ELECTRONIC PUBLISHING, VOL. 6(4), 1–12 (DECEMBER 1993)

MarkItUp! An incremental approach to document structure recognition

PETER FANKHAUSER AND YI XU

GMD-IPSI
Dolivostr. 15
D-64293 Darmstadt, Germany
email: {fankhaus, xuyi}@darmstadt.gmd.de

SUMMARY

This paper presents *MarkItUp!*, a system to recognize the structure of untagged electronic documents which contain sub–documents with similar format. For these kinds of documents manual structure recognition is a highly repetitive task. On the other hand, the specification of recognition grammars requires significant intellectual effort. Our approach uses manually structured examples to incrementally generate recognition grammars by means of techniques for learning by example. Users can structure example portions of a document by inserting mark–ups. *MarkItUp!* then abstracts and unifies the structure of the examples. On this basis it tries to structure another example with similar format. Users can correct or accept the produced structure. With every accepted example thereby a grammar is acquired and gradually refined, which can be used to successfully structure the other portions of the document.

KEY WORDS Document structure recognition Learning by example Structure unification SGML

1 INTRODUCTION

The purpose of document recognition is to extract information from documents. This ranges from character recognition and the identification of the layout of printed documents [1,2,3,4,5,6], the recognition of the (logical) structure of documents [7], and at the high end to the (largely domain dependent) extraction of semantic content. In this paper we focus on recognizing the logical structure of untagged electronic documents to transform them into structured SGML (Standard Generalized Markup Language) documents [8,9,10]. This application plays an important role in the publication cycle. Structured documents can much better be exchanged and further processed to produce hyper–documents, individualized printed documents or databases.

Our approach aims at publicly available electronic information sources, such as public databases, bulletin boards, and electronic mail. These sources usually provide *repetitively* structured documents, i.e., the documents consist of (nested) sequences of sub–documents with similar structure. However, the information providers use different and inconsistent formatting conventions to express the structure, even within one source or document. In order to incorporate such documents into an integrated publication environment, their structure has to be made explicit and homogenized, arriving at a consistent structure expressed by means of a formal language, in our case SGML. Structuring the documents

0894–3982/93/040001–12$11.00

Received 15 August 1993
Revised 1 December 1993

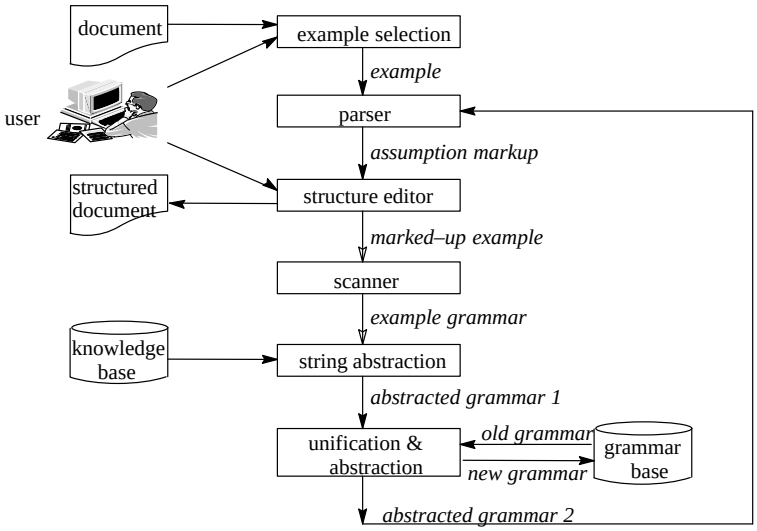

Figure 1. Markup cycle

manually is a highly repetitive task. But also the specification of recognition grammars [11] based on a complete definition of the structure requires significant intellectual effort due to structural differences and formatting inconsistencies among the sub–documents.

The repetitive structure of the documents under consideration lends itself to a machine learning approach, using manually structured example portions to *generate* recognition grammars for automatically structuring the rest of the document. In this paper we present *MarkItUp!*, a system which uses techniques for editing by example [12,13] and more generally learning by example [14,15] to gradually acquire recognition grammars. *MarkItUp!* supports a structure editor which can be used to transform example portions of documents into SGML–documents by inserting *markups*. Predefined recognition patterns, such as delimiters or application specific formats, are used to abstract from the content and format of the example, and rewrite rules are used to abstract from its structure. Thereby a preliminary recognition grammar is generated. This grammar can be used to parse another example by means of the parser generator DREAM [11] developed at our institute, which has specific features to support fault–tolerant parsing. The resulting structure assumption can be accepted or modified by the user, and a refined recognition grammar can be generated by unifying the original structure with the new structure. By repeating this cycle of (manually) marking up, and (automatically) abstracting, unifying and parsing, the grammar is gradually refined, and the amount of manual structuring is gradually reduced.

2 OVERALL APPROACH

Figure 1 represents the overall approach of *MarkItUp!* according to the data flow.

The system always “starts” when the **user** tries to mark up an example portion of the document. For this purpose, the user selects an example (**example selection**), which is sent to the **parser**. Since initially there is no grammar, the parser does not further structure the

example. Otherwise, it tries to parse the example with as much structure as possible. If the user is not satisfied with the produced structure, the user can provide new markups by means of a simple yet comfortable **structure editor**.

The marked–up example is then stored as **structured document**, and passed to a **scanner**, which generates a grammar describing the format and structure of the example. A nonterminal of the grammar corresponds to a markup (tag). Its definition in the form of a rule is generated on the basis of the example structure. A terminal stands for an atomic string. We distinguish two kinds of terminals: "*cut string*" which was deleted explicitly, or which is not properly surrounded by begin and end tags; and "*copy string*" for the rest. Only "*copy string*"s constitute the contents of the structured document.

The initial grammar thus can parse and mark up only the original example. In order to mark up subsequent examples, the grammar has to be abstracted such that it parses different content and slightly different structure.

For this purpose, we first abstract the terminals (**string abstraction**) by using a predefined but extensible knowledge base containing a lattice of typical string patterns, such as delimiters or alphabetic characters (see Section 3).

Then we unify the new grammar with the grammar acquired from previous examples (initially empty) and generalize it at structural level (**unification** & **abstraction**), in order to reflect structural deviations, such as missing or multiply occurring elements. For this purpose, we use heuristic rewrite rules derived from our experience in specifying recognition grammars for DREAM [11]. The new grammar is memorized and can be used to parse a new example, whereby the cycle is closed.

3 LEARNING AT THE STRING LEVEL

The goal of abstracting a string is to arrive at patterns for the content level which accept similar strings also, i.e., strings which belong to the same concept. As we are not interested in analyzing the semantics of the document but only in the syntactic structure, fairly syntactic concepts like "alphanumeric character", "word", "delimiter", suffice for our purposes. For specific classes of documents (like public databases, or tex–files) one can even identify more specific concepts like "field–name" or "tex–macro". These string concepts can be easily described by means of regular expressions. Based on the partial ordering of regular expressions [16] the concepts can be organized in a lattice with root <any string>. A concept thereby describes a superset of the set of strings described by its subconcepts.

A string is abstracted by determining the most specific sequence of concepts which matches the string. If the concept lattice is deeply nested, this first abstraction may still be too specific. Thus, we produce a number of more generic abstractions for each nesting depth, from which the user may choose the most suitable one - by default we take the most specific one. Finally, subsequent occurrences of equal patterns are abstracted to an arbitrarily long repetition of this pattern.

The result of applying string abstraction to the marked up example is thus a sequence of markups, cut patterns, and copy patterns. What remains to be done, is to determine the context of a cut pattern. Given a rule like *element = ..., element1, cut(pattern), element2, ... , cut(pattern)* can be either included into the preceding *element1*, the following *element2*, or remain within the context of element. We have considered three strategies: (1) let the user decide by explicitly cutting a string within the context of an element, (2) recognize not explicitly cut strings when they occur between two (different) begin (or end) tags, and

<bibentry><code>\ rp</code>
! <author><fname>Abadi</fname> <lname>Martin</lname></author>
<location>@ Stanford Un. * CSD</location>
<title>" Temporal-Logic Theorem Proving "</title>
<category>> DBDlogic</category>
</bibentry>

Figure 2. Marked-up example

leave them where they are, (3) keep the common substrings in all (not explicitly) cut strings of a sequence within the sequence, and include the non common prefixes into the preceding element and the non common suffixes into the following element, respectively. Currently, only strategies (1) and (2) are supported.

Figures 2 and 3 give an example of applying string abstraction and the strategies for determining the context of a cut string. For simplicity, we assumed a very simple set of recognition concepts only ([A-Z]; [a-z]; [a-zA-Z]; and single character patterns for all special characters). The "! " and the " " in element **<author>** are recognized as a "cut string" by strategy (2) whereas all other "cut strings" (denoted in Figure 2 by string) are cut explicitly.

4 LEARNING AT THE STRUCTURAL LEVEL

We use a subset of SGML [9] for representing the logical structure of documents – also called document type definitions (DTDs). DTDs are grammars consisting of rules which break down logical elements into more simple elements using a number of operators for *sequence*(','), *repeated elements*('+'), *optional elements*('?'), and *alternatives*('|').

The goal of learning at structural level is to incrementally construct a grammar from a finite number of structured examples in such a way that also the similar but not necessarily identical structure of other examples can be recognized automatically. We can distinguish two steps in this process: First, we describe how new example structures are *unified* with an already existing grammar such that the resulting grammar can recognize *exactly* the structure of the additional example. Then, we extend the strict unification to heuristics for *abstraction*, whereby the resulting grammar can anticipate small structural deviations also.

The most straightforward way of unifying grammars derived from several structured examples would be to simply form a top level disjunction. However, such a grammar would soon get highly redundant and, more gravely, give no clue for further abstraction. The simple

<bibentry><code>cut(\), copy([a-z]+)</code>
cut(!)<author><fname>copy([A-Z][a-z]+)</fname>
cut( )<lname>copy([A-Z][a-z]+)</lname></author>
<location>cut(@), copy([a-zA-Z *.]+)</location>
<title>cut("), copy(([A-Z][a-z]+(\-[A-Z][a-z]+)?( )?)+), cut(")</title>
<category>cut(>),copy([A-Z]+[a-z]+)</category>
</bibentry>

Figure 3. String abstracted example

disjunction (enumeration) of all different example substructures of each element does not carry us much further. Each element would be defined as an alternative of highly overlapping sequences. The unification rules described in the following *merge* new structures with existing element–rules such that the *commonalities* are represented only once and the *structural deviations* are made explicit.

Rules for Unification:

All unification rules take two parameters. One parameter is the already acquired definition of some element, the other parameter is the (possibly empty) sequence of elements derived from the new example. The two parameters are separated by symbol ";" in a rule. By applying the rules exhaustively in order of their specification we arrive at grammars where for each element definition the following holds: For each alternative $(A_1|\ldots|A_n)$ and for all $A_i, Aj, i \neq j, unify(A_i; Aj) = A_i|Aj$, or in other words, the grammar can not be further simplified.

1. Unification of elements or sequences with an empty element ε:
   $unify(A;\varepsilon) = A?$
   A is an arbitrary expressions (except $A = \varepsilon$).
2. Unification of an expression with a more specific expression:
   $unify(A; B) = A$ if $A \leq B$
   A, B are arbitrary expressions. The relationship $\leq$ denotes the usual partial ordering on regular expressions (see Section 3)
3. Unification of optional elements:
   $$unify(A?; B) = \begin{cases} A* & \text{if } B = A+ \\ (unify(A; B))? & \text{otherwise} \end{cases}$$
   A, B are arbitrary expressions.
4. Unification of sequences with a common prefix or suffix:
   (a) $unify(A, B; A', C) = A, unify(B; C)$ if $A' \leq A$
   (b) $unify(B, A; C, A') = unify(B; C), A$ if $A' \leq A$
   A, A', B, C are arbitrary expressions
5. Unification of alternatives:
   $$unify(A_1|...|A_n; B) = \begin{cases} A_1|...|A_n|B & \text{if } unify(A_i; B) = A_i|B \text{ for all i} \\ C_1|...|C_n & C_i = \begin{cases} unify(A_i; B) & \\ \quad \text{if } unify(A_i; B) \neq A_i|B & \\ A_i \quad \text{otherwise} & \end{cases} \end{cases}$$
   A_i, B are arbitrary expressions except alternative, $1 \leq i \leq n$.

   If $unify(A_i; B) = A_i|B$ for all i, i.e., there exist only *trivial* unifications between B and A_i, then we simply add B as an additional alternative. Otherwise, we merge B with all those A_i for which there exists a *non trivial* unification of A_i and B.
6. Trivial unification:
   if none of the above rules applies
   $unify(A; B) = A|B$
   A, B are arbitrary expressions

In addition to these unification rules, we utilize a number of simplification rules which take the associativity of sequence and alternative into account.

Rules for Abstraction:

We distinguish three kinds of abstraction: (1) Implicit abstraction arises from the fact that we unify each right–hand side of each rule with an eventually occurring new example, regardless of the nesting depth. Obviously, the resulting grammar will be more generic than the top level disjunction of the old grammar with the grammar derived from the new example. (2) Another kind of abstraction takes place during merging when only *trivial* unification is possible. (3) The third kind of abstraction is applied *after* the example has been merged into the old grammar to further simplify the inferred grammar.

For merging new examples with the existing grammar in a more tolerant way (2) we introduce three additional rules: Whereas the unification Rules 4a and 4b merge only sequences with a common prefix or suffix, the abstraction Rules 7a-c merge sequences with a number of common subsequences interleaved by distinct subsequences.

7. Abstraction merge of sequences:
   (a) *abstract-merge*$(A, B_1, \ldots, B_n; C_1, \ldots, C_n)$
   $= A?, unify(B_1, \ldots, B_n; C_1, \ldots, C_n)$
   A, B_j, C_h are arbitrary expressions, B_j, C_h no sequences
   $A \neq C_1, \ldots, C_n$ and $B_1, \ldots, B_n, B_i < C_i$ (or $C_i < B_i$)
   (this restriction excludes the applicability of unification rules).
   (b) *abstract-merge*$(B_1, \ldots, B_m, A; C_1, \ldots, C_n)$
   $= unify(B_1, \ldots, B_m; C_1, \ldots, C_n), A?$
   with similar restrictions as above
   (c) *abstract-merge*$(A_1, \ldots, A_k, B_1, \ldots, B_n; D_1, \ldots, D_k, C_1, \ldots, C_n)$
   $= unify(A_1, \ldots, A_h; D_1, \ldots, D_k), unify(B_1, \ldots, B_n; C_1, \ldots, C_n)$
   A_i, B_j, Cp, Dq are arbitrary expressions except sequence
   if $A_1, \ldots, A_k \neq D_1, \ldots, D_k$ and $B_i < C_i$ (or $C_i < B_i$)
   or $B_1, \ldots, B_n \neq C_1, \ldots, C_n$ and $A_j < D_j$ (or $D_j < A_j$)

In 7a and 7b subsequences, which do not occur in the new example and vice versa, become optional, in 7c some binary partition of the old sequence and the new sequence is chosen for further unification. By imposing the restrictions we achieve a deterministic merge of maximum common subsequences and consequently identify only distinct subsequences with minimum length as optional. We have also experimented with a consistent weighting scheme for measuring the degree of abstraction performed by a certain rule and with using the weight for pruning abstractions more continuously. However, as the goal of identifying only *maximal* common subsequences is *crisp*, at least this kind of structural abstraction can be achieved by a *crisp* strategy.

To further simplify the grammar (3), Rule 8 groups finite sequences of consecutive equal subsequences in a similar way as for abstraction at string level.

8. Abstraction of repeated elements:
   $abstract(A, (B_1, \ldots, B_m), (B_1, \ldots, B_m), \ldots, (B_1, \ldots, B_m), C)$
   $= A, (B_1, \ldots, B_m)+, C$
   A, C are arbitrary expressions, B_i not an ordered sequence, $1 \leq i \leq m$

Examples:

The following examples shall illustrate the usage of the above rules. For sparing some parentheses we assume the binding precedence (? + |).

Example 1 Unification of sequences with a maximum common prefix or suffix:
$unify(a, b, c, d, e; a, d) \stackrel{4a}{\Rightarrow} a, unify(b, c, d, e; d) \Rightarrow a,$ ***abstract–merge***$(b, c, d, e; d)$
$\stackrel{7a}{\Rightarrow} a, (b, c)?, unify(d, e; d)$
$\stackrel{4a}{\Rightarrow} a, (b, c)?, d, unify(e; \varepsilon)$
$\stackrel{1}{\Rightarrow} a, (b, c)?, d, e?$

Example 2 Unification of alternative elements:
$unify(a|(b, d)|c; d) \stackrel{5}{\Rightarrow} a|unify(b, d; d)|c$
$\stackrel{4b}{\Rightarrow} a|(unify(b; \varepsilon), d)|c$
$\stackrel{1}{\Rightarrow} a|(b?, d)|c$

5 IMPLEMENTATION AND EXAMPLE SESSION

The main components of *MarkItUp!*, i.e., the structure editor, the scanner, the algorithms for string abstraction and structure unification and abstraction are implemented in Smalltalk. The actual parser DREAM is implemented in C++. These components interact as follows:

A document to be marked up is first loaded into the *structure editor* where the user can select and mark up parts of a document. The marked up example (see Figure 2) is passed to the *scanner* which generates an initial grammar. Each terminal element (copy string) in the grammar rules is abstracted based on the concept lattice (see Section 3).

From this grammar a DREAM grammar is generated in two steps: First, the SGML structure is built, then the cut–copy groups at string level are used to form the expressions at string level. To parse also subsequent examples with slightly deviating structure and content which are not yet properly described by the abstractions at string level, each ELEMENT name gets associated with a fallback rule (anything), which parses all those parts which can not be parsed by one of the available ELEMENT definitions. Such portions, which are surrounded by **<anything>** and **</anything>**, can then be easily identified and further disambiguated by the user.

Figure 4 shows a complete DREAM grammar (DSD) generated (with the full set of abstraction concepts).

The effect of the fallback rule **<anything>** when applying the DSD to a new example is shown in Figure 5. For instance, in the new example there are now two **<author>**s (lines starting with "!") instead of one and no **<location>** (lines start with "@"). As the original example contained only one **<author>** and one **<location>**, the second **<author>** and the following items are now marked up as **<anything>**. However, the subsequent **<category>** ("> DBDkb") is properly marked up.

The user can correct the result (see Figure 6) by means of the *structure editor*.

The corrected example passes the above components step–by–step, producing the following rule for **<bibentry>**:

<!ELEMENT bibentry - - code, author, author, title, category>

```
<!DOCTYPE bibdoc [
<!ELEMENT bibentry - - code, author, location, title, category >
<!ELEMENT code      - - (anything#, cut(^?"\\ "), copy(([a-z])+ | ([a-zA-Z])+))? >
<!ELEMENT location  - - (anything#, cut($^"@ "),
                         copy([A-Z]([a-z])+[\ ][A-Z][a-z][.][\ ][\*][\ ]([A-Z])+
                         | ([a-zA-Z])+[\ \* \ ]([a-zA-Z])+[.]([\ \* \ ])+([a-zA-Z])+))? >
<!ELEMENT fname     - - (anything#, copy([A-Z]([a-z])+ | ([a-zA-Z])+))? >
<!ELEMENT title     - - (anything#, cut($^"\" "),
                         copy([A-Z]([a-z])+[\-]([A-Z]([a-z])+[\ ])+[A-Z]([a-z])+
                         | ([a-zA-Z])+[\-]([a-zA-Z])+[\ \* \ ])+([a-zA-Z])+))? >
<!ELEMENT category  - - (anything#, cut(" \""$^"> "),
                         copy(([A-Z])+([a-z])+ | ([a-zA-Z])+), cut($^))? >
<!ELEMENT lname     - - (anything#, cut(" "), (copy([A-Z]([a-z])+ | ([a-zA-Z])+)))? >
<!ELEMENT author    - - (anything#, cut(" "$^"! "), fname, lname)? >
<!ELEMENT anything  - - copy(.#) > ]>
```

Figure 4. The DREAM DSD of the example grammar

```
<!DOCTYPE bibdoc>
<bibentry><code>rp</code>
<author><fname>Abarbanel</fname>
<lname>Robert</lname>
</author>
<location></location>
<title></title>
<category><anything>! John Williams
" A Relational Representation for Knowledge Bases "</anything>
> DBDkb</category>
</bibentry>
```

Figure 5. An incompletely marked-up example generated by DREAM DSD

```
<!DOCTYPE bibdoc>
<bibentry><code>rp</code>
<author><fname>Abarbanel</fname>
<lname>Robert</lname>
</author>
<author><fname>John</fname>
<lname>Williams</lname>
</author>
<title>A Relational Representation for Knowledge Bases</title>
<category>DBDkb</category>
</bibentry>
```

Figure 6. The corrected markup

Since there are repeated elements for **<author>** the new element of **<bibentry>** is abstracted by Rule 8 first, resulting in

<!ELEMENT bibentry - - code, author+, title, category>

This rule is unified with the original rule for **<bibentry>** (see Figure 4) as follows:

$$
\begin{aligned}
& unify(code, author, location, title, category; code, author+, title, category) \\
& \stackrel{4a}{\Rightarrow} code, author+, unify(location, title, category; title, category) \\
& \stackrel{4b}{\Rightarrow} code, author+, unify(location; \varepsilon), title, category \\
& \stackrel{1}{\Rightarrow} code, author+, location?, title, category
\end{aligned}
$$

The other elements in the SGML–DSDs are abstracted and unified in a similar manner, but not exhibited here due to space limitations.

6 CONCLUSION

We have presented a system for the incremental generation of structure recognition grammars from example structures. Techniques for generating initial recognition grammars from marked-up examples, for abstracting them at string level and for merging the structure of multiple examples have been devised for this purpose. *MarkItUp!* incorporates these techniques into a simple structure editor, which can be used to structurally enrich inconsistently formatted electronic documents with repetitive but implicit structure. Future work will be devoted to the following extensions of the presented approach:

(a) Integration with the techniques for structure recognition for scanned documents (as opposed to electronic documents without a layout component) developed at our institute in CAROL (Cataloging by Automated Recognition Of Literature) [7]: In contrast to *MarkItUp!* CAROL uses much more powerful recognition styles for evaluating layout information of scanned document blocks, such as font (size, family), and (two dimensional) position. On the other hand, CAROL is currently limited to flat document types as used by title pages of PhD theses and research report titles. Thus, *MarkItUp!* can significantly benefit from CAROL with respect to the expressive power of recognition styles, and CAROL can benefit from the learning approach used in *MarkItUp!*, especially from the techniques devised for structural abstraction.

(b) Improvement of the implementation: Currently, *MarkItUp!* does not represent the incrementally structured document as structured object. Thus, examples have to be structured completely, before an according grammar can be generated. Often a more flexible strategy is desirable, i.e., first, a relatively flat structure is applied to the entire document, which is then gradually refined on demand. For this purpose selective access to arbitrary portions of already marked–up documents is required.

(c) Inclusion of restructuring: In many applications, documents have to be restructured to fit into some other application, for example, a reader's view, a hyperdocument with pre-existing document type, a (virtually) integrated database. At our institute currently a parser generator for restructuring SGML documents is being developed [17]. Just as *MarkItUp!* generates grammars for DREAM by example markups, it could be very useful to generate restructuring rules by example restructurings, adapting techniques developed in the field of database integration for coping with structural discrepancies [18]. In fact, the

regularity of SGML as opposed to the cyclicity of object–oriented database schemas and the more specific mapping which can be inferred from example restructurings should make simplified solutions to the restructuring problem feasible.

REFERENCES

1. J. Handley and S. Weibel, 'ADAPT: Automated Document Analysis Processing and Tagging', in *Proceedings of the International Conference on Electronic Publishing, Document Manipulation, and Typography (EP90)*, ed., R. Furuta, 183–192, Cambridge University Press, Cambridge, UK, (1990).
2. R. Ingold, 'Text Structure Recognition in Optical Reading', in *Structured Documents*, 133–141, Cambridge University Press, Cambridge, UK, (1989).
3. R. Ingold, R.P. Bonvin, and G. Coray, 'Structure recognition of printed documents', in *Document Manipulationd and Typography: Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography (EP88)*, ed., J.C. van Vliet, 59–70, Cambridge University Press, Cambridge, UK, (1988).
4. H. Kida, O. Iwaki, and K. Kawada, 'Document recognition system for office automation', in *Proceedings of the Eighth International Conference on Pattern Recognition*, 446–448, IEEE Computer Society Press, Washington, DC, (1986).
5. S.N. Srihari and G.W. Zack, 'Document Image Analysis', in *Proceedings of the Eighth International Conference on Pattern Recognition*, 434–436, IEEE Computer Society Press, Washington, DC, (1986).
6. L.D. Wilcox and A.L. Spitz, 'Automatic recognition and representation of documents', in *Document Manipulation and Typography: Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography (EP88)*, ed., J.C. van Vliet, 47–57, Cambridge University Press, Cambridge, UK, (1988).
7. J. Schmidt and W. Putz, 'Knowledge Acquisition and Representation for Document Structure Recognition: the CAROL Project', in *Proceedings of the Ninth IEEE Conference on Artificial Intelligence in Applications*, eds., F. Bancilhon and D. DeWitt, 183–194, IEEE Computer Society Press, (1993).
8. D. Barron, 'Why use SGML?', *Electronic Publishing*, **2**(1), 3–24, (1989).
9. M. Bryan, *An Author's Guide to the Standard Generalized Markup Language*, Addison–Wesley, Wokingham, 1989.
10. C.F. Goldfarb, *The SGML Handbook*, Oxford University Press, Oxford, 1990.
11. T. Göttke and P. Fankhauser, 'Dream 2.0 user manual', Technical report, Gesellschaft für Mathematik und Datenverarbeitung (GMD), Germany, (1992). Arbeitspapiere der GMD, No. 660.
12. D.H. Mo and I.H. Witten, 'Learning text editing tasks from examples: a procedural approach', *Behavior & Information Technology*, **11**(1), 32–45, (1992).
13. R. Nix, 'Editing by example', in *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, 186–195, (January 1984).
14. P.R. Cohen and E.A. Feigenbaum, *The handbook of artificial intelligence*, volume 3, William Kaufmann, Los Altos, CA, 1982.
15. I.H. Witten and B.A. MacDonald, 'Using concept learning for knowledge acquisition', *International Journal of Man–Machine Studies*, **29**(2), 171–196, (1988).
16. A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison–Wesley, Reading, MA, 1986.
17. G. Ihnofeld, *Spezifikation eines Hypertext Composing Prozesses und Entwicklung einer Regelsprache für die Transformation von strukturierten Dokumenten anhand einer beispielhaften Anwendung*, Master's thesis, TH–Darmstadt, Germany, 1991. (Specification of a Hypertext Composing Process and Development of a Rule Language for the Transformation of Structured Documents based on an Example Application – in German).
18. E.J. Neuhold and M. Schrefl, 'Dynamic Derivation of Personalized Views', in *Proceedings of the 14th International Conference on Very Large Databases (VLDB '88)*, eds., F. Bancilhon and D. DeWitt, 183–194, (August/September 1988).