic values" (*i. e.*, tag names, booleans, integers, etc.) are allowed as attribute values, then context-dependent transformations would be cleanly expressed by using conditionals on the semantic values. Descriptional composition for AGs with conditionals has been studied by Boyland and Graham [7] and they presented a relaxed SSUR condition, called a SAMODUR condition, which allows the same attribute to be referred to many times but at most once in each different branch of conditionals. We would benefit from the increased expressiveness, if we can extend the present method so that it can apply to the conditional SAMODUR AGs.

5 Conclusion and Future Work

We have given an algorithm which derives a one-pass event-based document transformation program from a tree-based specification of a document transformation. We have formalized the problem in the framework of attribute grammars (AGs) and have solved it by an application of the descriptional composition followed by a derivation of an interactive attribute evaluator based on an analysis of attribute dependency graphs. The contribution of the present paper is the algorithm for deriving the evaluator. The algorithm generates a

finite state transition machine and translates it to an interactive event-based transformer.

The authors implemented the algorithm in a prototype program which generates event-based transformers over the simple XML-like markup language given in Section 2.1. They are currently working for the extension mentioned in Section 4.2. The extension would bring a great increase in the expressiveness of document tree transformation. The extension would be also useful for enriching document structure with embedded plain texts and unordered labeled data (*i. e.*, *character data* and *attributes*, *resp.*, in XML jargon). We hope that we will be able to report the result of the extension in a foreseeable future.

References

[1] Extensible markup language (XML) 1.0 W3C recommendation. `http://www.w3.org/TR/1998/REC-xml-19980210`, 1998.

[2] XSL transformations (XSLT) version 1.0 W3C recommendation. `http://www.w3.org/TR/1999/REC-xslt-19991116`, 1999.

[3] Document object model (DOM). `http://www.w3.org/DOM/`, 2000.

[4] SAX 2.0: The simple API for XML. `http://www.megginson.com/SAX/`, 2000.

[5] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web*. Morgan Kaufmann, 2000.

[6] A. V. Aho and J. D. Ullman. *The theory of parsing, translation, and compiling*, volume I & II. Prentice-Hall, 1972.

[7] J. Boyland and S. L. Graham. Composing tree attributions. In *Proceedings of the* 21^{th} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 375–388. ACM Press, 1994.

[8] L. Correnson, E. Duris, D. Parigot, and G. Roussel. Symbolic composition. Technical Report 3348, INRIA, Jan. 1998. available from `ftp://ftp.inria.fr/INRIA/publication/RR/RR-3348.ps.gz`.

[9] L. Correnson, E. Duris, D. Parigot, and G. Roussel. Declarative program transformation: A deforestation case-study. In *Principles and Practice of Declarative Programming*, volume 1702 of *LNCS*, pages 360–377. Springer Verlag, 1999.

[10] H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, volume 19 of *SIGPLAN Notices*, pages 157–170, June 1984.

[11] R. Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25(4):355–423, May 1988.

[12] U. Kastens. Ordered attribute grammars. *Acta Informatica*, 13(3):229–256, 1980.

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

[14] P. Wadler. Deforestation: Transforming programs to eliminate trees. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, pages 344–358. Springer Verlag, 1988.