ELECTRONIC PUBLISHING, VOL. 6(4), 361–372 (DECEMBER 1993)

SIMON: A grammar-based transformation system for structured documents

AN FENG AND TOSHIRO WAKAYAMA

Xerox Corporation
Webster Research Center
Webster, New York 14580, USA

SUMMARY

SIMON is a grammar-based transformation system for restructuring documents. Its target applications include meta-level specification of document assembly, view definition and retrieval for multiview documents, and document type evolution. The internal document model is based on attribute grammars, and it interfaces with external document models such as SGML through input and output conversion. The transformation engine of SIMON is an amalgamation of syntax-directed computation and content-oriented computation: the former is through higher-order (and related) extensions of attribute grammars whereas the latter is done by externally defined programs and it is for computation not naturally amenable to the syntax-directed paradigm. The current implementation of SIMON employs the higher-order extension proposed in [1] for the syntax-directed computation, and C++ for the content-oriented computation.

KEY WORDS Structured documents Document transformation Document type evolution Document assembly Multiview documents Attribute grammars

1 INTRODUCTION

One of the challenging issues in the study of structured documents is how to reuse documents and document components across heterogeneous types (e.g., manuals, letters, articles within SGML [2]) and meta-types (e.g., SGML, ODA, and possibly LaTeX). The main difficulty of the problem is that document components to be reused have their own internal structures which are in general foreign to the new document being constructed. If the generic structure of the new document is to remain the same and not to be extended or modified to accommodate the structures of the reuse components, the reuse components themselves must be structurally transformed to fit into the new generic structure. Thus, the idea of document transformation becomes a key technical issue, as pioneered by [3,4].

A major advantage of structured documents is that documents are not simply structured, but also the generic structural description itself is given a formal representation (e.g., Document Type Definition in SGML). The generic structure descriptions enable document transformation to be specified at meta-level. Our study leverages such meta capabilities of structured documents in order to strengthen and broaden the effectiveness and scope of applications of document transformation. To this end, we have been developing a system called *SIMON*[1] based on attribute grammars and their higher-order extensions (and other related extensions), which have been receiving much attention recently in the area of programming language studies [1,5,6,7].

[1] “SI” and “MON” stand for *restructuring* and *documents* respectively in Japanese.

CCC 0894–3982/93/040361–12
Received 15 August 1993
Revised 1 December 1993

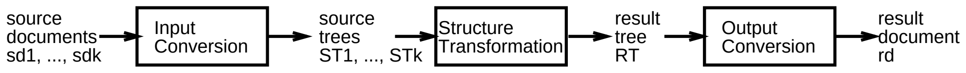


Figure 1. Framework for document transformations

Figure 1 depicts the general framework of how SIMON works. The internal document representation model of SIMON is exactly as in [8]. Namely, documents are given by attributed trees (*i.e.*, derivation trees where each node is associated with a set of attribute values) and their type definitions by attribute grammars [9]. The transformation specification in SIMON has three components: a collection of *source* attribute grammars, a *result* attribute grammar, and a mapping that takes a collection of *source* trees (of the source grammars) and produce a *result* tree (of the result grammar). Such mappings are specified by higher-order (and related) extensions of attribute grammars. In the current implementation, the transformation engine of SIMON is based on the higher-order extension proposed by [1]. The SIMON transformation engine will be discussed in detail in later sections. SIMON also interfaces with external document models such as SGML through *input* and *output conversions* (see Figure 1).

Some of the target applications of SIMON are:

- Document Assembly Specification
  It is often the case that the process of document reuse itself is highly structured, i.e., instances of the same component classes are combined into a new document in the same manner. For instance, monthly project reports may be constructed, to a large extent, from individual monthly reports, trip reports, publication databases, tables of experimental raw data, etc. Since these documents are, in general, in different meta-types, we first convert them into the uniform internal representation of SIMON as attributed trees and then apply the assembly specification given by a higher-order attribute grammar.
- Document View Retrieval
  While document assembly tends to build more complex documents out of simpler ones, document view retrieval refers to the inverse process of extracting simpler documents out of complex *multiview* documents. One of the problems with such complex documents is that they intertwine multiple views in a way quite cumbersome and sometimes nearly impossible for the human reader to comprehend. In SIMON, these views can be specified by higher-order attribute grammars, and they can be computationally retrieved through attribute evaluation.
- Document Type Evolution
  When a document type is designed, it reflects the properties of the objects and events of documentation and the designer's perception and understanding of them. As time goes by, the nature of these objects and events as well as the designer's perception and understanding of them may change. In fact, such changes may accumulate to an extent that the old type is no longer appropriate and a new type definition is desirable. This is analogous to the problem of schema evolution in database design, and the issue has been discussed in document settings as well [4]. In [4], the objects of transformation are derivation trees without attributes. By extending them to attributed trees, we enrich the class of transformations that can be specified and computed. For instance, [8] identifies various classes of *context-dependent* transformations.

```
<!DOCTYPE  article [
<!ELEMENT  article    0  O   (title, (sentence | citation)*)   >
<!ELEMENT  title      -  O   (#PCDATA)                         >
<!ELEMENT  sentence   -  O   (#PCDATA)                         >
<!ELEMENT  citation   -  -   (#PCDATA)                         > ]>
<title> AG   <sentence> AG was introduced in 1968. <citation> intro </citation>
<sentence> It has been widely used. <citation> app </citation><citation> else </citation>
```

(a) article sd_1 with citation in SGML style

```
@book{app,      author = "A.Guy",    title = "Application systems" }
@article{tran,  author = "S.Lady",   title = "Structure transformation" }
@article{intro, author = "G.Prof",   title = "Attribute grammars" }
```

(b) bibliography document sd_2 in BibTeX style

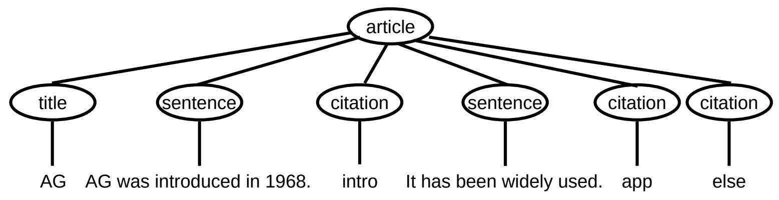
```
\documentstyle{article}
\begin{document}
\title{AG}
        AG was introduced in 1968.[2]    It has been widely used.[1][?]
\section*{Reference}
        \begin{itemize}
                \item{[1]} A.Guy: "Application Systems".
                \item{[2]} G.Prof: "Attribute grammars".
        \end{itemize}
\end{document}
```

(c) article *rd* with reference in LaTeX style

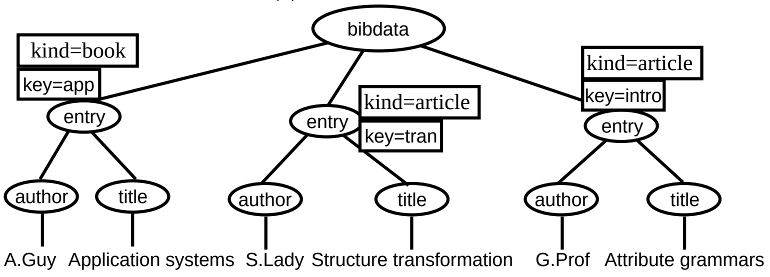
Figure 2. Transformation from SGML and BibTeX documents to LaTeX document

Another important feature of SIMON is that its transformation engine is an amalgamation of two computational paradigms: syntax-directed computation through attribute grammars and content-oriented, domain-specific computations through programs external to the grammar-based specification. Although higher-order attribute grammars can express arbitrary computations, certain computations become quite unnatural in the syntax-directed setting. Moreover, some of these computations are well understood and their efficient implementations are known in other programming paradigms (e.g., sorting, text retrieval, certain optimization problems). The interface between the two paradigms is through *free* functions of the attribute specification language, which correspond to *free* predicates in [10]. [10] gives a more formal treatment of such amalgamations. In the current implementation of SIMON, we have chosen C++ [11] for writing programs to define such functions.

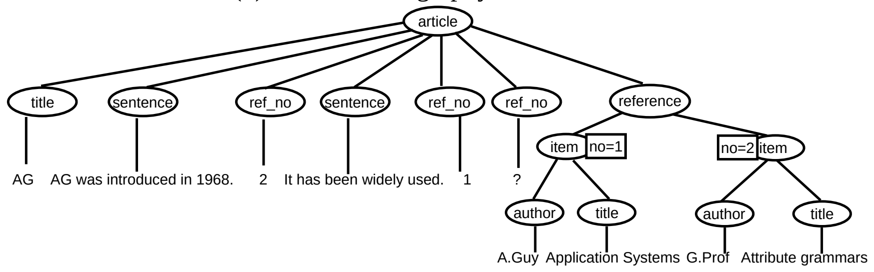
SIMON has been applied in several examples of document assembly, view retrieval, and document formatting. The previous experience indicates that SIMON enables us to write down transformation specifications that are easy for human beings to understand, and efficient for computers to execute. For example, for the formatting process of technical articles, our specification consists of 600 lines, among which 250 lines are attribute grammar



(a) ST_1 for article sd_1



(b) ST_2 for bibliography document sd_2



(c) RT for article rd

Figure 3. Attributed trees for source and result documents

and 350 lines are C++ code. Based on this specification, SIMON can derive the layout structure of a technical article from its logic structure within a few seconds.

In the following sections, we show, in greater detail, how SIMON works through a single common example, which has been tested under the current implementation of SIMON. Although the primary focus of this paper is the SIMON transformation engine, the example also illustrates the input and output conversion process.

2 AN EXAMPLE: ASSEMBLING DOCUMENTS OF MULTIPLE META-TYPES

Assume that we have (1) a technical article sd_1 in SGML style as shown in Figure 2(a), and (2) a bibliography document sd_2 in BibTeX style. The article sd_1 contains three citations {intro, app, else}, each of which indicates a reference to external documents. The bibliography document sd_2 contains a list of reference entries, each of which is either a book or an article. Each reference entry is associated with a citation key (e.g., *app* for the first entry) and has an author list and a title. The goal is to construct the LaTeX article *rd* of Figure 2(c) from sd_1 and sd_2 computationally. The result article *rd* contains a reference list at the end and several cross reference numbers. The first character of all words (e.g.,

		imported = { enum {book, article} entry.kind; char *entry.key;}	
article	$\longrightarrow$ title (sentence | citation)*	bibdata	$\longrightarrow$ entry*
title	$\longrightarrow$ PCDATA	entry	$\longrightarrow$ author title
sentence	$\longrightarrow$ PCDATA	author	$\longrightarrow$ PCDATA
citation	$\longrightarrow$ PCDATA	title	$\longrightarrow$ PCDATA

(a) SG_1 for articles with citations (b) SG_2 for bibliography documents

imported = { int item.no; }

article	$\longrightarrow$ title sentence* reference		
sentence	$\longrightarrow$ PCDATA ref_no*	title	$\longrightarrow$ PCDATA
ref_no	$\longrightarrow$ PCDATA	reference	$\longrightarrow$ item*
item	$\longrightarrow$ author title	author	$\longrightarrow$ PCDATA

(c) RG for articles with references

Figure 4. Attribute grammars for source and result documents

system of the first reference) in the reference's title is translated into a capital letter iff the reference is a book.

In order to deal with structured documents of various meta-types (e.g., SGML, LaTeX, troff), SIMON applies the transformation framework shown in Figure 1. First, source documents $sd_1, \cdots, sd_k$ of various meta-types are converted into attributed trees $ST_1, \cdots, ST_k$, respectively. These trees are then transformed into a result attributed tree RT through structure transformation. Finally, the result tree RT is converted into a result document rd. Figure 3 shows the attributed trees ST_1, ST_2 and RT for the SGML, BibTeX and LaTeX documents of Figure 2. In the figure, tree nodes are denoted by circles, and their associated attributes are shown in boxes. For example, in the attributed tree ST_2, the left-most child of the root is a node labelled by symbol *entry* which is associated with instances of two attributes *entry.kind* and *entry.key*, whose assigned values are *book* and *app*, respectively.

Figure 4 illustrates the attribute grammars $\{SG_1, SG_2, RG\}$ for the article with citation, the bibliography document and the article with reference (Figure 2). In Figure 4, nonterminal symbols are shown in lower-case letters, and terminal symbols in upper-case letters. As in SGML, the terminal symbol PCDATA means zero or more data characters. An attribute grammar usually contains three parts: (1) a context-free grammar (CFG) which expresses the generic document structure, (2) a set of attributes associated with symbols of CFG, and (3) a set of semantic rules associated with production rules of CFG. An attribute b associated with a symbol X is denoted as $X.b$. $X.b$ can be an *imported attribute*, *inherited attribute*, or *synthesized attribute*. The value of an imported attribute is provided by external means (e.g., the user). The values of synthesized and inherited attributes are defined by semantic rules of productions. In Figure 4, CFGs are expressed in extended BNF. Two imported attributes *entry.kind* and *entry.key* are declared in SG_2, and one imported attribute *item.no* is declared in RG. The types of attributes are declared as in C++. No semantic rules are defined in these grammars.

```
imported = { char *item.key;}            inherited = { int item.no;}

reference → item*
      { item.no = if is_left_most(item) then 1 else left(item).no +1; }
item → author title
author → PCDATA                          title → PCDATA
```

Figure 5. Attribute grammar IG for reference lists

Figure 5 shows an attribute grammar *IG* that specifies reference lists. In addition to an imported attribute *item.key*, *IG* declares an inherited attribute *item.no* which represents the ordered number of a reference item. The value of *item.no* is defined by the semantic rule between "{" and "}": it is incremented by 1 from left to right. For a given *item*, function *is_left_most(item)* returns a value *true* iff the item is the left-most child of its parent, and function *left(item)* returns the sibling of *item* immediately to its left.

3 HIGHER-ORDER ATTRIBUTE GRAMMAR SCHEMES

3.1 A syntax overview

The SIMON transformation engine is based on a *scheme* extension of attribute grammars in the sense of [10]: *i.e.*, the grammars are augmented with *free* functions which are (freely) interpreted by some externally defined programs. Currently its implementation employs the higher-order attribute grammar (HAG) of [1] and C++ to define free functions. Figure 6 shows an example of such HAG schemes. The scheme constructs a reference list from an article with citation and a bibliography document. The C++ code (Figure 6(b)) defines the functions (e.g., *contains*) and data types (e.g., STRING_SET) appearing in the grammar.

The grammar part of the scheme consists of *attribute declaration* (cf. lines (01) – (04) of Figure 6(a)) and a set of *productions* (cf. lines (05) – (31) of Figure 6(a)). Each production consists of a production rule in the form of "$X_0 \rightarrow X_1 \cdots X_m$", and several semantic rules within "{" and "}". When a symbol appears more than once in a production, each occurrence of the symbol is associated with a unique number in order to distinguish them from each other (e.g., content[1] and content[2] in the line (11) of Figure 6). Each semantic rule takes the form of "$\alpha = f(\cdots, \beta, \cdots)$", where f is the name of a function. The defined identifier α is either a synthesized attribute of the left-hand side symbol X_0, or an inherited attribute of a right-hand side symbol X_i ($1 \leq i \leq m$), or a right-hand side nonterminal symbol itself. Defining a nonterminal X in a semantic rule means that the subtree with root X is constructed according to the semantic rule, rather than syntactical parsing of a given string. Such a symbol X (e.g., **reference** in the first production of Figure 6(a)) is called a *nonterminal attribute*. Each argument β of a semantic rule is a symbol of the production or its attribute.

A production can be associated with a unique identifier, called *production operator* [12], in order to obtain a linear notation for trees [1,5,6]. For example, the production in the line (31) of Figure 6(a) is associated with an operator **item**. This operator denotes a function that takes a pair of attributed trees, namely, an `author`-tree and a `title`-tree, and constructs an `item`-tree with those two trees as its children. As a function, a production

```
(01) imported ={ char *entry.key, *item.key;     enum { book, article} entry.kind; }
(02) synthesized ={ item entry.ref;   item_list bibdata.ref, entry_list.ref;
(03)            STRING_SET article.cite, content.cite, citation.cite; }
(04) inherited ={ STRING_SET bibdata.cite, entry_list.cite; }

(05)         ref_cons → article bibdata reference
(06)           { bibdata.cite = article.cite;    reference = refer(bibdata.ref); }
(07)         article → title content              { article.cite = content.cite; }
(08)         title → PCDATA
(09)         content[1] → content[2] sentence     { content[1].cite = content[2].cite; }
(10)         sentence → PCDATA
(11)         content[1] → content[2] citation
(12)           { content[1].cite = content[2].cite + citation.cite; }
(13)         content →                            { content.cite = STRING_SET(); }
(14)         citation → PCDATA                    { citation.cite = STRING_SET(PCDATA); }
(15)         bibdata → entry_list
(16)           { entry_list.cite = bibdata.cite;   bibdata.ref = entry_list.ref; }
(17)         entry_list[1] → entry entry_list[2]
(18)           { entry_list[2].cite = if (entry_list[1].cite.contains(entry.key))
(19)                                 then entry_list[1].cite - STRING_SET(entry.key)
(20)                                 else entry_list[1].cite;
(21)             entry_list[1].ref = if (entry_list[1].cite.contains(entry.key))
(22)                                 then itlist(entry.ref, entry_list[2].ref) else entry_list[2].ref; }
(23)         entry_list →                         { entry_list.ref = nolist(); }
(24)         entry → author title
(25)           { entry.ref = if (entry.kind==book) then item(author, capitalize(title), key=entry.key)
(26)                                 else item(author, title, key=entry.key); }
(27)         author → PCDATA
(28) refer:  reference → item_list
(29) itlist: item_list[1] → item item_list[2]
(30) nolist: item_list →
(31) item:   item → author title
```

(a) Higher order attribute grammar

```
struct STRING_SET {
        int no;                         /*no of strings*/
        char *str[100];                 /*string data*/

        STRING_SET()                    { no = 0; }
        STRING_SET(char *x)             { no = 1; str[1] = new char[strlen(x)+1]; strcpy(str[0], x); }
        int contains(char *x)           { for (int i = 0; (i < no); i++);
                                                if (!strcmp(str[i],x)) return 1;
                                            return 0; }
        friend STRING_SET operator+(STRING_SET x, STRING_SET y)    { · · · }
        · · ·    }
```

(b) C++ code

Figure 6. HAG scheme specification G_1 for reference construction

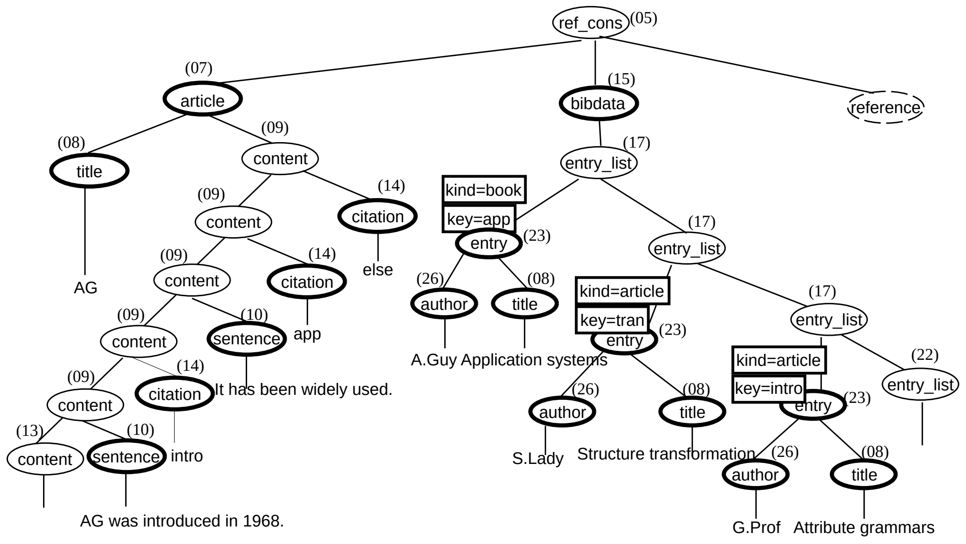


Figure 7. Basic tree T of HAG G_1

operator can be used in any semantic rule. For example, the production operator **item** is used in the semantic rule of the lines (25)-(26) in Figure 6(a). That semantic rule uses a function *capitalize* which translates the first letter of each word into capital. Three other production operators **refer, itlist, nolist** are also declared and used in Figure 6.

3.2 Attribute evaluation

Let G be a HAG scheme. An *attributed tree* of G is a derivation tree of G (*i.e.*, a derivation tree according to the productions of G) in which each node x labelled by a nonterminal X is associated with declared attribute instances $x.b$ and their values $val(x.b)$.

An attributed tree T of G is called a *basic tree* iff T satisfies the following conditions: Each leaf of T is labelled by a nonterminal attribute or a terminal; No internal node of T is labelled by a nonterminal attribute; All imported attribute instances of T are assigned values. Figure 7 illustrates a basic tree of the attribute grammar G_1 given in Figure 6(a). In the figure, a node labelled with a nonterminal attribute is denoted by a dashed circle. Each internal node has the line number of the production rule that expands that node.

An attributed tree T of G is *complete* if every leaf of T is labelled by a terminal symbol, and every attribute instance (synthesized, inherited, or imported) of T is assigned a value. An attributed tree T is *consistent* iff T satisfies the following conditions: (1) For every inherited or synthesized attribute instance, its assigned value is equal to the value defined by its semantic rule; (2) For every node of T labelled by a nonterminal attribute, the value of this nonterminal attribute instance, *i.e.*, the maximal basic subtree rooted at this node, is equal to the tree defined by the semantic function for this nonterminal attribute.

Given a basic tree T of a higher-order attribute grammar, the *attribute evaluation* is a process of constructing a complete and consistent extension of the tree, denoted *cons(T)*, by associating attribute instances with each node; assigning a value to an instance of

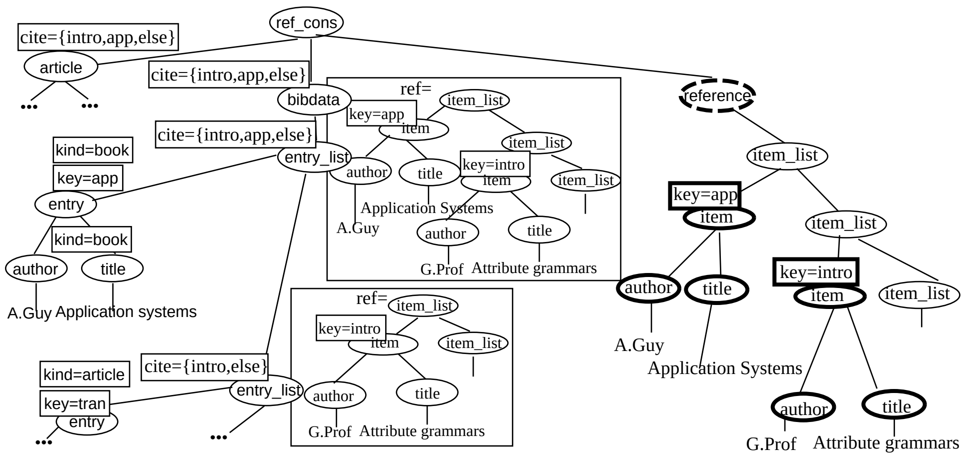


Figure 8. Complete and consistent tree cons(T) of HAG G_1

an inherited or synthesized attribute; and grafting a basic subtree to a leaf labelled by a nonterminal attribute.

For the basic tree T of Figure 7, its attribute evaluation with grammar G_1 of Figure 6 generates the tree $cons(T)$ given in Figure 8. For the space limitation, some parts of the tree have been omitted in Figure 8. Note that a subtree has been grafted to the node labelled by nonterminal attribute *reference*. Also, various attribute instances such as *article.cite* and *bibdata.cite* have been assigned values. As an example of nonterminal attributes, consider the node labelled by the nonterminal attribute *reference*. The maximal subtree of *cons(T)* with that node as a root forms a basic subtree, which is equal to the tree define by the semantic rule *reference* $= \cdots$ in the line (06) of Figure 6(a). Note also that, for some inherited or synthesized attribute instances (e.g., *bibdata.ref*), their assigned values are attributed trees.

4 DOCUMENT TRANSFORMATION IN SIMON

Let $SG_1, \cdots, SG_k$ be source attribute grammars, and RG a result attribute grammar. Then a document transformation in SIMON is a mapping of the following type:

$$\mid SG_1 \mid \times \cdots \times \mid SG_k \mid \longrightarrow \mid RG \mid ,$$

where $\mid X \mid$ denotes the collection of all attributed trees valid under the grammar X. As shown in Figure 9, the transformation consists of three parts: (1) Parsing: embed source trees into a basic tree of the HAG scheme which defines the transformation engine; (2) Attribute evaluation: construct the complete and consistent extension of the basic tree; and (3) Unparsing: take a projection of the extended tree with respect to the result grammar.

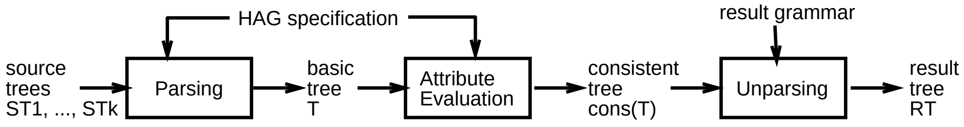


Figure 9. HAG based structure transformation

4.1 Parsing (embedding) and unparsing (projection)

Given source attribute grammars and a result attribute grammar as above, let G be a HAG scheme that specifies a transformation. We assume that G satisfies the following conditions: (1) The symbols of G include all symbols of grammars SG_i's and RG; (2) every imported attribute of RG is an attribute of G; and (3) every attribute of a source grammar SG_i is an imported attribute of G.

As an example, consider SG_1 and SG_2 in Figure 4 as source grammars, and IG of Figure 5 as the result grammar. With respect to these grammars, the HAG scheme specification in Figure 6 satisfies those constraints stated above.

Since attribute evaluation has been explained in Subsection 3.2, we now describe the other two steps in Figure 9: *parsing* and *unparsing*. Let ST_i be a tree of the grammar SG_i, and ST a tree whose root has ST_i's, $1 \leq i \leq k$, as its children. Then we require a one-to-one mapping φ from the nodes of ST to the nodes of the basic tree that the parsing process constructs out of ST_i's such that:

C1: φ is a syntactic embedding, *i.e.*, it preserves both ancestral and left-to-right orders;
C2: φ preserves labels, and no node of T outside the φ-image of ST has a symbol of a source grammar as its label; and
C3: φ preserves values of imported attributes.

As an example, consider the trees ST_1 and ST_2 of Figure 3, and the HAG scheme G_1 of Figure 6. In this case, the parsing process produces the basic tree given in Figure 7. A node x is shown in a bold circle in Figure 7 iff x is an image of some node in ST_1 or ST_2 under the mapping φ.

The unparsing process, on the other hand, is a process of constructing a complete and consistent tree RT of the source grammar RG out of the complete and consistent extension $cons(T)$ of the basic tree T. As in the parsing process, we require a one-to-one mapping ψ from the nodes of RT to the nodes of $cons(T)$ which satisfy three conditions analogous to C1, C2, and C3: correponding to C3, however, the inverse of ψ, rather than ψ itself, must preserve values of imported attributes.

As an example, consider the tree of Figure 8, and the grammar IG of Figure 5. The unparsing process derives the attributed tree IT given in Figure 10. A node x is shown in a bold circle in Figure 8 iff x is an image of some node in IT under the one-to-one mapping. Note that the values of attribute instances *item.no* have been added into IT according to grammar IG. IT expresses the list of references that exist within the bibliography document of Figure 2(b) and are cited in the article of Figure 2(a).

By giving a HAG scheme specification G (and a result grammar RG), source trees $\{ST_1, ST_2, \cdots, ST_k\}$ can be transformed into a result tree RT through the three steps shown in Figure 9. We write RT = trans($G, RG, ST_1, \cdots, ST_k$) to denote this transformation process. For example, source trees ST_1 and ST_2 of Figure 3 are transformed into IT of Figure 10

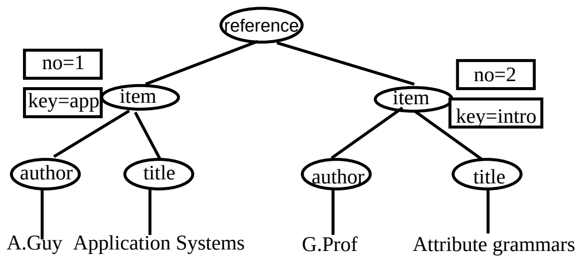


Figure 10. Attributed tree IT for derived references

with the grammars G_1 of Figure 6 and IG of Figure 5. IT = trans(G_1, IG, ST_1, ST_2) denotes this transformation.

For the implementation of such transformations, we can adopt the incremental and efficient algorithms originally developed for the syntactical and semantical analysis in the programming language field. Thus, when the source trees are modified, the result tree can be incrementally updated by recomputing only the portion of the result tree that is affected by the modification.

4.2 Composition of transformations

It is known that higher-order attribute grammars can express arbitrary computation [7]. Thus, in theory, any computable document transformation can be specificed in a HAG and hence in a HAG scheme. However, it is also known that by decomposing a transformation, one can obtain a sequence of computationally much simpler transformations [5].

Thus, SIMON also supports composition of transformations. For example, consider the transformation from articles with citation and bibliography documents into articles with reference (Figure 3). The transformation process can be composed of two subprocesses: (1) construct a reference list (e.g., Figure 10) from the article with citation (e.g., Figure 3(a)) and bibliography document (e.g., Figure 3(b)); (2) create an article with reference (e.g., Figure 3(c)) from the derived reference list and the given article with citation. The subprocess (1) is specified by the HAG scheme specification of Figure 6. In the same way, we can write a HAG specification, say G_2, for the subprocess (2).

After specifying all transformation subprocesses, we can perform the total transformation process by composing them. For example, for the HAG scheme specifications G_1 of Figure 6 and G_2 mentioned above, we can execute the following composition,

$$RT = \text{trans}(G_2, RG, ST_1, \text{trans}(G_1, IG, ST_1, ST_2)),$$

to assemble the source trees ST_1 and ST_2 into result the tree RT (Figure 3).

5 CONCLUSION: TOWARDS A VIEW MANAGEMENT SYSTEM FOR COMPLEX DOCUMENTS

One of the daunting document problems today is how to manage large complex documents as typified by the technical documents of, for instance, aerospace, automobile, and pharmaceutical industries. It is our belief that the notion of structure is an essential ingredient of any reasonable solution to the problem: any person who has to deal with such complex documents must do so component by component, which are, by the very definition of complex documents, intricately related to each other. One key issue here is how to define human comprehensible *views* of the complex whole out of its components dynamically as various needs arise. In this regard, the work presented in this paper may be considered a step towards providing a *document programming language* in which one can program various views as needed. Once a document management system acquires such flexibilities, there has to be a way of managing those layers of (mutually dependent) views: e.g., propagating updates through the layers and maintaining some notion of consistency among them. Our intention is to eventually develop SIMON into a more comprehensive system of view management.

One related issue not discussed in this paper is the conversion problem of unstructured documents. We simply note that there has been a significant progress on this issue, particularly, on the conversion from unstructured documents to (weakly) structured documents [13]. We hope to augment our system with such converters to build more powerful document transformation/conversion systems.

REFERENCES

1. H.H. Vogt, S.D. Swierstra, and M.F. Kuiper, ‘Higher-Order Attribute Grammars’, in *Proceedings of the ACM SIGPLAN’89 Conference on Programming Language Design and Implementation*, 131–145, ACM Press, New York, (1989).
2. C.F. Goldfarb, *The SGML Handbook*, Oxford University Press, Oxford, 1990.
3. R. Furuta and P.D. Stotts, ‘Specifying Structured Document Transformations’, in *Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography (EP88)*, ed., J.C. van Vliet, 109–120, Cambridge University Press, Cambridge, UK, (1988).
4. E. Akpotsui and V. Quint, ‘Type Transformation in Structured Editing Systems’, in *Proceedings of Electronic Publishing, 1992 (EP92)*, eds., C. Vanoirbeek and G. Coray, 27–41, Cambridge University Press, Cambridge, UK, (1992).
5. H. Ganzinger and R. Giegerich, ‘Attribute coupled grammars’, *ACM SIGPLAN Notices*, **19**(6), 157–170, (1984).
6. T. Teitelbaum and R. Chapman, ‘Higher-Ordered Attribute Grammars and Editing Environments’, in *Proceedings of the ACM SIGPLAN’90 Conference on Programming Language Design and Implementation*, 197–208, ACM Press, New York, (1990).
7. H.H. Vogt, *Higher-Order Attribute Grammars*, Ph.D. dissertation, Department of Computer Science, University of Utrecht, 1993.
8. A.L. Brown, T. Wakayama, and H. Blair, ‘A Reconstruction of Context-Dependent Document Processing in SGML’, in *Proceedings of Electronic Publishing, 1992 (EP92)*, eds., C. Vanoirbeek and G. Coray, 1–26, Cambridge University Press, Cambridge, UK, (1992).
9. P. Deransart, M. Jourdan, and B. Lorho, *Attribute Grammars – Definitions, Systems, and Bibliography*, Lecture Notes in Computer Science, No. 323, Springer, Berlin, 1988.
10. A.L. Brown and S. Mantha and T. Wakayama, ‘The declarative semantics of document processing’, in *Proceedings of the First International Workshop on Principles of Document Processing*, (1992).
11. B. Stroustrup, *The C++ Programming Language*, Addison–Wesley, Reading, MA, 1991.
12. T. Reps and T. Teitelbaum, *The Synthesizer Generator – A system for constructing language-based editors*, Springer, Heidelberg, 1988.
13. G. Porter and E.V. Rainero, ‘Document Reconstruction: A System for Recovering Document Structure from Layout’, in *Proceedings of Electronic Publishing, 1992 (EP92)*, eds., C. Vanoirbeek and G. Coray, 127–141, Cambridge University Press, Cambridge, UK, (1992).