Electronic Style Sheets*

Anne Brüggemann-Klein† Derick Wood‡

Abstract

Document processing systems must provide formatted versions of documents, where the specification of formats is the task of the *document designer*. To match the stylistic quality expected in the traditional publishing process, *electronic style sheets* need to support the design mechanisms that have evolved over the centuries. The designer's craft should not depend on the formatter, in particular it should not involve programming the formatter.

We propose four basic mechanisms called *transcription types* that are sufficient to express a wide range of layouts. Building on these four transcription types, we have defined a layout specification language, *Designer*, that is declarative and formatter-independent.

Putting on the agony, putting on the style.
— Lonnie Donnagon (c. 1960)

1 Introduction

In the traditional publishing process, a *style sheet* is a running account of rules about diction and language usage adopted for a particular manuscript. It is kept by the manuscript editor [Chi82, Ano85, McL80]. *Electronic style sheets*, however, deal with the graphical layout of documents: They facilitate "the setting, changing, and standardization of

*This work was partially supported under the NSERC and ITRC grants of the second author.

†Institut für Informatik, Universität Freiburg, Rheinstr. 10–12, 7800 Freiburg, Germany. E-mail: brueggemann@informatik.uni-freiburg.de.

‡Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada. E-mail: dwood@watdragon.waterloo.edu.

properties that control the layout and appearance of document content" [JB88]. A set of page, paragraph, and character templates in *Framemaker* or the LaTeX collection of article macros are both style sheets in this sense.

Many current document-processing systems are based on *structured documents* [Gol81, FSS82, CRD87, AFQ89]. Essentially, the author marks the logical elements of a document using *descriptive tags*, and in a separate process, a graphic artist specifies the layout of the document in an electronic style sheet. Subsequently, the style sheet guides the transformation of the logically marked-up source document into formatter input. We call this process *transcription* and the program that performs it a *transcription program*. By analyzing the work of document designers as described in books on graphic art and typography [Lee79, McL80, Wil83], we have identified four high level *transcription types* that can serve as building blocks in electronic style sheets:

1. Calls of parameterized formatting tasks.
2. Automatic numbering.
3. Generation of boilerplate and derived text.
4. Rearrangement of subelements.

The application of each of these transcription types can depend on the *context*. The four context-dependent transcription types provide domain-specific support for document designers, such that run-of-the-mill designs can be expressed straightforwardly.

To some extent, our approach to style-sheet support resembles the investigation of editing systems by Meyrowitz and van Dam [MvD82] in that we describe the fundamental operations—transcription types in our case—on the semantic level. We also have a syntactic realization, namely a specification language for electronic style sheets, *Designer*, which is described elsewhere [BW92].

This abstract is organized as follows. In the next section, we explain our document, formatter, and style-sheet model. Then we present the four transcription types. Section 3 deals with parameterized formatting tasks. Since this type of transcription, as well as the remaining three, is subject to context dependency, Section 4 defines our notion of context. The remaining transcription types are described in Sections 5 to 7.

2 Models

Document model We use the well-established model of structured documents [Gol81, FSS82, CRD87, AFQ89] in which a document is a hierarchical, tree-like structure with symbolic cross references and each node is tagged with the name of a generic element. Documents that share the same structure are combined into a document type, which, depending on the system that implements the model, may or may not be formally defined by a document grammar. A style sheet can then be applied to all documents belonging to the same document type.

Formatter model We assume that the formatter is capable of certain tasks, such as hyphenation (turning character sequences into character sequences), typeface attribution (turning characters into glyphs), line breaking (assembling sequences of glyphs into sequences of lines), or page breaking (assembling lines into pages) [FSS82]. The formatting tasks have parameters such as type size, line length, first-line indentation, or page height. Formatter input is a stream of formatter commands and text. The area of the input to which a formatting task is applied is surrounded by parameter settings and formatter commands. Therefore, formatter input is a hierarchy of parameterized formatting tasks. It depends on the individual formatter which formatting tasks are supported, how they are parameterized, what syntactic conventions are used, and which hierarchies of formatting tasks are admissible. As with structured source documents, the admissible task hierarchies can be constrained by a formal grammar but they do not have to be.

Style-sheet model In our model, a style sheet is a *table with an entry for each generic element* that may occur in the document. Following Johnson and Beach [JB88], an entry is called a *style rule*, and each style rule lists its associated transcriptions. Thus, our style-sheet mechanism is *structure-based* rather than *grammar-based* [MvD82]. (Other terms in use are markup-sensitive rather than structure-controlled [Gol90] or event-driven rather than grammar-driven [WvV91].) In computer science terminology, style sheets in our model operate on structured documents in the form of parse trees. They do not depend on document grammars.

In the grammar-based model, style rules are assigned not to generic elements but to occurrences of generic elements in the document grammar. Therefore, grammar-based systems are capable of specifying some context-dependent processing. Yet they are not able to accommodate all the examples of context dependencies that are listed in Section 4. Thus, the structure-based model and the grammar-based model both have to be enhanced to handle context dependencies. For grammar-based systems, attributed grammars [Fra89] and grammar-based coordination between attributed grammars [BWB92] have been used. In our model, *each transcription in a style rule may depend on a context predicate* whose evaluation depends on the structure tree of the document instance.

As book designers usually explain their work by example, it is hard to argue which formal model comes closest to their work. We believe that they follow a structure-based approach, rather than a grammar-based one, since the notion of a document grammar does not come up at all, even if the layout for a whole series of books such as the Penguin books [McL75] is defined.

So far we have discussed the binding of style rules to generic elements. Now we turn to the transcriptions used in a style rule. Here, current style-driven systems vary between the purely substitutional and the fully programmable approach. Substitutional systems replace structural markup tags with settings for visual properties; these settings are then used as parameters for formatting tasks. Many early versions of commercial word processors and desktop-publishing systems (*Xerox Star*, *Word*, *Wordperfect*, *Ventura Publisher*, *Interleaf*, *Framemaker*) were purely substitutional. A more recent example is *Lector* [Ray92]. One has to give credit to substitutional systems for being declarative and, thus, easy to learn and to use. Some authors have enhanced the purely substitutional approach: Kipeläinen et al. [KLMN90] use syntax-directed translation schemes and Reid [Rei80] has introduced inheritance of formatting parameters.

Historically, however, the scope of graphic design is substantially wider. Book designers rearrange the order of elements given by a manuscript; they determine which material goes into various tables and indexes and how they should be sorted; they include boilerplate text; and they mark up some instances of elements, depending on their context, for special typographic treatment. The transformational power of the systems described so far is too weak to implement such functions.

One approach to overcome this difficulty is the introduction of programmable systems, which associate more general semantic actions with the structural tags. This approach has been implemented with macro packages built directly on top of programmable formatters (*ms* on top of *troff*, LaTeX on top of TeX, *GML* on top of *Script*). Here each element instance calls a macro program implementing the design of the structural element. SGML formatting applications such as *Daphne* [SS86], *Grif* [QVB86], *Markup* [Pri88], or *Rita* [dVSC89] use essentially the same approach: Macro programs are substituted for structural markup tags. Some systems (*Quill* [CHP88], *Lilac* [Bro91], and *FFL* [Han90]) use a programming language instead of formatter macros, thus avoiding some of the problems of the macro approach.

Programmable systems are usually computationally complete. They allow document designers to implement whatever transcription they have in mind. Yet they lay upon them the burden of programming their methods, even the most common ones. Thus, the design and its implementation become irretrievably intertwined. We believe that the separation of concerns between design specification and design implementation is best served by providing *high-level transcriptions* in a transcription program.

3 Parameterized formatting tasks

The most common kind of transcription supported by any style-driven system is to call a parameterized formatting task, which is applied to the subelements of the current element instance. Modern commercial word processors and desktop publishing systems such as *Word*, *Wordperfect*, *Ventura Publisher*, or *Framemaker* provide at least the following formatting tasks: Typeface attribution (applied to strings of characters), line breaking (applied to paragraphs, represented as strings of characters between line feeds), page breaking (applied to files), and document compilation; that is, collecting sequences of pages into complete documents, including title pages, tables of contents and indexes (applied to sets of files).

It is characteristic for these systems that exactly one formatting task has to be associated with each generic element. Hence, the logical structures of documents are constrained by the admissible hierarchical relationships of formatting tasks. (The effect of this restriction can, for example, be observed in the way word processors handle enumerated lists. A list cannot be marked up as a distinct element. Rather, each list item is marked up as a paragraph. The paragraph formatting task prints and advances the list-item counter. Since none of the formatting tasks available is applicable to the list as a whole, the list-item counter has to be manually reset to its initial value each time a new list starts in the document.)

To summarize, we think it essential that a style-sheet mechanism allows the book designer to *associate any number of parameterized formatting tasks to a generic element*. These formatting tasks can be applied to an element instance in the order in which they have been specified.

4 Context

General design rules suggest that elements of the text that are logically identical should also be laid out identically. Yet there are exceptions to this rule, due to tradition or aesthetics, or because the context of an element requires non-standard treatment. Thus, document designers must be provided with the functionality to make the transcriptions of a generic element conditional to the context of its instantiation in the document. This applies not only to the call of parameterized formatting tasks, but also to the other three transcription types that are discussed in later sections. We give some examples of designs that call for context-dependent transcriptions:

1. In some designs, all paragraphs are indented, with the only exception being paragraphs immediately preceded by a heading.
2. The headings in an appendix might be labeled A), B), and C), whereas the headings in the main part are labeled 1., 2., and 3.
3. Three coauthors like Aho, Sethi, and Ullman are referred to as "Aho et al.", whereas two coauthors such as Hopcroft and Ullman are referred to as "Hopcroft and Ullman".
4. In footnotes, lists are linearized instead of placing each item on a new line.

5. Cross references (see "Theorem 1.2" but "see Lemma 1.1") are automatically prefixed with the name of the object to which they refer.

In these examples, the style rule that is to be applied to a specific element depends on the position of the current element instance within the structure of the document. Its position, however, is in general not solely characterized by the generic elements higher up in the document hierarchy. Rather, it is often necessary to take the siblings (Examples 1 and 3) or the internal structure of the current element instance into account or even to consider the structure of the element it refers to (Example 5). Finally, in some cases the direct approach is to express context in terms of the current parameter values instead of in terms of structural properties.

Macro packages such as LaTeX have some context dependencies hardwired into them in an ad-hoc manner. Other systems, for example *Bravox* [JB88], support subordinate style rules; that is, they allow the user to locally redefine a style rule within the scope of another generic element. Subordinate style rules are also provided by USELINK declarations in SGML [Gol90]. However, these systems can only deal with Examples 2 and 4. The same restriction holds for *Scribe*.

To cover all the examples above, we propose to make *each transcription of a style rule conditional to a context predicate*; that is, a Boolean expression that is evaluated each time the style rule is applied. Only those transcriptions whose context predicate evaluates to *true* are enabled for the current instance of the element. Context predicates are Boolean expressions built from the variable cE initialized with the current element instance and functions such as $parent(e)$, $leftSibling(e)$, $leftMostSibling(e)$, $rightSibling(e)$, $rightMostSibling(e)$, and $ref(e)$ that map elements to elements so that we can navigate in the multiway tree structure of the document.

All our examples of context-dependent document design can be captured in an expression language for context predicates, given a suitable set of primitive functions and predicates. We wish to emphasize that in our approach, the context in which a style rule is to be applied is defined explicitly in the corresponding style sheet and that the context mechanism is orthogonal to the four transcription types; that is, each type of transcription can be made context-dependent.

5 Numbering

In addition to the hierarchical structure of nested elements, logically marked-up documents also have a cross-reference structure. Usually, the cross references are indicated with symbolic names. Unique identifiers can be attached to element instances, and other element instances can build a cross reference to a target element instance by referring to the unique identifier of the target; see the `ID` and `IDREF` attributes in SGML and the `\label` and `\ref` commands in LaTeX. In a printed document, target elements of cross references are often numbered and referred to by their number. The scheme used for numbering is part of the document design.

The simplest numbering scheme is that the element instances of one type are numbered consecutively throughout the document. Often, however, nested enumeration schemes require that elements of one type are numbered relatively to elements of another type higher up in the hierarchy of the document. Finally, we must be able to specify enumeration schemes for nested enumerated lists where the numbering of first-level items must be independent from the numbering of second-level items although items on both levels may be tagged with the same name. As far as chapters and sections are concerned, this functionality is provided by many current systems. However, it is necessary to provide a numbering method that applies to each possible element type, to enumerate, for example, sidebars and other floating figures within the same document independently.

We propose the following specification scheme to cover all these cases. The document designer declares the counters intended to be used. The declaration of a counter means specifying a name, an environment, and a start value. The environment of the counter is a generic element. A style rule for any generic element e may then contain a *transcription for enumeration that specifies which counter should be used for numbering element instances* of that type. The transcription program driven by the style sheet allocates numbers to element instances according to the following rules. Let us assume we have declared a counter with name c, environment e, and start value s. Each time an element instance of type e is encountered, numbering of element instances using c is started afresh from s. At the end of element instance e, numbering according to c resumes at the higher level of the document hierarchy. For example, by declaring a counter *cntr* with environment *section* and start value 1 and by specifying use of *cntr* in the style rules for generic elements *definition*, *lemma*, and *theorem*, these logical elements are numbered consecutively within sections. Numbering will be independent of the section structure if *cntr* is declared with environment *document* instead.

6 Boilerplate and derived text

Some printed documents contain textual fragments that are not present in the logically marked-up source document. They are called *form* or *boilerplate text*. Examples are letter headings, the keywords "To" and "From" in memoranda, the bullets in lists, and the end-of-proof markers ("tombstones") in mathematical texts. Inserting such text is a typical design task largely ignored in current systems.

In our approach, a style rule may contain the call of an *insertion transcription together with textual content*. This content is inserted at the beginning or the end of each element instance the style rule is applied to. The content associated with such an insertion transcription is logically marked-up text whose layout is again specified in style rules. Thus, the inserted text can be laid-out differently from the element it is inserted into. Insertion transcriptions can also be used to actually print the numbers that have been assigned to element instances according to an enumeration transcription.

Automatic insertions are needed to generate tables of contents and indexes; their contents are derived from different places in a document. We equip insertion transcriptions with a *trigger predicate*. Similar to a context predicate, the value of a trigger predicate is a Boolean expression that selects the element instances from which material must be included into the newly created element. The same expression language is used for both context and trigger predicates.

7 Subelement rearrangement

In some printed documents, element instances appear in an order different from how they appear in the logically marked-up source document. For example, the position of authors and title may be reversed, or bibliographic references may be sorted according to date or authors. We propose that a style rule contains an *arrangement transcription specifying the order of the subelements* of the element instance the style rule is applied to. As a special case, the subelements can be suppressed altogether. Structural aspects of order can be specified with the expression language for context and trigger predicates. For aspects of alphabetical order, however, the language has to be enhanced with functions operating on the textual contents of elements (#PCDATA in SGML). These functions can be modeled after *MakeIndex*, a formatter-independent indexing tool [CH88].

References

[AFQ89] J. André, R. Furuta, and V. Quint. *Structured Documents. The Cambridge Series on Electronic Publishing*, Cambridge University Press, Cambridge, 1989.

[Ano85] Anonymous. The Canadian style—A guide to writing and editing. 1985. The Department of the Secretary of State of Canada.

[Bro91] K. P. Brooks. Lilac: A two-view document editor. *Computer*, 24(6):7–19, June 1991.

[BW92] A. Brüggemann-Klein and D. Wood. Extended style sheets for grammar-based full-text databases. Submitted for publication, 1992.

[BWB92] A. L. Brown, Jr., T. Wakayama, and H. A. Blair. A reconstruction of context-dependent document processing in SGML. In C. Vanoirbeek and G. Coray, editors, *EP92*, pages 275–290, Cambridge University Press, Cambridge, 1992.

[CH88] P. Chen and M. P. Harrison. Index preparation and processing. *Software—Practice and Experience*, 18(9):897–915, September 1988.

[Chi82] The Chicago manual of style. The University of Chicago Press, Chicago, 1982.

[CHP88] D. D. Chamberlin, H. F. Hasselmeier, and D. P. Paris. Defining document styles for WYSIWYG processing. In J. C. van Vliet, editor, *Document Manipulation and Typography*, pages 121–137, Cambridge University Press, Cambridge, 1988. Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography, Nice, France, April 20–22, 1988.

[CRD87] J. H. Coombs, A. H. Renear, and S. J. DeRose. Markup systems and the future of scholarly text processing. *Communications of the ACM*, 30(11):933–947, November 1987.

[dVSC89] G. de V. Smit and D. D. Cowan. Manipulating partial documents in a syntax-directed environment. Submitted for publication, 1989.

[Fra89] P. Franchi-Zannettacci. Attribute specifications for graphical interface generation. In G. X. Ritter, editor, *Information Processing 89*, pages 149–155, North Holland, Amsterdam, 1989.

[FSS82] R. Furuta, J. Scofield, and A. Shaw. Document formatting: Survey, concepts, and issues. *Computing Surveys*, 14(3):417–172, September 1982.

[Gol81] C. F. Goldfarb. A generalized approach to document markup. *SIGPLAN Notices of the ACM*, June 1981.

[Gol90] C. F. Goldfarb. *The SGML Handbook*. Clarendon Press, Oxford, 1990.

[Han90] B. Stig Hansen. A function-based formatting model. *Electronic Publishing—Origination, Dissemination and Design*, 3(1):3–28, February 1990.

[JB88] J. Johnson and R. J. Beach. Styles in document editing systems. *IEEE Computer*, 32–43, January 1988.

[KLMN90] P. Kilpeläinen, G. Lindén, H. Mannila, and E. Nikunen. A structured document database system. In R. Furuta, editor, *EP90*, Cambridge University Press, Cambridge, 1990. Proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography, Gaithersburg, Maryland.

[Lee79] M. Lee. *Book Making: The Illustrated Guide to Design/Production/Editing.* R. R. Bowker Company, New York, 1979.

[McL75] R. McLean. *Jan Tschichold: Typographer.* Lund Humphries, London, 1975.

[McL80] R. McLean. *The Thames and Hudson Manual of Typography.* Thames and Hudson, London, 1980.

[MvD82] N. Meyrowitz and A. van Dam. Interactive editing systems (Part I and II. *Computing Surveys*, 14(3):321–415, September 1982.

[Pri88] L. A. Price. Using SGML and TEX for user documentation. In C. Thiele, editor, *TEXniques, Number 7*, 1988. Proceedings of the Ninth Annual Meeting, TEX Users Group.

[QVB86] V. Quint, I. Vatton, and H. Bedor. Grif: An interactive environment for TEX. In J. Désarménien, editor, *TEX for Scientific Documentation*, pages 145–158, Springer, Berlin, 1986. Proceedings of the Second European Conference, Strasbourg, France, June 1986.

[Ray92] D. R. Raymond. Lector—An interactive formatter for tagged text. *IEEE Computer*, 25(8), 1992. To appear.

[Rei80] B. K. Reid. A high-level approach to computer document formatting. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 24–30, January 1980.

[SS86] A. Scheller and C. Smith. *Document Application Processing in a Heterogeneous Network Environment.* DFN-Bericht 41, Hahn-Meitner-Institut, July 1986.

[Wil83] H. Williamson. *Methods of Book Design.* Yale University Press, New Haven, 1983.

[WvV91] J. Warmer and H. van Vliet. Processing SGML documents. *Electronic Publishing—Origination, Disseminatin and Design*, 4(1):3–26, March 1991.