Inferring Structure Information from Typography

C. Fuß, F. Gatzemeier, M. Kirchhof, O. Meyer*

{cfuss, fxg, kirchhof, omeyer}@i3.Informatik.RWTH-Aachen.de

RWTH Aachen
Lehrstuhl für Informatik III
Ahornstraße 55

52074 Aachen, Germany

phone: +49 (2 41) 4 40 - 2 13 13

Abstract. One of the key benefits of digital documents is the possibility to add further structure. With current authoring tools, however, this imposes additional effort on the author. We propose a technique which uses everyday editing actions to infer structure. For this, we rely on the widespread use of typographical markup by authors.
We state the assumptions underlying its application, design, and possibilities. In concluding remarks, we draw connections to further work which will provide actual benefits to authors based on the `inferred` structure.

1 Introduction

Digital documents are frequently preferred over analog ones for their ease of transmission and processing. Automated processing depends on explicit structural information. Publishers use markup languages internally for workflow management and cross-platform publishing. Submissions from the authors, however, undergo costly conversion into a marked-up format after being accepted for publication.

With the advent of the more cheaply processable XML, it is now realistic to implement tools that may be distributed to a wide audience of authors to help them submit marked-up documents to the publisher. Several attempts have been made in this direction (see section 2), but have generally failed to reach wider acceptance, as they forced authors to change, however minimally, their accustomed working styles or tools.

* The work presented here has been supported by the German Ministry for education, research and technology as part of the Global-Info project (`http://www.global-info.org`).

This text presents a technique which may be used to reasonably infer XML structure from the text as it is written by the author the way he is used to. As structure is derived from typographical information, only simple structures can be inferred. The results are used for the development of the authoring tool aTool[1]. It is the core of a cooperation with Springer-Verlag and Technische Universität München in the context of the Global-Info project funded by the German Ministry of Education, Research and Technology (BMBF)[2]. At the current stage, the project is focused on scientific authors writing article-sized documents.

This paper starts out with the discussion of related work in the area of structured document generation. The body of this paper begins by stating the basic assumptions we make for the application of this technique. On this basis, the technique itself is presented. The application context of its implementation is discussed towards the end.

2 Related Work

Today's approaches to XML authoring can be classified into two categories: XML editors and batch converters for proprietary documents. Historically, syntax-directed editors are precursors of the first type of applications.

Syntax-directed Editing

A paradigmatic syntax-directed editing system is the family of synthesizer editors and the Synthesizer Generator [13] that generates editors from descriptions of document syntax and editing commands. This structure is quite similar to validating XML editors, which are parameterized at runtime with a DTD. Pure syntax-directed editing has proven to be too restrictive for authors, so there must be at least a free editing mode, in which the document or parts of it can be manipulated at will. Such systems, however, need to rely on their own visualization and interaction mechanisms and are therefore no option for integration with existing word processors.

As the discipline of psychology has established quite rigid authoring guidelines, there have been attempts to develop authoring applications that ensure the correctness of documents according to the formal parts of these guidelines. Manuscript Manager [11] is a commercial product based on this idea. It did not reach widespread acceptance, presumably due to its nature of being another stand-alone word processor. Its current successor, the APA-Style helper

[1] `http://www11.in.tum.de/aTool/`
[2] `http://www.global-info.org/`

[1], takes a different approach: Dialog boxes guide the author through entering the required information and references, from which a RTF document framework is generated. This can be of help to novice authors, but its restriction of very tight guidance and completely free-form editing gives no support in the core stages of document creation.

XML Editors

The purpose of XML editors is to create correct XML documents. The generally possible complexity of these documents and the lack of abstraction from XML concepts demand a rather complex user interface. Hence reviews of most XML editors (like XML Pro, XMetaL, etc.) stress that today's applications are not made for novices, but rather assume a good working knowledge of XML and a general understanding of the nomenclature.

Some applications make use of widely used word processing applications as the UI to edit XML documents (for example i4i's S4/Text [8]). The content author is still forced to declare the structure of the document explicitly and in a detailed way, by specifying the type of every inserted element which interrupts the author while writing.

Batch Conversion Applications

Common batch conversion applications fail on the task of inferring deep XML structure from the input documents (mainly MS Word or RTF documents). Only some sophisticated tools, like Schema's MarkupKit [7] build up more than a two-layer hierarchy.

Most of them map character formatting like bold, italic etc. to generic element types expressing the same *presentation*, disregarding structure expressed by the typography. Electronic Book Technologies (EBT)[3] have developed a suite of tools based on this idea. The Rainbow DTD describes this information and the Rainbow Maker converts documents of some formats, for example RTF to instances of this DTD. As a second step, higher-level structure can be added with DynaTag. These applications offer the user an automatic mode to extract structure. Still, most of these algorithms rely heavily on named styles and format templates rather than inferring structure from typography.

As batch mechanisms, the systems rely on appropriate configuration and entail relatively high costs for user intervention, namely offline fine-tuning. None of these methods can build logical structure as a true hierarchy by just evaluating typography without burdening the content author with explicitly specifying structure.

[3] Now Electronic Business Technologies, `www.ebt.com`.

The CIDRE project [3] at the university of Fribourg pursues a probabilistic approach adopted from optical document recognition. The n-grams of sequential texts are generalized to include hierarchy information. Probabilities for such generalized n-grams can be inferred from existing document bases. This approach overcomes the disadvantageous extensive manual configuration and flat hierarchy of the other batch conversion mechanisms, but is not integrated into authoring systems.

3 Basic Assumptions

As only a human being can understand a text enough to introduce semantical markup in the general case, there must be some assumptions we make in comparison to that. We claim that these assumptions do not severely limit generality in our application domain of scientific publishing.

Assumption 1 (Known target document type) *There is a description of the expected structure, that is: a DTD for the* XML *document.*

As we are discussing documents that are to be fed into a well-defined processing pipeline, it is safe to assume that the interchange format is well-defined on the syntactical level. The technique relies on the DTD as a source for informative element type names and, in a validating mode, to limit the number of possible types.

Assumption 2 (Textual nature) *The author produces textual documents, not general data.*

While XML can be used and is used for general data modeling (as in RDF [9]), we use it in the application domain of natural-language documents. Moreover, these documents still resemble the linear documents found in printed books and journals.

Using this assumption, we distinguish element types into those corresponding to paragraphs and those corresponding to sub-paragraph constructs.

Assumption 3 (Typography) *The author applies some typographic markup.*

This is based on the experience that authors frequently do so. They often use direct formatting instead of named styles as the applied changes are more obvious. Headlines, emphasis and other categories are usually typographically marked, but not necessarily as in the final publication. We use these marks to determine element boundaries and types. We do not require usage of named styles, but can incorporate them where present.

Authors that do not use typography may still use typewriter-style markup such as whitespace. While we do not address such markup here, we do feel confident that an extension of the typography determination routines will suffice to support these authors, too.

Assumption 4 (Correspondence) *The typographic markup corresponds with the semantical structure.*

That is, parts of the text that look different are probably used in different roles. Often the author is not aware of these roles and therefore cannot be convinced to use named styles. This assumption does not determine the degree of difference required to be classified differently. For example, an author may tweak font sizes to fit that extra paragraph on a page in his draft printouts without being required to associate some different structure to it.

Parts of the text that look the same are, special situations exempt, structurally equivalent. A sequential content model for example may lead to equally formatted paragraphs being interpreted differently.

4 Deriving Structure

In our approach, we extend the user's authoring application to maintain data structures *incrementally*, that is, while he is editing and formatting his text. These data structures can then be used to extract a structured document.

Inferring structure information from typography involves two subproblems: locating the structural elements and assigning types to these elements. Both subproblems also involve hierarchical ordering of the elements.

4.1 Locating the Elements

To solve the first problem, the flat character stream is partitioned into syntactical fragments on typographic and structural boundaries. According to assumption 4, these boundaries imply semantical boundaries. From these fragments, a hierarchy of typographic elements is constructed. The top-level elements represent the paragraphs, while nested elements correspond to typographical markup within the paragraphs (see assumption 2). Inferring the hierarchy of the paragraphs (for

example chapters or sections) depends on the underlying DTD and is beyond the scope of this paper.

To determine the typographic boundaries, the style of each character is expressed as a tuple covering a selection of the typographic properties, for example a style name, font name, size, weight, and variation. An example for a format tuple of a typical heading character is

$$FT = (\text{'Heading'}, \text{Helvetica}, 16\text{pt}, \text{bold}, \text{roman})$$

Figure 1 shows the pseudocode for locating the syntactical fragments. It expects an input stream of formatted characters and produces an XML stream with mixed-content elements, whose types are discussed in the next subsection. This batch algorithm can be applied to single paragraphs for incremental operation, as the algorithm returns to its initial state at the end of each paragraph. For the sake of simplicity, paragraph breaks are determined by a special character (CR) and we expect a CR as the last character in the document.

```
oStack.Empty
WHILE Stream.HasChars
    curChar := Stream.GetNextChar
    IF curChar == CR THEN            -- close all elements
        WHILE oStack.NotEmpty DO
            XML.CloseElement(oStack.Head)
            oStack.Pop
        ENDWHILE
    ELSE
        forTup := FormatTuple(curChar)
        IF forTup in oStack THEN     -- close contained elements
            WHILE forTup != oStack.Head DO
                XML.CloseElement(oStack.Head)
                oStack.Pop
            ENDWHILE
        ELSE                         -- open new element
            oStack.Push(forTup)
            XML.OpenElement(forTup)
        ENDIF
    ENDIF
ENDWHILE
```

Fig. 1. Pseudocode for identifying text fragments.

The algorithm scans the input stream for changes in format tuples and opens or closes elements accordingly. A stack of format tuples (oStack) correspond-

ing to the currently open elements is maintained to determine whether to open or close elements. Every paragraph break causes all open elements to be closed and the stack to be cleared.

A change of the format tuple (`forTup`) may indicate the end of one or more open elements or a new nested element. If the format tuple has been seen before, it is assumed that the author wants to switch back to the corresponding level. This means that the head of the stack is popped and the current element is closed until the format tuple on the stack equals the new format tuple. If the format tuple has no corresponding entry in the stack, a new element is opened and its format tuple is pushed onto the stack.

The algorithm is presented here as working on streams for reasons of simplicity. When creating a DOM-like document tree, some more anchors in the input file have to be remembered on a stack of open elements.

As inferring types from the DTD is deferred to another part of the overall algorithm, we create only preliminary dummy types reflecting only the different format tuples. The type information is stored in a table called FORMAT-TO-TYPE-MAP. Figure 2 shows an example.

Format Tuple	Element Type
('Heading', Helvetica, 16pt, bold, roman)	`<Dummy_Heading16ptBold>`
('Standard', Times, 10pt, regular, roman)	`<Dummy_Standard>`
('Standard', Times, 10pt, regular, italic)	`<Dummy_StandardItalic>`

Fig. 2. An example of a FORMAT-TO-TYPE-MAP

In building this map, the algorithm introduces *types* of elements, therefore abstracting from instances. Two elements in the text with the same format tuple are of the same type and therefore get the same temporary type name (see assumption 4).

4.2 Generalization of Format Tuples

As not all elements of the tuples are relevant for determining the type, wildcards ($*$) may be used in the table to allow arbitrary matches on single fields.

When a new dummy type is created, a minimal *format pattern* for the map entry is generated by comparing the format of the dummy element (fmt) with that of the immediately preceding (pre) and following text (suc). For a single property value of these formats, the operator Δ is defined as follows:

$$\Delta(pre, fmt, suc) ::= \begin{cases} fmt, \text{ iff } pre \neq fmt \vee suc \neq fmt \\ * \quad , \text{ iff } pre = fmt = suc \end{cases}$$

If the text is at the start or end of a paragraph the preceding or following text is not considered:

$$\Delta(pre, fmt, \bot) ::= \Delta(pre, fmt, fmt)$$
$$\Delta(\bot, fmt, suc) ::= \Delta(fmt, fmt, suc)$$

As we regard multiple format properties we extend the Δ operator to handle tuples of properties $\overline{fmt} ::= (fmt_1, \ldots, fmt_n)$:

$$\overline{\Delta}(\overline{pre}, \overline{fmt}, \overline{suc}) ::= (\Delta(pre_1, fmt_1, suc_1), \ldots, \Delta(pre_n, fmt_n, suc_n))$$

Figure 3 shows two examples. When calculating the pattern for "this", its format ($\overline{t_2}$) is compared to $\overline{t_1}$ and $\overline{t_3}$. As $\overline{t_1}$ and $\overline{t_3}$ describe the same format, we get $\overline{\Delta}$(('Standard', Times, 11pt, regular, roman), ('Standard', Times, 11pt, bold, roman), ('Standard', Times, 11pt, regular, roman)) = (∗, ∗, ∗, bold, ∗). The pattern for "that" is more interesting. Style, font name and size are equal for $\overline{s_1}$ and $\overline{s_3}$, while the variation differs. Our generalization operator yields $\overline{\Delta}(\overline{s_1}, \overline{s_2}, \overline{s_3}) =$ ('∗', ∗, ∗, bold, roman). This captures all differences of $\overline{s_2}$ to surrounding texts.

Fig. 3. Calculating the minimal pattern.

4.3 Inferring Types of Elements

The next task is to determine real element types, that is, element types that are defined in the DTD. This is carried out by the so-called *inference function* which computes DTD element types for given format tuples. In order not to make too many assumptions about the structure of documents in general and naming conventions, the element types are derived only from the DTD attached to this document (see assumption 1). There can be no automatic mapping from dummy types to DTD-defined types. So, somebody (the author or the publisher, for example) has to indicate which dummy types represent which element types in the DTD resulting in a map like that in figure 4.

Note that the format pattern ('Standard', ∗, ∗, ∗, ∗) occurs twice in the map, so `<Paragraph>` and `<Author>` are equally possible. In general, one format tuple may be mapped to multiple element types and one element type may have multiple matching format tuples.

Format Tuple	Element Type
(‘Heading’, ∗, 16pt, ∗, ∗)	`<Title>`
(‘Standard’, ∗, ∗, ∗, ∗)	`<Paragraph>`
(‘Standard’, ∗, ∗, ∗, ∗)	`<Author>`
(‘Standard’, ∗, ∗, ∗, italic)	`<InlineEmphasize>`

Fig. 4. An example for a FORMAT-TO-TYPE-MAP with element types derived from the DTD.

This results in the following characterization of the inference function: The function computes a list of element types based on the text fragment, its context, and its format tuple. This *match list* is ordered by match confidence, which is in a simple form determined by the number of matching properties. A more sophisticated variant could take into account that certain properties (for example the style name) are more important than others.

The author may want to ensure that his document is valid (see [2]) with respect to a given DTD, in which case a validating mode is entered. In this mode, the inference function queries the DTD to determine which element types are allowed in the context of the text fragment. If the intersection of the match list and the set of allowed element types is not empty, the match list is restricted to this intersection. If it is empty, nothing is changed.

The element type with the highest rank in the match list is the result of the inference function and assigned to the text fragment. If several entries have an equal rank, the author has to decide which type should be assigned.

5 Application Context

We are using the technique described above in the implementation of the authoring tool aTool as an extension of MS Word since it is the word processor most authors use for non-mathematical text, according to the Springer-Verlag’s experience. The implementation adopts the principles of MS Word and hides XML jargon wherever possible.

To apply their journal-specific guidelines, our project partner Springer-Verlag aims at receiving XML documents from authors. The XML documents must at least conform to a DTD provided by the publisher. In this context assumption 1 “Known target document type” and assumption 2 “Textual nature” are true. We are additionally looking at author guidelines which can not be expressed with a DTD, but can be formalized, for example length constraints. These formalized guidelines beyond the expressive power of a DTD are called *additional constraints*.

The authoring tool draws its parameter settings from the so called *aToolKit*, which consists of the DTD, the additional constraints, an initial FORMAT-TO-TYPE-MAP and other data required for presentation and formatting features. The publisher is responsible for an appropriate configuration shipped in the aToolKit.

The initial map is expected to map MS Word's named styles directly to element types of the DTD. In order to relieve the author from the burden to associate dummy types with DTD definitions the map also contains heuristic entries like ('$*$', $*$, 16pt, bold, $*$) $\mapsto$ `<title>` that map directly formatted text onto structural entities.

The aTool application keeps the additional structural information in an attributed tree which correlates to the XML structure and accompanies every MS Word document processed by aTool. The position and bounds of the elements are maintained by references in the word document model [4] which are stored using bookmarks (figure 5). The attributed tree is stored in a DOM-conforming way, using a third-party XML parser.

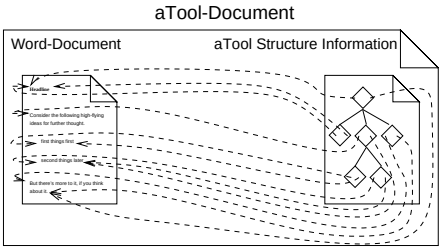

Fig. 5. Internal structure of an enriched document

The main goal of the implementation is to remain as unobtrusive as possible. We therefore avoid additional user interface elements and interruptions to the usual workflow wherever possible. The tool is supposed to work incrementally; the aTool structure is updated whenever the author changes the format of a text portion, creates a new paragraph or edits the text.

It is up to the author when to change a dummy type to a type given by the DTD. We suggest to use the find and replace metaphor for this purpose. For this purpose the *mapping dialog* is opened to display the format tuples and the possible matches. The author can find elements of a specific type and change their type individually or collectively. In the first case the same format pattern is associated with different types. In the latter case, the entry in the FORMAT-TO-TYPE-MAP is changed to map the pattern to the type indicated by the author.

Any formatting matching this pattern after that change will create an element typed conforming to the DTD.

The FORMAT-TO-TYPE-MAP in the aToolKit is modified to reflect user decisions in the mapping dialog. All elements of the aToolKit are stored in XML files to allow modification by an advanced user with any editing tool of his choice.

Querying the Author

User interaction is required to resolve ambiguous type matches from the FORMAT-TO-TYPE-MAP. We delay the query through the concept of warning markers that has proven to be very useful in our PROGRES development environment [10]. A simpler form of this concept is also used in continuous spell checking or the PowerPoint 2000 light bulb tips.

Instead of an immediate query, we mark the element internally. The author does not necessarily see that mark, he can just type ahead. When he wishes to view the marks, these elements are highlighted, for example shown in a special color or underlined. The errors and warnings are also visualized in additional views provided by aTool.

The author can then select a mark in any view and correct the document to match the DTD. For ambiguous element types he is presented the list of possible types produced by the inference function. The tool supports the author with a function to navigate through the marks. This provides him with a systematic way to handle all warnings in the document.

6 Conclusion and Plans

The technique described in section 4 achieves the goals outlined in the introduction: based on a consistent application of typography and a description of the expected structure, an inference function based on a FORMAT-TO-TYPE-MAP can be constructed that returns appropriate element types given a specific format. By providing a specific FORMAT-TO-TYPE-MAP for user groups, the correctness of structure inference can be improved, which will also increase acceptance of aTool among authors.

As the inference function maps from format tuples, not only from style names, we do not require the usage of styles, which has proven to be a noticeable barrier for many authors.

The publisher can fit the description of the structure to his needs and therefore is in control of the resulting XML document. The elements typically required are bibliographic information, hierarchical structure, and literature references. All these element types can be determined using the technique described here.

6.1 Plans

While the inferred structure is valuable for the publisher, the gain for the author is still marginal. He has some more structural views on the document, with attached navigation and editing facilities on structural elements. The structure information may also be used to easily derive variants of the document formatted in various output forms of the further publishing chain.

If speed of processing is important, the author may value *extended validation* functionality that validates the XML structure against the DTD and checks additional constraints implied by author guidelines. Error reports could be propagated into the document by using markers as described in section 5.

The algorithm described in section 4 focuses on elements below the paragraph level. A natural extension would use paragraph format properties to derive XML elements for paragraphs. This would extend the algorithm from inline to block elements as they are distinguished by other tools for structured documents like HTML-Tidy [12]. The preceding and subsequent paragraphs can then be used to trim the match list. If the element may only appear at this position nested into other ones, these may be generated automatically. This approach may be generalized based on the results of the CIDRE project [3], attaching probabilities to optional elements.

Some semantical structure is also the prerequisite of taking word processors to the level of text processors or authoring applications. This would take up research in the line of the Writing Environment [14] up to SEPIA [15]. Simple measures would be instantiation and maintenance of patterns, visualization of semantical structure and dependencies [5], complex derivation of audience-specific documents and alternative access routes [6].

6.2 Thanks

We would like to thank Ms. Prof. Dr. Brüggemann-Klein and Ms. Gross from Technische Universität München and Ms. Hepper, Mr. Dr. Wilson, and Mr. Schreiber from Springer-Verlag for the helpful, interesting and sometimes lively discussions about these topics.

References

1. American Psychological Association. *APA-Style Helper 2.0*, 11 2000. `http://www.apa.org/apa-style/`. 2
2. Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen, editors. *Extensible Markup Language (XML) 1.0*. W3C, February 1998. 4.3
3. Rolf Brugger, Frédéric Bapst, and Rolf Ingold. A DTD Extension for Document Structure Recognition. In Roger D. Hersch, J. André, and H. Brown, editors, *Electronic Publishing, Artistic Imaging, and Digital Typography*, volume 1375 of *Lecture Notes in Computer Science*, page 343ff. Springer, 1998. 3, 6.1
4. Microsoft Corporation. *Microsoft® Office 2000*. Microsoft Press, 1999. 5
5. Felix Gatzemeier. Patterns, Schemata, and Types — Author Support Through Formalized Experience. In B. Ganter and G. Mineau, editors, *Proc. International Conference on Conceptual Structures 2000*, volume 1867 of *LNAI*. Springer, 2000. 6.1
6. Felix Gatzemeier and Oliver Meyer. Improving the Publication Chain through High-Level Authoring Support. In Manfred Nagl, Andy Schürr, and Manfred Münch, editors, *Applications of Graph Transformation with Industrial Relevance (AGTIVE)*, volume 1779 of *LNCS*, Heidelberg, 2000. Springer. 6.1
7. Schema GmbH. Word as HTML/XML/SGML-editor with the MarkupKit. `http://www.schema.de/sitehtml/site-e/wordasht.htm`. 2
8. i4i. S4 Text. Infrastructures for Information Inc. `http://www.i4i.com/tech_s4text.htm`. 2
9. Ora Lassila and Ralph R. Swick, editors. *Resource Description Framework (RDF) Model and Syntax Specification*. W3C, Feb 1999. `http://www.w3.org/TR/REC-rdf-syntax`. 3
10. Manfred Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *LNCS*. Springer, Heidelberg, 1996. 5
11. Pergamon Press, Elmsford, NY. *Manuscript Manager (APA Style) user guide*, 1988. 2
12. Dave Raggett. HTML Tidy. `http://www.w3.org/People/Raggett/tidy`. 6.1
13. Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, 1989. 2
14. J. B. Smith and M. Lansman. A Cognitive Basis for a Computer Writing Environment. In B. K. Britton and S. M. Glynn, editors, *Computer Writing Aids: Theory, Research, & Practice*, pages 17–56. Erlbaum, Hillsdale, NJ, 1989. 6.1
15. Norbert A. Streitz, Jörg M. Haake, Jörg Hannemann, Andreas Lemke, Wolfgang Schuler, Helge Schütt, and Manfred Thüring. SEPIA – A Cooperative Hypermedia Authoring System. In D. Lucarella, J. Nanard, M. Nanard, and P. Paolini, editors, *Proceedings of ECHT'92, the Fourth ACM Conference on Hypertext*, pages 11–22, Milano, Italy, 1992. ACM. 6.1