Department of Computer Science
Series of Publications A
Report A-2000-2

Generalization of Document Structures and Document Assembly

Barbara Heikkinen

University of Helsinki
Finland

Department of Computer Science
Series of Publications A
Report A-2000-2

Generalization of Document Structures and Document Assembly

Barbara Heikkinen

To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism in Auditorium XIV, University Main Building, on April 8th, 2000, at 10 o'clock.

University of Helsinki
Finland

Contact information

Postal address:
Department of Computer Science
P.O.Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki
Finland

Email address: postmaster@cs.Helsinki.FI (Internet)

URL: http://www.cs.Helsinki.FI/

Telephone: +358 9 1911

Telefax: +358 9 191 44441

ISSN 1238-8645
ISBN 951-45-9278-6
Computing Reviews (1998) Classification: I.7.1, I.7.2, I.7.4, F.4.3, H.3.3, H.3.7
Helsinki 2000
Helsinki University Printing House

Generalization of Document Structures and Document Assembly

Barbara Heikkinen

Department of Computer Science
P.O. Box 26, FIN-00014 University of Helsinki, Finland
Barbara.Heikkinen@cs.Helsinki.FI
http://www.cs.Helsinki.FI/Barbara.Heikkinen/

PhD Thesis, Series of Publications A, Report A-2000-2
Helsinki, April 2000, 179 pages
ISSN 1238-8645, ISBN 951-45-9278-6

Abstract

The accelerating evolution of the World Wide Web has made numerous digital document collections widely available for the public. There is a clear need for new tools that assist the user to gather, combine, and reuse information from existing document collections. On the other hand, the amount of fine-structured documents will enormously increase in the near future, since the Extensible Markup Language (XML) is rapidly gaining popularity in various communities. Compared to HTML, XML makes more versatile processing and customization of documents possible. However, explicit structuring using XML leads to heterogeneously structured document collections, which causes problems when combining and reusing fragments of documents.

Document assembly is the computer-aided construction of new documents from existing document collections. Such reuse includes finding relevant document fragments, modifying them as needed, and combining the fragments. This thesis describes a document assembly model based on versatile recognition and manipulation of document fragments that are coherent, contiguous, and relatively independent document parts used as the basis for new assemblies. We also introduce a general document assembly system *SAW* and a specialized system for tailoring textbooks via the Web.

If the assembled documents are to be further processed, the heterogeneous structures of the original documents also have to be unified. This work presents an *element-type classification* method that facilitates uniform processing of heterogeneous structures. The method contains a decision procedure for mapping an arbitrary structure element to a predefined *generic class*. The generic classes are defined in a Document Type Definition (DTD) called *generic DTD*, which can be seen as a metastructure definition describing typical logical structures of electronic documents.

The element-type classification extracts information from document instances by inspecting element relations and average text lengths of element instances. In this way various structures, such as hierarchies and element containers wrapping logical units, can be recognized. The method is formally presented by using the concept of grammar morphism. Various practical examples of applying the generic classes are provided, and the method is applied to several well-known public document types. In addition to document assembly, the results of the element-type classification method can be used, for instance, in automatic generation of stylesheets for structured documents.

Computing Reviews (1998) Categories and Subject Descriptors:

I.7.1 Document and Text Processing: Document and Text Editing
I.7.2 Document and Text Processing: Document Preparation
I.7.4 Document and Text Processing: Electronic Publishing
F.4.3 Mathematical Logic and Formal Languages: Formal Languages
H.3.3 Information Storage and Retrieval: Information Search and Retrieval
H.3.7 Information Storage and Retrieval: Digital Libraries

General Terms:

Documentation, Algorithms, Experimentation, Design

Additional Key Words and Phrases:

Document management, Document assembly, Metastructures, Context-free grammars, XML, SGML, Semistructured data, Digital libraries

Acknowledgments

First of all, I am grateful to my supervisor, Professor Heikki Mannila, who encouraged me to continue studies as well as inspired and guided my work with this thesis. I am deeply grateful to my mentor, Professor Helena Ahonen. You believed in me, you worked with me, and you shared your warm guidance as a colleague and a friend when ever I needed. I would also like to thank Professor Derick Wood, who gave insightful and fruitful comments when I needed them most.

My special thanks belong to PhD Mika Klemettinen and Oskari Heinonen, who are dear colleagues and friends from our "bohm" group composed of me, Oskari, Helena, and Mika. With "bohm" atmosphere, research is worth doing. I would also like to thank the members of our SID research group, namely Professor Pekka Kilpeläinen, Jani Jaakkola, PhD Greger Lindén, Kimmo Paasiala, and Jyrki Niemi, for their work and support. Furthermore, thanks to Professors Jukka Paakki and Inkeri Verkamo, PhD Eila Kuikka, and all of you, who gave invaluable comments to my thesis text.

For the financial support I would like to thank the Finnish Cultural Foundation, the National Technology Agency (Tekes), industrial partners (Aamulehti Group, Edita, National Board of Education, WSOY, Helsinki Media, Lingsoft, and MTV3), and the Helsinki Graduate School of Computer Science and Engineering. Moreover, I would like to thank the Department of Computer Science at the University of Helsinki, headed by Professors Martti Tienari, Esko Ukkonen, and Timo Alanko, for providing the working conditions, and the whole personnel for giving the crucial background support to everyday work.

Finally, I would like to give my thanks and appreciation to my mother, Salli Gargoulas, and to my parents-in-law, Valma and Toivo Heikkinen, who have supported me and my family over the years. Dearest thanks and thoughts belong to my family, to my husband Seppo, and to my daughters Heli and Sonja. You are the sunshine of my life.

Contents

Chapter 1

Introduction

Electronic documents have become a part of everyday life and work for masses of people, since the companies and organizations of today produce their documentation using computer systems, and since more and more people use the World Wide Web both for work and for recreation. The recent evolution of the Web has made numerous digital document collections reachable for ordinary people as well as for experts. The availability of large collections inspires to combine information and reuse existing document fragments in order to assemble new documents. Moreover, the amount of fine-structured documents, especially on the Web, will be increased dramatically in the near future, since the Extensible Markup Language (XML) [W3C98c] is rapidly gaining popularity among various communities. Compared to HTML, XML makes more versatile processing and customization possible. However, explicit structuring of documents also raises new challenges due to the heterogeneous structure of different document collections.

Document assembly. There is a need for new tools that support the on-demand generation of individualized documents through dynamic document assembly. We define *document assembly* (Figure 1.1) as the computer-aided construction of new documents from a collection of electronic documents. Such reuse includes finding the relevant document building blocks, modifying them as needed, and combining the pieces to make a new document. The lack of general computational and conceptual models for document assembly has prevented the development of widely applicable systems.

Document assembly could facilitate and speed up reuse of existing document fragments within many application areas. For instance, a group of teachers could maintain a course material pool consisting of textbooks, ex-

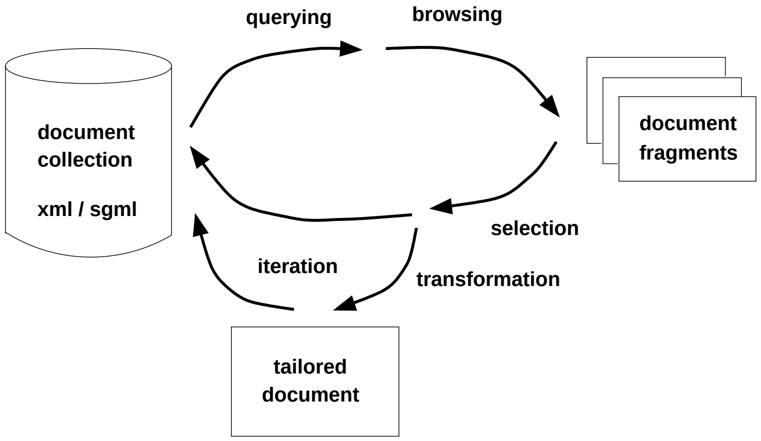


Figure 1.1: Document assembly process.

amples, exercises, examinations, and animations. More general external material, like news, articles, journals, encyclopedias and books, might also be available. When a teacher is starting a new course, he or she could configure a tailored textbook for the current students. Also, teachers could provide their own material, like exercises and examples, for the common pool. Additionally, a journalist collecting background material for an article or an editor of an electronic book could utilize assembly tools to facilitate their work. The payment policies and the copyright issues in this kind of reuse naturally need careful considerations.

An example of an industrial assembly application is composition of technical manuals: A client may order different configurations of a complex machine, like an airplane, a train, an elevator or a paper machine. The user manual could be assembled from heterogeneously structured modules of a large text database. Also, the composition of printed and online manuals could differ considerably.

This thesis describes a novel document assembly model and architecture that have been developed to study principles and methods of intelligent document reuse. Between 1995 and 1998 at the University of Helsinki, a research and development project called "Structured and Intelligent Documents (SID)" studied document assembly and related issues such as document transformation languages. The project was funded by the National Technology Agency (Tekes) and seven Finnish enterprises[1]. Our prototype

[1]The Aamulehti Group, Oy Edita Ab, the National Board of Education, WSOY, Helsinki Media, Lingsoft, Inc., and MTV3

document assembly system called *SID Assembly Workbench (SAW)* and a specialized system for tailoring textbooks via the Web are presented in Chapter 6. Standard Generalized Markup Language (SGML) [ISO86] was chosen as the basis for the project. This work is extended to consider also XML documents.

XML and SGML are metalanguages for defining application and platform independent document markup languages. XML is a simplified subset of SGML especially designed for Internet publishing. One of the key differences between XML and SGML is that the processing of XML documents does not necessarily require a schema specification, which is called Document Type Definition (DTD): DTD defines the structure of a class of related XML or SGML documents and the markup used in document instances for indicating the structure. For over a decade, SGML has been widely adopted by industry, since it offers a framework for creating structured documents with high potential for longevity and multi-purpose reuse. Since the birth of the XML specification in the beginning of 1998, the evolution of companion specifications enhancing the properties of XML as well as individual markup languages conforming to XML has been extremely fast. XML is also used as a data representation and exchange format resulting in applications for various areas other than document management.

What XML and SGML technologies do not provide are ready solutions for managing and processing large heterogeneously structured document collections. Although one of the strengths of the technologies is to allow several representations and formats to be generated from the same documents, reuse of documents by dynamic assembly of structure elements is still an unsolved problem, and hardly any tools or general techniques exist. The necessity of document assembly is, however, clearly considered to be of importance [Kim96, McF96, Mur98, PVH98, VP98, W3C00d].

We have developed a document assembly framework based on versatile recognition and manipulation of *document fragments*, which are consistent and relatively independent document parts used as building blocks for the tailored document to be composed. Similar ideas recur at document manipulation and information retrieval meetings under different names, such as passages, semantic fragments, information units, minimal revisable units, or micro documents. The assembly framework is not inherently XML or SGML specific, but the possibility of using generic semantic document markup and the existence of tools for a standardized formalism makes it much more easier to implement a document assembly system based on the fragment framework.

Generalization of document structures. If the new document assemblies are to be further processable, the heterogeneous structures of the original documents also have to be unified. This thesis presents an *element-type classification* method, which facilitates uniform processing of heterogeneous document structures by offering a decision procedure to map an arbitrary structure element to a predefined *generic class*. The generic classes are defined in an XML DTD called *generic DTD*, which can be seen as a metastructure definition describing typical logical structures of electronic documents.

The approach is based on the fact that although no semantics can be inferred from the names of XML or SGML elements, some knowledge can be extracted from the element hierarchy, e.g., which other elements an element can contain and how long the text content is on average. To be more precise, the mapping of original element types into generic element types is based on element relations and on average text lengths of element instances in document instances. The logical decision rules of this work decide, which generic class would be closest to the original element type.

The element-type classification recognizes various document structures, such as element hierarchies and element containers without any text content of their own. The basic elements of the generic hierarchy are *sections*, *paragraphs* and *strings*. Typically, strings are short sequences of characters, text paragraphs contain some text and marked strings, and sections in turn consist of paragraphs and other sections. Furthermore, container elements wrap logical units together. For example, a list might consist of items, which may consist of one or several paragraphs. In this case, the generic structure would be a *paragraph group* consisting of *paragraph containers*, which in turn consist of one or several paragraphs.

When the element types of a document type are classified, and when classified element names are used instead of the original ones in the document instances, the hierarchical structure is preserved. In addition, the document instances follow the structure of the generic DTD. The element-type classification may simplify the original structure, as several original element types may be mapped into one generic class. Nevertheless, the original element names can be saved in attributes, and thus the original structure can be transformed back, if necessary. The method itself is simple, easy to implement, and very efficient.

The set of generic classes should be as minimal as possible, since, in practice, the user cannot manage too complicated generic structures. In typical electronic publishing or industry DTDs there are tens or over a hundred different element types. Hence, a user who is not familiar with the specific structure may use the generic classes along with the original ones, which helps to get an overview of the complicated structures. The element-type classification is also useful for other applications than document assembly such as automatic generation of stylesheets for XML and SGML documents. More generally, inferring and extracting useful information from heterogeneous document structures facilitates the search and query of documents, which is one of the key issues when finding information from the Web.

One of the main contributions of this dissertation is the novel element-type classification method and the generic DTD for generalization of document structures. I formally present the method, and I also describe the test results of the method for several public document types. These issues are published for the first time in this thesis. Another contribution of the dissertation is the detailed and coherent presentation of the general document assembly framework that was developed and implemented in the SID project. I participated in designing and implementing the assembly framework as a member of the project. One of the key ideas was to apply my element-type classification for manipulating heterogeneous document structures in document assembly. I also implemented the specialized system for tailoring textbooks that was developed in the early stages of SID. I have participated in producing the following document assembly related articles [AHH+96a, AHH+96b, AHHK97a, AHHK97b, AHH+98a, AHH+98b].

The rest of the thesis is organized as follows. Chapter 2 gives an overview of document management including the basic concepts of structured documents, some aspects of XML and SGML technologies, and a description of related work. Chapter 3 presents the basic principles of the element-type classification method and the document assembly process in detail. The element-type classification is formally presented in Chapter 4 by using the concept of grammar morphism, while results of applying the method in practice are described in Chapter 5. A general document assembly prototype called SAW and a system for tailoring textbooks via the Web are described in Chapter 6. The SAW system offers several possibilities to find and select interesting document fragments. As a result, a uniformly structured XML document is automatically constructed by using generic classes. Finally, Chapter 7 contains a short conclusion and a discussion of future directions.

Chapter 2

Document management

Document management offers facilities for a user to manage electronic documents: The user needs to create new documents, edit and browse them, and finally store the work. Furthermore, the user needs to find information from an individual document as well as from document bases. When the system prints the content of a document, it requires formatting instructions for producing the layout. The formatting instructions may be an inherent part of the system, or the user may provide a stylesheet containing the formatting instructions. A basic need is document interchange, which is enabled by widely spread use of international document standards.

Companies and organizations of today produce nearly all of their documentation using computer systems, and a common phenomenon is that some parts of the documentation may be available on the Web for public or restricted use. Currently, not only the production of manuals is done using advanced systems, such as structure editors, but typically the customer is provided with both printed and electronic versions. Electronic manuals may be incorporated with intelligent guiding and searching facilities. Already for a long time, libraries have provided information retrieval facilities for their material. Typically, only the bibliographic information has been saved to a text database. However, more and more digital libraries with massive text archives and advanced search facilities have become available on the Web.

Electronic publishing is rising in the publishing industry. Many electronic books and encyclopedias are available on CD-ROM. Also, several printed journals have their online versions on the Web. Examples of electronic research journals are D-lib Magazine[1], The Magazine of Digital Library Research, and JEP[2], The Journal of Electronic Publishing, as well as

[1]http://www.dlib.org/
[2]http://www.press.umich.edu/jep/index.html

the Web editions of printed journals, like Sigmod Record[3] and The VLDB Journal[4]. Examples of Web versions of magazines for a wider public are Scientific American[5], New Scientist[6], and TechWeek[7]. Some of the Web newspapers are available only for the subscribers of the printed copies, such as the largest newspaper in Finland, Helsingin Sanomat[8]. Publishers also provide online news services that filter and categorize information for the user, like the WOW[9] service in Finland.

This thesis studies structured electronic documents. There is a common mechanism to identify structures in a document, namely markup languages. The *Extensible Markup Language (XML)* [W3C98c] and the *Standard Generalized Markup Language (SGML)* [ISO86] are two widely used *metalanguages* for defining markup languages. XML is a subset of SGML. Both of them offer document representations that are independent of the document processing software, the operating system, and the hardware. A basic principle when using XML or SGML is to separate the text structure from its presentation. Typically, the markup is used to indicate the logical structure of the text; such markup is called *descriptive markup*. The presentation of structured text is typically defined in stylesheets discussed later.

This chapter is organized as follows. The basics of XML and SGML documents are presented in Section 2.1. XML specification and related specifications are introduced in Section 2.1.1, whereas SGML and its companions are described in Section 2.1.2. Some related studies in document management are introduced in Section 2.2. The research of document management is closely related to work in several research fields, such as digital libraries, information retrieval, multimedia and hypertext, databases, semistructured data, data mining in text, and computational linguistics. This chapter is concluded by introducing some relevant work in digital libraries (Section 2.3), databases (Section 2.4), and semistructured data (Section 2.5).

2.1 Basic concepts of structured documents

The document structure can be utilized in multiple ways: A structure editor, for instance, guides the writing by prompting the required structural

[3]http://www.acm.org/sigmod/record/
[4]http://sunsite.informatik.rwth-aachen.de/dblp/db/journals/vldb/index.html
[5]http://www.sciam.com/index.html
[6]http://www.newscientist.com/
[7]http://www.techweek.com/index.html
[8]http://www.helsinginsanomat.fi/
[9]http://www.wow.fi/

parts and by validating the resulting document structure. In addition to full-text search the user may compose structural queries. It is also possible to generate different layout formats — HTML, PDF, PostScript, RTF — and different document compositions for different kinds of user groups by using transformation languages. Various tasks can be automated, such as the above-mentioned transformation of original documents to different formats enabling automatic or semi-automatic publication of printed paper copies or of online or CD-ROM versions. Since the basic idea is the separation of structure and presentation, it is possible to ensure that related documents have a uniform style. The documents may be stored in document bases, like XML or SGML databases, which are typically accompanied by editing, query, work-flow, and versioning facilities. Finally, a very important aspect of structural markup is that the documents are software and system independent, which enables interchange between different environments. The described characteristics of structured documents lead to longevity of stored information.

The basic structural parts of XML and SGML documents are called *elements*, and together they compose an element tree of the document. An element may contain text and other elements, or it may be empty. Elements are separated by start-tags and end-tags in the text. For example, on row 17 of Figure 2.1 the start-tag is "<title>" and the end-tag is "</title>". Typically, information which is not treated as text content may be attached to elements as *attributes* located in the start-tag. On row 16 the start-tag contains the attribute *explanat*, whose value is "The first act of Shakespeare's Hamlet".

An element appearing in a document instance is called element instance. A document instance belongs to a document class, or a document type defined by the *Document Type Definition (DTD)*, which is a grammar defining the element types and their content models. An excerpt of an XML DTD is presented on rows 4–11 of Figure 2.1. A *document type declaration* contains or points to the DTD. In this example the declaration contains the DTD (rows 2–13). The *XML declaration* on row 1 specifies the version of XML as 1.0 and the character encoding as UTF-8. The name of the document type is *play* (row 2). It must be the same as the name of the root element (row 14).

The content model of the element type *act* on row 4 defines that *act* contains one *title*, zero or more (the operator "$*$") *subtitle*s, zero or one (the operator "?") *prologue*, one or more (the operator "+") *scene*s, and finally, zero or one *epilogue*. The attribute *explanat* of the element type *act* is defined on row 5. Elements of type *title*, *stagedir*, and *speaker* (rows 6, 8,

```
 1: <?xml version="1.0" encoding="UTF-8" ?>
 2: <!DOCTYPE play [
 3:   ...
 4:   <!ELEMENT act      (title, subtitle*, prologue?, scene+, epilogue?)>
 5:   <!ATTLIST act       explanat    CDATA    #IMPLIED>
 6:   <!ELEMENT title    (#PCDATA)>
 7:   <!ELEMENT scene    (title, subtitle*, (speech | stagedir | subhead)+)>
 8:   <!ELEMENT stagedir (#PCDATA)>
 9:   <!ELEMENT speech   (speaker+, (line | stagedir | subhead)+)>
10:   <!ELEMENT speaker  (#PCDATA)>
11:   <!ELEMENT line     (#PCDATA | stagedir)*>
12:   ...
13: ]>
14: <play>
15: ...
16:   <act explanat="The first act of Shakespeare's Hamlet">
17:     <title>ACT I</title>
18:     <scene>
19:       <title>SCENE I.  Elsinore. A platform before the castle.</title>
20:       <stagedir>FRANCISCO at his post. Enter to him BERNARDO</stagedir>
21:       <speech>
22:         <speaker>BERNARDO</speaker>
23:         <line>Who's there?</line>
24:       </speech>
25:       <speech>
26:         <speaker>FRANCISCO</speaker>
27:         <line>Nay, answer me: stand, and unfold yourself.</line>
28:       </speech>
29:       ...
30:     </scene>
31:     ...
32:   </act>
33: ...
34: </play>
```

Figure 2.1: An XML document containing the DTD.

and 10) contain character data, i.e., text (#PCDATA). Elements of type *line* (row 11) may contain text (#PCDATA) and elements of type *stagedir* mixed together. The operator “|” means a choice. Finally, rows 14–34 contain the element tree of the document instance.

A DTD includes no semantics — the processing orders may be, for example, specified in a stylesheet or coded into a document processing system. A DTD is compulsory only when processing SGML documents, since processing well-formed XML documents does not require a DTD.

As already mentioned, XML and SGML are metalanguages for defining markup languages. The structure used in a specific application is designed, for example, by the document manager, the application programmer, or the user. This means that any element names, which are represented in the document markup as tags, can be used. The *HyperText Markup Lan-*

guage (HTML) [W3C99b] is an SGML application in which the tag set is fixed. HTML is designed to be layout-oriented, whereas XML and SGML documents are typically content-oriented.

XML, originally intended for Web publishing, is a restricted form of SGML. Nowadays, XML is also used as a general data format, for example, in electronic commerce transactions, data communications, application programming interfaces, mathematical equations, and vector graphics. Major software companies are not only participating in developing XML but they are also developing new low-cost XML tools for the wider public as well as using it as a representation and storage format for software products. SGML, in turn, is a large and complex standard. For instance, SGML parsers are much more laborious to implement than XML parsers. For over a decade, SGML has been widely used in various areas, such as publishing, libraries, government administration, large-scale industry, and defense contractors.

In the next two sections we provide more details concerning XML and SGML technologies.

2.1.1 XML and its companions

The XML specification [W3C98c] was completed in early 1998 by the World Wide Web Consortium, also called W3C. Since then, a great deal of new related specifications and applications have been developed and published. Online versions of XML and its companion specifications can be found on the Web pages of the W3C[10]. Moreover, Robin Cover and OASIS[11], a non-profit international consortium, maintain "The SGML/XML Web Page", which is an online database containing organized reference information[12]. Well-organized information on XML can also be found, for example, from *XML*INFO Web pages[13].

By the time we started the research on intelligent document reuse in the SID project in 1995, there were no signs of XML nor its success in the publishing or software industry. Today, the new XML-related specifications offer advanced frameworks and methodologies for a broad variety of applications. Many of these new specifications are not yet finished; many technical details will change during the development work of the specifications which are not W3C recommendations. It is not at all clear which of these new specifications and approaches will win the approval of the industry and the

[10] http://www.w3.org/XML/
[11] The Organization for the Advancement of Structured Information Standards
[12] http://www.oasis-open.org/cover/
[13] http://www.xmlinfo.com/

public and hence will be supported by widespread and inexpensive software.

First, we introduce a few technical aspects of XML. Then, several XML-related specifications, which are noteworthy with respect to document assembly and the element-type classification method, will be introduced. For more detailed technical description of XML, see, for instance, the book by Charles Goldfarb and Paul Prescod [GP98a]. There are also many introductory and tutorial articles available on the Web[14], e.g., [Wal98].

Some design goals of XML were that XML should be usable over the Internet, it should support a wide variety of applications, and be compatible with SGML. It should also be easy to create XML documents and write programs processing them, while the specification of XML should be formal and concise.

An XML document has both a physical and a logical structure. Physically, the document consist of storage units called *entities*, which contain parsed or unparsed data. Parsed data contains both character data and markup. Unparsed data may or may not be text, and if text, may not be XML. Each unparsed entity has an associated notation, which is identified by name. The logical structure is composed of declarations, elements, comments, character references, and processing instructions indicated by explicit markup.

The *XML processor* reads XML documents and passes the data to an *application*. The XML specification defines how the processor should read XML data and what information it must provide to the application.

A document is *well-formed* if the following holds: the document meets all the well-formedness constraints given in the XML specification, each of the parsed entities referenced directly or indirectly within the document is well-formed, the document contains one or more elements, and the elements nest properly within each other. The nesting is proper, if there is exactly one *root* element, no part of which appears in the content of any other element. For other elements than the root the following rule applies: if the start-tag is in the content of another element, then the end-tag is in the content of the same element. As a consequence, for each non-root element C there is one other element P such that C is in the content of P, but is not in the content of any other element that is in the content of P. P is called *parent* of C, and C is called *child* of P.

A document is *valid*, if it has an associated document type declaration (e.g. Figure 2.1 rows 2–13) that contains or points to the DTD, and if the document obeys the DTD. Valid XML documents are also SGML documents.

[14] see http://www.oasis-open.org/cover/xmlIntro.html

W3C specification or proposal	Description
Resource Description Framework (RDF)	A framework for representing metadata
XML Namespaces	A mechanism providing unique names across the Internet
Document Object Model (DOM)	A programming interface for manipulating documents
Cascading Style Sheets (CSS)	A simple-to-use stylesheet language
Extensible Stylesheet Language (XSL)	A versatile stylesheet and document transformation language
XSL Transformations (XSLT)	The transformation language of XSL
XML Linking Language (XLink)	Constructs for links in XML resources
XML Pointer Language (XPointer)	A language for pointing into XML resources
XML Fragment Interchange	A mechanism for specifying and interchanging fragments
XML Schema	A schema specification language extending the properties of DTDs
Extensible Hypertext Markup Language (XHTML)	Reformulation of HTML as an XML application
Modularization of XHTML	A framework for creating hybrid document types
Synchronized Multimedia Integration Language (SMIL)	A language for creating interactive multimedia presentations

Table 2.1: Some XML-related recommendations and working drafts by the W3C.

In the following, we introduce some recommendations and working drafts of the W3C that are designed to accompany the use of XML in practice. Note that the content of a working draft may change substantially during the development, whereas a recommendation is stable. A short description of these specifications and proposals is given in Table 2.1.

The Resource Description Framework (RDF) [W3C99e, W3C99f] gives the means to define structural metadata for XML documents and to interchange the metadata between different applications on the Web. RDF can be used for various purposes, such as representing the structure of a thesaurus and other library classification schemes, describing the logical hierarchy of the pages in Web sitemaps, and describing the contents of a single Web page. Search engines may use this metainformation to improve the search results significantly. RDF can also be used by intelligent soft-

```
<h:html xmlns:h="http://www.w3.org/TR/REC-html40/"
        xmlns:n="http://www.newrider.net/"
        xmlns:r="http://www.ridelessons.com/">
  <h:head><h:title>New Rider</h:title></h:head>
  <h:body>
    <h:h1>Horse riding</h:h1>
    <n:h2>How to start</n:h2>
    <n:p>There are three important personal safety items you must
    consider: head protection, footwear, and body protectors...</n:p>
    ...
    <r:h2>Riding basics</r:h2>
    <r:p>Walk, trot, and canter are the three basic paces you will
    learn...</r:p>
    ...
```

Figure 2.2: An example of XML Namespaces.

ware agents in facilitating knowledge sharing and in describing intellectual property rights, privacy preferences, and privacy policies on the Web. One key application will be digital signatures for electronic commerce and collaboration.

RDF is defined in two parts: The first part, the RDF Model and Syntax Specification is already a W3C recommendation, but the second part, the RDF Schema Specification, is work in progress. No substantial changes are anticipated to the latter specification, however, so it is likely that there will be solid implementations of RDF in the near future. It remains to be seen if a wider public will endorse the capabilities RDF offers, because, in most cases, the attachment of RDF metadata is the responsibility of the creator of a Web page or site. For more information concerning RDF, see [DuC99, Mil98, Bra98].

XML Namespaces [W3C99d] provide a simple mechanism to give context to element and attribute names used in XML documents. An XML namespace is a collection of names, identified by a URI (Uniform Resource Identifier) reference [IET98a], which are used in XML documents as element types and attribute names. A namespace is declared using reserved attributes, the names of which must either be *xmlns* or have *xmlns:* as a prefix.

In the example of Figure 2.2, the namespace declaration is given in the attributes of the root element *h:html*. The namespace prefix "h" is associated with the namespace name "http://www.w3.org/TR/REC-html40/", the namespace prefix "n" is associated with the namespace name "http://www.newrider.net/", and the namespace prefix "r" is associated with the namespace name "http://www.ridelessons.com/".

The purpose of the XML Namespaces mechanism is to have unique names across the Internet. While processing documents from different sources, the elements and attributes can be identified by the unique names.

The Document Object Model (DOM) [W3C98b, W3C00b] is a platform-neutral and language-neutral interface that allows programs and scripts to access XML and HTML documents dynamically. DOM offers a standard internal representation of the document structure and a consistent programming interface for manipulating the components of the documents.

The *DOM level one*, already a W3C recommendation, contains the basic functions for document navigation and manipulation. In addition to language-independent specification, language bindings for Java and ECMAScript are provided. The *DOM level two* extends with, e.g., support for XML Namespaces, abstract views, generic stylesheets, Cascading Style Sheets, events, and traversing the document structure. It is at present a candidate recommendation, the content of which may change. The DOM level three is expected to address, e.g., DTDs, schemas and document validation. Further levels are expected to specify, e.g., an interface with the underlying window system and a query language interface. All these different level specifications compose the Document Object Model Specification.

Cascading Style Sheets (CSS) [W3C99a, W3C98a] is a simple-to-use language with which authors and users may attach style to XML and HTML documents. With *CSS level 1* the user can design the appearance and layout of documents, e.g., control the color, font, borders, spacing, lining, and indenting of document building blocks. *CSS level 2* extends with media-specific stylesheets, content positioning, downloadable fonts, table layout, features for internationalization, automatic counters and numbering, and properties related to user interfaces. A fundamental feature of CSS is that the presentation can be influenced by more than one stylesheet. For instance, the author can attach a preferred stylesheet, while the user may attach a personal stylesheet to adjust the specifications of the author.

The Extensible Stylesheet Language (XSL) [W3C00c] is designed for XML documents. It contains a transformation language called **XSL Transformations (XSLT)** [W3C99j] and an XML vocabulary for specifying formatting semantics. An XSL stylesheet specifies the layout of a class of documents by describing the transformation of a document instance into a document that uses the XML formatting vocabulary. An XML document accompanied with an XSL stylesheet is passed to an *XSL stylesheet processor*, which produces the presentation desired. Firstly, the processor constructs a result tree from the XML source tree by applying *template* *rules* on the source tree. Secondly, it interprets the result tree and produces a formatted presentation on a display, on paper, in speech or onto other media. The former subprocess is called *tree transformation*, while the latter is called *formatting*. The nodes of the result tree are *formatting objects* that are typographic abstractions such as page, paragraph, and rule. A set of *formatting properties*, like indents, spacing, and window, orphan, and hyphenation control, provide an exact control over the presentation of formatting objects.

The XSL working group was formed in the beginning of year 1998, and it published the current working draft of XSL specification in March 2000. The work is based on the prior work on Cascading Style Sheets and on the Document Style Semantics and Specification Language (DSSSL) [ISO96], which will be introduced in Section 2.1.2. Most of the formatting properties of CSS are also in XSL, some of them are extended, and some properties are available only in XSL. DSSSL was also used as a source of inspiration in many property extensions and in designing the processing model and syntax of XSL.

In CSS the result tree is almost the same as the source tree, but in XSL the trees can be radically different. In other words, CSS has limited facilities to attach some text and images to the result tree, while XSL offers powerful transformation capabilities to rearrange the components of the source tree, to collate and sort selected components, and to attach information on the result tree. On the other hand, mature implementations of CSS are available, while XSL does not yet have mature browser and content-authoring support, because the XSL specification is currently under development. However, CSS is designed for dynamic formatting of online HTML or XML documents, while XSL is intended for extensive formatting of highly-structured and data-rich XML documents. In addition, XSL and CSS can be used together. XSL can be used, for example, on the Web server to transform XML data into HTML documents with CSS stylesheets. For more details on using CSS and XSL together, see [W3C98d].

Recently, there has been a debate whether CSS and DOM together are better than XSL; see, e.g., [Wal99a, Hol99, Lev99, Wal99b]. One side of the coin is that XSL offers facilities needed for versatile publishing and document reuse: the same content can be transformed into different purposes and media using XSL. Another side of the coin is that CSS and DOM together will do the same things in a different manner, because DOM offers the programming facilities needed to manipulate the content and structure of documents which are lacking from CSS. However, all of the three technologies are still developing, and both approaches — CSS with DOM and XSL — have their own advantages and advocates as well as their own applications. It remains to be seen, how software industry, developers, programmers and ordinary users will choose.

Robin Cover's XSL page[15] is a good starting point for more information on XSL. A recent article on the use of XSL can be found from the PC Magazine [Ran99].

The XML Linking Language (XLink) [W3C00g] specifies constructs for representing links in XML documents. A link is defined to be a relationship between resources or portions of resources encoded as an XLink linking element. XLink provides mechanisms which extend the basic ID/IDREF linking inside a document instance defined in the XML specification. XLink provides a framework for asserting linking relationships among more than two resources, associating metadata with a link, and creating link databases that reside in a location separate from the linked resources. The design of XLink was influenced by HTML and HyTime [ISO92, ISO97], which are introduced in Section 2.1.2, and by the Text Encoding Initiative (TEI) Guidelines [TEI94].

XLink uses the **XML Pointer Language (XPointer)** [W3C99i] for addressing the internal structures of XML documents. XPointer, in turn, builds upon the **XML Path Language (XPath)** [W3C99h], which is also used by XSLT, the transformation language of XSL introduced earlier in this section. In fact, XPath was developed to satisfy the needs of XPointer and XSLT, both of which extend the features of XPath to their own purposes. James Clark, who is the editor of XSLT specification and co-editor of XPath specification, has implemented a freeware prototype version of XSLT according to the specification[16].

The XML Fragment Interchange [W3C99g] specification provides means to specify smaller logical units than a document in order to process only the desired parts of the entire document, since the parts alone may not contain enough information to be parsed correctly. This specification defines the structure units which may constitute fragments, the XML notation for describing the information needed to parse, view, and edit fragments, and finally, some mechanisms for associating the information with fragments.

XML Schema [W3C00h, W3C00i] is a schema definition language defined in two parts: Structures and Datatypes. The idea is to offer more sophisticated mechanisms than a DTD offers to represent the structure definition. The first part Structures specifies the XML Schema definition language that offers facilities for describing the structure and constraining

[15]http://www.oasis-open.org/cover/xsl.html
[16]http://www.jclark.com/xml/xt.html

the contents of XML documents. The second part Datatypes specifies a language for defining datatypes. XML Schema provides a superset of the capabilities found in XML DTDs.

The Extensible Hypertext Markup Language (XHTML) [W3C00f] is a reformulation of HTML as an XML application. This means that XHTML 1.0 defines three XML DTDs which correspond to the three SGML DTDs defined in HTML 4. In addition to the XHTML 1.0 recommendation, there is a working draft called **Modularization of XHTML** [W3C00d], which is a decomposition of XHTML 1.0 into a collection of abstract modules that provide specific types of functionality. The mechanism for defining the abstract modules is provided in **Building XHTML Modules** [W3C00a]. The modules may be combined with each other and with other modules to create XHTML subset and extension document types that qualify as members of the XHTML family of document types.

The primary purpose of the modularization framework is to give means to combine elements from multiple abstract modules into a hybrid document type, to develop documents conforming to the hybrid document type, and to validate documents against the associated hybrid document type definition. A goal is to ease the development of document types based upon XHTML. The document types may extend XHTML by integrating additional capabilities, or they may define a subset of XHTML for use in a specialized device. However, the modularization framework does not solve the problem of validation of XML documents with different schemas from multiple XML Namespaces in different portions of the document, but it defines methods for creating hybrid DTDs from a set of abstract modules.

This kind of reformulation and modularization of HTML is urgently needed, because content providers need to deliver the content on a considerable number and diversity of platforms. Currently there is activity to move HTML onto, for instance, mobile devices, like handheld computers and portable phones, and television devices, like digital television and tv-based Web browsers. Each device has different requirements and constraints. The modularization framework gives means to specify element types supported by a device and the element types actually used in a specific application.

The Synchronized Multimedia Integration Language (SMIL) [W3C00e] is a simple XML-based language for creating interactive multimedia presentations such as training courses on the Web. With SMIL the temporal behavior, hyperlinks, and layout of the presentation can be described and controlled. SMIL is designed to be usable in other XML-based languages such as XHTML, when timing and synchronization is needed.

All the XML-related specifications presented so far are made by the W3C. Many of them are in a developing stage and very likely to be revised and changed. Only XML 1.0, the first part of RDF, XML Namespaces, DOM level 1, CSS levels 1 and 2, XSLT, XPath, and XHTML 1.0 are W3C recommendations and therefore stable.

Table 2.2 lists some XML applications that other organizations than the W3C have proposed. The current development is very rapid and the applications are both diverse and numerous. However, many of these applications have already been under development for some years. The purpose of the listing is to give a taste of the variety of XML applications under development. A comprehensive listing including the presented applications can be found from Robin Cover's XML reference pages[17].

2.1.2 SGML and its companions

SGML became an ISO standard as early as 1986 [ISO86]. Since then, it has been one of the most popular ISO standards, and has been widely used, for instance, in the publishing industry; in organizations producing large amounts of texts, like government administration; in electronic archiving of old literature, and in digital libraries. SGML is also adopted by industry producing machinery and technical devices, where it is used for technical documentation of products, like airplanes, trains, elevators, paper machines, computers, and cellular phones. Hence, there have been sophisticated systems processing SGML documents for over a decade. But the evolution of the Internet and the recent evolution of XML is bringing the ideas of SGML into the hands of ordinary users all over the world.

This section mainly introduces some important companions of SGML, namely HyTime, DSSSL and HTML. A referenced and appreciated tutorial to the technical basics of SGML can be found from a chapter of the TEI Guidelines [TEI94] called "A Gentle Introduction to SGML". A comprehensive information source is a book by Charles Goldfarb [Gol90], who is often referred to as an inventor of SGML. Though XML is an extensive subset of SGML, there are some differences, which affect both document instances and DTDs. To help the users of SGML in adapting XML, James Clark has listed some technical differences between SGML and XML [W3C97].

HyTime [ISO92, ISO97] is a very large and complex ISO standard providing multimedia properties for SGML documents. It offers facilities to create hyperlinks within and between documents and other information objects as well as mechanisms for scheduling multimedia information in

[17] http://www.oasis-open.org/cover/xml.html

XML application	**Purpose**
XML Metadata Interchange Format (XMI)	exchanging programming data created with object technology; by the Object Management Group (OMG)
Open eBook specification	representing electronic books; by the Open eBook Initiative
WAP Wireless Markup Language (WML)	specifying content and user interface for narrowband devices such as cellular phone; by the WAP Forum
Predictive Model Markup Language (PMML)	describing models used by data mining and modeling community
Chemical Markup Language (CML)	managing molecular information
Bioinformatic Sequence Markup Language (BSML)	encoding and display of DNA, RNA and protein sequence information
Weather Observation Markup Format (OMF)	encoding weather observation reports
Open Financial Exchange (OFX/ OFE)	online transfer of financial data
Bank Internet Payment System (BIPS)	sending payment instructions to banks safely over the Internet
Music Markup Language (MML)	encoding musical content
Astronomical Instrument Markup Language (AIML)	commanding and controlling astronomical instruments; by NASA and Century Computing
News Industry Text Format (NITF)	interchange of news information
Real Estate Transaction Markup Language (RETML)	exchange of real estate transaction information

Table 2.2: XML applications from different areas.

time and space. HyTime also introduces the concept of architectural forms, which give means to define metastructures for SGML documents. Moreover, the HyTime metastructures for creating SGML documents with multimedia properties are defined as architectural forms.

The Document Style Semantics and Specification Language (DSSSL) [ISO96] provides methods for defining the layout of SGML documents and the transformation capabilities needed for processing them. Like HyTime, it is also a very large and complex ISO standard. The DSSSL language consists of two main components: a style language for describing the layout of documents and a transformation language for transforming the structure of documents. DSSSL also contains a query language called Structured Document Query Language (SDQL), which it shares with the

HyTime standard, and an expression language closely modeled on a programming language called Scheme.

The HyperText Markup Language (HTML) [W3C99b] was originally designed to be a layout oriented language with a flat structure tree for simple Web pages. In the latest version of HTML, which is version 4.01, there are three different flavors: *transitional*, *strict* and *frameset*. Transitional HTML supports features of the older versions 2.0 and 3.2 as well as new features such as stylesheets. Strict HTML offers structural markup free from tags associated with layout, and should be used with Cascading Style Sheets. Frameset HTML provides mechanisms to partition the browser window into two or more frames. Each flavor has its own SGML DTD.

2.2 Related document management research

In the document assembly process, structure transformations are needed when saving the chosen fragments of arbitrary structure into one resulting assembly document. Structured document transformation is a well-studied area in document management; see, e.g., [Kui96, Lin97] for further details and references to relevant work. A novel approach introduces a powerful formalism for both document and schema transformation with patterns and contextual conditions [Mur97b, Mur97a, Mur98]. The approach is based on forest-regular language theory. One of the motivations of this work is the possibility to apply the transformation model for document assembly, since the model also allows schema transformation.

Reuse of existing documents has been studied in [Lev93]. Dynamic reuse of different data sources, such as HTML, XML, SGML, and object-oriented database sources, has been studied in [PVH98, VP98]. Their approach is to use a specific middleware format to integrate query results from various resources and to map the results into a new virtual document.

In [Aho96] DTDs are generated for tagged text lacking structure schema by using grammatical inference methods. Document structure analysis is reported in [KF97], which presents an approach to extract the logical structure of flat text or documents structured according to proprietary formats by transforming these documents into SGML documents. Expressive Description Logic is used to represent SGML and XML documents thus providing a general reasoning mechanism for inferring information concerning DTSs, such as the verification of equivalences between DTDs [CGL97, CGL99]. In order to improve the retrieval of most relevant fragments of structured documents, a modeling of uncertainty with respect to structural components of a document is presented in [Lal97].

2.3 Digital libraries

A digital library contains collections of digital material that can be searched and browsed with the assist of software. Association of Research Libraries has defined the term digital library [ARL95]:

> The digital library is not a single entity. The digital library requires technology to link the resources of many. The linkages between the many digital libraries and information services are transparent to the end users. Universal access to digital libraries and information services is a goal. Digital library collections are not limited to document surrogates: they extend to digital artifacts that cannot be represented or distributed in printed formats.

An international research agenda for digital libraries [NSF98] summarizes major future research problems in the field. Many of the problems are related to research problems in the document management field. Moreover, it is suggested in [Ren97] that the digital library community should pay more attention to structured document research and applications, and the investigation of documents should be in the core of the research agenda of digital libraries. Indeed, the recent development in the document management field, like the fast evolution of XML technology, will bring new techniques and mechanisms usable in building future digital libraries.

However, SGML has been widely applied in building existing digital libraries [PK97, PW97, Che97, HY97, SB97, DLF98, Obe99]. The application of HyTime and DSSSL along with SGML is reported in [RvOHB98]. The TULIP [Els96] research project studied the use of SGML in electronic publishing of scholarly journals together with Elsevier Science. The legacy of the TULIP project is investigated in the PEAK experiment [BLMMR99a, BLMMR99b]. Furthermore, some recent projects [Sos99, TBSN99] have investigated the use of XML.

The extraction of a set of multimedia documents from a digital library and their presentation as a single hypermedia document is investigated in [DH95]. This approach is closely related to our approach to document assembly. Our approach differs, e.g., in the employment of generic XML and SGML structures in order to allow the assembly result to be further transformed to the final delivery format, be it digital or hard-copy.

It is also suggested that (object-oriented) database techniques should be used in building future digital libraries, since database research provides many applicable solutions to the organization and access of massive volumes of distributed data [Abi97a]. The problems caused by the heterogeneous nature of digital library sources are considered in [PCGMW98, CGM98, CGM99]. For instance, the Stanford Digital Library project [BCGP97, PBC$^+$99] has studied the interoperability among heterogeneous digital library services.

Currently, huge amounts of large text collections marked up with HTML, XML or SGML are available for free on the Web. For instance, the link page of the University of Pennsylvania[18] and the ATHENA reference page of the University of Geneva[19] are good starting points to electronic text archives. The reference page of D-Lib Magazine[20] contains a wealth of up-to-date pointers to Web sites concerning digital library research.

The Dublin Core (DC) metadata element set for descriptive metadata is proposed by the Dublin Core initiative [WL97, WH98, IET98b, BMR$^+$99, Wei99, DCM99]. The DC metadata is especially valuable in finding information from electronic resources, and may be used, for instance, to improve information retrieval over the Web. The Dublin Core has been described as a metadata for the masses, so it should be acceptable to a broad variety of communities, like librarians, content experts, digital library researchers, computer networking specialists and text-markup experts.

The DC element set contains 15 elements, which describe the content, intellectual property, and instantiation of the resource. *Title*, *Subject*, *Description*, *Source*, *Language*, *Relation*, and *Coverage* are content related DC elements, *Creator*, *Publisher*, *Contributor*, and *Rights* are intellectual property related, and finally, *Date*, *Type*, *Format*, and *Identifier* are instantiation related elements.

RDF, already presented in Section 2.1.1, is an architecture for representing metadata in XML documents. The Dublin Core initiative and the RDF working groups share a number of members, and the evolution of DC has affected the development of RDF and vice versa. Furthermore, the DC community is in the front line applying RDF technology.

2.4 Databases and documents

In the database community the storage of structured documents in databases has been studied. One of the key issues is the ability to query documents on structure and text content. Also, database techniques offer

[18]http://digital.library.upenn.edu/books/
[19]http://un2sg4.unige.ch/athena/html/booksite.html
[20]http://www.dlib.org/reference.html

powerful facilities for document management. In the following, we present some studies on the area.

The translation from structured text files to object-oriented databases (OODBS) is studied in [ACM93, ACM98]: The context-free grammar describing the text structure is exploited in constructing the database representation. When answering a query, it is possible to compose a database view of the file without having to load the entire file into the database. The updates specified at the database level can be translated into updates of the file. In addition to mapping SGML documents into OODBs, extensions of OODB query languages for the retrieval of SGML texts are studied in [CACS94, ACC$^+$97]. With the query extensions, the data can be queried without exact knowledge of the schema. Database queries are also used to navigate through hyperlinks [CR94].

An alternative approach to store and query structured documents containing hyperlinks is provided in [BANY97], which introduces the HyperStorM database application framework. A model to query document databases both by structure and by content is presented in [NBY97]. It is shown that this model is expressive and efficiently implementable. However, query language for end-users and query optimization remains to be developed. Constraining grammars, which are specified by adding boolean conditions to the productions of a grammar, extend the modelling capability of simple context-free grammars [ST99]. Constraining grammars can be used in specifying data definition and data manipulation operations for a text database. In this framework, the semantics of the operations may be precisely defined.

2.5 Semistructured data

Our approach concerning generalized document structures and document assembly is related to the research of semistructured data management [Abi97b, Bun97, Suc97]. However, the research problems are different. In semistructured data research, the emphasis is on the database issues, the goal is to handle also implicitly structured non-XML documents, and XML is mainly considered as a data representation and exchange format. We consider XML as a document representation format. The element-type classification accepts only XML or SGML documents whose element names, by definition, do not include any semantics.

Some representative research issues include:

- extracting implicit structures in semistructured data collection [NAM97, NAM98];

- discovering typical structures of similarly structured documents using data-mining techniques [WL98];
- extracting structure of HTML pages and text files semi-automatically [Ade98];
- discovering coherent fragments in Web documents using heuristic approach[EJN99];
- providing type checking and type inference for queries on semistructured data [MS99];
- forming queries that access the views of semistructured data [PV99a];
- integrating semistructured and structured data [BCV99, LAW99];
- presenting Knowledge Representation (KR) formalisms for modeling semistructured data [CGL98].

Documents conforming to different DTDs can be flexibly stored and maintained in a document management system by using object-oriented semantic networks [CDF97]. For storing semistructured data one can use a relational database management system, in which the mapping between semistructured data model and relational data model is done automatically using data-mining techniques [DFS99]. If a DTD is available, it is exploited to improve performance.

Modeling and querying the World Wide Web. Application of database techniques for managing and querying the huge volumes of data on the Web has been a popular research topic. A comprehensive survey [FLM98] divides the information management tasks into three categories: modeling and querying the Web, information extraction and integration, and Web site construction and restructuring. The survey contains a representative list of references to work in these three areas.

A recent technique, XML views, facilitates the processing of XML data in the Web context [Abi99]. XML views offer ways to transform and restructure data from heterogeneous sources. The views are considered with respect to data modeling, queries, objects, combining semistructured and structured data, updates, data distribution, and incomplete information. Views and data integration over the Web are also studied in [CGLV99].

In the end of year 1998, the W3C organized a workshop on query languages [Rei99]. There were 66 position papers[21] considering the issue of an XML query language. The participants represented companies as well as

[21]see http://www.w3.org/TandS/QL/QL98/pp.html

universities, and they came from different communities — the workshop was clearly one of the first steps towards an XML query standard. For recent articles on an XML query language, see [DFF+99, MW99, GMW99]. The new research opportunities on data management inspired and enabled by the emerging XML technology are outlined in [Wid99].

Integration of heterogeneous information sources. The goal in this area is to provide a framework and tools for users to process heterogeneous data of diverse and dynamic information sources; see, e.g., [PGMW95, PAGM96, ACM97, GMPQ+97, VP97]. Generally, the data is unstructured or semistructured, and the integration needs human interaction. The integration system typically provides a model for exchanging data objects, tools for querying sources, and mediator programs that collect information from sources, process the information, and export it to applications. For a survey classifying existing approaches to query heterogeneous data sources, see [DD99].

Queries over heterogeneous databases on the Internet are studied in [AGMPY98, GP98b]. Recent articles present an adaptive data integration system [IFF+99], a schema mapping language that can be used to specify views to heterogeneous data sources [HST99], and a web-based architecture and user interface for processing large distributed files resulting from scientific simulation [PWDN99]. A novel approach uses XML in information mediation [PV99b, BGL+99, LPV99]. An XML view of the heterogeneous XML data is provided by inferring the view DTD from the view definition and the source DTDs. The mediator administrator customizes the view to the user needs by using the query and view definition languages of the mediator.

Chapter 3

Document assembly using extracted metastructures

This chapter presents a document assembly framework and a method called element-type classification that is used to extract metastructures of typical electronic documents. The basic concepts and ideas of document assembly and element-type classification are introduced in Section 3.1 with several practical examples. In Section 3.2 we present the element-type classification method and the generic schema defined as an XML DTD, which can be seen as a metastructure definition. In Section 3.3 we describe the document assembly process.

3.1 Basic ideas of assembly and element-type classification

The number of heterogeneously structured XML documents will increase on the Web. Already, many organizations have made part of their proprietary document collections publicly available. Many organized digital libraries reach masses of users via the Web. However, when moving from proprietary document collections to widely-available digital collections, the problem of *heterogeneous document structures* will be faced. It cannot be assumed that the user knows how the documents are structured and what the exact names of the elements are. When several document collections are reachable at the same time, the natural desire for uniform and standardized processing also arises. For instance, there is a need for new techniques that allow versatile processing such as *structured queries* and *structured formatting.*

XML and SGML give means for defining logical structures of documents. Naming of structures is not unique: document managers give differ-

```
Collection A:
  <article>
    <person>Hermann Hesse</person>
    <biography>Hermann Hesse was born 1877 in Calw
    and died 1962 in Montagnola. He was awarded the
    Nobel Prize for Literature in 1946.</biography>
    <production>
      <work>Rosshalde (1914)</work>
      <work>Siddhartha (1922)</work>
      <work>Steppenwolf (1927)</work>
      <work>The Glass Bead Game (1945)</work>
      ...
    </production>
  </article>

Collection B:
  <poem>
    <author>Hermann Hesse</author>
    <book>The Glass Bead Game</book>
    <ptitle>Steps</ptitle>
    <stanza>
      <poemline>As every blossom fades</poemline>
      <poemline>and all youth sinks into old age,</poemline>
      <poemline>so every life's design,</poemline>
      <poemline>each flower of wisdom,</poemline>
      <poemline>every good attains its prime</poemline>
      <poemline>and cannot last forever.</poemline>...
    </stanza>...
  </poem>
```

Figure 3.1: Excerpts from the collections A and B.

ent names for common structures, and they typically use their own language in naming. For instance, in Finnish a paragraph object might be named as *kappale* and in English *para* or *p*. Thus, there is a clear need to separate logical structures from the names chosen for them.

Assume that there are two document collections A and B, which are structured using XML (or SGML). The collection A contains articles of authors, whereas the collection B contains samples of their texts. Figure 3.1 presents two excerpts from the collections.

If the user would like to combine all the information regarding Hermann Hesse, a query like the following would have to be stated to A and B.

> **select** *article* elements **from** A containing *person* element
> **where** the content of the *person* element contains "Hermann Hesse"
> **union**

> **select** *poem* elements **from** B containing *author* element
> **where** the content of the *author* element contains "Hermann Hesse"

As can be seen from this query, the user has to know exactly what the names of the structure elements are and how the elements in the structure hierarchy are located.

Even if the user gives a simple string query, e.g., "Hesse and Hermann", another problem arises. The system finds elements *person*, *biography*, and *author*, which contain the words in the query, but what should it return as a result? Should it return these elements only, or some surrounding elements in addition, or maybe the largest element containing these elements? If the size of the elements in the collection varies from short strings to books or book collections, it is not at all obvious which elements should be returned. Some choices, such as returning the whole book collection, would certainly be rather annoying for the user.

Assuming that a result can be returned, there is still a problem: how should the result be presented to the user? An XML or SGML document itself does not contain any formatting information. A collection may contain rendering mechanisms for formatting entire documents, but they are probably not prepared to handle fragments of documents. Moreover, different collections certainly will have different styling principles.

As more and more digital collections become available, a natural desire also arises: the user would like to search for pieces of information in several collections and construct new documents reusing existing document fragments. This kind of use, which is called *document assembly*, fits the general trend in which the nature of documents is changing from static, clearly defined artifacts to dynamic documents that are created using some existing building blocks. Document assembly is seen as an interactive process where a user, with the help of tools, searches for appropriate sources and then forms the intended document.

When the desired fragments are combined into one document, a new problem arises: the new document does not necessarily conform to any Document Type Definition (DTD). Especially, when concerning SGML documents, a DTD is an inherent part of the document. Many SGML tools, such as editors, formatters, and database engines, require a DTD for processing purposes. Even though a well-formed XML document does not need to have a DTD, generally it is useful to have structure definitions for processing and querying.

As an example, consider the assembled document of Figure 3.2, which is extracted from the excerpts of Figure 3.1. The poem "Steps" from the book "The Glass Bead Game" is added into an article describing the work of

```
<article>
  <person>Hermann Hesse</person>
  <biography>Hermann Hesse was born...</biography>
  <production>
    <work>Rosshalde (1914)</work>
    <work>Siddhartha (1922)</work>
    <work>Steppenwolf (1927)</work>
    <work>The Glass Bead Game (1945)</work>
    ...
  </production>
  <poem>
    <book>The Glass Bead Game</book>
    <ptitle>Steps</ptitle>
    <stanza>
      <poemline>As every blossom fades</poemline>
      <poemline>and all youth sinks...</poemline>...
    </stanza>
  </poem>
</article>
```

Figure 3.2: A document assembled from the collections A and B.

Hermann Hesse. The element *author* in *poem* is left out, since the element *person* in *article* contains the same information.

Assume that the XML DTD of the collection A defines the element *article* and the XML DTD of the collection B defines the element *poem* as follows.

```
Collection A:
  <!ELEMENT article (person, biography, production)>

Collection B:
  <!ELEMENT poem (author, book, ptitle, stanza)>
```

The definition of the element type *article* tells that it has to contain elements *person*, *biography*, and *production*, while the definition of the element type *poem* tells that it has to contain elements *author*, *book*, *ptitle*, and *stanza*. This means, for instance, that the element *poem* cannot be included within the element *article*. The new assembly is not valid according to either of the XML DTDs.

Solutions for managing and manipulating heterogeneous document structures will be presented in this chapter. The approach is based on the fact that although no semantics can be inferred from the names of XML or SGML elements, some knowledge can be extracted from the elements: which other elements an element can contain, and how long the text content of

```
Collection A:
  <ParaCont>
    <Title>Hermann Hesse</Title>
    <Para>Hermann Hesse was born 1877 in Calw
    and died 1962 in Montagnola. He was awarded the
    Nobel Prize for Literature in 1946.</Para>
    <StrCont>
      <String>Rosshalde (1914)</String>
      <String>Siddhartha (1922)</String>
      <String>Steppenwolf (1927)</String>
      <String>The Glass Bead Game (1945)</String>
      ...
    </StrCont>
  </ParaCont>

Collection B:
  <StrGrp>
    <Title>Hermann Hesse</Title>
    <String>The Glass Bead Game</String>
    <String>Steps</String>
    <StrCont>
      <String>As every blossom fades</String>
      <String>and all youth sinks into old age,</String>
      <String>so every life's design,</String>
      <String>each flower of wisdom,</String>
      <String>every good attains its prime</String>
      <String>and cannot last forever.</String>...
    </StrCont>...
  </StrGrp>
```

Figure 3.3: The original element names in *article* and *poem* are replaced with the names of the generic classes.

the element is. This information is extracted from the document instances, which may be the only source available. Various components, like strings, paragraphs and sections, as well as various structures, like element hierarchies and element containers wrapping logical components together, can be recognized. Using the extracted knowledge each element type is mapped to some predefined element type called *generic class*. This mapping process is called *element-type classification*.

Figure 3.3 presents the elements *article* and *poem* of Figure 3.1, when the original element names are replaced with the names of the generic classes. The mapped generic classes are *ParaCont* (paragraph container), *Title*, *Para* (paragraph), *StrCont* (string container), *String*, and *StrGrp* (string group).

When the generic class for each element type is known, the documents can be queried uniformly. The following simple query could be stated to collections A and B after knowing the generic classes for each element type in the collections.

> **select** elements containing *Title* element
> **where** the content is "Hermann Hesse"

Each generic element has an element type definition, which describes the structure of the element type in terms of other generic classes. The definitions of the generic classes constitute the generic DTD. It will be shown in Chapter 4 that all the document instances, where the generic element names produced by the element-type classification are used instead of the original ones, obey the generic DTD. Now one can design a stylesheet that enables browsing and printing any XML or SGML document according to the generic structure. As the classification may map several original element types into one generic class, part of the structure can be lost, and hence some fine distinctions may not be visible in the resulting documents. However, the classification guarantees that the result is clear and readable.

Basic needs in document processing are browsing and printing. If the elements of document instances are classified and if the generic element names are used instead of the original ones, the document instances can be formatted by using the formatting instructions of the generic classes. This enables browsing and printing of heterogeneously structured XML documents from different sources even if the appropriate formatting instructions are not available.

For instance, the following XSL [W3C00c] formatting instructions define that the font size is 12pt in text paragraphs and 20pt in titles, and the titles are printed in bold. The generic classes *Para* and *Title* are attached to the XSL flow object (fo) *block*.

```
<xsl:template match="Para">
  <fo:block font-size="12pt">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

<xsl:template match="Title">
  <fo:block font-size="20pt" font-weight="bold">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

Another application is constructing *automatic stylesheet generators* for different DTDs. If the generic classes of the original element types are known, the formatting instructions designed for the generic classes can be used in the generation of a stylesheet template for the original structure. If necessary, the generated stylesheet can be tuned by the user. Automatic stylesheet generation is useful especially if the DTDs are large and updated often.

For example, the element *article* in Figure 3.1 contains the element *biography*. Because *biography* is mapped to *Para*, the XSL formatting rules of the *article* structure could contain the following rule. As can be seen, the XSL formatting instruction of *Para* is copied, and the target-element type is set to *biography*.

```
<xsl:template match="biography">
  <fo:block font-size="12pt">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>
```

The element-type classification facilitates document assembly in various ways. In addition to the benefits for searching the interesting fragments, valid XML or SGML documents can be constructed, because the new documents obey the generic DTD. Although document assembly is considered as a user-driven process, the meta knowledge gathered from the elements makes it possible to design automatic tools for including some document fragments in the assembly. For instance, if an element is identified as a cross-reference, then the target of the reference can be included.

The element-type classification can also be used to aid general *document transformations*. Instead of writing specific translators between all the different document types, one only needs to write them between each type and the generic type. Thus, the original document instances are transformed into the generic structure, which is transformed into the desired final structure.

3.2 Extracting document metastructures

3.2.1 Overview

In this section we present a method for extracting metastructures that can be used to view any XML (or SGML) document as belonging to a predefined generic document type. When given an arbitrary document type consisting of the set of document instances and the DTD, each element type, which is in the DTD and has element instances in the document instances, is mapped to one element type of the *generic XML DTD* shown in Figure 3.4.

```
<!ENTITY % sec      "Section|SectGrp|StrSect">
<!ENTITY % par      "Para|ParaCont|ParaGrp">
<!ENTITY % sml      "meta|Empty|Ref|Formula|FormCont|math|Fig|FigCont|
                     String|Title|StrCont|StrGrp|Unknown">
<!ENTITY % sml2     "meta|Empty|Ref|Fig|FigCont|String|Title|StrCont|
                     StrGrp|Unknown">
<!ENTITY % sml3     "meta|Empty|Ref|String|Title|StrCont|StrGrp|Unknown">

<!ELEMENT VirtDoc  ((#PCDATA)|%sml;|%par;|%sec;|table)*>
<!ELEMENT Unknown  ANY>

<!ELEMENT StrSect  (%sml;|StrSect|table)*>
<!ELEMENT SectGrp  (%sml;|%sec;|table)*>
<!ELEMENT Section  ((#PCDATA)|%sml;|%par;|%sec;|table)*>

<!ELEMENT ParaGrp  (%sml;|ParaCont|ParaGrp|table)*>
<!ELEMENT ParaCont (%sml;|%par;|table)*>
<!ELEMENT Para     ((#PCDATA)|%sml;|%par;|table)*>

<!ELEMENT StrGrp   (%sml3;|table)*>
<!ELEMENT StrCont  (meta|Empty|Ref|String|Title|Unknown)*>
<!ELEMENT String   ((#PCDATA)|%sml;|%par;|table)*>
<!ELEMENT Title    ((#PCDATA)|%sml;|%par;|table)*>

<!ELEMENT FigCont  (%sml2;|table)*>
<!ELEMENT Fig      EMPTY>
<!ELEMENT FormCont (%sml;|table)*>
<!ELEMENT Formula  EMPTY>

<!ELEMENT Ref      EMPTY>
<!ELEMENT Empty    EMPTY>

<!ELEMENT meta     EMPTY>
<ENTITY % tableX PUBLIC "-//W3C//DTD XHTML Table Model//EN">
<ENTITY % mathML PUBLIC "-//W3C//DTD Mathematical Markup Language//EN">
%tableX; %mathML;
```

Figure 3.4: Definition of the generic XML DTD.

This process is called *element-type classification*, and the element types in the generic DTD are called *generic classes* [AHH+98b]. If the DTD is not available in the case of well-formed XML documents, the names of the element types to be processed are collected from the element names in the document instances.

The basic elements of the generic hierarchy are sections (*Section*), paragraphs (*Para*) and strings (*String*). Typically, strings are short sequences of characters, text paragraphs consist of a certain amount of strings, and sections in turn consist of paragraphs and other sections. Some container elements without any text content of their own are also needed to wrap a logical unit together. For example, a typical list structure in a document consists of items, which consist of one or several paragraphs. In this case, the generic container structure would be a paragraph group (*ParaGrp*) consisting of paragraph containers (*ParaCont*), which in turn consist of one or several paragraphs. The corresponding container structures for strings are a string group (*StrGrp*) and a string container (*StrCont*). Furthermore, there are two different types of containers wrapping sections, namely section groups (*SectGrp*) and string sections (*StrSect*). Finally, generic representations for special objects in documents are needed: figures (*Fig*), formulas (*Formula* and *math*), tables (*table*), references (*Ref*), titles (*Title*), and a representation for metadata (*meta*) such as the Dublin Core metadata.

In the following, we introduce the notions used in the generic XML DTD. The operator "|" means a choice, and the operator "*" means that the operand may occur zero or more times. For instance, $(a|b)*$ means that a or b may occur zero or more times, which results in the following different possibilities: $\emptyset$, a, b, aa, ab, ba, bb, aaa, aab, aba, baa, ... In the beginning of the generic DTD there are five parameter entities, "sec", "par", "sml", "sml2", and "sml3". For example, the value of "sec" is the string "Section|SectGrp|StrSect" — every occurrence of "%sec;" will be replaced with "Section|SectGrp|StrSect". The root element type is *VirtDoc*. The content model of *VirtDoc* defines that it may contain text (#PCDATA), element types "sml" (which is "meta|Empty|Ref|Formula|..."), *StrGrp*, "par", "sec" and "table". In fact, *VirtDoc* may contain every element type of the generic DTD except *VirtDoc* itself. The content model "ANY" in the element type *Unknown* allows it to contain any element types of the DTD in an arbitrary order. The last two entities in the generic DTD are references to other public DTDs, where the element types *table* and *math* are defined.

The basic idea of the element-type classification is the following: In the beginning of the classification process information is collected from the document instances of a document type. For each element type, which is in the DTD and has element instances in the document instances, the following information is extracted: the names of the elements that the element instances contain directly, the average text length of all the element instances, and finally, the average text length of the subtrees of all the element instances. This information is then used to decide which generic class would be closest to the original element type.

We define that the *text length of an element instance* is the number of characters the element instance contains directly, while the *text length of the subtree of an element instance* is the text length of the element instance plus the number of characters contained in the element instances in the subtree. As an example, see Figure 3.5, where the element instance *Para*

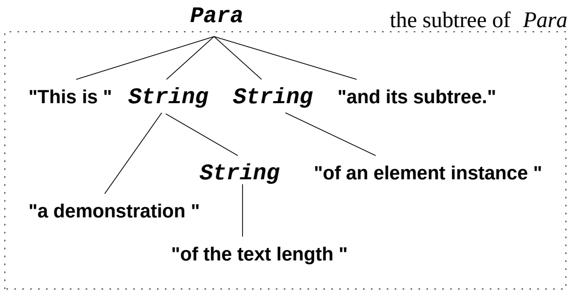


Figure 3.5: The structure tree of the element *Para*.

has four children: two character sequences “This is ” (8 characters) and “and its subtree.” (16 characters), and two *String* elements. Hence, the text length of the instance *Para* is 24 characters. The text length of the subtree of the instance *Para* is the length of the five sequences “This is ” (8 characters), “a demonstration ” (16 characters), “of the text length ” (19 characters), “of an element instance ” (23 characters), and “and its subtree.” (16 characters). Thus, the text length of the subtree is 82 characters.

If the original structure is a properly structured document hierarchy, it is likely that the generic classes mapped to the original element types are suitable and the resulting metastructures are sensible. The original hierarchical structure is preserved, when the generic element names are used instead of the original element names in the document instances. Moreover, the document instances obey the structure of the generic DTD designed to be very permissive, as shown in Chapter 4. On the other hand, classification simplifies the original structure, as several original element types may be mapped to one generic class. Nevertheless, the original element names can be saved in attributes. Thus, the original structure can be restored, if necessary.

Appendix A gives an example of the use of the generic classes. The first document in Appendix A is an excerpt of a biography [Bor93] structured according to the Book DTD of ISO 12083 DTD [ISO94]. In the latter document the element names are replaced with the generic element names, and the structure obeys the generic DTD. As another example, Figure 3.6 presents the structure tree of the element *example* that contains mathematical text, and Figure 3.7 presents the structure tree, where the original element names are replaced with the generic element names.

It is typical to make a distinction between *block-level* and *inline* elements: block-level elements generally begin a new line, whereas inline ele-

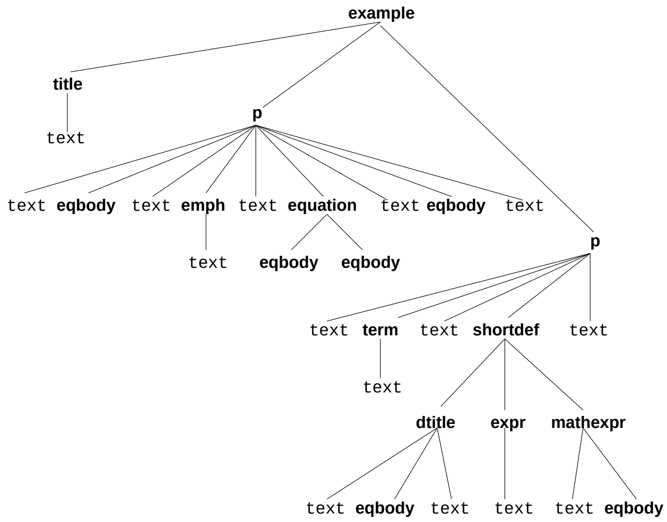


Figure 3.6: The structure tree of the element *example*.

ments do not. On the other hand, block-level elements may contain both inline and block-level elements, whereas inline elements contain text and other inline elements. Block-level structures are generally larger than inline structures, because block-level elements are typically interior nodes of the structure tree, whereas inline-elements are often leaves or located near leaves.

Typically, the generic classes *String*, *Fig*, *Formula*, *math*, and *Ref* behave like inline elements, whereas the rest of the generic classes, such as *Section*, *Para*, *ParaCont*, and *StrCont*, are more clearly block-level elements. However, some elements may act as inline or block-level elements depending on the context. As an example, consider strings: if a string is located inside a paragraph, it is clearly an inline element, whereas if a *StrCont* has several strings as children, each string should begin a new line; see Figure 3.8, which is a formatted presentation of the structure tree of Figure 3.7.

The presented generic DTD is only one solution to describe metastructures. It is suitable especially for *typical electronic documents* such as books, articles, reports, news, encyclopedias, and manuals. There can be, of course, different meta-DTDs for different needs. XML is widely used as a data representation and exchange format, and some application area, e.g., banking, medicine, or data communications, might need a representation for common data structures easily defined in a specific meta-DTD. On the other

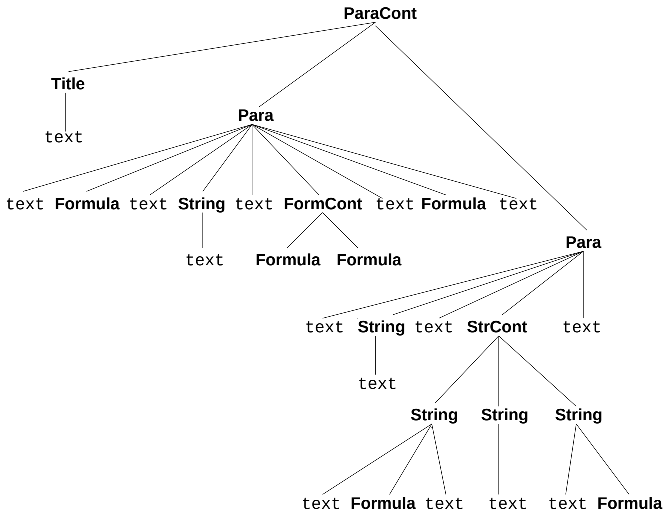


Figure 3.7: The structure tree of the generic element *ParaCont*.

hand, a computer program is an example of a structured document which is not marked up with SGML or XML. However, we could also design metastructures that describe common structures of programming languages and a mapping procedure that maps the structures of a programming language to the metastructures. The element-type classification is designed to map element types of typical electronic documents. However, the use of text length and element relations can be applied to different mappings, even if the mapping rules are significantly different than the rules of the classification method presented in the next section.

3.2.2 Decision rules

In this section we present rules for mapping an element type of a document type to a certain generic class. In defining the decision function we use the following notations:

- E is an element type in the original (arbitrary) DTD,
- T_E is the *average text length of the element instances* of the element type E,
- R_E is the *average text length of the subtrees of the element instances* of the element type E,

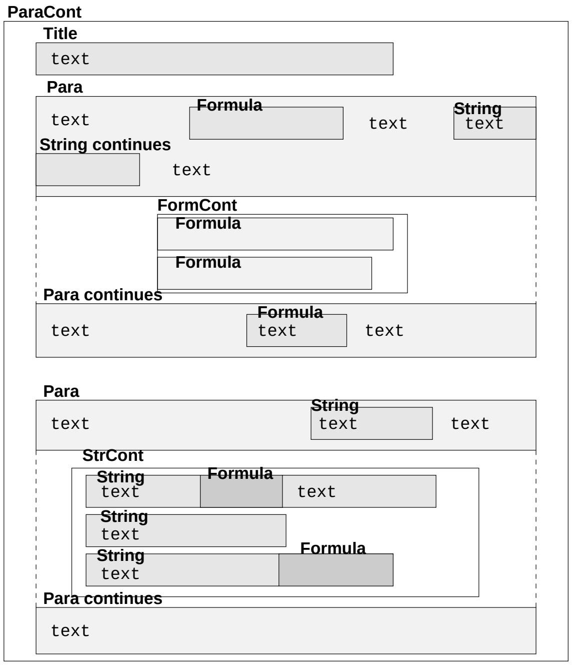


Figure 3.8: A demonstration of inline and block-level elements.

- The set D_E contains the names of the elements which the element instances of E contain directly, a terminal symbol text, if text appears in some element instance of E, and λ, if all the element instances of E are empty. During the mapping process, each original element name in D_E is replaced with the mapping result, i.e., the name of the generic class.

The average text length of the element instances of the element type E, denoted as T_E, is counted as follows: the sum of the text lengths of the element instances of E in the document instances is divided by the number of the element instances. For the element type E, R_E is counted as follows: the sum of the text lengths of the subtrees of the element instances of the element type E is divided by the number of the subtrees of the element instances.

We use paras as a shorthand for "$Para, ParaCont, ParaGrp$". The threshold maxstr is used for distinguishing strings from paragraphs. It is the maximum value for the average text length of the element instances of the generic class *String*. The threshold minsect is used for recognizing sections. It is the minimum value for the average text length of the subtrees of the

$(R_E <$ minsect$) \wedge (\forall e \in D_E, R_e <$ minsect$)$	
$T_E > 0$	$(0 < T_E \leq$ maxstr$) \Rightarrow \hat{h}_G(E) = \mathit{String}$
	$($maxstr $< T_E <$ minsect$) \Rightarrow \hat{h}_G(E) = \mathit{Para}$
$T_E = 0$	$(D_E = \lambda) \Rightarrow \hat{h}_G(E) \in \{\mathit{Empty}, \mathit{Ref}, \mathit{Fig}, \mathit{Formula}\}$
	$(\mathit{Para} \in D_E) \Rightarrow \hat{h}_G(E) = \mathit{ParaCont}$
	$((\mathit{Para} \notin D_E) \wedge (\{\mathit{ParaCont}, \mathit{ParaGrp}\} \cap D_E \neq \emptyset)) \Rightarrow \hat{h}_G(E) = \mathit{ParaGrp}$
	$(\{$paras$\} \cap D_E = \emptyset) \Rightarrow \hat{h}_G(E) \in \{\mathit{FormCont}, \mathit{FigCont}, \mathit{StrCont}, \mathit{StrGrp}\}$
$R_E >$ minsect	
$T_E > 0$	$\hat{h}_G(E) = \mathit{Section}$
$T_E = 0$	$(\{$paras$\} \cap D_E \neq \emptyset) \Rightarrow \hat{h}_G(E) = \mathit{Section}$
	$((\{$paras$\} \cap D_E = \emptyset) \wedge (\{\mathit{Section}, \mathit{SectGrp}\} \cap D_E \neq \emptyset)) \Rightarrow \hat{h}_G(E) = \mathit{SectGrp}$
	$(\{$paras$, \mathit{Section}, \mathit{SectGrp}\} \cap D_E = \emptyset) \Rightarrow \hat{h}_G(E) = \mathit{StrSect}$
else	$\hat{h}_G(E) = \mathit{Unknown}$

Table 3.1: An overview of the decision rules.

element instances of the generic class *Section*. The testing of the classification method (see Chapter 5) shows that in practice, a suitable value for maxstr is 60 characters and for minsect 2 000 characters.

The decision function is denoted by $\hat{h}_G$. An overview of the rules of $\hat{h}_G$ is presented in Table 3.1.

In the following, we use as an example the formatted text fragment of Figure 3.9, which demonstrates the use of the generic classes. The text was originally structured according to the DocBook SGML DTD [OAS91], and the original element names are saved in the attribute *ORIGEL*. In order to simplify the figure, the end tags of the markup have been left out.

Next we present the rules of the element-type classification. The value of the decision function $\hat{h}_G$ is defined by twelve rules. In the algorithm that implements the decision rules (presented in Chapter 4) the order of the rules is relevant. For instance, in the rule 5 for *ParaGrp* there is the condition *ParaCont in* D_E. Hence, the rule 4 for *ParaCont* must be applied before the rule 5. The dependencies between the different rules are taken into account when presenting the rules of $\hat{h}_G$: the processing order is from the rule 1 to the rule 12 in the algorithm.

In the first rule, element types that are totally empty are identified, namely the classes *Empty*, *Ref* (reference), *Fig* (figure), and *Formula*:

1: <VirtDoc ORIGEL="BOOK"><Title ORIGEL="TITLE"><String ORIGEL="PRODUCTNAME">

Digital UNIX

2: <ParaGrp ORIGEL="BOOKINFO"><ParaCont ORIGEL="BOOKBIBLIO"><Title ORIGEL="TITLE">

System Administration

...
*3: <StrCont ORIGEL="PUBLISHER"><String ORIGEL="PUBLISHERNAME">***Digital Equipment Corporation**
*4: <String ORIGEL="ADDRESS"><String ORIGEL="CITY">***Maynard,** *<String ORIGEL="STATE">***Massachusetts**
...
5: <Section id="INTRO-MAINT-CH" ORIGEL="CHAPTER"><Title ORIGEL="TITLE">

2. Starting Up and Shutting Down the System

*6: <Para ORIGEL="PARA">*This chapter describes procedures for starting up and shutting down the operating...
...
7: <Section ORIGEL="SECT1"><Title ORIGEL="TITLE">

2.2 Preparing to Boot the Installed System

*8: <Para ORIGEL="PARA">*As the system administrator, you set up or encounter various preboot or postshutdown states. This section describes and recommends procedures for preparing and initiating a reboot from a variety of system states. The states discussed here include the following:
9: <ParaGrp ORIGEL="ITEMIZEDLIST">

- *10: <ParaCont ORIGEL="LISTITEM"><Para ORIGEL="PARA">*A powered–down system
- *11: <ParaCont ORIGEL="LISTITEM"><Para ORIGEL="PARA">*A powered–up, halted system

...
12: <Section ORIGEL="SECT2"><Title ORIGEL="TITLE">

2.2.1 Preparing to Boot a Powered–Down System

13: <Para ORIGEL="PARA"><StrCont ORIGEL="INDEXTERM">
14: <String ORIGEL="PRIMARY">boot preparation<String ORIGEL="SECONDARY">powered–down systems
A system is powered down when the hardware (processor, devices, and peripherals) is turned off. Administrators power down the hardware periodically for routine maintenance or to configure new devices.
...
*15: <Para ORIGEL="PARA">*The following*<String ORIGEL="LITERAL">***mount** command syntax is for all file systems:
*16: <StrGrp ORIGEL="CMDSYNOPSIS"><Title ORIGEL="COMMAND">***mount**

- *17: <StrGrp ORIGEL="GROUP"><String ORIGEL="OPTION">*adflruv
- *18: <StrGrp ORIGEL="GROUP"><String ORIGEL="OPTION">*o, *<String ORIGEL="REPLACEABLE">*option
- *19: <StrGrp ORIGEL="GROUP"><String ORIGEL="OPTION">*t, *<String ORIGEL="REPLACEABLE">*type
- *20: <Title ORIGEL="ARG"><String ORIGEL="REPLACEABLE">***file_system**
- *21: <Title ORIGEL="ARG"><String ORIGEL="REPLACEABLE">***mount_point...**

Figure 3.9: Formatted text fragment with the generic classes.

1. if $(D_E = \lambda)$ **then** $\hat{h}_G(E)$ **in** $\{Empty, Ref, Fig, Formula\}$.

Mapping of the classes *Ref*, *Fig*, and *Formula* requires using the attribute information of the DTD. There is a standard way of representing references and figures in XML (or SGML) DTDs: references (*Ref*) have an attribute of type IDREF, and figures (*Fig*) have a graphics NOTATION attribute. However, it is impossible to determine automatically whether a figure is a formula (*Formula*). Hence, human interaction or element name based identification is needed, if such formulas are mapped. If the element type does not have an IDREF or a graphics NOTATION attribute, the resulting class is *Empty*.

String is a short sequence of text. The average text length of the element instances must be more than zero but less or equal than maxstr characters (condition $0 < T_E \leq \mathsf{maxstr}$):

> **2.** **if** $(0 < T_E \leq \mathsf{maxstr})$ **and** $(R_E < \mathsf{minsect})$ **and**
> (**for each** e **in** $D_E, R_e < \mathsf{minsect}$) **then**
> $\hat{h}_G(E) = \textit{String}$.

A typical use for *String* is, for example, to emphasize some word in a paragraph, like on row 15 of Figure 3.9, where the element *LITERAL* contains the word "mount". *String* may also contain other *String*s like on row 4, where the element *ADDRESS* contains the elements *CITY* and *STATE*. *ADDRESS* is classified as *String* even if the element instance on row 4 does not contain directly any text. Hence, to satisfy the condition $T_E > 0$, some other element instance(s) must contain text increasing the T_E, e.g, like "<*ADDRESS*>Rosegate 10</*ADDRESS*>". A less common situation is when a string contains a paragraph; a string could contain a footnote, which in turn contains a paragraph. There is a restriction that the average text length of the subtrees of the element instances of *String* must be less than $\mathsf{minsect}$ characters (condition $R_E < \mathsf{minsect}$). The average text length of the subtrees of the element instances which the element instances of *String* may contain must also be less than $\mathsf{minsect}$ (condition *for each e in* $D_E, R_e < \mathsf{minsect}$). These two conditions prevent a string from containing a section.

Paragraphs (*Para*) have to contain directly text more than maxstr characters on average:

> **3.** **if** $(\mathsf{maxstr} < T_E < \mathsf{minsect})$ **and** $(R_E < \mathsf{minsect})$ **and**
> (**for each** e **in** $D_E, R_e < \mathsf{minsect}$) **then**
> $\hat{h}_G(E) = \textit{Para}$.

The classes *String* and *Para* have identical content models in the generic DTD. The basic difference is that *Para* is a longer sequence of characters (condition $\mathsf{maxstr} < T_E < \mathsf{minsect}$) than *String*. In Figure 3.9 there are *Para*s, e.g., on rows 6, 8, 13, and 15.

A container wraps other elements, and it is not allowed to contain directly any text (condition $T_E = 0$). Next, the classes *ParaCont* (paragraph container) and *ParaGrp* (paragraph group) are presented:

> **4.** **if** (*Para* **in** D_E) **and** $(T_E = 0)$ **and** $(R_E < \mathsf{minsect})$ **and**
> (**for each** e **in** $D_E, R_e < \mathsf{minsect}$) **then**
> $\hat{h}_G(E) = \textit{ParaCont}$.

5. **if** ($Para$ **not in** D_E) **and**
($\{ParaCont, ParaGrp\} \cap D_E \neq \emptyset$) **and** ($T_E = 0$) **and**
($R_E <$ minsect) **and** (**for each** e **in** $D_E, R_e <$ minsect) **then**
$\hat{h}_G(E) = ParaGrp$.

The class *ParaCont* must contain paragraphs (condition *Para in* D_E). In the generic DTD the content models of *Para* and *ParaCont* are the same except that *ParaCont* may not contain text (#PCDATA). The class *ParaGrp*, in turn, must contain *ParaCont*s or *ParaGrp*s, or both (condition $\{ParaCont, ParaGrp\} \cap D_E \neq \emptyset$), but it is not allowed to contain any *Para*s, since that characteristic was used for identifying *ParaCont*s.

ParaGrp is typically a list structure, an example of which may be seen in Figure 3.9: The paragraph starting from row 8 contains a list (*ParaGrp*) starting from row 9. *ParaCont* is a list item (rows 10, 11), which may contain one ore more *Para*s. Lists should be formatted according to the context, which means that the formatting depends on the location in the structure tree. In the beginning of the book on row 2 of Figure 3.9, there is a *ParaGrp*, whose father is the root of the document, *VirtDoc*. It could be inferred from the context that this *ParaGrp*, originally *BOOKINFO*, contains the front cover information of the book, and the title "System Administration" in the element *ParaCont*, originally *BOOKBIBLIO*, should be formatted with a larger font.

Container structures *FormCont* (formula container), *FigCont* (figure container), *StrCont* (string container), and *StrGrp* (string group) are not allowed to contain paras:

6. **if** ($\{\text{paras}\} \cap D_E = \emptyset$) **and**
($\{Formula, FormCont\} \cap D_E \neq \emptyset$) **and** ($T_E = 0$) **and**
($R_E <$ minsect) **and** (**for each** e **in** $D_E, R_e <$ minsect) **then**
$\hat{h}_G(E) = FormCont$.

7. **if** ($\{Formula, FormCont, \text{paras}\} \cap D_E = \emptyset$) **and**
($\{Fig, FigCont\} \cap D_E \neq \emptyset$) **and** ($T_E = 0$) **and**
($R_E <$ minsect) **and** (**for each** e **in** $D_E, R_e <$ minsect) **then**
$\hat{h}_G(E) = FigCont$.

8. **if** ($D_E \subseteq \{String, Empty, Ref\}$) **and** ($T_E = 0$) **and**
($R_E <$ minsect) **then**
$\hat{h}_G(E) = StrCont$.

9. if $(\{Formula, FormCont, Fig, FigCont, \mathsf{paras}\} \cap D_E = \emptyset)$ **and** $(\{StrCont, StrGrp\} \cap D_E \neq \emptyset)$**and** $(T_E = 0)$ **and** $(R_E < \mathsf{minsect})$ **and** (**for each** e **in** $D_E, R_e < \mathsf{minsect}$) **then**
$\hat{h}_G(E) = StrGrp$.

The most unrestrictive of the above classes is *FormCont*: if at least one element type in D_E is mapped to *Formula* or a *FormCont*, the element type is classified as *FormCont*. Element types containing at least one *Fig* or *FigCont* are identified as *FigCont*. A basic container for strings is *StrCont*, and it is allowed to contain only element types mapped to the classes *String*, *Empty* or *Ref*. *StrGrp* must contain at least one *StrCont* or *StrGrp*, and it may also contain the classes *String*, *Empty*, and *Ref*. *StrGrp* is typically a list of short items.

On rows 3–4 of Figure 3.9 there is a *StrCont*, originally *PUBLISHER*, which contains two strings *PUBLISHERNAME* and *ADDRESS*. Also, the *Para* of row 13 contains a *StrCont*, originally *INDEXTERM*, which contains the strings *PRIMARY* and *SECONDARY*. Examples of the class *StrGrp* are on rows 16–21: the element *CMDSYNOPSIS* is a *StrGrp* list containing other *StrGrp*s (originally *GROUP*) and *Title*s (originally *ARG*). In this context, one would expect that these *GROUP* elements should be classified as *StrCont*, since they contain strings only, but in some other element instances *GROUP* elements must have contained elements mapped to the classes *StrCont* or *StrGrp*. A better example of the class *StrGrp* can be found in Figures 3.1 and 3.3 in Section 3.1: the element *poem* was classified as *StrGrp*, the element *stanza* as *StrCont*, and the element *poemline* as *String*.

The class *Section* is a basic element type forming the skeleton of a typical document hierarchy:

10. if $((T_E > 0)$ **and** $(R_E \geq \mathsf{minsect}))$ **or** $((\mathsf{paras} \cap D_E \neq \emptyset)$ **and** $(T_E = 0)$ **and** $(R_E \geq \mathsf{minsect}))$ **then**
$\hat{h}_G(E) = Section$.

There are two conditions for a *Section*: the element type may contain text and the average text length of the subtrees of the element instances of *Section* is at least minsect, or the element type may not contain text, but contain element types mapped to *Para*, *ParaCont* or *ParaGrp* and the average text length of the subtrees of the element instances of *Section* is at least minsect. In most practical cases, *Section*s are containers for paras and other *Section*s, like on rows 5 (originally *CHAPTER*), 7 (originally *SECT1*), and 12 (originally *SECT2*) of Figure 3.9.

There are two section level containers, namely *SectGrp* (section group) and *StrSect* (string section):

11. if $(\{\mathsf{paras}\} \cap D_E = \emptyset)$ **and** $(\{\mathit{Section}, \mathit{SectGrp}\} \cap D_E \neq \emptyset)$ **and** $(T_E = 0)$ **and** $(R_E \geq \mathsf{minsect})$ **then**
$\hat{h}_G(E) = \mathit{SectGrp}$.

12. if $(\{\mathsf{paras}, \mathit{Section}, \mathit{SectGrp}\} \cap D_E = \emptyset)$ **and** $(T_E = 0)$ **and** $(R_E \geq \mathsf{minsect})$ **then**
$\hat{h}_G(E) = \mathit{StrSect}$.

The class *SectGrp* contains *Section*s or *SectGrp*s, or both, but it is not allowed to contain *Para*s, *ParaCont*s or *ParaGrp*s. A typical example of a *SectGrp* is a body element which contains chapters in a book. The class *StrSect* may contain only *String*s and related containers, and other *StrSect*s. Typically, poem collections contain *StrSect*s. A good example of *StrSect* is a play of Shakespeare, where acts and scenes would be classified as *StrSect*s, because they contain only short lines of speech.

Finally, if there are still element types which do not have a generic class, they are mapped to *Unknown* ($\hat{h}_G(E) = \mathit{Unknown}$). Note that the class *Unknown* may appear anywhere in the metastructure.

3.2.3 Refining the mapping

The decision rules identify the basic classes of the generic DTD. There are some situations, where more specific classes are needed and, hence, the mapping must be refined.

As an example, consider the identification of titles. Usually sections and some container structures contain directly a title as a first item, which the decision rules map to *String* or *StrCont*. If, for instance, a section contains directly a title, the title element is typically the first child of the section element. The set D_E contains the names of the elements which element instances of E contain directly, i.e., the names of the children of element instances of E. To identify titles, for each element name e in D_E we have to know, if an element instance named e has been the first child of some element instance of E. This information can be collected while gathering the information from the document instances. In the following rule, E' is the generic class of an element type E, e is a element name in D_E, and $e.\mathsf{first}$ is true, if an element instance named e has been the first child of some element instance of E.

if $((E'$ **in** $\{VirtDoc, Section, SectGrp, StrSect\})$ **and**
(**for some** e **in** $D_E, e.\mathsf{first}$ **and** e **in** $\{String, StrCont\}))$ **or**
$((E'$ **in** $\{ParaGrp, ParaCont, StrGrp\})$ **and**
(**for some** e **in** $D_E, e.\mathsf{first}$ **and** e' **in** $\{String\}))$ **then**
$\hat{h}_G(E) = Title.$

The extraction of titles has to face a trade-off: If the conditions are too permissive, some element types which are not originally meant to be titles are also mapped to the class *Title*. If the conditions are too strict, some of the original titles may not be found. Note that the content models of *Title* and *String* are the same in the generic DTD, and *Title* appears in other content models in the same way as *String* does.

The table model of the XHTML DTD [W3C00f] is chosen to be the table model for the generic XML DTD in Figure 3.4. It must be noted that the CALS Table Model [OAS95], which is an SGML construct, is mostly used by the industry. For uniform table formatting, a transformation script should transform all CALS tables of document instances into XHTML tables or vice versa. In practice, if a table and its subtree of the chosen table model are recognized, an element type belonging to the table structure is mapped to itself.

Formulas are typically either pictures identified as *Formula*s, or marked up with some description language. In the generic DTD in Figure 3.4 there is a reference to *Mathematical Markup Language (MathML)* [W3C99c], which is a recommendation of the W3C. It can be used to encode both the presentation and the mathematical content. The element type *math* is assumed to be the root of a MathML structure, and it appears in other content models in the same way as *Formula* does. Similarly to the table recognition, if a MathML structure is recognized, an element type belonging to the structure is mapped to itself.

The Dublin Core (DC) metadata was introduced in Section 2.3. Next, we present examples of how DC elements are encoded in HTML 4.0, and how they can be represented using RDF and XML Namespaces. We also describe how these representations are mapped to the generic classes.

The markup of the following excerpt is according to HTML 4.0. The extraction contains four DC elements, namely *Title*, *Creator*, *Subject*, and *Description*, which describe this thesis. The example is encoded using the element type *meta*, which in HTML 4.0 is defined to be empty[1]. The attribute *name* identifies the DC element, while the attribute *content* contains the metainformation.

[1]http://www.w3.org/TR/REC-html40/struct/global.html#h-7.4.4

```
<meta name="DC.Title" content=
      "Generalization of Document Structures and Document Assembly">
<meta name="DC.Creator"
      content="Barbara Heikkinen">
<meta name="DC.Subject"
      content="document assembly, metastructures, xml">
<meta name="DC.Description" content=
      "computer science PhD Thesis on structured document assembly">
```

The next example uses XML Namespaces and the XML representation of RDF. Prefix "rdf" identifies the RDF namespace, while prefix "dc" identifies the Dublin Core namespace. According to the RDF syntax, the construct begins with the element *RDF*, which contains the element *Description* and the attribute *about*. The DC elements *Title*, *Creator*, *Subject*, and *Description* are encoded as separate XML elements in the Dublin Core namespace.

```
<rdf:RDF
  xmlns:rdf="http://www.w3.org/TR/PR-rdf-syntax#"
  xmlns:dc="http://purl.oclc.org/dc#">
  <rdf:Description about="http://www.cs.Helsinki.FI/Barbara.Heikkinen">
    <dc:Title>Generalization of Document Structures and Document
    Assembly</dc:Title>
    <dc:Creator>Barbara Heikkinen</dc:Creator>
    <dc:Subject>document assembly, metastructures, xml</dc:Subject>
    <dc:Description>computer science PhD Thesis on structured document
    assembly</dc:Description>
</rdf:RDF>
```

Examples of using RDF to encode Dublin Core properties can be found from the RDF Model and Syntax Specification [W3C99e]. A real-life example can be found from the Web pages of the Electronic Libraries Programme (eLib)[2].

Finally, the mapping of the above excerpts is discussed. The generic DTD of Figure 3.4 contains the element type *meta*, which is equivalent to *meta* in HTML 4.0 (and XHTML 1.0). Hence, the element type *meta* in the former excerpt is mapped to the generic class *meta*. The processing of the latter excerpt according to RDF requires processors, which are both RDF and XML Namespaces aware. If such tools are used, then the classification method should leave RDF structures as they are without mapping them to the generic classes.

[2]http://www.ukoln.ac.uk/services/elib/metadata.html

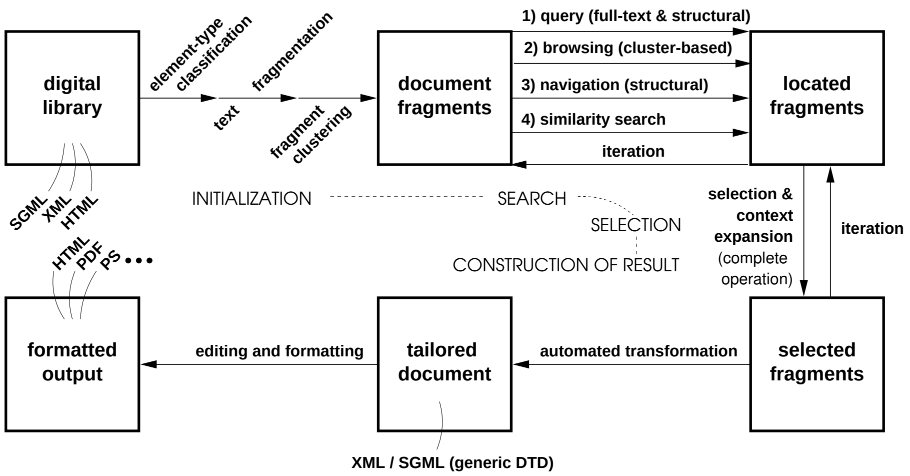


Figure 3.10: The document assembly process.

3.3 The assembly process

This section presents the key ideas of document assembly and describes some situations where document assembly can be applied.

Document assembly can be defined as the computer-aided construction of new documents from a collection of electronic documents [AHH+98a]. The assembly is usually an *interactive process*, where an author or editor uses various tools to find appropriate sources and to configure the intended document. Figure 3.10 presents the document assembly process, which consists of the initialization actions that analyze and preprocess the collection, the search of interesting text fragments, the selection of desired fragments, and the construction of a coherent assembly result. The document composer works iteratively on a document collection until a result document is obtained. Even though producing carefully edited publications requires human creativity, the assembly tasks can be supported by computerized methods.

3.3.1 Initialization of the document collection

The actual assembly process is preceded by the *initialization actions* that analyze and preprocess the documents for the assembly (Figure 3.10). Heterogeneous document collections which contain documents with differing document types can be imported. The set of DTDs is not fixed, and no DTD-specific semantics is assumed to be available. Therefore, the assembly process has to rely on the knowledge that can be extracted directly from the document instances and their DTDs. The processing of well-formed XML documents does not necessarily require a DTD, but the DTD is used if available. The first initialization phase is the *element-type classification* presented in the previous section: the document structure is analyzed and each element type is mapped to a generic class.

The second initialization phase, the *text fragmentation*, divides documents into fragments, which are coherent, contiguous, and relatively independent parts of documents. To get reasonable results for document reuse, each fragment, like a chapter or a section, should be authored to discuss a well-defined topic. However, it is often not possible to manually re-structure an existing document base. To enable fragment-based assembly from existing document collections, automatic statistical methods for the recognition and markup of probably useful document fragments have been developed [AHHK97b, Hei98]. The fragmentation uses knowledge of the generic classes: lowest-level fragments are usually sequences of elements that are mapped to the class *Para*. Additionally, the fragment hierarchy consists of the ancestor elements of the minimal fragments. The size of the lowest-level fragments, i.e., the granularity of the minimal coherent document parts, can be adjusted during the initialization phase. The user can also search and manipulate smaller units than fragments called document components, or assembly components. A component can be any document element and its subtree, while fragment is a coherent part of the document.

The third initialization phase, *fragment clustering*, groups fragments into clusters, which are collections of mutually similar document fragments. Similarity of two fragments is measured in a standard fashion by calculating the proportion of their common words.

The preprocessing, or the authoring phase, may also include adding supplementary information that can guide the assembly. Adding intelligence to documents is called *seeding* [ET94, AHH+96a]. Manual seeding has been experimented with a document collection of three text books on control engineering. This experimentation will be discussed in detail in Section 6.2. The contents of the books have been classified by the authors according to the educational level and the nature of the content. The content classification was attached as attributes of the corresponding document elements. Information was also added automatically to the books, e.g., keyword lists for the document fragments were computed.

3.3.2 Searching interesting fragments

In the first actual assembly phase, the user *searches* interesting fragments or components to be included in the resulting document. In our framework there are four ways to search elements (Figure 3.10). The first one is a rather standard search that includes full-text search with conditions on the structure and attribute values. For instance, within our collection of text books on control engineering, the user might want to create an assembly of exercises for college students majoring in data communications.

The second way is based on browsing fragment clusters. Clusters are used as a way to present large portions of the collection as browsable groups of similar content. In the preprocessing phase the collection is already partitioned into clusters, which are a starting point for interactive cluster browsing. This approach, known as Scatter/Gather, was introduced by Cutting et al., who also presented efficient algorithms for it [CPKT92, CKP93]. Scatter/Gather browsing has been reported to be an effective way to utilize unfamiliar information collections in a way that is complementary to traditional query-based retrieval of documents [PSHD96]. There are also other approaches to categorize and explore large text collections; see, e.g., [LHKK96, DMRA97, HK97, Mer97].

The third possibility is to browse the document collection by navigating through the hierarchical document structures and searching interesting fragments or components manually. Application of this search method usually requires that the two previous search methods have already been used to reduce the document collection.

Finally, the fourth way is to start from the selected fragments or components and let the system search for similar or related fragments in the collection. Again, measuring the similarity of fragments can be based on comparing the sets of their keywords.

3.3.3 Selecting fragments and constructing a result

In the assembly framework there are methods that assist the user to *select* desired document fragments or components and to construct a *coherent assembly result* (Figure 3.10). To facilitate getting a view of the document structures the user can see both the original element names and the generic class names. This is important, since the collection may consist of differently structured documents from different document types, and the user may not be familiar with the structures. The user can also recognize which components are fragments and hence coherent units. To get a reasonable result it is recommended to use fragments as building blocks of the assembly, but the user can choose smaller components if needed. After having found a group of document components, the user may manipulate them. The components are presented as an ordered list, and the user may select, discard and rearrange the components. Each component in the component list is actually a pointer to the corresponding element structure.

Although any list of components can be exported as a valid XML (or SGML) document, sometimes it is useful to expand the assembly with non-selected elements that increase the readability of the resulting document. This expansion is called the *Complete* operation. The general idea of *Complete* is that some context is included in the assembly. The context in question can be, for example, the nearest section title and some introductory paragraphs from this section, or for a reference, the target of the reference such as a figure or a formula. The user selects the desired components for the *Complete* operation.

The heuristic principles guiding the completion are: For a selected component in the component list the nearest ancestor mapped to *VirtDoc*, *SectGrp*, *Section* or *StrSect* is searched, and the *Title* child of the ancestor is retrieved. If there are several consecutive elements in the component list under the same ancestor, they are joined as one component called *Completed Sequence*. If there is only one element under an ancestor, a component called *Completed <element-class>* is formed. If the ancestor has a child mapped to *Para*, *ParaCont* or *ParaGrp* after the *Title* element, this child is also retrieved. If there are references in some component element, the targets of the references are also retrieved. If the target is large, such as *SectGrp*, *Section* or *StrSect*, only the title of the target is included.

An example of the *Complete* operation is presented in Figure 3.11. The user has selected the components mapped to *ParaCont* and *ParaGrp*. The nearest ancestor mapped to *Section* is searched, and the first child mapped to *Title* and the second child mapped to *Para* are retrieved with the selected components. The retrieved components form a *Completed Sequence*.

Additional heuristics may be designed for the *Complete* operation. For instance, if the component of *Complete* is very small, such as a *String* or a *StrCont*, or it is part of an entity, such as a *ParaGrp* forming a list, it may be reasonable to apply *Complete* to the father. Hence, if the father is mapped to, e.g, *Para*, *ParaCont*, *ParaGrp*, *String*, *StrCont* or *StrGrp*, the target element may be replaced with its father. In the case of Figure 3.11, the target *ParaGrp* would be replaced with its father *Para*.

When a completed component is saved to a result document, it is saved inside a wrapper that is a *Section*, *ParaCont*, *StrSect* or *StrGrp* depending on the generic classes of the element types of the corresponding component.

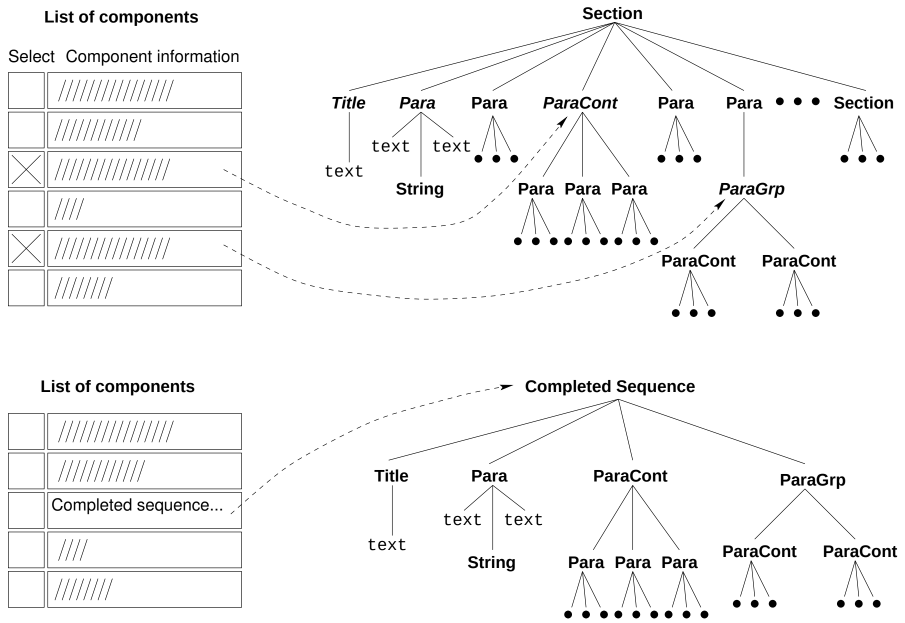


Figure 3.11: *Complete* operation for the components under the same *Section* ancestor.

For example, in Figure 3.11 the wrapper for the *Completed Sequence* would be *ParaCont*. Information of the executed *Complete* operation is saved to an attribute of the wrapper to inform the user of the formed composite element.

Intermediate assemblies can be presented to the user, e.g., as HTML pages, by adorning each generic element type with suitable formatting markup. However, the final assembly should be a valid and coherent XML (or SGML) document, which can be properly processed by other applications. Thus, after the iterative selection and modification process, the selected fragments may be *automatically transformed* to one structured tailored document that conforms to the presented generic DTD (Figure 3.10). Of course, transformation scripts could be made to further transform the assembly result from the known structure of the generic DTD to some other structure. The tailored document can now be further *edited* with some structure editor in order to get more trimmed content and structure. The resulting assembly may also be *formatted* and *printed* using the formatting information attached to the generic DTD. Hence, the document may be automatically transformed to, e.g., HTML, PDF, or PostScript format.

3.3.4 Applications of document assembly

Document assembly tools could facilitate the working processes of writers, such as journalists collecting background material for articles and editors of electronic books or magazines. Consider an example where an electronic composition called “Special issue on nightingales” is to be prepared using a heterogeneously structured document collection consisting of the following types of material:

- pictures and text of anatomy and physiology of the nightingale,
- samples of singing,
- maps of distribution,
- poems, novels, plays, and songs of nightingales,
- news stories about individual nightingales, and
- statutes of law, e.g., protection.

The journalist who writes an individual article uses the assembly system, first to *search* the desired information from the document base, then to *select* the most appropriate information fragments considering the subject of the article, and finally to *combine* the pieces into a single document. The journalist may print the assembly result and use the information, such as some statutes of law related to the subject, as a supporting context for the story; he or she may also use some or all parts of the assembly, such as pictures, maps and text excerpts from poems or songs, as building blocks of the new text. The journalist may also create some new material by making, e.g., some interviews in the zoo, and a photographer may go to a park or a forest where nightingales live, and take some nice pictures for the story. After the article is ready, the new information may be stored into the document base for further use.

In an industrial application a client may order different configurations of a complex machine, such as an airplane, a train, an elevator or a paper machine. The user manual could be assembled from differently structured modules of a large text database. Moreover, the database and manual structure may differ from each other considerably. The database structure may imitate the mechanical structure of a machine, while the user manual may be structured semantically into different manual parts such as maintenance, safety and mechanics, as well as according to logical structures such as sections, lists, tables and paragraphs. The compositions of printed and online manuals could also be somewhat different. In this kind of a demanding publishing process, our assembly tools can greatly facilitate the composition and dissemination of manuals.

A future application will be the retrieval of Web pages into a cellular phone: the Web pages have to be divided into smaller information fragments, which will fit onto the display of the phone. In partitioning the information into smaller fragments, a method using similar reasoning mechanisms as the element-type classification together with the text fragmentation method presented in Section 3.3.1 could be utilized. Of course, the new method should also extract information, e.g., from the formatting-oriented HTML elements or from the CSS and XSL stylesheets, and use metadata of the source if available.

Chapter 4

Generalization of document structures

In this chapter we present the element-type classification method formally by using the concept of *grammar morphism*. The main task of this chapter is to prove that after applying the classification method, any document instance in which the original element names are replaced with the generic class names follows the structure of the generic DTD. Section 4.1 introduces the basic concepts needed to define grammar morphisms. Section 4.2 defines an elementary grammar morphism function that maps element types into generic classes. Section 4.3 defines an advanced grammar morphism function that uses the average text lengths of the element instances and their subtrees in mapping the element types. The decision rules of this function are identical to the decision rules of the element-type classification method introduced in Section 3.2.

4.1 Basic concepts of context-free grammars and grammar morphisms

In this section we present some basics of context-free grammars and the definitions needed for a function mapping element types to the generic classes. In particular, we show that if a function g is a grammar morphism mapping grammar G to grammar G', then the language generated by grammar $g(G)$ is a subset of the language generated by grammar G'.

An *alphabet* Σ is a finite nonempty set of symbols. A *string* is a finite ordered sequence of symbols over an alphabet. The *length* of a string x, i.e., the number of symbols it contains, is denoted by $|x|$. A special symbol λ indicates an *empty string*, and $|\lambda| = 0$. The set of all strings over Σ is denoted by Σ^*. For example, if $\Sigma = \{0, 1\}$, then $\Sigma^* = \{\lambda, 0, 1, 00, 01, 10, 11, ...\}$. A *language* is any subset of Σ^*.

A basic operation, *catenation*, joins strings one after another, and it is denoted by xy (or $x \cdot y$), where x and y are strings. A string consisting of n copies of $a \in \Sigma$ is denoted by a^n. *Iteration* of a string x k times means catenation of k copies of x, and it is denoted by x^k. For instance, if $x = ab$ and $k = 2$, then $x^k = (ab)^2 = abab$.

Definitions 4.1–4.14 are based on [HU69, HU79, Woo87].

Definition 4.1 Let L and L' be languages over an alphabet Σ. Then

(i) The *union* of L and L' is the language

$$L \cup L' = \{x \in \Sigma^* \mid x \in L \vee x \in L'\}.$$

(ii) The *catenation* of L and L' is the language

$$LL' = \{xy \in \Sigma^* \mid x \in L, y \in L'\}.$$

(iii) The *powers* of L are defined iteratively:

$$L^k = \begin{cases} L^0 = \{\lambda\}, & k = 0 \\ L^k = LL^{k-1} = \{x_1 \dots x_k \mid x_i \in L \ \forall i = 1, \dots, k\}, & k \geq 1. \end{cases}$$

(iv) The *closure* of L is the language

$$L^* = \bigcup_{k \geq 0} L^k = \{x_1 \dots x_k \mid k \geq 0, x_i \in L \ \forall i = 1, \dots, k\}.$$

□

Definition 4.2 The *regular expressions* over an alphabet Σ are defined inductively by the following rules:

(i) $\emptyset$ and λ are regular expressions over Σ,

(ii) a is a regular expression over Σ for all $a \in \Sigma$,

(iii) if r and s are regular expressions over Σ, then $(r \cup s)$, (rs) and r^* are regular expressions over Σ,

(iv) there are no other regular expressions over Σ.

□

Definition 4.3 Each of the regular expressions r over Σ describes a language $L(r)$, which is defined as follows. Let r and s be regular expressions and let $a \in \Sigma$.

(i) $L(\emptyset) = \emptyset$,

(ii) $L(\lambda) = \{\lambda\}$,

(iii) $L(a) = \{a\}$ for all $a \in \Sigma$,

(iv) $L(r \cup s) = L(r) \cup L(s)$,

(v) $L(rs) = L(r)L(s)$,

(vi) $L(r^*) = (L(r))^*$.

□

Definition 4.4 A language L is *regular*, if there is a regular expression r such that $L = L(r)$. □

Definition 4.5 A *context-free grammar* is a quadruple $G = (N, \Sigma, P, S)$, where:

(i) N is a finite set of *nonterminals*,

(ii) Σ is a finite set of *terminals*, and N and Σ are disjoint,

(iii) $P \subseteq N \times (N \cup \Sigma)^*$ is a finite set of *productions* written as $A \to \omega$, where $A \in N, \omega \in (N \cup \Sigma)^*$,

(iv) $S \in N$ is the start symbol.

□

Definition 4.6 A string $\gamma \in (N \cup \Sigma)^*$ *derives directly* string $\gamma' \in (N \cup \Sigma)^*$ in grammar G, denoted by

$$\gamma \underset{G}{\Rightarrow} \gamma',$$

if $\gamma = \alpha A \beta$ and $\gamma' = \alpha\omega\beta$, where $\alpha, \beta, \omega \in (N \cup \Sigma)^*, A \in N$, and the grammar G has the production $A \to \omega$. □

Definition 4.7 A string $\gamma \in (N \cup \Sigma)^*$ *derives* string $\gamma' \in (N \cup \Sigma)^*$ in grammar G, denoted by

$$\gamma \underset{G}{\Rightarrow}^{+} \gamma',$$

if for some $k \geq 1$ there is a sequence of strings $\gamma_0, \gamma_1, \ldots, \gamma_k$ in $(N \cup \Sigma)^*$ such that

$$\gamma = \gamma_0 \underset{G}{\Rightarrow} \gamma_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} \gamma_k = \gamma'.$$

We write $\gamma \underset{G}{\Rightarrow}^* \gamma'$ if either $\gamma = \gamma'$ or $\gamma \underset{G}{\Rightarrow}^+ \gamma'$. □

Definition 4.8 A string $\gamma \in (N \cup \Sigma)^*$ is a *sentential form* of grammar G, if $S \underset{G}{\Rightarrow}^* \gamma$ holds. A sentential form $x \in \Sigma^*$ containing only terminals is a *sentence* of G. The language generated by grammar G, denoted by $L(G)$, is the set of all sentences, namely,

$$L(G) = \{x \in \Sigma^* \mid S \underset{G}{\Rightarrow}^+ x\}.$$

□

Definition 4.9 A language $L \subseteq \Sigma^*$ is *context-free*, if there is a context-free grammar G such that $L = L(G)$. □

Definition 4.10 A *parse tree* for a context-free grammar $G = (N, \Sigma, P, S)$ is an ordered tree T such that

(i) The nodes of T are labeled with symbols of the set $N \cup \Sigma \cup \{\lambda\}$.

(ii) The label of the root of T is S.

(iii) The labels of the interior nodes of T are nonterminals.

(iv) The labels of the leaves are terminals or λ. If a node n has a label λ, then it is the only child of its parent.

(v) If A is the label of an interior node n, and $X_1, X_2, \ldots, X_k$ are the labels of the children of n in the order from left to right, then $A \to X_1 X_2 \cdots X_k$ is a production in P.

□

Definition 4.11 The set of all parse trees for a context-free grammar $G = (N, \Sigma, P, S)$ is denoted by *ptrees*(G). □

Definition 4.12 For a context-free grammar $G = (N, \Sigma, P, S)$ we required that the set of productions P is finite. An *extended context-free grammar* is defined in the same way as a context-free grammar, with the exception that the set P can be infinite, as long as for all $A \in N$ the language $L(A) = \{\alpha \mid A \to \alpha \in P\}$ is regular. Equivalently, in such a grammar the right-hand side of a rule, α, is allowed to be a regular expression over $N \cup \Sigma$.
□

Proposition 4.13 [Woo87] For each extended context-free grammar G there is a context-free grammar G' such that $L(G) = L(G')$. □

Definition 4.14 Parse trees of an extended context-free grammar $G = (N, \Sigma, P, S)$ are similar to parse trees of an ordinary context-free grammar, except that parent with label A can have children labeled $X_1, X_2, \ldots, X_k$ when there is a production $A \to \alpha \in P$ such that $X_1 X_2 \cdots X_k \in L(\alpha)$. □

Definitions 4.15 and 4.16 are based on [Rei86, Woo87].

Definition 4.15 Let N and N' be finite sets of nonterminals and let Σ and Σ' be finite sets of terminals. A function $g : (N \cup \Sigma)^* \to (N' \cup \Sigma')^*$ is a *morphism* if:

(i) $g(\lambda) = \lambda$,

(ii) $g(a) \in \Sigma'$, if $a \in \Sigma$,

(iii) $g(A) \in N'$, if $A \in N$,

(iv) $g(X_1 X_2 \cdots X_k) = g(X_1) g(X_2) \cdots g(X_k)$, for all $X_1, X_2, \ldots, X_k$.

□

Definition 4.16 Let $G = (N, \Sigma, P, S)$ and $G' = (N', \Sigma', P', S')$ be context-free grammars. A function g is a *grammar morphism* from G to G', denoted

$$G \underset{g}{\to} G',$$

if the following conditions hold:

(i) $g(S) = S'$,

(ii) g is a morphism $g : (N \cup \Sigma)^* \to (N' \cup \Sigma')^*$, and

(iii) for all productions $A \to \alpha \in P$, we have $g(A) \to g(\alpha) \in P'$.

□

The definition of a morphism can be expanded to languages and parse trees.

Definition 4.17 Let g be a morphism. If L is a language, then $g(L) = \{g(x) \mid x \in L\}$. □

Definition 4.18 Let g be a morphism. If $G = (N, \Sigma, P, S)$ is a context-free grammar and t is a parse tree for G, then the parse tree $g(t)$ is defined to be the tree that has the same tree structure as t, and for all nodes n of t labeled A, the corresponding node of $g(t)$ is labeled $g(A)$. □

Definition 4.19 Let G be a context-free grammar and g be a grammar morphism. Then $g(\mathit{ptrees}(G)) = \{g(t) \mid t \in \mathit{ptrees}(G)\}$. □

Theorem 4.20 Let $G = (N, \Sigma, P, S)$ and $G' = (N', \Sigma', P', S')$ be context-free grammars. If there is a grammar morphism g from G to G', $G \underset{g}{\rightarrow} G'$, then

(i) If $A \underset{G}{\Rightarrow} \alpha_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} \alpha_k$ is a derivation such that $A \in N$ and $\alpha_i \in (N \cup \Sigma)^*$ for each $i = 1, \ldots, k$, then $g(A) \underset{G'}{\Rightarrow} g(\alpha_1) \underset{G'}{\Rightarrow} \cdots \underset{G'}{\Rightarrow} g(\alpha_k)$.

(ii) $g(L(G)) \subseteq L(G')$.

(iii) $g(\mathit{ptrees}(G)) \subseteq \mathit{ptrees}(G')$.

Proof

(i) The claim is proved by induction on the length n of the derivation.

Basis: $n = 1$. As $A = \alpha_0 \underset{G}{\Rightarrow} \alpha_1$, there exists in P a rule $A \rightarrow \alpha_1$. According to Definition 4.16 (iii), there exists a rule $g(A) \rightarrow g(\alpha_1)$ in P'. Hence, $g(A) \underset{G}{\Rightarrow} g(\alpha_1)$, as well.

Induction hypothesis: Assume the claim holds for all derivations $A \underset{G}{\Rightarrow}^k \alpha_k$ of length at most n, for some $n \geq 1$.

Induction step: Let $A = \alpha_0 \underset{G}{\Rightarrow} \alpha_1 \underset{G}{\Rightarrow} \cdots \underset{G}{\Rightarrow} \alpha_{n+1}$ be a derivation. According to the induction hypothesis, there exists a derivation

$$g(A = \alpha_0) \underset{G'}{\Rightarrow} g(\alpha_1) \underset{G'}{\Rightarrow} \cdots \underset{G'}{\Rightarrow} g(\alpha_n).$$

Next we show that from $\alpha_n \underset{G}{\Rightarrow} \alpha_{n+1}$ follows that $g(\alpha_n) \underset{G'}{\Rightarrow} g(\alpha_{n+1})$.

Assume $\alpha_n = \beta C \gamma$ and $\alpha_{n+1} = \beta \omega \gamma$, in which $\alpha_n, \beta, \gamma, \omega \in (N \cup \Sigma)^*$and $C \in N$. By the definition of direct derivation (Definition 4.6) grammar G has a production $C \rightarrow \omega$ in P. Since g is a morphism (Definition 4.15 (iv)), $g(\beta C \gamma) = g(\beta)g(C)g(\gamma)$ and $g(\beta \omega \gamma) = g(\beta)g(\omega)g(\gamma)$. Since g is a grammar morphism (Definition 4.16 (iii)), $g(C) \rightarrow g(\omega)$ is a production in P'. Thus, by the definition of direct derivation $g(\alpha_n) = g(\beta)g(C)g(\gamma) \underset{G'}{\Rightarrow} g(\beta)g(\omega)g(\gamma) = g(\alpha_{n+1})$. Hence, the derivation $g(A) \underset{G'}{\Rightarrow} g(\alpha_1) \underset{G'}{\Rightarrow} \cdots \underset{G'}{\Rightarrow} g(\alpha_{n+1})$ exists.

(ii) It is sufficient to show that for any $x \in L(G)$ we have $g(x) \in L(G')$. If $x \in L(G)$, then there exists a derivation $S \underset{G}{\Rightarrow}^k x$, with some $k \geq 1$. According to part (i), there exists a derivation $g(S) \underset{G'}{\Rightarrow}^k g(x)$, which means that $g(x) \in L(G')$.

(iii) We show that if $t \in \mathit{ptrees}(G)$, then $g(t)$ is a parse tree for G'. Since g is a morphism (Definition 4.15), the nodes of $g(t)$ are labeled with symbols of the set $N' \cup \Sigma' \cup \{\lambda\}$. If $S \in N$ is the label of the root of t, then $g(S) = S' \in N'$ is the label of the root of $g(t)$ (Definitions 4.16 (i) and 4.18). If B is the label of an interior node of $g(t)$, then $B \in N'$, whereas if l is the label of a leaf, then $l \in \Sigma' \cup \{\lambda\}$ (Definition 4.15 and 4.18).

If C is the label of an interior node m in t, and $Y_1, Y_2, \ldots, Y_k$ are the labels of the children of m in the order from left to right, then $g(C)$ is the label of an interior node m' in $g(t)$ and $g(Y_1), g(Y_2), \ldots, g(Y_k)$ are the labels of the children of m' in the order from the left to right (Definition 4.18). In order $g(t)$ to be a parse tree for G', the production $g(C) \rightarrow g(Y_1)g(Y_2)\cdots g(Y_k)$ has to be in P'. Since t is a parse tree for G, the production $C \rightarrow Y_1Y_2\cdots Y_k$ is in P (Definition 4.10). Since g is a grammar morphism (Definition 4.16 (iii)), $g(C) \rightarrow g(Y_1Y_2\cdots Y_k)$ is a production in P'. Since g is a morphism (Definition 4.15 (iv)), $g(Y_1Y_2\cdots Y_k) = g(Y_1)g(Y_2)\cdots g(Y_k)$. Hence, $g(C) \rightarrow g(Y_1)g(Y_2)\cdots g(Y_k)$ is in P'. Thus, $g(t)$ is a parse tree for G'.

We have shown that if $t \in \mathit{ptrees}(G)$, then $g(t) \in \mathit{ptrees}(G')$. Thus $g(\mathit{ptrees}(G)) \subseteq \mathit{ptrees}(G')$.

□

4.2 A grammar morphism mapping element types

In this section we define a simplified grammar morphism function h_G that maps the element types of a Document Type Definition (DTD) to the generic classes described in Section 3.2. The mapping is produced by an algorithm that contains decision rules based on the relationships of the elements only. Before an element type may be classified, all the element types it may contain must be classified.

The structure schema of an XML or SGML markup language is defined in a DTD as discussed in Chapters 2 and 3. A DTD is essentially an extended context-free grammar where the element types and their content models are specified[1]. A content model is the right-hand side of a production. It is a regular expression that defines either the structure of an element type in terms of other element types, or for the unstructured parts of the text, the type of data that can occur in an element type. A document instance is a parse tree of its grammar. For more information concerning SGML- and XML-grammars, see, e.g., [Woo87, BK93a, BK93b, BKW98, KW96, Kil99].

We assume that the classification method is given an arbitrary document type, which consists of a set of document instances and a DTD. If the DTD is not provided in the case of an XML document type, the names of the element types to be processed are collected from the document instances. Using the information extracted from the document instances we construct an extended context-free grammar denoted by $G = (N, \Sigma, P, S)$. This is done as follows. The set N is the set of nonterminals consisting of the element types of the DTD which also have element instances in the document instances. The set Σ consists of the terminals formed from the characters appearing in the texts of the document instances, and S is the start symbol, which is the label of the root of the XML (or SGML) document structure.

The set P of productions is defined as follows. For each element type $E \in N$ we form a set D_E and a production $E \rightarrow \alpha \in P$, where α is a regular expression. The set D_E contains the names of the elements which the element instances of E contain directly, the terminal symbol `text`, if text appears in some element instance of E, and λ, if all the element instances of E are empty. If $D_E = \{i_1, i_2, \ldots, i_k\}$, then the regular expression α is $(i_1 \cup i_2 \cup \cdots \cup i_k)^*$. During the mapping process of Algorithm 4.1, each original element name in D_E is replaced with the mapping result, i.e., the name of the generic class.

The extended context-free grammar obtained from the generic DTD (see Section 3.2) is denoted by $G' = (N', \Sigma', P', S')$. Let N' be the set of nonterminals, which consists of a subset of the predefined generic classes of the generic DTD. That is, $N' = \{$*Empty*, *String*, *StrCont*, *StrGrp*, *Para*, *ParaCont*, *ParaGrp*, *Section*, *SectGrp*, *Unknown*, *VirtDoc*$\}$. Let $\Sigma' = \Sigma$ be the set of terminals consisting of the characters appearing in the texts of the document instances. The start symbol S' is *VirtDoc*.

[1]SGML DTDs can have inclusions and exclusions, which go beyond extended context-free grammars. In this work, the SGML DTD of the given document type is used only for constructing the set of element types and for parsing the document instances in the tests. Some attribute information is also extracted from the DTD. Thus we do not have to address the issues of inclusions and exclusions here. On the other hand, XML does not allow inclusions and exclusions — the generic XML DTD is an extended context-free grammar.

The set P' of productions is defined as follows. For each element type $E' \in N'$ we define a set $D_{E'}$ and a production $E' \to \alpha' \in P'$, where α' is a regular expression. The set $D_{E'}$ and the regular expression α' are obtained from the content model of E' in the generic DTD. The set $D_{E'}$ contains all the nonterminals of the content model of E', the terminal symbol text, if #PCDATA is in the content model of E', and λ, if the content model of E' is defined as EMPTY. If $D_{E'} = \{i'_1, i'_2, \ldots, i'_k\}$, then the regular expression α' is $(i'_1 \cup i'_2 \cup \cdots \cup i'_{k'})^*$.

The set N' contains only a subset of the generic classes of the generic DTD to simplify and clarify the issues presented in this chapter. An element type belonging to the XHTML table structure (*table* as root), to the MathML structure (*math* as root), or to the RDF structure (*rdf* as root) is mapped to itself, i.e., the original structure remains the same. Also, the HTML element type *meta* is mapped to itself. The generic class *Title* is recognized in the refining phase after the basic mappings have been done. These specific cases are not considered in this chapter. The generic classes *Ref*, *Formula*, *FormCont*, *Fig*, *FigCont*, and *StrSect* are considered in the next section.

We use the following abbreviations in this section:

- Sm for "$\mathit{Empty} \cup \mathit{String} \cup \mathit{StrCont} \cup \mathit{StrGrp} \cup \mathit{Unknown}$",
- Pa for "$\mathit{Para} \cup \mathit{ParaCont} \cup \mathit{ParaGrp}$",
- Se for "$\mathit{Section} \cup \mathit{SectGrp}$",
- smalls for "$\mathit{Empty}, \mathit{String}, \mathit{StrCont}, \mathit{StrGrp}$",
- paras for "$\mathit{Para}, \mathit{ParaCont}, \mathit{ParaGrp}$", and
- sects for "$\mathit{Section}, \mathit{SectGrp}$".

In the following, we give the set P' containing the productions. The production for each element type E' can be obtained from the content model of E' in the generic DTD (or from the corresponding set $D_{E'}$).

$$
\begin{aligned}
&P' = \{\\
&\quad \mathit{VirtDoc} \to (\mathsf{text} \cup \mathsf{Sm} \cup \mathsf{Pa} \cup \mathsf{Se})^*,\\
&\quad \mathit{Unknown} \to (\mathsf{text} \cup \mathsf{Sm} \cup \mathsf{Pa} \cup \mathsf{Se})^*,\\
&\quad \mathit{SectGrp} \to (\mathsf{Sm} \cup \mathsf{Se})^*,\\
&\quad \mathit{Section} \to (\mathsf{text} \cup \mathsf{Sm} \cup \mathsf{Pa} \cup \mathsf{Se})^*,\\
&\\
&\quad \mathit{ParaGrp} \to (\mathsf{Sm} \cup \mathit{ParaCont} \cup \mathit{ParaGrp})^*,\\
&\quad \mathit{ParaCont} \to (\mathsf{Sm} \cup \mathsf{Pa})^*,
\end{aligned}
$$

```
<!ELEMENT cookbook  (title?, (editor* | author*), recipes+)>
<!ELEMENT recipes   (title?, (editor* | author*), recipe*)>
<!ATTLIST recipes   course (starter | maincour | dessert | baked |
                            barbq | drink) "maincour">
<!ELEMENT recipe    (title, author*, info?, ingreds+, instruct+)>
<!ELEMENT title     (#PCDATA)>
<!ELEMENT editor    (fname?, surname, address?, url?)>
<!ELEMENT author    (fname?, surname, address?, url?)>
<!ELEMENT fname     (#PCDATA)>
<!ELEMENT surname   (#PCDATA)>
<!ELEMENT address   (#PCDATA)>
<!ELEMENT url       (#PCDATA)>
<!ELEMENT info      (#PCDATA | p)*>
<!ELEMENT ingreds   (title?, item+)>
<!ELEMENT item      (measure*, ingred, descr?)>
<!ELEMENT measure   (#PCDATA)>
<!ELEMENT ingred    (#PCDATA)>
<!ELEMENT descr     (#PCDATA)>
<!ELEMENT instruct  (p+)>
<!ELEMENT p         (#PCDATA | emph)*>
<!ELEMENT emph      (#PCDATA)>
```

Figure 4.1: An XML DTD for cookbooks.

$$Para \to (\mathsf{text} \cup \mathsf{Sm} \cup \mathsf{Pa})^*,$$

$$StrGrp \to (\mathsf{Sm})^*,$$
$$StrCont \to (Empty \cup String \cup Unknown)^*,$$
$$String \to (\mathsf{text} \cup \mathsf{Sm} \cup \mathsf{Pa})^*,$$
$$Empty \to \lambda$$

$\}$.

Next we give an example of the notions used. The cookbook document type consists of the document instance given in Appendix B and the DTD presented in Figure 4.1. The context-free grammar $G = (N, \Sigma, P, S)$ is constructed from the document instance as follows. The set of nonterminals is N containing the element types that have element instances in the document instance[2], i.e., $N = \{$*cookbook*, *recipes*, *recipe*, *title*, *editor*, *author*, *fname*, *surname*, *url*, *ingreds*, *item*, *measure*, *ingred*, *descr*, *instruct*, $p\}$. The set of terminals is Σ containing the characters that may appear in the cookbooks, the start symbol is $S =$ *cookbook*, and the set of productions is P.

[2]Note that the element types *address* and *info* do not have element instances in Appendix B.

In the following, we provide the sets $D_{cookbook}$, D_{recipe}, D_{item}, D_p, and D_{emph}, which are obtained from the document instance of Appendix B.

$$\begin{aligned}
&D_{cookbook} = \{title, editor, recipes\};\\
&D_{recipe} = \{title, ingreds, instruct\};\\
&D_{item} = \{measure, incred, descr\};\\
&D_p = \{\mathsf{text}, emph\};\\
&D_{emph} = \{\mathsf{text}\}.
\end{aligned}$$

The regular expression for each element type E in the set of productions P can be obtained from the corresponding set D_E. We have

$$\begin{aligned}
P = \{&\\
&cookbook \rightarrow (title \cup editor \cup recipes)^*,\\
&recipe \rightarrow (title \cup ingreds \cup instruct)^*,\\
&item \rightarrow (measure \cup incred \cup descr)^*,\\
&p \rightarrow (\mathsf{text} \cup emph)^*,\\
&emph \rightarrow \mathsf{text},\\
&\ldots\\
\}.&
\end{aligned}$$

Definition 4.21 Let $G = (N, \Sigma, P, S)$ be the extended context-free grammar obtained from the document instances of an arbitrary document type as described, and let $G' = (N', \Sigma', P', S')$ be the extended context-free grammar obtained from the generic DTD. The function h_G is defined by the following rules, which define a mapping from the element types of a document type into the element types of the generic DTD.

(a) $h_G(S) = VirtDoc$.

(b) $h_G(\lambda) = \lambda$.

(c) For each $a \in \Sigma$, $h_G(a) = a$. Thus, $h_G(\mathsf{text}) = \mathsf{text}$.

(d) Algorithm 4.1 provides the decision rules for mapping an element type $E \in N$ to the generic class $h_G(E) = E' \in N'$.

(e) For an element type $E \in N$, for which $h_G(E)$ is undefined after processing all the previous rules, $h_G(E)$ is *Unknown*.

□

The following decision rules of Algorithm 4.1 define the mapping of the element types to the generic classes. A rule is denoted as R, conditions of a rule are denoted as c, the original element type is denoted as e, and the resulting generic class is denoted as g. The processing order of the rules is from the rule 1 to the rule 9. For each rule, element types are processed until no new mappings are found.

Algorithm 4.1

$M := \emptyset$;
for each rule $R_i : c_i \Rightarrow h_G(e) = g_i$, where $1 \leq i \leq 9$ **do**
  for each $E \in N \setminus M$ **do until** no new mappings to h_G occur
    if c_i **and** each item of D_E except E has been mapped **then**
      $h_G(E) := g_i$;
      for each $X \in N \setminus M$ **do**
        if $E \in D_X$ **and** $g_i \notin D_X$ **then**
          replace E with g_i in D_X;
        elseif $E \in D_X$ **and** $g_i \in D_X$ **then**
          delete E from D_X;
        endif
      endfor
      add E to M;
    endif
  endfor
endfor

Rules: Rule format: condition $c \Rightarrow$ mapping of nonterminal E.

1. $(D_E = \{\lambda\}) \Rightarrow h_G(E) = \mathit{Empty}$.
2. $((\mathsf{text} \in D_E) \wedge (\mathit{String} \notin D_E)) \Rightarrow h_G(E) = \mathit{String}$.
3. $((\{\mathit{String}, \mathit{Empty}\} \cap D_E \neq \emptyset) \wedge (\{\mathsf{text}, \mathit{StrCont}\} \cap D_E = \emptyset)) \Rightarrow h_G(E) = \mathit{StrCont}$.
4. $((\{\mathit{StrCont}, \mathit{StrGrp}\} \cap D_E \neq \emptyset) \wedge (\mathsf{text} \notin D_E)) \Rightarrow h_G(E) = \mathit{StrGrp}$.
5. $(\{\mathsf{text}, \mathsf{smalls}\} \cap D_E \neq \emptyset) \Rightarrow h_G(E) = \mathit{Para}$.
6. $((\mathit{Para} \in D_E) \wedge (\mathsf{text} \notin D_E)) \Rightarrow h_G(E) = \mathit{ParaCont}$.
7. $((\mathit{ParaCont} \in D_E) \wedge (\{\mathsf{text}, \mathit{Para}, \mathit{ParaGrp}\} \cap D_E = \emptyset)) \Rightarrow h_G(E) = \mathit{ParaGrp}$.
8. $(\{\mathsf{text}, \mathsf{paras}\} \cap D_E \neq \emptyset) \Rightarrow h_G(E) = \mathit{Section}$.

9. $((\{Section, SectGrp\} \cap D_E \neq \emptyset) \wedge (\{\mathsf{text}, \mathsf{paras}\} \cap D_E = \emptyset)) \Rightarrow h_G(E) = SectGrp$.

As an example, consider the result of Algorithm 4.1 for the cookbook document type (the document instance is in Appendix B and the DTD is in Figure 4.1). In the following, we give for each element type $E \in N$ the value of the function h_G and the final value of the set D_E. The initial value D_E^0 of the set D_E is also given, if it is different from the final value.

Rule 2: $D_{title} = D_{fname} = D_{surname} = D_{url} = D_{measure} = D_{ingred} = D_{descr} = D_{emph} = \{\mathsf{text}\}$.
$h_G(title) = h_G(fname) = h_G(surname) = h_G(url) = h_G(measure) = h_G(ingred) = h_G(descr) = h_G(emph) = String$.

Rule 3: $D^0_{item} = \{measure, incred, descr\}$;
$D^0_{editor} = \{fname, surname\}; D^0_{author} = \{surname, url\}$;
$D_{item} = D_{editor} = D_{author} = \{String\}$.
$h_G(item) = h_G(editor) = h_G(author) = StrCont$.

Rule 4: $D^0_{ingreds} = \{title, item\}$; $D_{ingreds} = \{String, StrCont\}$.
$h_G(ingreds) = StrGrp$.

Rule 5: $D^0_p = \{\mathsf{text}, emph\}$; $D_p = \{\mathsf{text}, String\}$.
$h_G(p) = Para$.

Rule 6: $D^0_{instruct} = \{p\}$; $D_{instruct} = \{Para\}$.
$h_G(instruct) = ParaCont$.

Rule 7: $D^0_{recipe} = \{title, ingreds, instruct\}$;
$D_{recipe} = \{String,\ StrGrp,\ ParaCont\}$.
$h_G(recipe) = ParaGrp$.

Rule 8: $D^0_{recipes} = \{author, recipe\}; D_{recipes} = \{StrCont,\ ParaGrp\}$.
$h_G(recipes) = Section$.

Rule 9: $D^0_{cookbook} = \{title, editor, recipes\}$;
$D_{cookbook} = \{String,\ StrCont,\ Section\}$.
$h_G(cookbook) = SectGrp$.

Because the element *cookbook* is the root of the cookbook structure, the start symbol S is *cookbook* in the context free grammar G. Thus, $h_G(cookbook) = VirtDoc$.

Theorem 4.22 Let $G = (N, \Sigma, P, S)$ be the extended context-free grammar obtained from the document instances of an arbitrary document type as described in the beginning of the section, and let $G' = (N', \Sigma', P', S')$ be the extended context-free grammar obtained from the generic DTD. The function h_G defined in Definition 4.21 is a grammar morphism from G to G'.

Proof We consider the conditions of Definition 4.16 for the function h_G one by one.

1. The first rule (a), $h_G(S) = \mathit{VirtDoc}$, satisfies the condition (i) of Definition 4.16, since $\mathit{VirtDoc} = S'$.

2. The condition (ii) of Definition 4.16 demands that h_G is a morphism. This is examined step by step:

   (i) The second rule (b), $h_G(\lambda) = \lambda$, is the same as the condition (i) of Definition 4.15.

   (ii) The third rule (c) defines that for all $a \in \Sigma$, $h_G(a) = a$. Since $\Sigma = \Sigma'$, the condition (ii) of Definition 4.15 is satisfied.

   (iii) The mapping from nonterminals of N into nonterminals of N' is done in the rules (a) and (d). Since the rule (e) associates $h_G(E) = \mathit{Unknown}$ for all the remaining unmapped items of N, finally each nonterminal in the set N will be associated with one nonterminal in the set N'. Hence, the condition (iii) of Definition 4.15 holds.

   (iv) Assume that P contains a production $A \to (X_1 \cup X_2 \cup \cdots \cup X_k)^*$, where $i = 1, 2, \ldots, k$, $X_i \in N \cup \{\mathsf{text}\} \cup \{\lambda\}$. Let $X_1 X_2 \cdots X_m \in L(A)$ for some $m \leq k$. We have to show that $h_G(X_1 X_2 \cdots X_m) = h_G(X_1) h_G(X_2) \cdots h_G(X_m)$. This claim holds, since the mapping function h_G for $X_j, 1 \leq j \leq m$, depends entirely on the items of the set D_{X_j} (Algorithm 4.1), and not on any $X_l, 1 \leq l \leq m$.

   As all the conditions of Definition 4.15 are satisfied, the presented function h_G is a morphism.

3. When applied to extended context-free grammars, the condition (iii) of Definition 4.16 requires that for each element type $E \in N$ and production $E \to \alpha \in P$ there is a generic class $h_G(E) \in N'$ and a production $h_G(E) \to \alpha' \in P'$ such that $L(h_G(\alpha)) \subseteq L(\alpha')$. When all the rules from (a) to (e) of h_G are applied, each of the productions of

P is associated with exactly one production of P' (see 2. (iii)). In the following, we show that if all the rules from (a) to (e) of Definition 4.21 are processed, then for each $E \to \alpha \in P$ there is $h_G(E) \to \alpha' \in P'$ such that $L(h_G(\alpha)) \subseteq L(\alpha')$.

$\boldsymbol{h_{\mathrm{G}}(S) = \mathit{VirtDoc}}$.

For $S' = \mathit{VirtDoc}$ and $\alpha' = (\mathsf{text} \cup \mathsf{Sm} \cup \mathsf{Pa} \cup \mathsf{Se})^*$ we have $S' \to \alpha' \in P'$ and $D_{\mathit{VirtDoc}} = \{\mathsf{text}, \mathsf{smalls}, \mathsf{paras}, \mathsf{sects}\}$. The set D_{VirtDoc} contains all the other nonterminals of N' except $\mathit{VirtDoc}$, the start symbol in G'. Since S is the start symbol in G, the only element instances of S are the root elements of the documents, which no element instances can contain. Since $h_G(S) = \mathit{VirtDoc}$, $\mathit{VirtDoc} \notin D_S$. Thus $D_S \subseteq D_{\mathit{VirtDoc}}$, and $L(h_G(\alpha)) \subseteq L(\alpha')$.

1. $(D_E = \{\lambda\}) \Rightarrow \boldsymbol{h_{\mathrm{G}}(E) = \mathit{Empty}}$.

   For $E' = \mathit{Empty}$ and $\alpha' = \lambda$ we have $E' \to \alpha' \in P'$ and $D_{\mathit{Empty}} = \{\lambda\}$. Condition $D_E = \{\lambda\}$ implies that $\alpha = \lambda$, as well. Hence $D_E = D_{\mathit{Empty}}$, and $L(h_G(\alpha)) = L(\alpha')$.

2. $((\mathsf{text} \in D_E) \wedge (\mathit{String} \notin D_E)) \Rightarrow \boldsymbol{h_{\mathrm{G}}(E) = \mathit{String}}$.

   For $E' = \mathit{String}$ and $\alpha' = (\mathsf{text} \cup \mathsf{Sm} \cup \mathsf{Pa})^*$ we have $E' \to \alpha' \in P'$ and $D_{\mathit{String}} = \{\mathsf{text}, \mathsf{smalls}, \mathsf{paras}\}$. Each item in D_E has to have a mapping before processing the rule 2. The previous rule 1 and the conditions of the rule 2 imply that $D_E \subseteq \{\mathit{Empty}, \mathsf{text}\}$. Hence $D_E \subseteq D_{\mathit{String}}$, and $L(h_G(\alpha)) \subseteq L(\alpha')$.

3. $((\{\mathit{String}, \mathit{Empty}\} \cap D_E \neq \emptyset) \wedge (\{\mathsf{text}, \mathit{StrCont}\} \cap D_E = \emptyset)) \Rightarrow$ $\boldsymbol{h_{\mathrm{G}}(E) = \mathit{StrCont}}$.

   For $E' = \mathit{StrCont}$ we have $D_{\mathit{StrCont}} = \{\mathit{String}, \mathit{Empty}, \mathit{Unknown}\}$. Each item in D_E has to have a mapping before processing the rule 3. The previous rules 1–2 and the conditions of the rule 3 imply that $D_E \subseteq \{\mathit{Empty}, \mathit{String}\}$. Hence $D_E \subseteq D_{\mathit{StrCont}}$, and $L(h_G(\alpha)) \subseteq L(\alpha')$.

4. $((\{\mathit{StrCont}, \mathit{StrGrp}\} \cap D_E \neq \emptyset) \wedge (\mathsf{text} \notin D_E)) \Rightarrow$ $\boldsymbol{h_{\mathrm{G}}(E) = \mathit{StrGrp}}$.

   For $E' = \mathit{StrGrp}$ we have $D_{\mathit{StrGrp}} = \{\mathsf{smalls}\}$. Each item in D_E has to have a mapping before processing the rule 4. The previous rules 1–3 and the conditions of the rule 4 imply that $D_E \subseteq \{Empty, String, StrCont, StrGrp\}$. Hence $D_E \subseteq D_{StrGrp}$, and $L(h_G(\alpha)) \subseteq L(\alpha')$.

5. $(\{\mathsf{text}, \mathsf{smalls}\} \cap D_E \neq \emptyset) \Rightarrow \boldsymbol{h_{\mathrm{G}}(E) = Para}$.

   For $E' = Para$ we have $D_{Para} = \{\mathsf{text}, \mathsf{smalls}, \mathsf{paras}\}$. Each item in D_E has to have a mapping before processing rule 5. The previous rules 1–4 and the conditions of the rule 5 imply that $D_E = \{Empty, \mathsf{text}, String, StrCont, StrGrp, Para\}$. Hence $D_E \subseteq D_{Para}$, and $L(h_G(\alpha)) \subseteq L(\alpha')$.

6. $((Para \in D_E) \wedge (\mathsf{text} \notin D_E)) \Rightarrow \boldsymbol{h_{\mathrm{G}}(E) = ParaCont}$.

   For $E' = ParaCont$ we have $D_{ParaCont} = \{\mathsf{smalls}, \mathsf{paras}\}$. Each item in D_E has to have a mapping before processing rule 6. The previous rules 1–5 and the conditions of the rule 6 imply that $D_E \subseteq \{Empty, String, StrCont, StrGrp, Para\}$. Hence $D_E \subseteq D_{ParaCont}$, and $L(h_G(\alpha)) \subseteq L(\alpha')$.

7. $((ParaCont \in D_E) \wedge (\{\mathsf{text}, Para, ParaGrp\} \cap D_E = \emptyset)) \Rightarrow \boldsymbol{h_{\mathrm{G}}(E) = ParaGrp}$.

   For $E' = ParaGrp$ we have $D_{ParaGrp} = \{\mathsf{smalls}, ParaCont, ParaGrp\}$. Each item in D_E has to have a mapping before processing rule 7. The previous rules 1–6 and the conditions of the rule 7 imply that $D_E \subseteq \{Empty, String, StrCont, StrGrp, ParaCont\}$. Hence $D_E \subseteq D_{ParaGrp}$, and $L(h_G(\alpha)) \subseteq L(\alpha')$.

8. $(\{\mathsf{text}, \mathsf{paras}\} \cap D_E \neq \emptyset) \Rightarrow \boldsymbol{h_{\mathrm{G}}(E) = Section}$.

   For $E' = Section$ we have $D_{Section} = \{\mathsf{text}, \mathsf{smalls}, \mathsf{paras}, \mathsf{sects}\}$. Each item in D_E has to have a mapping before processing rule 8. The previous rules 1–7 and the conditions of the rule 8 imply that $D_E \subseteq \{\mathsf{text}, \mathsf{smalls}, \mathsf{paras}, Section\}$. Hence $D_E \subseteq D_{Section}$, and $L(h_G(\alpha)) \subseteq L(\alpha')$.

9. $((\{Section, SectGrp\} \cap D_E \neq \emptyset) \wedge (\{\mathsf{text}, \mathsf{paras}\} \cap D_E = \emptyset)) \Rightarrow \boldsymbol{h_{\mathrm{G}}(E) = SectGrp}$.

   For $E' = SectGrp$ we have $D_{SectGrp} = \{\mathsf{smalls}, \mathsf{sects}\}$. Each item in D_E has to have a mapping before processing rule 9. The previous rules 1–8 and the conditions of the rule 9 imply that

$D_E \subseteq \{\textsf{smalls}, \mathit{Section}, \mathit{SectGrp}\}$. Hence $D_E \subseteq D_{SectGrp}$, and $L(h_G(\alpha)) \subseteq L(\alpha')$.

$\boldsymbol{h_{\mathrm{G}}(E) = Unknown}$.

For $E' = \mathit{Unknown}$ we have $D_{Unknown} = \{\textsf{text}, \textsf{smalls}, \textsf{paras}, \textsf{sects}\}$, since the content model of E' is ANY in the generic DTD. Now α' is the set of all possible combinations of items of $D_{Unknown}$, and $D_{Unknown}$ contains all the other nonterminals of N' except *VirtDoc*, which is the start symbol. Hence for all $h_G(E) = \mathit{Unknown}$, $D_E \subseteq D_{Unknown}$, and $L(h_G(\alpha)) \subseteq L(\alpha')$. All content models in the generic DTD, except those defined as EMPTY, contain the generic class *Unknown*. This implies that *Unknown* is allowed to be anywhere in the document tree.

It has been shown that after processing all the rules of h_G, for each $E \to \alpha \in P$ there is $h_G(E) \to \alpha' \in P'$ such that $L(h_G(\alpha)) \subseteq L(\alpha')$.

□

The grammar morphism function h_G is based purely on structural relationships between elements. However, the approach is not sufficient. To achieve a method classifying real-life structures properly, information of the average text lengths of the element instances and their subtrees is needed. For example, strings are typically short sequences of characters, and text paragraphs consist of strings and longer sequences of characters than strings. On the other hand, sections consist of text paragraphs and other sections, and the average text length of a section has to exceed a certain threshold. This information will be considered in the grammar morphism of the next section.

4.3 A grammar morphism using average text lengths

In this section we define a grammar morphism $\hat{h}_G$. It has the same decision rules as the element-type classification method informally presented in Section 3.2. Compared to the function h_G defined in the previous section, $\hat{h}_G$ uses the average text length of the element instances of an element type and the average text length of the subtrees of the element instances of an element type. Our tests show that the method of this section classifies real-life document structures as expected; this is discussed in detail in Chapter 5.

We define that the *text length of an element instance* is the number of characters the element instance contains directly, while the *text length of the subtree of an element instance* is the text length of the element instance plus the number of characters contained in the element instances in the subtree.

In addition to the notations of the previous section, we use the following notations introduced in Section 3.2:

- T_E is the *average text length of the element instances* of the element type E,
- R_E is the *average text length of the subtrees of the element instances* of the element type E,

The threshold maxstr is used for distinguishing strings from paragraphs. It is the maximum value for the average text length of the element instances of the generic class *String*. The threshold minsect is used for recognizing sections. It is the minimum value for the average text length of the subtrees of the element instances of the generic class *Section*.

The extended context-free grammar obtained from the generic DTD is denoted by $G'' = (N'', \Sigma'', P'', S'')$. The set of nonterminals denoted by N'' consists of a more comprehensive subset of the predefined generic classes than the set of nonterminals in the previous section, namely $N'' = \{$*Empty*, *Ref*, *Formula*, *FormCont*, *Fig*, *FigCont*, *String*, *StrCont*, *StrGrp*, *Para*, *ParaCont*, *ParaGrp*, *Section*, *SectGrp*, *StrSect*, *Unknown*, *VirtDoc*$\}$. Let $\Sigma'' = \Sigma$ be the set of terminals consisting of the characters appearing in the texts of the document instances. The start symbol S'' is *VirtDoc*.

The set of productions is P'' defined in the same way as the set of productions P' in the previous section. For each element type $E'' \in N''$ we define a set $D_{E''}$ and a production $E'' \to \alpha'' \in P''$, where α'' is a regular expression. The set $D_{E''}$ and the regular expression α'' are obtained from the content model of E'' in the generic DTD in the same manner as $D_{E'}$ and α' in the previous section.

We use the following abbreviations in this section:

- Sm for "$\mathit{Empty} \cup \mathit{Ref} \cup \mathit{Formula} \cup \mathit{FormCont} \cup \mathit{Fig} \cup \mathit{FigCont} \cup \mathit{String} \cup \mathit{StrCont} \cup \mathit{StrGrp} \cup \mathit{Unknown}$",
- Pa for "$\mathit{Para} \cup \mathit{ParaCont} \cup \mathit{ParaGrp}$",
- Se for "$\mathit{Section} \cup \mathit{SectGrp} \cup \mathit{StrSect}$",
- smalls for "*Empty*, *Ref*, *Formula*, *FormCont*, *Fig*, *FigCont*, *String*, *StrCont*, *StrGrp*",

- paras for "*Para*, *ParaCont*, *ParaGrp*", and
- sects for "*Section*, *SectGrp*, *StrSect*".

In the following we list the items of P'' which are different from the items of the set P' of productions in the previous section.

$$
\begin{aligned}
P'' = \{ & \\
& \mathit{StrSect} \to (\mathsf{Sm} \cup \mathsf{Se})^*, \\
& \mathit{StrGrp} \to (\mathit{Empty} \cup \mathit{Ref} \cup \mathit{String} \cup \mathit{StrCont} \cup \mathit{StrGrp})^*, \\
& \mathit{FigCont} \to (\mathit{Empty} \cup \mathit{Ref} \cup \mathit{Fig} \cup \mathit{FigCont} \cup \mathit{String} \cup \\
& \qquad\qquad \mathit{StrCont} \cup \mathit{StrGrp})^*, \\
& \mathit{Fig} \to \lambda, \\
& \mathit{FormCont} \to (\mathsf{Sm})^*, \\
& \mathit{Formula} \to \lambda, \\
& \mathit{Ref} \to \lambda, \\
& \ldots \\
\}. &
\end{aligned}
$$

Definition 4.23 Let $G = (N, \Sigma, P, S)$ be the extended context-free grammar obtained from the document instances of an arbitrary document type as described in the previous section, and let $G'' = (N'', \Sigma'', P'', S'')$ be the extended context-free grammar obtained from the generic DTD. The function $\hat{h}_G$ using the average text lengths of the element instances and their subtrees is defined by the following rules, which define a mapping from the element types of a document type into element types of the generic DTD.

(a) $\hat{h}_G(S) = \mathit{VirtDoc}$.

(b) $\hat{h}_G(\lambda) = \lambda$.

(c) For each $a \in \Sigma$, $\hat{h}_G(a) = a$. Thus, $\hat{h}_G(\mathsf{text}) = \mathsf{text}$.

(d) Algorithm 4.2 provides the decision rules for mapping an element type $E \in N$ to a generic class $\hat{h}_G(E) = E'' \in N''$.

(e) If there are such element types $E \in N$, for which $\hat{h}_G(E)$ is undefined after processing all the previous rules, then $\hat{h}_G(E) = \mathit{Unknown}$.

□

The following decision rules of Algorithm 4.2 define the mapping of the element types to the generic classes. A rule is denoted as R, conditions of a rule are denoted as c, the original element type is denoted as e, and the resulting generic class is denoted as g. The processing order of the rules is from the rule 1 to the rule 12. For each rule, element types are processed until no new mappings are found.

Algorithm 4.2

```
M := ∅;
for each rule R_i : c_i ⇒ ĥ_G(e) = g_i, where 1 ≤ i ≤ 12 do
    for each E ∈ N \ M do until no new mappings to ĥ_G occur
        if c_i then
            ĥ_G(E) := g_i;
            for each X ∈ N do
                if E ∈ D_X and g_i ∉ D_X then
                    replace E with g_i in D_X;
                elseif E ∈ D_X and g_i ∈ D_X then
                    delete E from D_X;
                endif
            endfor
            add E to M;
        endif
    endfor
endfor
```

Rules: $c \Rightarrow \hat{h}_G(e) = g$

1. $((D_E = \lambda) \wedge (T_E = 0)) \Rightarrow \hat{h}_G(E) \in \{\mathit{Empty}, \mathit{Ref}, \mathit{Fig}, \mathit{Formula}\}$.
2. $((0 < T_E \leq \mathsf{maxstr}) \wedge (R_E < \mathsf{minsect}) \wedge (\forall e \in D_E, R_e < \mathsf{minsect})) \Rightarrow \hat{h}_G(E) = \mathit{String}$.
3. $((\mathsf{maxstr} < T_E < \mathsf{minsect}) \wedge (R_E < \mathsf{minsect}) \wedge (\forall e \in D_E, R_e < \mathsf{minsect})) \Rightarrow \hat{h}_G(E) = \mathit{Para}$.
4. $((\mathit{Para} \in D_E) \wedge (T_E = 0) \wedge (R_E < \mathsf{minsect}) \wedge (\forall e \in D_E, R_e < \mathsf{minsect})) \Rightarrow \hat{h}_G(E) = \mathit{ParaCont}$.
5. $((\mathit{Para} \notin D_E) \wedge (\{\mathit{ParaCont}, \mathit{ParaGrp}\} \cap D_E \neq \emptyset) \wedge (T_E = 0) \wedge (R_E < \mathsf{minsect}) \wedge (\forall e \in D_E, R_e < \mathsf{minsect})) \Rightarrow \hat{h}_G(E) = \mathit{ParaGrp}$.
6. $((\{\mathsf{paras}\} \cap D_E = \emptyset) \wedge (\{\mathit{Formula}, \mathit{FormCont}\} \cap D_E \neq \emptyset) \wedge (T_E = 0) \wedge (R_E < \mathsf{minsect}) \wedge (\forall e \in D_E, R_e < \mathsf{minsect})) \Rightarrow \hat{h}_G(E) = \mathit{FormCont}$.

7. $((\{Formula, FormCont, \mathsf{paras}\} \cap D_E = \emptyset) \wedge (\{Fig, FigCont\} \cap D_E \neq \emptyset) \wedge (T_E = 0) \wedge (R_E < \mathsf{minsect}) \wedge (\forall e \in D_E, R_e < \mathsf{minsect})) \Rightarrow \hat{h}_G(E) = FigCont$.

8. $((D_E \subseteq \{String, Empty, Ref\}) \wedge (T_E = 0) \wedge (R_E < \mathsf{minsect})) \Rightarrow \hat{h}_G(E) = StrCont$.

9. $((\{Formula,\ FormCont,\ Fig,\ FigCont,\ \mathsf{paras}\} \cap D_E = \emptyset) \wedge (\{StrCont, StrGrp\} \cap D_E \neq \emptyset) \wedge (T_E = 0) \wedge (R_E < \mathsf{minsect}) \wedge (\forall e \in D_E, R_e < \mathsf{minsect})) \Rightarrow \hat{h}_G(E) = StrGrp$.

10. $(((T_E > 0) \wedge (R_E \geq \mathsf{minsect})) \vee ((\{\mathsf{paras}\} \cap D_E \neq \emptyset) \wedge (T_E = 0) \wedge (R_E \geq \mathsf{minsect}))) \Rightarrow \hat{h}_G(E) = Section$.

11. $((\{\mathsf{paras}\} \cap D_E = \emptyset) \wedge (\{Section, SectGrp\} \cap D_E \neq \emptyset) \wedge (T_E = 0) \wedge (R_E \geq \mathsf{minsect})) \Rightarrow \hat{h}_G(E) = SectGrp$.

12. $((\{\mathsf{paras}, Section, SectGrp\} \cap D_E = \emptyset) \wedge (T_E = 0) \wedge (R_E \geq \mathsf{minsect})) \Rightarrow \hat{h}_G(E) = StrSect$.

Theorem 4.24 Let $G = (N, \Sigma, P, S)$ be the extended context-free grammar obtained from the document instances of an arbitrary document type as described in the previous section, and let $G'' = (N'', \Sigma'', P'', S'')$ be the extended context-free grammar obtained from the generic DTD. The function $\hat{h}_G$ in Definition 4.23 is a grammar morphism from G to G''.

Proof The conditions of Definition 4.16 for the function $\hat{h}_G$ are considered one by one.

1. – 2. The conditions (i)–(ii) of Definition 4.16 are proved in the same manner as in Theorem 4.22.

3. In the following we show that if all the rules of Definition 4.23 are processed, then for each $E \to \alpha \in P$ there is $\hat{h}_G(E) \to \alpha'' \in P''$ such that $L(\hat{h}_G(\alpha)) \subseteq L(\alpha'')$.

   $\boldsymbol{\hat{h}_{\mathrm{G}}(E = S) = VirtDoc}$:

   This is proved in the same manner as in Theorem 4.22.

   1. $((D_E = \lambda) \wedge (T_E = 0)) \Rightarrow \boldsymbol{\hat{h}_{\mathrm{G}}(E) \in \{Empty, Ref, Fig, Formula\}}$.

      This is proved in the same manner as in Theorem 4.22.

2. $((0 < T_E \leq \mathsf{maxstr}) \wedge (R_E < \mathsf{minsect}) \wedge (\forall e \in D_E, R_e < \mathsf{minsect})) \Rightarrow \boldsymbol{\hat{h}_{\mathrm{G}}(E) = String}$.

   For $E'' = \mathit{String}$ and $\alpha'' = (\mathsf{text} \cup \mathsf{smalls} \cup \mathsf{paras})^*$ we have $E'' \to \alpha'' \in P''$ and $D_{\mathit{String}} = \{\mathsf{text}, \mathsf{smalls}, \mathsf{paras}\}$. Condition $\forall e \in D_E, R_e < \mathsf{minsect}$ implies that $\{\mathsf{sects}\} \cap D_E = \emptyset$, because $R_{\mathsf{sects}} \geq \mathsf{minsect}$. Condition $\mathit{VirtDoc} = S'' = \hat{h}_G(S)$ implies that $\mathit{VirtDoc} \notin D_E$. Hence, $\{\mathsf{sects}, \mathit{VirtDoc}\} \cap D_E = \emptyset$, and $D_E \subseteq D_{\mathit{String}}$. Thus $L(h_G(\alpha)) \subseteq L(\alpha'')$.

3. $((\mathsf{maxstr} < T_E < \mathsf{minsect}) \wedge (R_E < \mathsf{minsect}) \wedge (\forall e \in D_E, R_e < \mathsf{minsect})) \Rightarrow \boldsymbol{\hat{h}_{\mathrm{G}}(E) = Para}$.

   The proof for *Para* is the same as for *String* (see item 2.).

4. $((\mathit{Para} \in D_E) \wedge (T_E = 0) \wedge (R_E < \mathsf{minsect}) \wedge (\forall e \in D_E, R_e < \mathsf{minsect})) \Rightarrow \boldsymbol{\hat{h}_{\mathrm{G}}(E) = ParaCont}$.

   For $E'' = \mathit{ParaCont}$ we have $D_{\mathit{ParaCont}} = \{\mathsf{smalls}, \mathsf{paras}\}$. Condition $T_E = 0$ implies that $\mathsf{text} \notin D_E$. Since $\forall e \in D_E, R_e < \mathsf{minsect}$ and $\mathit{VirtDoc} = \hat{h}_G(S)$, $\{\mathsf{sects}, \mathit{VirtDoc}\} \cap D_E = \emptyset$. Hence $D_E \subseteq D_{\mathit{ParaCont}}$, and $L(\hat{h}_G(\alpha)) \subseteq L(\alpha'')$.

5. $((\mathit{Para} \notin D_E) \wedge (\{\mathit{ParaCont}, \mathit{ParaGrp}\} \cap D_E \neq \emptyset) \wedge (T_E = 0) \wedge (R_E < \mathsf{minsect}) \wedge (\forall e \in D_E, R_e < \mathsf{minsect})) \Rightarrow \boldsymbol{\hat{h}_{\mathrm{G}}(E) = ParaGrp}$.

   For $E'' = \mathit{ParaGrp}$ we have $D_{\mathit{ParaGrp}} = \{\mathsf{smalls}, \mathit{ParaCont}, \mathit{ParaGrp}\}$. Conditions $T_E = 0$ and $\mathit{Para} \notin D_E$ imply that $\{\mathsf{text}, \mathit{Para}\} \cap D_E = \emptyset$. Since $\forall e \in D_E, R_e < \mathsf{minsect}$ and $\mathit{VirtDoc} = \hat{h}_G(S)$, $\{\mathsf{sects}, \mathit{VirtDoc}\} \cap D_E = \emptyset$. Hence $D_E \subseteq D_{\mathit{ParaGrp}}$, and $L(\hat{h}_G(\alpha)) \subseteq L(\alpha'')$.

6. $((\{\mathsf{paras}\} \cap D_E = \emptyset) \wedge (\{\mathit{Formula}, \mathit{FormCont}\} \cap D_E \neq \emptyset) \wedge (T_E = 0) \wedge (R_E < \mathsf{minsect}) \wedge (\forall e \in D_E, R_e < \mathsf{minsect})) \Rightarrow \boldsymbol{\hat{h}_{\mathrm{G}}(E) = FormCont}$.

   For $E'' = \mathit{FormCont}$ we have $D_{\mathit{FormCont}} = \{\mathsf{smalls}\}$. Conditions $T_E = 0$ and $\mathsf{paras} \cap D_E = \emptyset$ imply that $\{\mathsf{text}, \mathsf{paras}\} \cap D_E = \emptyset$. Since $\forall e \in D_E, R_e < \mathsf{minsect}$ and $\mathit{VirtDoc} = \hat{h}_G(S)$, $\{\mathsf{sects}, \mathit{VirtDoc}\} \cap D_E = \emptyset$. Hence $D_E \subseteq D_{\mathit{FormCont}}$, and $L(\hat{h}_G(\alpha)) \subseteq L(\alpha'')$.

7. $((\{Formula, FormCont, \mathsf{paras}\} \cap D_E = \emptyset) \wedge (\{Fig, FigCont\} \cap D_E \neq \emptyset) \wedge (T_E = 0) \wedge (R_E < \mathsf{minsect}) \wedge (\forall e \in D_E, R_e < \mathsf{minsect})) \Rightarrow \boldsymbol{\hat{h}_G(E) = FigCont}$.

   For $E'' = FigCont$ we have $D_{FigCont} = \{Empty, Ref, Fig, FigCont, String, StrCont, StrGrp\}$. Conditions $T_E = 0$ and $\{Formula, FormCont, \mathsf{paras}\} \cap D_E = \emptyset$ imply that $\{\mathsf{text}, Formula, FormCont, \mathsf{paras}\} \cap D_E = \emptyset$. Since $\forall e \in D_E, R_e < \mathsf{minsect}$ and $VirtDoc = \hat{h}_G(S)$, $\{\mathsf{sects}, VirtDoc\} \cap D_E = \emptyset$. Hence $D_E \subseteq D_{FigCont}$, and $L(\hat{h}_G(\alpha)) \subseteq L(\alpha'')$.

8. $((D_E \subseteq \{String, Empty, Ref\}) \wedge (T_E = 0) \wedge (R_E < \mathsf{minsect})) \Rightarrow \boldsymbol{\hat{h}_G(E) = StrCont}$.

   For $E'' = StrCont$ we have $D_{StrCont} = \{Empty, Ref, String\}$. Conditions $D_E \subseteq \{String, Empty, Ref\}$ and $T_E = 0$ restrict that $D_E = \{String, Empty, Ref\}$. Hence $D_E \subseteq D_{StrCont}$, and $L(\hat{h}_G(\alpha)) \subseteq L(\alpha'')$.

9. $((\{Formula, FormCont, Fig, FigCont, \mathsf{paras}\} \cap D_E = \emptyset) \wedge (\{StrCont, StrGrp\} \cap D_E \neq \emptyset) \wedge (T_E = 0) \wedge (R_E < \mathsf{minsect}) \wedge (\forall e \in D_E, R_e < \mathsf{minsect})) \Rightarrow \boldsymbol{\hat{h}_G(E) = StrGrp}$.

   For $E'' = StrGrp$ we have $D_{StrGrp} = \{Empty, Ref, String, StrCont, StrGrp\}$. Conditions $T_E = 0$ and $\{Formula, FormCont, Fig, FigCont, \mathsf{paras}\} \cap D_E = \emptyset$ imply that $\{\mathsf{text}, Formula, FormCont, Fig, FigCont, \mathsf{paras}\} \cap D_E = \emptyset$. Since $\forall e \in D_E, R_e < \mathsf{minsect}$ and $VirtDoc = \hat{h}_G(S)$, $\{\mathsf{sects}, VirtDoc\} \cap D_E = \emptyset$. Hence $D_E \subseteq D_{StrGrp}$, and $L(\hat{h}_G(\alpha)) \subseteq L(\alpha'')$.

10. $(((T_E > 0) \wedge (R_E \geq \mathsf{minsect})) \vee ((\mathsf{paras} \cap D_E \neq \emptyset) \wedge (T_E = 0) \wedge (R_E \geq \mathsf{minsect}))) \Rightarrow \boldsymbol{\hat{h}_G(E) = Section}$.

    For $E'' = Section$ we have $D_{Section} = \{\mathsf{text}, \mathsf{smalls}, \mathsf{paras}, \mathsf{sects}\}$. Since $VirtDoc = \hat{h}_G(S)$, $VirtDoc \notin D_E$. Hence $D_E \subseteq D_{Section}$, and $L(\hat{h}_G(\alpha)) = L(\alpha'')$.

11. $((\mathsf{paras} \cap D_E = \emptyset) \wedge (\{Section, SectGrp\} \cap D_E \neq \emptyset) \wedge (T_E = 0) \wedge (R_E \geq \mathsf{minsect})) \Rightarrow \boldsymbol{\hat{h}_G(E) = SectGrp}$.

    For $E'' = SectGrp$ we have $D_{SectGrp} = \{\mathsf{smalls}, \mathsf{sects}\}$. Conditions $T_E = 0$ and $\{\mathsf{paras}\} \cap D_E = \emptyset$ imply that $\{\mathsf{text}, \mathsf{paras}\} \cap D_E = \emptyset$. Since $VirtDoc = \hat{h}_G(S)$, $VirtDoc \notin D_E$. Hence

$D_E \subseteq D_{SectGrp}$, and $L(\hat{h}_G(\alpha)) \subseteq L(\alpha'')$.

12. $((\{\mathsf{paras}, \mathit{Section}, \mathit{SectGrp}\} \cap D_E = \emptyset) \wedge (T_E = 0) \wedge (R_E \geq \mathsf{minsect})) \Rightarrow \boldsymbol{\hat{h}_{\mathrm{G}}(E) = \mathit{StrSect}}$.

   For $E'' = \mathit{StrSect}$ we have $D_{\mathit{StrSect}} = \{\mathsf{smalls}, \mathit{StrSect}\}$. Conditions $T_E = 0$ and $\{\mathsf{paras}, \mathit{Section}, \mathit{SectGrp}\} \cap D_E = \emptyset$ imply that $\{\mathsf{text}, \mathsf{paras}, \mathit{Section}, \mathit{SectGrp}\} \cap D_E = \emptyset$. Since $\mathit{VirtDoc} = \hat{h}_G(S)$, $\mathit{VirtDoc} \notin D_E$. Hence $D_E \subseteq D_{\mathit{StrSect}}$, and $L(\hat{h}_G(\alpha)) \subseteq L(\alpha'')$.

$\boldsymbol{h_{\mathrm{G}}(E) = \mathit{Unknown}}$.

This is proved in the same manner as in Theorem 4.22.

We have shown that after processing all the rules of $\hat{h}_G$ there is for each $E \to \alpha \in P$ associated $\hat{h}_G(E) \to \alpha'' \in P''$ such that $L(\hat{h}_G(\alpha)) \subseteq L(\alpha'')$.

□

As an example, consider the result of Algorithm 4.2 for the cookbook document type. It is assumed that the threshold maxstr in Algorithm 4.2 is set to 60 characters and the threshold minsect is set to 2 000 characters. For each element type E the values of T_E and R_E in the document instance of Appendix B are given in Table 4.1. In the following, we give for each element type $E \in N$ the value of the function $\hat{h}_G$ and the final value of the set D_E. The initial value D_E^0 of the set D_E is also given, if it is different than the final value.

Rule 2: $D_{\mathit{title}} = D_{\mathit{fname}} = D_{\mathit{surname}} = D_{\mathit{url}} = D_{\mathit{measure}} = D_{\mathit{ingred}} = D_{\mathit{descr}} = D_{\mathit{emph}} = \{\mathsf{text}\}$.
$\hat{h}_G(\mathit{title}) = \hat{h}_G(\mathit{fname}) = \hat{h}_G(\mathit{surname}) = \hat{h}_G(\mathit{url}) = \hat{h}_G(\mathit{measure}) = \hat{h}_G(\mathit{ingred}) = \hat{h}_G(\mathit{descr}) = \hat{h}_G(\mathit{emph}) = \mathit{String}$.

Rule 3: $D_p^0 = \{\mathsf{text}, \mathit{emph}\}$; $D_p = \{\mathsf{text}, \mathit{String}\}$.
$\hat{h}_G(p) = \mathit{Para}$.

Rule 4: $D_{\mathit{instruct}}^0 = \{p\}$; $D_{\mathit{instruct}} = \{\mathit{Para}\}$.
$\hat{h}_G(\mathit{instruct}) = \mathit{ParaCont}$.

Rule 5: $D_{\mathit{recipe}}^0 = \{\mathit{title}, \mathit{ingreds}, \mathit{instruct}\}$;
$D_{\mathit{recipe}} = \{\mathit{String}, \mathit{StrGrp}, \mathit{ParaCont}\}$.

$E \in N$	T_E	R_E	Alg. 4.2 with or without *emph*	Alg. 4.1 with *emph*	Alg. 4.1 without *emph*
cookbook	0	34 565	*SectGrp*	*SectGrp*	*StrGrp**
recipes	0	8 626	*Section*	*Section*	*StrGrp**
recipe	0	1 105	*ParaGrp*	*ParaGrp*	*StrGrp**
title	10	10	*String*	*String*	*String*
editor	0	16	*StrCont*	*StrCont*	*StrCont*
author	0	60	*StrCont*	*StrCont*	*StrCont*
fname	7	7	*String*	*String*	*String*
surname	8	8	*String*	*String*	*String*
url	52	52	*String*	*String*	*String*
ingreds	0	203	*StrGrp*	*StrGrp*	*StrGrp*
item	0	23	*StrCont*	*StrCont*	*StrCont*
measure	5	5	*String*	*String*	*String*
ingred	11	11	*String*	*String*	*String*
descr	15	15	*String*	*String*	*String*
instruct	0	846	*ParaCont*	*ParaCont*	*StrCont**
p	294	294	*Para*	*Para*	*String**
emph	15	15	*String*	*String*	—

Table 4.1: Text lengths and mappings for element types of a cookbook.

$\hat{h}_G(recipe) = ParaGrp.$

Rule 8: $D^0_{item} = \{measure, incred, descr\};$
$D^0_{editor} = \{fname, surname\}; D^0_{author} = \{surname, url\};$
$D_{item} = D_{editor} = D_{author} = \{String\}.$
$\hat{h}_G(item) = \hat{h}_G(editor) = \hat{h}_G(author) = StrCont.$

Rule 9: $D^0_{ingreds} = \{title, item\}$; $D_{ingreds} = \{String,\ StrCont\}.$
$\hat{h}_G(ingreds) = StrGrp.$

Rule 10: $D^0_{recipes} = \{author, recipe\}$; $D_{recipes} = \{StrCont,\ ParaGrp\}.$
$\hat{h}_G(recipes) = Section.$

Rule 11: $D^0_{cookbook} = \{title, editor, recipes\};$
$D_{cookbook} = \{String,\ StrCont,\ Section\}.$
$\hat{h}_G(cookbook) = SectGrp.$

Because the element *cookbook* is the root of the cookbook structure, the start symbol S is *cookbook* in the context free grammar G. Hence,

$\hat{h}_G(cookbook) = VirtDoc$.

As can be seen from Table 4.1, the mapping results of Algorithms 4.1 and 4.2 are the same in this cookbook example. Since the average text lengths affect the mapping produced by Algorithm 4.2, the results are more advanced than the results of Algorithm 4.1 in general. For instance, if the element instances of p did not contain instances of *emph*, Algorithm 4.1 would map the element type p to *String*; see the sixth column of Table 4.1, where the different mappings are marked with "*". In this case, Algorithm 4.2 would map p to *Para*, since T_p is 294 characters. Clearly, the latter mapping result is better than the former; see the elements p in Appendix B. If Algorithm 4.1 mapped p to *String*, it would affect to the other mapping results of the cookbook example, as well. For instance, *instruct* would be mapped to *StrCont* instead of *ParaCont*.

Chapter 5

Applying element-type classification

This chapter describes the results of applying the element-type classification method presented in Chapters 3 and 4. We have tested the method on various kinds of document structures, and the results are discussed in detail in Section 5.1. In Section 5.2 we consider some implementation issues, the mapping of attributes, and finally, how architectural forms defined in the HyTime standard [ISO92, ISO97] can be used to represent the generic DTD and the mapping of element types to the generic classes.

5.1 Results of identifying metastructures

For implementing the element-type classification method we used the languages TranSID [JKL97] and perl [WS91]. TranSID is a transformation language that models documents as tree structures and allows any part of the document tree to be accessed. TranSID is used to collect all the information from the document instances, whereas the calculation of average information and the decision process is implemented in perl.

In the first phase of classifying element types, the TranSID query of Figure 5.1 is executed to the document instances of a document type. The query produces for every element instance in a document instance the name of the element (Figure 5.1 row 2), the names of the children (row 3), the length of the textual data in the element instance (row 7), and finally, the length of the textual data in the whole subtree of the element instance (row 8). Then a perl program computes average information from the output of the executed query for every element type in the document type: the names of the elements which the element instances of the element type contain directly, the average text length of the element instances of the element type, and the average text length of the subtrees of the element instances

```
query (source,source.descendants).elements.map(true;
     "<",this.name,">\n"
     , "\tChildren:",this.children.elements.map(
             true;
             " ",this.name
     ),"\n"
     , "\tTextlength: ",this.children.data.strcat.strlen,"\n"
     , "\tSubtree: ",this.descendants.data.strcat.strlen,"\n"
);
```

Figure 5.1: A TranSID query on document instances.

of the element type. Finally, the decision rules presented in Sections 3.2 and 4.3 are applied to the gathered information of the element types. The values for the thresholds maxstr and minsect of the classification method were set to 60 and 2 000 characters in the tests. The following sections contain some discussion about suitable values for maxstr and minsect.

The element-type classification method has been tested on the following material: textbooks on control engineering in Finnish, a multilingual version of a paper machine manual and some other manuals, a large collection of Finnish statutes, a Finnish encyclopedia, articles and news articles, as well as a large number of novels, books of poems, and plays of Shakespeare. The structures of these documents vary widely. However, most of the material has a book-like structure, such as the following publicly available and widely used document types: the TEI DTD and the TEI Lite DTD [TEI94] published by the Text Encoding Initiative, the Book DTD of ISO 12083 DTD [ISO94], and the DocBook DTD [OAS91].

The following sections present the results for each category of material. Evaluation of a mapping result requires understanding of the specific document type. Even though a plain element structure does not provide any semantics, public document types typically include documentation describing the meaning and use of element types. The generic classes define typical logical structures of electronic documents, and for a human who is familiar with the original structure and the generic classes, it is quite easy to assess a mapping result. Especially a clearly failed mapping, such as a section object mapped to a paragraph or a string object, is easy to identify. In the following, we point out the unclear or failed mappings found, and if nothing else is mentioned, it can be assumed that the mapping result is successful in terms of the generic classes.

Another way to evaluate the mappings is to format the documents using formatting instructions generated based on the mapping results. We have imported a large portion of the material tested into our assembly system SAW described in Chapter 6. SAW transforms the document content to be browsed into HTML format on the fly. The transformation is based on the information of the generic classes only. In most cases, the HTML formatting is successful and the resulting documents are readable, even in cases where the original structure is rich and complicated.

5.1.1 Results of the textbooks on control engineering

We first describe the test results for three textbooks. The textbooks are also used in the description of the textbook tailoring system in Chapter 6. The classification result for each element type in the textbook structure can be found in Tables 5.1, 5.2, and 5.3. The first column of the table gives the name of the element type, the second column gives the generic class the element type is mapped to, the third column gives the average text length of the element instances of the element type, the fourth column gives the average text length of the subtrees of the element instances of the element type, and finally, the fifth column gives the names of the elements which the element instances of the element type contain directly. To illustrate certain general aspects, we compare some results on the textbooks with the results on the other document types.

The total size of the textbooks is 2.669 megabytes, where the proportion of the text without tags is 0.681 megabytes. The total space required by the tags and attribute information inside the tags is, hence, 1.988 megabytes, which is almost three times as much as the actual size of the text. The reason for this is that the textbooks contain a considerable number of formulas and other figures, which have a large amount of attribute information, such as *file*, *align*, *angle*, *bloffset*, *cropped*, and *float*. An excerpt of tagged text with attributes is presented in Figure 5.2. The size of the output of the TranSID query (Figure 5.1) on the textbooks is 0.988 megabytes.

The size information is also presented in Table 5.6, which lists all the tested collections, the sizes of the document instances, the size of the text without tags, the size of the TranSID query result, and the tested documents.

The basic structure of the control engineering textbooks is the Book DTD of ISO 12083 DTD extended with element types for exercises, answers, examples, definitions, clauses, formulas, and computer code. The basic skeleton for a textbook consists of:

- the element type *front* classified as *Section*, which contains the cover information and the element types *preface* and *foreword*,

```
...
<item><p id = "C7p65" level = "c" content = "aa"
branch = "unspec">Kompleksikonjugaatti napapari aidosti
imaginaariakselin vasemmalla puolella:</p>
<p id = "C7p66" level = "c" content = "aa" branch = "unspec">esim.
 <eq id = "C7df143" level = "c" content = "bb" branch = "unspec"
    number = "no"><eqbody file = "eqbod535.eqn" align = "acenter"
    angle = "0.000" bloffset = "-0.153in" cropped = "0" float = "0"
    height = "0.412in" nsoffset = "0.000in" position = "inline"
    width = "3.109in"></eq><eq id = "C7df144" level = "c" content = "bb"
    branch = "unspec" number = "no"><eqbody file = "eqbod536.eqn"
    align = "acenter" angle = "0.000" bloffset = "-0.415in"
    cropped = "0" float = "0" height = "0.674in" nsoffset = "0.000in"
    position = "inline" width = "3.578in"></eq> impulssivaste:
 <eq id = "C7df145" level = "c" content = "bb" branch = "unspec"
    number = "no"><eqbody file = "eqbod537.eqn" align = "acenter"
    angle = "0.000" bloffset = "-0.384in" cropped = "1" float = "0"
    height = "0.889in" nsoffset = "0.000in" position = "inline"
    width = "2.555in"></eq></p></item>
<figgrp id = "C7fg8" level = "c" content = "aa" branch = "unspec"><fig
    file = "fig538.wmf" align = "acenter" angle = "0.000"
    bloffset = "-2.228in" cropped = "1" float = "0" height = "8.790in"
    nsoffset = "0.000in" position = "below" width = "5.932in"
    impang = "0.000" impby = "copy" impsize = "5.799in 8.599in"
    sideways = "0" xoffset = "0.066in" yoffset = "0.096in">
<caption id = "C7ft9">Siirtofunktiomallin napojen vaikutus sen
stabiilisuuteen, vasemmalla siirtofunktio, oikealla
impulssivaste.</caption></figgrp>
...
```

Figure 5.2: An excerpt from a textbook in Finnish.

- the element type *body* classified as *SectGrp*, which is the actual book content containing the *chapter–section* hierarchy,
- the element type *appmat* classified as *SectGrp*, which is the appendix material containing the *appendix–section* hierarchy, and
- the element type *back* classified as *Section*, which contains the bibliography (*biblist*) and after word (*afterwrd*).

As can be seen from Table 5.1, the element types mapped to *Section* are section type objects of the textbook structure, and the element types mapped to *SectGrp* are containers for them. To compare mappings to the generic class *Section* in different document types, the mappings of the document types of Appendix C are presented in Table 5.4. The names of typical sections are *front*, *preface*, *chapter*, *section*, *sect1*, *sect2*, *body*, *div*, *div1*, *div2*, *div3*, *div4*, and *appendix*. The average text length of the subtrees of a section type object varies between 2 004 and 85 721 characters; the threshold minsect was set to 2 000 characters. The names of the sections in

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
ophbooks	*VirtDoc*	0	226988	book
book	*SectGrp*	0	226988	front body appmat back
body	*SectGrp*	0	203815	chapter
appmat	*SectGrp*	0	26622	appendix
front	*Section*	0	2903	authgrp titlegrp corpauth pubfront foreword ack figgrp preface toc
chapter	*Section*	0	21837	title p section list figgrp example
section	*Section*	0	4905	title subsect1 p definit example clause exgroup figgrp list deflist codegrp eq table
appendix	*Section*	0	6655	title section p
back	*Section*	0	3782	biblist afterwrd
biblist	*Section*	0	3106	head citation note
preface	*Section*	0	2063	title p
list	*ParaGrp*	0	184	item figgrp
exgroup	*ParaGrp*	0	1069	title exerc figgrp
deflist	*ParaGrp*	0	494	term dd
codegrp	*ParaGrp*	0	791	code
pubfront	*ParaCont*	0	241	cpyrt contract pubname
foreword	*ParaCont*	0	1747	title p date author
subsect1	*ParaCont*	0	1862	title p example clause list figgrp table codegrp exgroup
example	*ParaCont*	0	582	p figgrp title codegrp answer
clause	*ParaCont*	0	154	p title
item	*ParaCont*	0	63	p list figgrp
footnote	*ParaCont*	0	128	p
definit	*ParaCont*	0	88	p
exerc	*ParaCont*	0	139	p figgrp answer

Table 5.1: Classification results for the modified Book DTD of ISO 12083 DTD for textbooks: part 1.

the paper machine manual tell something about the content, e.g., *electrics*, *maintenance*, *troubleshoot*, *operations*, *mechanics*, and *hydraulics*. The average text length of the subtrees of a section in the manual varies between 2 362 and 7 800 characters.

The list hierarchies *list* and definition list (*deflist*) as well as the element types exercise group (*exgroup*) and code group (*codegrp*) are mapped to *ParaGrp*; see Figure 5.3. The element types *pubfront* (information about the publisher), *example*, *clause*, *footnote*, *definit*, and *note* are mapped to

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
dd	*ParaCont*	0	43	p
code	*ParaCont*	0	254	p code
note	*ParaCont*	0	90	p
afterwrd	*ParaCont*	0	1351	title p
answer	*ParaCont*	0	326	title p figgrp
contract	*Para*	304	304	text
p	*Para*	86	94	text emph eqbody list footnote fig superscr figgrp eq eqgrp table subscr text userinp deflist xref
authgrp	*StrGrp*	0	17	author
cpyrt	*StrGrp*	0	51	cpyrtnme date
ack	*StrGrp*	0	423	poem
citation	*StrGrp*	0	115	author sertitle othinfo
author	*StrCont*	0	22	fname surname degree
titlegrp	*StrCont*	0	40	title
corpauth	*StrCont*	0	43	orgname
cpyrtnme	*StrCont*	0	23	fname surname orgname
poem	*StrCont*	0	423	poemline
pubname	*StrCont*	0	14	orgname
fname	*String*	7	7	text
surname	*String*	7	7	text
emph	*String*	8	8	text eqbody emph superscr subscr
orgname	*String*	34	34	text
date	*String*	10	10	text
superscr	*String*	1	1	text
caption	*String*	39	40	text eqbody emph userinp superscr subscr xref

Table 5.2: Classification results for the modified Book DTD of ISO 12083 DTD for textbooks: part 2.

ParaCont; see Tables 5.1 and 5.2.

The element types *foreword*, *subsect1*, and *afterwrd*, whose average text lengths of the subtrees of the element instances are 1 747, 1 862, and 1 351 characters, are also mapped to *ParaCont*. One would expect that especially *subsect1* were classified as *Section*. One reason for this problem is that the textbooks contain a great deal of inline formulas saved as figures, which do not increase the text length but are an inherent part of the formatted text. In order to fix the problem, the value of the threshold minsect can be decreased from 2 000. On the other hand, in our test material there is, for

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
text	*String*	4	4	text
subscr	*String*	1	1	text
poemline	*String*	23	23	text
degree	*String*	32	32	text
userinp	*String*	11	11	text
sertitle	*String*	41	45	text emph superscr
othinfo	*String*	36	53	text emph
title	*Title*	21	21	text emph superscr subscr
term	*Title*	5	7	text emph eqbody userinp
head	*Title*	13	13	text
figgrp	*FigCont*	0	39	fig caption
fig	*Fig*	0	0	λ
eq	*FormCont*	0	0	eqbody text
eqgrp	*FormCont*	0	0	eq
eqbody	*Formula*	0	0	λ
xref	*Ref*	0	0	λ
toc	*Empty*	0	0	λ
table	*table*	0	81	tgroup
tgroup	*tgroup*	0	81	colspec tbody thead
colspec	*colspec*	0	0	λ
tbody	*tbody*	0	81	row
row	*row*	0	14	cell
cell	*cell*	1	3	text p eqbody footnote
thead	*thead*	0	7	row

Table 5.3: Classification results for the modified Book DTD of ISO 12083 DTD for textbooks: part 3.

instance, a long list whose average text length of the subtrees of the element instances is 1 893; see the element type *list* in Section C.1 of Appendix C. However, the value of the threshold minsect is set to 2 000, because it works well for typical sections in the test material, and because a long list should not be classified as *Section*. In this case, either some compromises have to be made or mapping results of certain element types have to be manually changed.

In general, more sophisticated reasoning for the threshold minsect could be used. For example, the threshold value could depend on the average text lengths of paragraphs in the document type: if the paragraphs are typically long, the minsect value is increased, if they are typically short, the value is

DTD	$E \in N$	R_E	DTD	$E \in N$	R_E
ISO12083	front	10706	Papermch	variation	5053
ISO12083	preface	9783	Papermch	electrics	7800
ISO12083	chapter	53003	Papermch	maintenance	3139
ISO12083	section	10290	Papermch	ondemandmaint	2488
ISO12083	notes	11564	Papermch	troubleshoot	2583
DocBook	chapter	24852	Papermch	operations	3741
DocBook	sect1	3902	Papermch	procedures	2900
DocBook	sect2	2004	Papermch	automation	4409
DocBook	glossary	4905	Papermch	software	3278
DocBook	preface	6852	Papermch	mechanics	2362
DocBook	appendix	15385	Papermch	hydraulics	4015
TEI	front	6071	Papermch	parameters	6002
TEI	div	3194	Papermch	pneumatics	3335
TEI	body	85721	Papermch	principles	4126
TEI	div0	7053	Papermch	topic	2601
TEI	div1	4241	HTML	body	41337
TEI	div2	3470	HTML	div	5254
TEI	div3	2838	HTML	rset	60400
TEI	div4	2837			

Table 5.4: Element types mapped to *Section* in different document types.

decreased. See Table 5.5 for different average text lengths of paragraphs.

In these textbooks the basic paragraph element type is *p*, whose average text length of the element instances is 86 characters; see Table 5.2. The element type *p* mapped to class *Para* may contain text, strings of different types, such as emphasized words (*emph*) and superscripts (*superscr*), figures, formulas, footnotes, references, lists, and tables. On the other hand, the paragraph *p* appears typically in chapters, sections, examples, exercises, lists, and tables. Another paragraph type *contract*, whose average text length of the element instances is 304 characters, appears only in the element type *pubfront* in the front of the book. Table 5.5 lists the element types mapped to *Para* in the document types of Appendix C. The average text length of the element instances varies typically either between 70 and 100 or between 140 and 350. The smallest length is 63 and the largest one is 837 characters. The names of the basic paragraph types in the tested document types are *p*, *para*, and *body*, and the average text lengths are 86 (textbooks), 337 (ISO 12083), 161 (DocBook), 346 (TEI), 88 (Papermch), 199 (HTML), and 837 (Reuters).

In the textbooks there are four element types mapped to *StrGrp*, namely

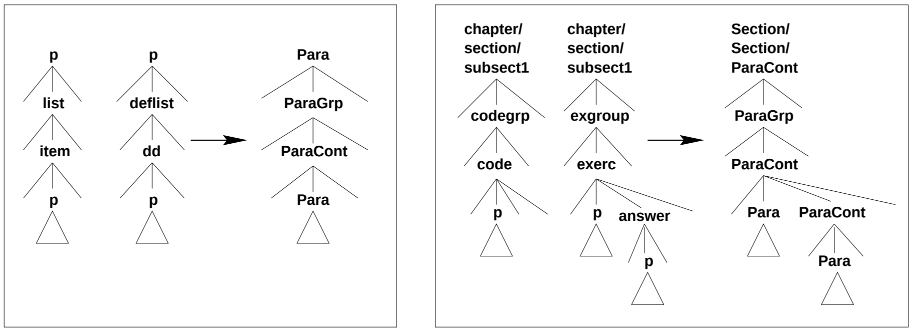


Figure 5.3: Element hierarchies mapped to *ParaGrp*.

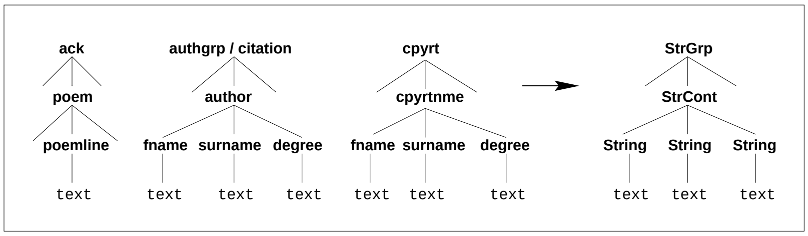


Figure 5.4: Element hierarchies mapped to *StrGrp*.

ack (acknowledgment), *authgrp* (author group), *citation*, and *cpyrt* (copyright notice); see Figure 5.4. The element types *corpauth*, *pubname* (publisher name), and *titlegrp* (title group) are mapped to *StrCont*. The element type *titlegrp* is not recognized as *Title*, because it appears inside element instances of *front* without being the first child.

The average text lengths of the element instances of the element types mapped to *String* in Tables 5.2 and 5.3 are 1, 4, 7, 8, 10, 11, 23, 32, 34, 36, 39, 41. The smallest average text length of the element instances of an element type mapped to the class *Para* is 86. Thus, in this material, the difference between strings and paragraphs is clear, and value 60 for the threshold maxstr is suitable. Furthermore, the element type *caption* mapped to *String* is easy to recognize as figure caption, since it appears inside a figure group (*figgrp*) mapped to *FigCont*. We tried to minimize the number of the generic classes. Hence, instead of having a generic class called *Caption*, we decided that the class *String* will have an attribute called *caption* that may be used in order to identify captions.

The element type *title* mapped to *Title* in Table 5.3 appears in many

DTD	$E \in N$	T_E	DTD	$E \in N$	T_E
ISO12083	catalog	562	TEI	attldecl	148
ISO12083	p	337	TEI	docimprint	71
DocBook	subtitle	90	Papermch	para	88
DocBook	para	161	Papermch	note	90
DocBook	programlisting	113	Papermch	caution	114
DocBook	literallayout	217	Papermch	warning	70
DocBook	screen	75	Papermch	explanation	77
DocBook	comment	144	HTML	p	199
TEI	p	346	HTML	li	72
TEI	distributor	68	HTML	blockquote	209
TEI	q	70	HTML	dd	88
TEI	note	112	HTML	pre	300
TEI	eg	331	HTML	cptr	561
TEI	desc	77	HTML	font	252
TEI	parents	205	HTML	small	98
TEI	decl	163	HTML	tag	278
TEI	valdesc	70	Reuters	unknown	63
TEI	children	173	Reuters	body	837
TEI	elemdecl	85			

Table 5.5: Element types mapped to *Para* in different document types.

different contexts as a title, such as in the element types *chapter*, *section*, *appendix*, *exgroup*, and *titlegrp*. Also, the classification method has correctly mapped the element types *term* appearing in *deflist* and *head* appearing in *biblist* to class *Title*.

In the textbook structure, figures are floating elements allowed to appear anywhere. A figure group (*figgrp*) mapped to *FigCont* occurs, e.g., in sections, paragraphs, and list items. The element type *fig* mapped to *Fig* contains a reference to the graphics file in an attribute. As already mentioned, formulas are represented as figures. In order to separate formulas from ordinary figures, name-based identification has been used. Thus, equation (*eq*) and equation group (*eqgrp*) are mapped to *FormCont*. Equation body (*eqbody*) mapped to *Formula* contains a reference to the graphics file in an attribute.

The reference *xref* is mapped to *Ref*, and *toc* (table of contents) is mapped to *Empty*. It is typical that the table of contents is defined to be empty, since text editing systems automatically generate it.

The CALS Table Model [OAS95] was chosen to be the common model for the generic DTD used in the tests as most of the tables in the test

DTD	*Document instances (MB)*	*Text without tags (MB)*	*TranSID query (MB)*	*Tested documents*
Textbook	2.669	0.681	1.988	3 textbooks
ISO 12083	0.789	0.680	0.224	1 book
DocBook	1.650	0.970	1.202	1 book, 2 manuals
TEI	34.921	26.837	15.006	86 books containing verses, 56 novels, 3 handbooks
Papermch	0.8 - 1	0.388	0.574	modules of a paper machine manual
HTML	2.018	1.365	0.597	38 HTML documents
Reuters	27.637	19.910	12.645	692 571 news articles
Shakespeare	7.531	4.691	7.122	38 plays
Statutes	24.500	14.299	14.438	3400 Finnish statutes
Encyclopedia	1.172	0.215	1.961	a part of a Finnish encyclopedia

Table 5.6: The test material.

material were CALS tables. Hence, an element type of the CALS table structure is mapped to itself; see Table 5.3. Of course, there are other possibilities in choosing the table model, such as the table model of the XHTML DTD [W3C00f], as discussed in Section 3.2.3.

Appendix C contains the test results from seven document types. Table 5.6 is a summary of the tested document instances and their sizes. Rows 2 to 8 contain the document types of Appendix C. Rows 9 and 10 give information about the law material and the encyclopedia used in the tests. The test results from this material are left out from the appendix, since the element names in the structure are in Finnish. As can be counted from the second column of Table 5.6, the size of all the tested material is approximately 103 megabytes.

5.1.2 Results of the Book document type of ISO 12083

Table C.1 of Appendix C contains the results of the biography of Alice Freeman Palmer [Bor93] structured by the Book DTD of ISO 12083 DTD. This is the same structure as the basic structure of the textbooks discussed in the previous section. An excerpt of the biography is given in Appendix A. In Table C.1 the element type *poem* is mapped to *StrGrp*, since *poem* contains both *poemline* (*String*) and *stanza* (*StrCont*). However, in the textbooks *poem* was mapped to *StrCont*. The element type *titlegrp* is mapped to *Title* unlike in the textbooks, since in this case *titlegrp* is the first child of some element instance of *front*. The element type *date* is mapped to *Title*, whereas it was mapped to *String* in the textbooks. Otherwise the classification results of the converging element types of the textbooks and Table C.1 are the same.

5.1.3 Results of the DocBook document type

The classification results of three books conforming to the DocBook DTD are presented in Table C.2 of Appendix C. A few specific situations will be discussed.

Four different kinds of list structures, namely *itemizedlist*, *variablelist*, *orderedlist*, and *calloutlist*, are mapped to *ParaGrp*. Section type structures *sect3* and *sect4* are mapped to *ParaCont*, since the average text lengths of the subtrees of the element instances are only 1 200 and 1 693 characters. On the other hand, the average text length of the subtrees of the element instances of *variablelist* is 1 197, which is close to the corresponding values of *sect3* and *sect4*. Hence, it is difficult to automatically find a good threshold value for minsect. However, the basic section types are clearly *chapter*, *sect1*, and *sect2*, which contain paragraphs, lists, tables and other sections.

Another observation from Table C.2 is that the difference between strings and paragraphs is clear. The largest average text length of the element instances of an element type mapped to *String* is 32 characters (*citation*), while the smallest average text length of the element instances of an element type mapped to *Para* is 75 characters (*screen*). The element type *para* mapped to class *Para* is the basic paragraph type that may contain lists, references, figures, footnotes, and various strings. In this document structure, a string (*entry*) and a title may contain a footnote, which is mapped to *ParaCont*.

5.1.4 Results of the TEI and TEI Lite document types

Table C.3 of Appendix C contains the results of 145 documents structured by the TEI DTD and the TEI Lite DTD — the latter DTD is a subset of the former. The tested document instances were 86 books containing verses, 56 novels, and 3 handbooks, which altogether constitute 35 megabytes of tagged text. The books containing verses are collected from the American Verse Project collection[1], the novels are from the Victorian Women Writers

[1] http://www.hti.umich.edu/bin/amv-idx.pl?page=main

Project collection[2], whereas the three handbooks are from the documentation of TEI[3].

The chapter–section hierarchy of the TEI structure is composed of the element types *body*, *div*, *div0*, *div1*, *div2*, *div3*, *div4*, which are mapped to *Section*; see Table C.3. The list of bibliographic citations (*listbibl*) is mapped to *StrSect*. The basic paragraph element type is *p*, whose average text length of the element instances is 346 characters. Altogether 12 element types are mapped to the generic class *Para*. Many element types of the TEI structure are mapped to the class *String*. Their average text length of the element instances is quite small: with 50 element types it is less than or equal to 30 characters, whereas with 8 element types it is between 41 and 52. Since the smallest average text length of the element instances of an element type mapped to *Para* is 68 characters (*distributor*), the distinction between the average length of strings and paragraphs is quite clear.

However, the mapping of the element types *figure*, *q* (quote), and *item* to the class *Unknown* is not desirable. The reason for this classification result is that the element types may contain the element type *text*, which in the TEI DTD is designed to wrap both elements of section type and elements of paragraph type (see [SMB95]); *text* is either an alternative to the element types *body* and *div*, or a component-level element. Since *text* may contain sections, it is classified as *SectGrp*. If *figure*, *q*, and *item* did not contain *text*, *figure* would be mapped to *ParaCont*, *q* to *Para*, and *item* to *String*. Neither of these generic classes may contain *SectGrp*s. On the other hand, the average length of the subtrees of the element instances of the element types *figure*, *q* and *item* are less than the threshold value minsect. Hence, the element types cannot be mapped to *Section* or to *SectGrp*. Thus, there is no generic class suitable for these element types, and they are mapped to the class *Unknown*.

If in some document collection many element types behave like paragraph- or string-level types but may contain section-level types, like *figure*, *q*, and *item*, it might be necessary to add a new generic class into the generic DTD used in such cases. An example of such a generic class would be *MixComp* (mixed component) that has the same content model as the class *Section*:

```
<!ELEMENT MixComp  ((#PCDATA)|%sml;|StrGrp|%par;|%sec;|table)*>
```

The new class *MixComp* could exist in the content models of string and paragraph levels, i.e., in the content models of the classes *String*, *StrCont*,

[2] http://www.indiana.edu/~letrs/vwwp/index.html
[3] http://www.tei-c.org/

StrGrp, *Para*, *ParaCont*, and *ParaGrp*. An alternative to a new generic class is to allow the generic classes of string and paragraph levels to contain section-level types. In general, this is not a good solution, because, for example, formatting of section-level objects differs considerably from formatting of paragraph- and string-level objects.

The mapping of the element type *list* in Table C.3 is not successful: *list* is mapped to *StrCont*, and *item* of *list* would be mapped to *String* without the element type *text* as a child. In this document structure, an ideal mapping for the element type *list* would be to *ParaCont* and for the element type *item* to *Para*. The average text length of the element instances of *item* is 47 characters, and the average text length of the subtrees of the element instances is 81 characters. Hence, the average text length is greater than for an element type mapped to *String* on average in the TEI structure. Also, the average text length of the subtrees is bigger than the threshold value for strings and paragraphs ($\mathsf{maxstr} = 60$). One remark is also that *item* contains the element type *p*, which is the basic paragraph type. It is not very usual that strings contain paragraphs. Thus, to improve the classification method, these kinds of situations could also be taken into account. Especially, the average text length of the subtrees should also have an influence on the mapping of strings and paragraphs.

Because some element instance of *l* (verse line) have been the first child of an element instance of *lg* (verse stanza), *l* is mapped to the class *Title*, which is not desirable. On the other hand, instances of *l* also appear in many other instances, where they are not the first children. Hence, one could reason that *l* should not be mapped to *Title*. To improve the recognition of titles, this kind of reasoning could be taken into account.

Our final remark is that the table structure *table* – *row* – *cell* in the TEI DTD is not a CALS Table. Hence, in the test it is mapped to the structure *StrGrp* – *StrCont* – *String*. To maintain the table information, it is possible to transform the tables of the document instances into CALS tables before applying the element-type classification.

5.1.5 Results of the paper machine manual

The results of the modules of a paper machine manual are summarized in Table C.4 of Appendix C. This document structure is an example of semantic markup; the element structure is designed to follow the structure of a manual of a complex machine. The first-level section *variation* contains the sections *electrics*, *operations*, *automation*, *mechanics*, *hydraulics*, and *pneumatics* — the element name of a section partly describes its content. Some of these section types contain the element types *safety*, *maintenance*, *drawings*, *spareparts*, *troubleshoot*, *malfunction*, *structure*, and *functions*, which are also meaningful parts of a manual.

Even if the markup is semantic, there are also element names related to logical structuring such as *chapter*, *section*, *para*, *orderedlist*, *unorderedlist*, *listitem*, *firstname*, *surname*, *year*, *month*, and *day*. However, most of the element names are related to the semantic structure of the paper machine manual.

The basic paragraph type in the manual is *para*. Other element types mapped to the class *Para* are *note*, *caution*, *warning*, and *explanation*. The average text length of paragraphs varies between 70 and 114 characters, whereas with strings it varies between 1 and 31 characters. The difference in the average text length of paragraphs and strings is clear also in this document structure. On the other hand, there are only 10 element types mapped to *String*. This number is small compared to, e.g., the TEI DTD, where 58 element types were mapped to *String*.

There are 47 element types mapped to *ParaGrp* and *ParaCont*, whereas only 16 element types are mapped to *SectGrp* and *Section*. This could indicate that some of the element types mapped to *ParaGrp* or *ParaCont* should have been mapped to the class *Section*. The average text length of the element instances of the basic paragraph type *para* is 88 characters, which is quite small compared to, e.g., the TEI DTD, where the average text length of the element instances of the basic paragraph type *p* is 346 characters. For this document type the value of minsect could be smaller. Element types whose average text length of the subtrees of the element instances is larger than 1 000 characters are *structure*, *function*, *malfunction*, *procedure*, *system*, *tableslist*, and *chapter*. As can be seen from the overall document structure, it would be better to map these element types to the class *Section*. This kind of reasoning requires human interaction, but as already mentioned, the classification method could be improved by setting the value of minsect dynamically after knowing the actual average text lengths of the element instances of the element types mapped to the class *Para*.

5.1.6 Results of the HTML document type

Table C.5 of Appendix C contains the results of 38 documents conforming to the HTML DTD [W3C99b]. The tested documents were structured according to HTML 2.0 and 3.2. Because the Web browsers accept HTML documents which do not conform to any HTML DTD, there are a great deal of Web pages, where the markup is incorrect. The tested HTML documents were, e.g., online books, manuals, and tutorials, from which the

SGML parser found a great number of parse errors.

Two element types are mapped to the class *Section*, namely *body*, which is the document body, and *div*, which is a generic container used to add structure to documents. In the documents *div* instances appear inside *body* instances and inside other *div* instances. Three list structures are mapped to *ParaCont*: unordered list *ul* and ordered list *ol*, which may contain the list item *li* mapped to *Para*, as well as definition list *dl*, which may contain definition term *dt* mapped to *Title* and definition description *dd* mapped to *Para*.

The basic paragraph type is *p*. Other element types mapped to the class *Para* are *blockquote* (long quotation), *pre* (preformatted text), *font* (local change to font), and *small* (small text style). Element types *font* and *small* are not designed to be paragraph objects, and they should contain only short inline elements, not paragraphs. However, this is not followed in practice.

The element type *table* is mapped to *StrGrp*, since *table* contains table row (*tr*) mapped to *StrCont*, which in turn may contain data cell (*td*) and header (*th*) mapped to *String*. The average text length of the element instances of *td* is 58, which is very near to the value of maxstr (60). The average length of the subtrees of the element instances of *td* is 362, and *td* may contain, e.g., paragraphs and tables. Thus, the mapping of *td* to *String* is not desirable. As already discussed, the average text length of the subtrees of the element instances would be useful when identifying paragraphs. The following is a slightly more sophisticated rule for identifying paragraphs (compare Definition 4.23), and it can also be used with other document types than HTML.

3. **if** $(\mathsf{maxstr} < T_E < \mathsf{minsect}$ **or**
$(T_E > \mathsf{maxstr} - 40$ **and** $R_E > \mathsf{maxstr} + 40))$ **and**
$R_E < \mathsf{minsect}$ **and** (**for each** e **in** $D_E, R_e < \mathsf{minsect}$) **then**
$\hat{h}_G(E) = Para$.

The difference to the reasoning presented in Section 3.2 is in the condition $(T_E > \mathsf{maxstr} - 40) \wedge (R_E > \mathsf{maxstr} + 40)$. When the value of maxstr is set to 60, the condition is satisfied, if the T_E is greater than 20 and the R_E is greater than 100. With this condition, the element type *center* would also be mapped to the class *Para* instead of the class *String*, which is desirable, because *center* may contain paragraphs.

The following list presents the element types mapped to *String* grouped according to the HTML specification. The element type *title* in the last item of the list should have been mapped to the class *Title*. The element types *h2*, *h3*, and *h4* are typically used as titles, but with our reasoning it is hard to find them, since an HTML structure tree is usually flat, and heading elements are seldom the first children of appropriate structures.

- Headings: *h2*, *h3*, and *h4*.
- Font style elements: *i* (italic text style), *b* (bold text style), *big* (large text style), *tt* (teletype or monospaced text style), and *u* (underlined text style).
- Phrase elements: *em* (emphasis), *cite* (citation), *strong* (strong emphasis), *code* (computer code fragment), *dfn* (instance definition), *samp* (sample program output, scripts, etc.), *var* (instance of a variable or program argument), and *kbd* (text to be entered by the user).
- Subscripts and superscripts: *sub* and *sup*.
- Others: *a* (anchor), *center* (shorthand for DIV align=center), *address* (information on author), and *title* (document title).

Half of the element types are mapped to the class *String*. This kind of mapping simplifies the original structure very much. To improve the classification method, we can define new generic classes for string types which have different thresholds for the average text length of the element instances. The new generic classes could be, e.g., *ShString* (short string, $0 < T_E \leq 20$), *MeString* (medium string, $20 < T_E \leq 40$), and *LoString* (long string, $40 < T_E \leq 60$). The content models for these new string types would be the same as for the class *String*.

The element type *head* is the document head correctly mapped to the class *Title*. By accident, the element type *h1* has been the first child of some element instance of *div* mapped to *Section*. Hence, *h1* is mapped to the class *Title*, which is reasonable.

The element type *map* (client-side image map) is mapped to *StrCont*, since it contains *area* (client-side image map area) mapped to the class *Empty*. Name-based identifying is used in mapping the element type *img* to the class *Fig*. The element types mapped to the class *Empty* are the following: *hr* (horizontal rule), *br* (forced line break), and *link* (media-independent link). The element type *meta* (generic metainformation) is mapped to the class *meta*.

5.1.7 Results of the texts of Shakespeare

The results of 38 plays conforming to the Shakespeare XML DTD [Bos97] are summarized in Table C.6 of Appendix C. The basic structure of the texts is the following: A play contains acts that consist of scenes. A speech is a basic element of a scene containing the name of a speaker and several lines of text. This simple structure is classified as follows: The element types *act* and *scene* are mapped to the class *StrSect*, which is a section-level container for groups of strings. The element type *speech* is mapped to *StrCont*, whereas *speaker* and *line* are mapped to *String*. The element types *epilogue* and *prologue* also contain *speech* like *scene* does, but due to the average text lengths of the subtrees of their element instances, the mapping result is *StrGrp*.

The element type *personae* is mapped to *StrGrp*. It contains both introductions of persons in the play, namely the element type *persona* mapped to *String*, and groups of *persona*, namely the element type *pgroup* mapped to *StrCont*. For instance, the play "The Tragedy of Hamlet, Prince of Denmark" contains the following kind of *personae*:

```
<StrGrp origel="personae">
  <Title origel="title">Dramatis Personae</Title>

  <String origel="persona">CLAUDIUS, king of Denmark. </String>
  <String origel="persona">HAMLET, son to the late, and nephew to
  the present king.</String>
  <String origel="persona">POLONIUS, lord chamberlain. </String>
  <String origel="persona">HORATIO, friend to Hamlet.</String>
  <String origel="persona">LAERTES, son to Polonius.</String>
  <String origel="persona">LUCIANUS, nephew to the king.</String>

  <StrCont origel=""pgroup">
    <String origel="persona">VOLTIMAND</String>
    <String origel="persona">CORNELIUS</String>
    <String origel="persona">ROSENCRANTZ</String>
    <String origel="persona">GUILDENSTERN</String>
    <String origel="persona">OSRIC</String>
    <String origele="grpdescr">courtiers.</String>
  </StrCont>
...
</StrGrp>
```

The generalized play structure is introduced in two figures: Figure 5.5 shows an excerpt from the beginning of the play. The end tags of the markup are mainly abbreviated as "</>" to save space. Figure 5.6 presents a formatted view of the same excerpt in our document assembly system *SAW*, which will be introduced in Section 6.1.

5.1.8 Results of the Reuters news

The results of 692 571 Reuters news articles [Reu97] are located in Table C.7 of Appendix C. The news structure is very flat: the news article (*reuters*) mapped to *ParaCont* contains the news text (*text*) mapped to

```
<StrSect origel="act"><Title origel="title">ACT I</Title>
  <StrSect origel="scene">
    <Title origel="title">SCENE I. Elsinore. A platform before the castle.</>
    <String origel="stagedir">FRANCISCO at his post. Enter to him BERNARDO</>
    <StrCont origel="speech"><String origel="speaker">BERNARDO</>
      <String origel="line">Who's there?</></>
    <StrCont origel="speech"><String origel="speaker">FRANCISCO</>
      <String origel="line">Nay, answer me: stand, and unfold yourself.</></>
    <StrCont origel="speech"><String origel="speaker">BERNARDO</>
      <String origel="line">Long live the king!</></>
    <StrCont origel="speech"><String origel="speaker">FRANCISCO</>
      <String origel="line">Bernardo?</></>
    <StrCont origel="speech"><String origel="speaker">BERNARDO</>
      <String origel="line">He.</></>
    <StrCont origel="speech"><String origel="speaker">FRANCISCO</>
      <String origel="line">You come most carefully upon your hour.</></>
    <StrCont origel="speech"><String origel="speaker">BERNARDO</>
      <String origel="line">'Tis now struck twelve; get thee to bed,
        Francisco.</></>
    <StrCont origel="speech"><String origel="speaker">FRANCISCO</>
      <String origel="line">For this relief much thanks: 'tis bitter cold,</>
      <String origel="line">And I am sick at heart.</></>
    <StrCont origel="speech"><String origel="speaker">BERNARDO</>
      <String origel="line">Have you had quiet guard?</></>
    ...
  </StrSect>
  ...
</StrSect>
```

Figure 5.5: An excerpt from the beginning of Hamlet.

String and short sequences of metainformation mapped to *Para*, *StrCont*, *String*, *Title*, and *Empty*. The metainformation includes, e.g., element types *topics*, *places*, *people*, *author*, and *date*. The news text (*text*), in turn, may contain text and the element types *title*, *dateline*, *author*, and *body*, which is mapped to *Para* and contains the actual text content of the news article.

5.2 Implementation issues

In the following, we consider some aspects of implementing the element-type classification method.

5.2.1 Overview

The element-type classification method can be implemented without the information extracted from the DTD. The structural relationships and the average text lengths are extracted from the document instances. When im-

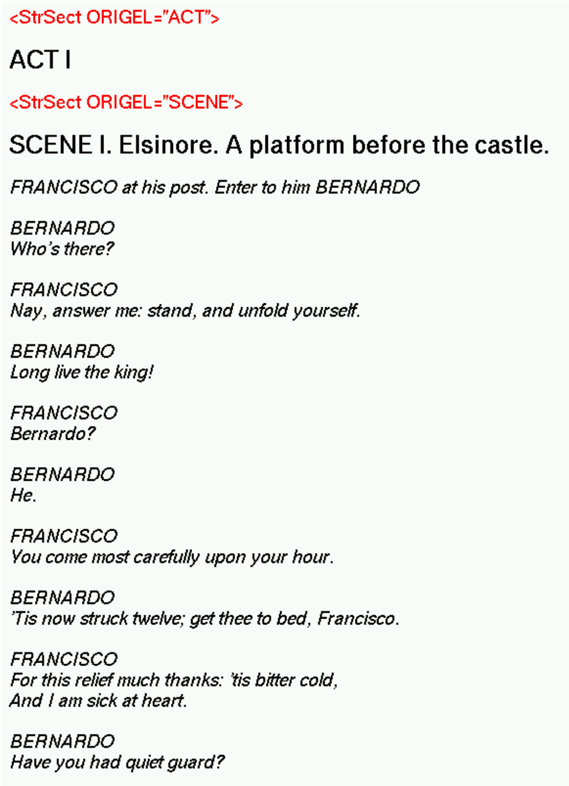


Figure 5.6: A dynamically formatted view in the SAW system.

plementing the method, the DTDs were used in parsing SGML documents, which is not necessary in parsing well-formed XML documents. Some attribute information related to references and figures were also extracted from the DTDs. However, apart from the attribute information, the implementation does not depend on the knowledge of the DTD when considering XML documents. The test material contained both XML and SGML documents. However, the majority of the material was SGML documents.

In practice, not all the element types of the DTD are used in the document instances. The classification method maps only the element types appearing in the document instances. Thus, there is no mapping for element types not appearing in the document instances. Because the method extracts information from the document instances, the classification results must be updated when adding new documents to the document collection.

It is typical that DTDs are quite permissive, and allow the user some flexibility that may result in slightly different mappings for a given DTD between different document bases. On the other hand, the classification results reveal the actual structure of the documents in a document base. Hence, document designers may study the actual use of the designed structure by exploring the classification results.

Generally, to improve the results of the element-type classification method, the thresholds maxstr and minsect, and even some of the decision rules can be changed by the document or software designer for certain document types. Also, the mistakes in the mappings of the method can be manually corrected by the user. It would be useful, if the document designers provided ideal mappings, which may be based on the results of the classification method.

The size of the document collection must also be taken into account. Instead of using hundreds of megabytes of document instances in the classification process, it is essential to sample the data. For instance, the Finnish law collection contains approximately 50 000 Finnish statutes, altogether hundreds of megabytes of text. In the classification process 3 400 statutes containing 24.5 megabytes of text were used. If the collection is very big, there is probably someone, who is responsible for the maintenance of the collection and has an insight into the document base. This knowledge is valuable in choosing the documents that best represent the whole collection, and are thus appropriate to be used in sampling.

It is encouraging that in the document types only a few element types were classified as *Unknown*. Altogether ten different document types containing hundreds of element types were tested (see Table 5.6). Only four element types in two document types were mapped to the class *Unknown*. This demonstrates that the classification rules are applicable in practice.

To demonstrate the time consumption of the method, we tested a collection of 56 novels structured by the TEI Lite DTD. The size of the collection was 15.4 megabytes. With an Alpha-based computer running Linux OS, the TranSID query of Figure 5.1 took 9 minutes and 50 seconds of real time, from which the parsing of the documents took 1 minute and 50 seconds. The size of the output of the TranSID query was 3.5 megabytes. The classification procedure implemented using Perl took 40 seconds of real time with a PC of 120 MHz running Linux OS.

5.2.2 Mapping of attributes

Quite a common design principle is that the text which will be presented to the user is coded into elements, whereas other information such as metadata is coded into attributes. Typically, attributes are system specific, which makes automatic reasoning of the meaning of attributes impossible. Some attributes, however, are very important, like the ones related to the source of a figure and to the formatting of a figure.

We have implemented an automatic method for identifying some of the attributes. The identification is based on the information extracted from the DTDs available. The method identifies the attributes of type ID (unique identifier), IDREF (reference to unique identifier), and IDREFS (references to unique identifiers), as well as the attributes of the CALS Table Model, and some attributes related to the source of a figure and to the formatting of a figure. Note that the identification of metadata was already discussed in Section 3.2.

Figure 5.7 presents an excerpt, which contains some mapping results of the TEI document type discussed in Section 5.1.4. In fact, our assembly system SAW imports the classification results from a file which is structured as the excerpt: The attribute *elem* of the element *map* contains the original element name, and the attribute *class* contains the generic class name. The attribute *origatt* of the element *attmap* contains the original attribute name, the attribute *classatt* contains the corresponding attribute name of the generic class, and the attribute *type* contains the type of the mapped attribute.

As can be seen from Figure 5.7, the element type *note* has the attributes *id*, *lang*, *target*, and *targetend*, which are mapped to the attributes *id*, *idref*, *idrefs*, and *idrefs2* of the generic class *Para*.

Figure 5.8 shows an excerpt, which contains the mapping results of the element types *p*, *xref*, *fig*, *table*, and *row*. The element types are from the textbook document type discussed in Section 5.1.1.

5.2.3 Architectural forms for representing metastructures

Architectural forms defined in the HyTime standard [ISO92, ISO97] offer mechanisms for representing metastructures of SGML documents. The Resource Description Framework (RDF) [W3C99e, W3C99f] designed for XML documents can also be used for defining document metastructures.Both architectural forms and RDF offer tools to represent the generic DTD and the mapping of element types to the generic classes described in Chapters 3 and 4.

In this section we show how the element-type classification can be used with the architectural forms facility of HyTime to create *architectural documents*, or *virtual documents*. For instance, if there is a need for a DTD to be more permissive resulting in a more general or simplified structure for a given document collection, then the original documents are transformed

```
<map elem="note" class="Para">
   <attmap origatt="id" classatt="id" type="ID">
   <attmap origatt="lang" classatt="idref" type="IDREF">
   <attmap origatt="target" classatt="idrefs" type="IDREFS">
   <attmap origatt="targetend" classatt="idrefs2" type="IDREFS">
</map>
<map elem="abbr" class="String">
   <attmap origatt="id" classatt="id" type="ID">
   <attmap origatt="lang" classatt="idref" type="IDREF">
   <attmap origatt="resp" classatt="idref2" type="IDREF">
</map>
<map elem="unclear" class="String">
   <attmap origatt="hand" classatt="idref" type="IDREF">
   <attmap origatt="id" classatt="id" type="ID">
   <attmap origatt="lang" classatt="idref2" type="IDREF">
</map>
<map elem="tagusage" class="String">
   <attmap origatt="id" classatt="id" type="ID">
   <attmap origatt="lang" classatt="idref" type="IDREF">
   <attmap origatt="render" classatt="idref2" type="IDREF">
</map>
```

Figure 5.7: Attributes identified from the TEI structure.

into architectural documents by retaining the contents of the documents but mapping the original element types into new ones. It is also possible to make mappings that alter the document contents, for example, by ignoring some original element types.

When creating virtual documents, the following components are needed: the original DTD, the *base architecture DTD*, and the *architectural form attributes*. Together they form a *meta-DTD* for a specified document collection. To be more precise: The original DTD provides the initial elements, the base architecture DTD gives the target elements and the document structure, and the architectural form attributes provide the mapping between initial and target elements. In our context, the element-type classification produces the mapping, whereas the base architecture DTD is the predefined generic DTD (see Section 3.2).

The architectural form attributes can be specified in the document itself or in conjunction with the DTD. If they are defined in the document, architectural form attributes for different base architectures are defined separately for every element instance, while in the latter case they are defined only once in the DTD in conjunction with the original element types. The latter case is chosen to our experiments. An example of an architectural form attributes list is presented in Figure 5.9. The list provides a mapping for a document according to the Book DTD of ISO 12083 DTD into a base architecture DTD *VirtDoc*, which is the generic DTD. The same mapping

```
<map elem="p" class="Para">
   <attmap origatt="exclpars" classatt="idrefs" type="IDREFS">
   <attmap origatt="id" classatt="id" type="ID">
</map>
<map elem="xref" class="Ref">
   <attmap origatt="rid" classatt="idref" type="IDREF">
</map>
</map>
<map elem="fig" class="Fig">
   <attmap origatt="entity" classatt="entity" type="ENTITY">
   <attmap origatt="file" classatt="file" type="CDATA">
   <attmap origatt="align" classatt="align" type="NAME">
   <attmap origatt="angle" classatt="angle" type="CDATA">
   <attmap origatt="bloffset" classatt="bloffset" type="CDATA">
   <attmap origatt="cropped" classatt="cropped" type="NUMBER">
   <attmap origatt="dpi" classatt="dpi" type="NUMBER">
   <attmap origatt="float" classatt="float" type="NUMBER">
   <attmap origatt="height" classatt="height" type="CDATA">
   <attmap origatt="nsoffset" classatt="nsoffset" type="CDATA">
   <attmap origatt="position" classatt="position" type="NAME">
   <attmap origatt="width" classatt="width" type="CDATA">
   <attmap origatt="xoffset" classatt="xoffset" type="CDATA">
   <attmap origatt="yoffset" classatt="yoffset" type="CDATA">
   ...
</map>
<map elem="table" class="table">
   <attmap origatt="colsep" classatt="colsep" type="NUMBER">
   <attmap origatt="frame" classatt="frame" type="all, top, bottom,
     topbot, sides, none">
   <attmap origatt="pgwide" classatt="pgwide" type="NUMBER">
   <attmap origatt="rowsep" classatt="rowsep" type="NUMBER">
   <attmap origatt="tabstyle" classatt="tabstyle" type="NMTOKEN">
</map>
<map elem="row" class="row">
   <attmap origatt="maxheight" classatt="maxheight" type="CDATA">
   <attmap origatt="minheight" classatt="minheight" type="CDATA">
   <attmap origatt="rowsep" classatt="rowsep" type="NUMBER">
   <attmap origatt="valign" classatt="valign" type="top, middle, bottom">
</map>
```

Figure 5.8: Attributes identified from the textbook structure.

can be found in a tabular format in Appendix C.1.

The element-type classification producing the list of Figure 5.9 was done using only one document [Bor93]. Excerpts of the original book and the resulting virtual document are presented in Appendix A. Note that although we only present one mapping, it is possible to refer to different base architecture DTDs from a DTD or from a document.

```
<!AFDR "ISO/IEC 10744:1992">

<!-- Architecture Base Declaration -->
<?ArcBase VirtDoc>

<!-- Architecture Notation Declaration -->
<!NOTATION VirtDoc PUBLIC
          "-//Heikkinen//NOTATION AFDR ARCBASE VirtDoc//EN">
<!ATTLIST #NOTATION VirtDoc
          ArcDTD   CDATA               #FIXED VirtDocDtd
          ArcAuto  (ArcAuto|nArcAuto)  #FIXED nArcAuto>

<!-- Architecture Entity Declarations -->
<!ENTITY VirtDocDtd PUBLIC "-//Heikkinen//DTD VirtDoc//EN">
<!ENTITY % original.dtd PUBLIC "ISO 12083:1993//DTD Book//EN">
%original.dtd;

<!ATTLIST book     VirtDoc NAME #FIXED "VirtDoc">
<!ATTLIST body     VirtDoc NAME #FIXED "SectGrp">
<!ATTLIST back     VirtDoc NAME #FIXED "SectGrp">
<!ATTLIST front    VirtDoc NAME #FIXED "Section">
<!ATTLIST preface  VirtDoc NAME #FIXED "Section">
<!ATTLIST chapter  VirtDoc NAME #FIXED "Section">
<!ATTLIST section  VirtDoc NAME #FIXED "Section">
<!ATTLIST notes    VirtDoc NAME #FIXED "Section">
<!ATTLIST list     VirtDoc NAME #FIXED "ParaGrp">
<!ATTLIST pubfront VirtDoc NAME #FIXED "ParaCont">
<!ATTLIST ded      VirtDoc NAME #FIXED "ParaCont">
<!ATTLIST bq       VirtDoc NAME #FIXED "ParaCont">
<!ATTLIST note     VirtDoc NAME #FIXED "ParaCont">
<!ATTLIST item     VirtDoc NAME #FIXED "ParaCont">
<!ATTLIST catalog  VirtDoc NAME #FIXED "Para">
<!ATTLIST p        VirtDoc NAME #FIXED "Para">
...
```

Figure 5.9: A list of architectural form attributes.

Chapter 6

Document assembly systems

Two implementations of document assembly will be described in this chapter. A general purpose document assembly system is described in Section 6.1, and a specialized system for tailoring textbooks via the Web is discussed in Section 6.2.

6.1 SAW — The SID Assembly Workbench

This section describes our prototype of a document assembly system, which we call the *SID Assembly Workbench*, or *SAW*. We have developed the system in order to test and to evaluate the conceptual model of the document assembly process discussed in Section 3.3.

The assembly starts from a collection of document fragments or document components. Fragment is a coherent and contiguous part of the document, whereas component can be any document element and its subtree. The user can recognize which components are fragments provided by the text fragmentation method in the initialization phase. The user assembles a result document by iteratively selecting and reordering components from the collection. When the user is satisfied with the result, he or she saves it. If the result document contains components conforming to several DTDs, the system transforms them to conform either to our generic DTD or to HTML DTD. SAW does not support editing of the assembly result. For producing a final deliverable, such as a printed book, an online document or a CD-ROM, the resulting document can be further edited and formatted using a structure editor or a publishing system.

6.1.1 The SAW user interface

The SAW user interface (Figure 6.1) is based on the concept of a *view*, which is a representation of the document collection or the result at some state of the assembly process. Conceptually, a view is an ordered sequence of *assembly components*, which can be document fragments, clusters of fragments, other components, or other views. The initial view presents the entire document collection. Operations on a view always create a new view. By modifying the view, the user gradually moves towards a more final result consisting of a sequence of useful document components. The user can also store intermediate views, e.g., a draft assembly of a chapter. Stored views can be retrieved and included in new result views.

The SAW user interface is divided into three logical parts; see Figure 6.1. The first part presents a view of the document collection, also called the *current view* of the collection. The current view consists of a table of rows; one row for each assembly component in the view. The second part consists of the commands for modifying the current view. They include selection and reordering of components, clustering and searching the collection, and advanced querying of the underlying database. The third part of the user interface provides operations for saving the assembled views, namely the operations *Save view* and *Export view as SGML*.

Figure 6.1 presents an initial view of a collection of 7 documents, each of which has a different structure. The documents and their DTDs are as follows:

1. A biography; the Book DTD of ISO 12083 DTD [ISO94].
2. A Finnish textbook on control engineering; the Book DTD of ISO 12083 DTD supplemented with element types for, e.g., exercises, answers, examples, definitions, clauses, formulas, and computer code.
3. A handbook; the DocBook DTD [OAS91].
4. A handbook; the TEI DTD [TEI94].
5. A novel; the TEI Lite DTD [TEI94].
6. A play; the Shakespeare (XML) DTD [Bos97].
7. A module of a manual in Finnish; the paper machine manual DTD.

Figure 6.2 presents components which are sections of different levels in a book[1] structured by the DocBook DTD, namely *chapter*, *sect1* and *sect2*.

[1]"Getting started with SGML" by Eve Maler; http://www.oasis-open.org/docbook/samples/index.html

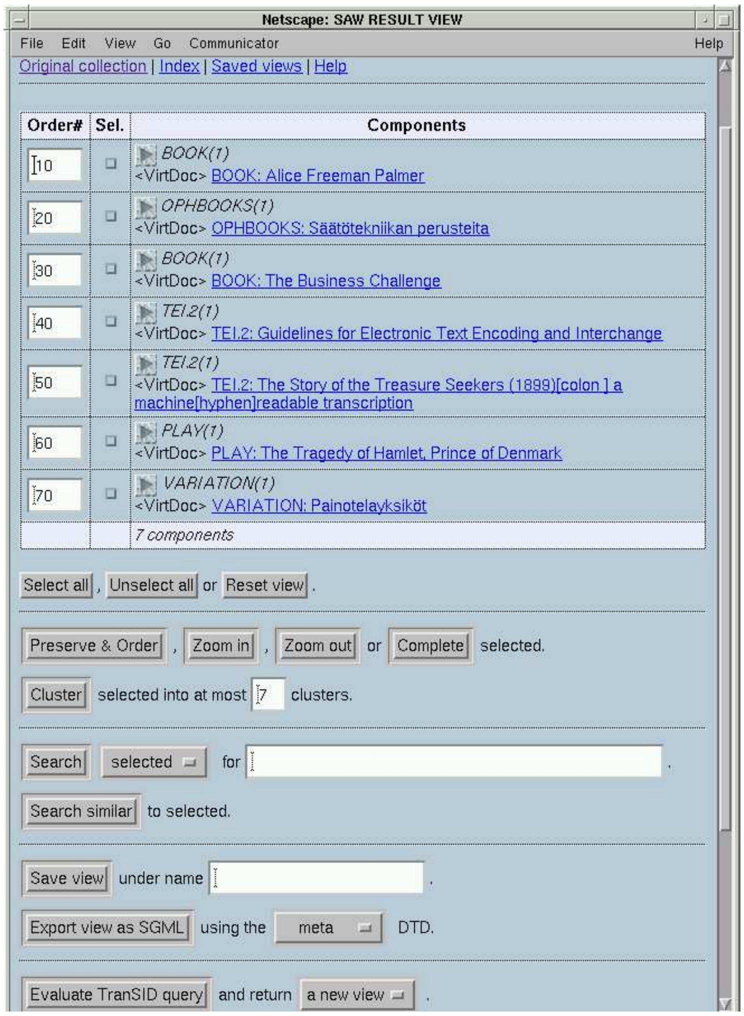
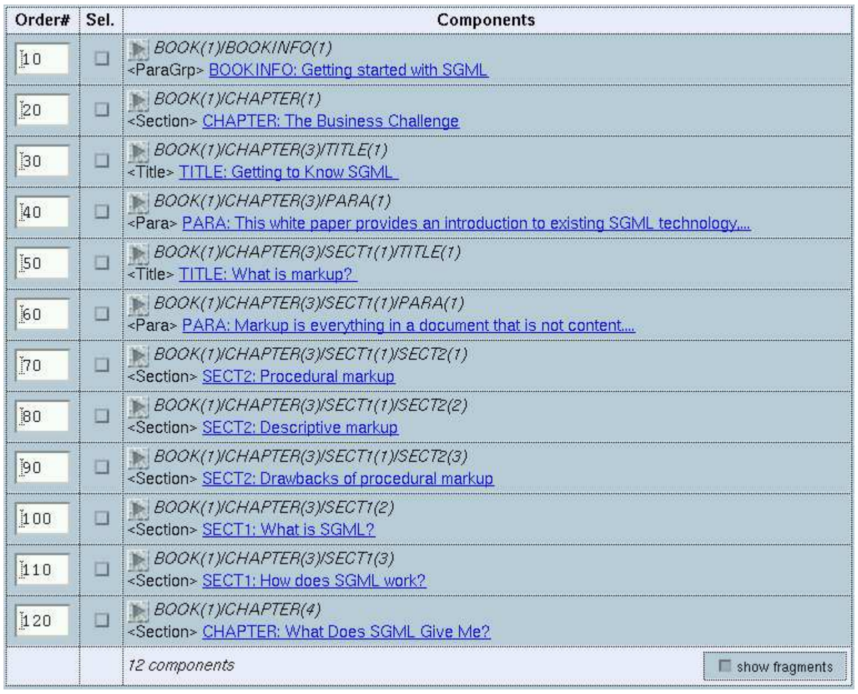


Figure 6.1: The user interface of SAW.

An arrow icon in a component row informs the location of the element in the structure tree. For instance, in the 8th component the location is given by a path *book(1)chapter(3)sect1(1)sect2(2)*, which tells that the element is the second subsection of the first section of the third chapter in the book. Below the arrow icon is the generic class of the component element, which for the 8th component is *Section*. Next to the generic class is the original name of the element (*sect2*) and the nearest *Title* sibling of the element (subsection title "Descriptive markup"); see the 8th component of



Order#	Sel.	Components
10	☐	BOOK(1)/BOOKINFO(1) <ParaGrp> BOOKINFO: Getting started with SGML
20	☐	BOOK(1)/CHAPTER(1) <Section> CHAPTER: The Business Challenge
30	☐	BOOK(1)/CHAPTER(3)/TITLE(1) <Title> TITLE: Getting to Know SGML
40	☐	BOOK(1)/CHAPTER(3)/PARA(1) <Para> PARA: This white paper provides an introduction to existing SGML technology...
50	☐	BOOK(1)/CHAPTER(3)/SECT1(1)/TITLE(1) <Title> TITLE: What is markup?
60	☐	BOOK(1)/CHAPTER(3)/SECT1(1)/PARA(1) <Para> PARA: Markup is everything in a document that is not content....
70	☐	BOOK(1)/CHAPTER(3)/SECT1(1)/SECT2(1) <Section> SECT2: Procedural markup
80	☐	BOOK(1)/CHAPTER(3)/SECT1(1)/SECT2(2) <Section> SECT2: Descriptive markup
90	☐	BOOK(1)/CHAPTER(3)/SECT1(1)/SECT2(3) <Section> SECT2: Drawbacks of procedural markup
100	☐	BOOK(1)/CHAPTER(3)/SECT1(2) <Section> SECT1: What is SGML?
110	☐	BOOK(1)/CHAPTER(3)/SECT1(3) <Section> SECT1: How does SGML work?
120	☐	BOOK(1)/CHAPTER(4) <Section> CHAPTER: What Does SGML Give Me?
		12 components ☐ show fragments

Figure 6.2: Chapters (*chapter*), sections (*sect1*), subsections (*sect2*), titles and paragraphs in a SAW view.

Figure 6.2. If an element does not have a *Title* sibling, like the paragraph of the 6th component, then some text from the beginning of the paragraph is included ("Markup is everything in a document that is not content...").

The user may browse the formatted content of the component by clicking the underlined title which is a link. The component is formatted dynamically using the formatting instructions for the generic classes. Hence, a browsable HTML page is generated on the fly using the information of the generic classes and the content of the component. As an example, the formatted page generated from the Chapter 3 of Figure 6.2 is presented in Figure 6.3.

6.1.2 The SAW commands

Each component is preceded in the current view by a field for an order number and a selection button. The order numbers can be changed to rearrange the view according to the numbering. Selections define which components are affected by the view manipulation commands. All components in the

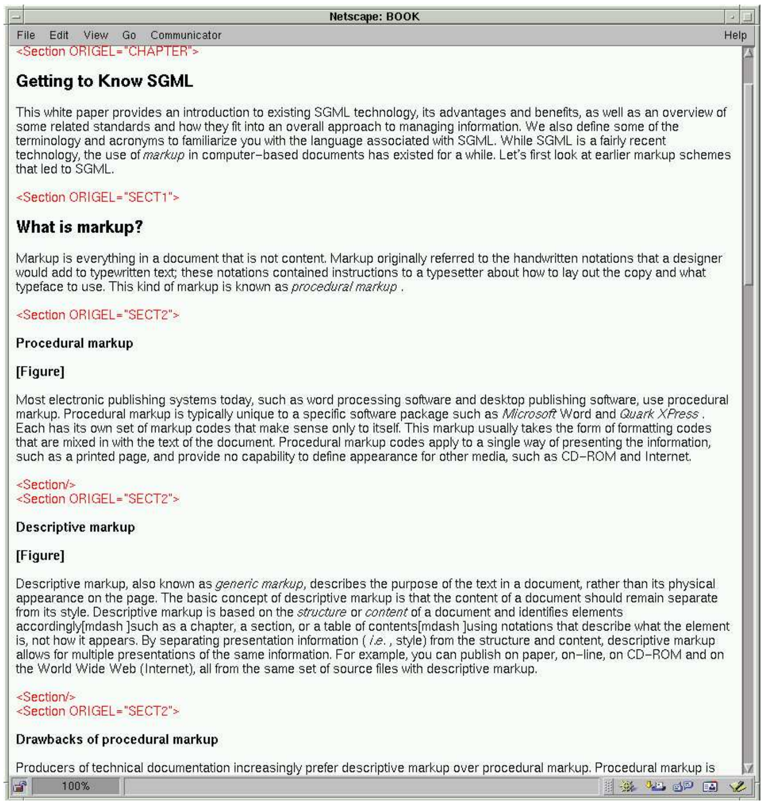


Figure 6.3: Formatted content of a component in SAW.

view can be selected or unselected at the same time using the buttons *Select all* and *Unselect all*. Any selecting or reordering of the current view can be undone using the button *Reset view*.

The *Preserve & Order* button discards all unselected components from the view, and arranges the remaining ones according to their numbers. The *Zoom in* button replaces selected document components by their immediate subelements, e.g., tables by their rows. *Zoom out* correspondingly allows the user to navigate in the document structure to a higher level by replacing, say, selected section elements by their enclosing chapters.

To gather the components of Figure 6.2 the following actions have been made over the view shown in Figure 6.1:

- The third book (component "The Business Challenge") has been selected by pressing the button *Sel.* in the corresponding row, and the command *Preserve & Order* has been executed. As a result, only the selected book has been included in the component list, since *Preserve & Order* keeps the selected components only.

- The command *Zoom in* has been used to replace the selected component by its immediate subelements, in this case the chapters of the book.

- The element *bookinfo*; row 1, and the chapter elements *chapter(1)*; row 2, *chapter(3)*, and *chapter(4)*; row 12, have been selected[2], and *Preserve & Order* has been executed.

- The command *Zoom in* has been executed to expand the third chapter resulting in the elements *title(1)*; row 3, *para(1)*; row 4, *sect1(1)*, *sect1(2)*; row 10, and *sect1(3)*; row 11.

- The first section of third chapter (*sect1(1)*) has been expanded resulting in the elements *title(1)*; row 5, *para(1)*; row 6, *sect1(1)*; row 7, *sect1(2)*; row 8, and *sect1(3)*; row 9.

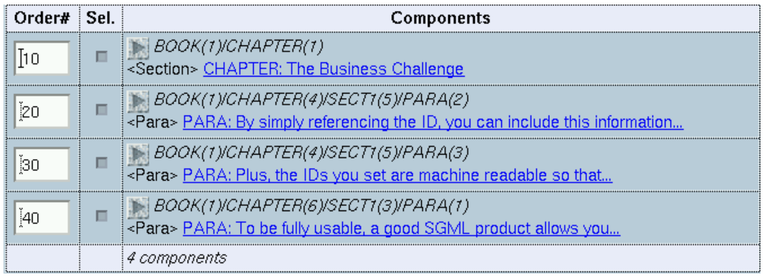
To produce a consistent result document, it is often reasonable to include chapter titles and introductory paragraphs that form the context of a chosen component. The *Complete* button includes components relevant to the selected ones in the current view. This helps the user construct a coherent document from the components chosen. More detailed discussion of the *Complete* operation can be found in Section 3.3.3.

In the example of Figure 6.4 the user has selected from our example book the whole Chapter 1, two paragraphs from Section 4.5, and one paragraph from Section 6.3. The *Complete* operation has been executed for all these components, giving the component list of Figure 6.5:

- Chapter 1 has become a *Completed <Section>*. The nearest ancestor of Chapter 1 is the element *book* (class *VirtDoc*); its first child, namely element *bookinfo* (class *ParaGrp*), has been retrieved.

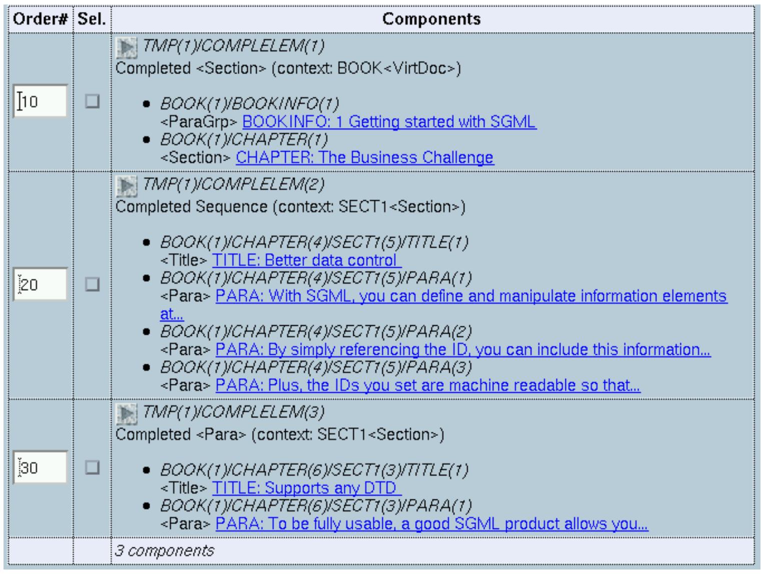
- Two paragraphs from Section 4.5 have formed a *Completed Sequence.* The ancestor of the paragraphs is naturally Section 4.5, and its title and first paragraph have been retrieved.

- The paragraph from Section 6.3 has become a *Completed <Para>*. The ancestor is Section 6.3, and its title has been retrieved. The chosen paragraph is the first one in the section, and thus no additional paragraphs have been retrieved.



Order#	Sel.	Components
10		BOOK(1)/CHAPTER(1) <Section> CHAPTER: The Business Challenge
20		BOOK(1)/CHAPTER(4)/SECT1(5)/PARA(2) <Para> PARA: By simply referencing the ID, you can include this information...
30		BOOK(1)/CHAPTER(4)/SECT1(5)/PARA(3) <Para> PARA: Plus, the IDs you set are machine readable so that...
40		BOOK(1)/CHAPTER(6)/SECT1(3)/PARA(1) <Para> PARA: To be fully usable, a good SGML product allows you...
		4 components

Figure 6.4: Components selected for the *Complete* operation.



Order#	Sel.	Components
10		TMP(1)/COMPLELEM(1) Completed <Section> (context: BOOK<VirtDoc>) • BOOK(1)/BOOKINFO(1) <ParaGrp> BOOKINFO: 1 Getting started with SGML • BOOK(1)/CHAPTER(1) <Section> CHAPTER: The Business Challenge
20		TMP(1)/COMPLELEM(2) Completed Sequence (context: SECT1<Section>) • BOOK(1)/CHAPTER(4)/SECT1(5)/TITLE(1) <Title> TITLE: Better data control • BOOK(1)/CHAPTER(4)/SECT1(5)/PARA(1) <Para> PARA: With SGML, you can define and manipulate information elements at... • BOOK(1)/CHAPTER(4)/SECT1(5)/PARA(2) <Para> PARA: By simply referencing the ID, you can include this information... • BOOK(1)/CHAPTER(4)/SECT1(5)/PARA(3) <Para> PARA: Plus, the IDs you set are machine readable so that...
30		TMP(1)/COMPLELEM(3) Completed <Para> (context: SECT1<Section>) • BOOK(1)/CHAPTER(6)/SECT1(3)/TITLE(1) <Title> TITLE: Supports any DTD • BOOK(1)/CHAPTER(6)/SECT1(3)/PARA(1) <Para> PARA: To be fully usable, a good SGML product allows you...
		3 components

Figure 6.5: Result of the executed *Complete* operation.

The SAW system also allows the user to browse the contents of the collection as *clusters* giving an extracted view of the documents. Thus, he or she can choose the appropriate clusters and fragments for re-clustering, and use these to search for appropriate fragments on new subsets of the document collection. The browsing mechanism employs the Scatter/Gather clustering techniques developed by Cutting et al. [CPKT92, CKP93]. SAW also offers a word index, which is useful in content-based searching. A link to the indexing service can be found in the upper left corner of the user

[2] The number in the parenthesis is the number of the element.

Figure 6.6: A word index of SAW.

interface; see Figure 6.1. The link is called *Index*.

Let us take an example of the word indexing and clustering facilities. In Figure 6.6 there is an index of 5 words beginning with letters “af”. If the user clicks the link containing the text “afganistan (15)”, the view of Figure 6.7 is returned with 15 fragments containing statutes from the Finnish law.

Next, if the user executes the command *Cluster selected into at most 3 clusters*, the result is the view of Figure 6.8. The clusters are very reasonable ones, since in the first cluster there are two acts on reduced duties on import from developing countries, in the second one regulations on tax-free travel expenses, and in the third cluster acts on restrictions in foreign trade. The representation of clusters includes a title link informing the number of fragments in the cluster, at most 3 central fragments closest to the cluster centroid, and a list of top-frequency terms. For instance, in the third cluster the keywords are “alankomaa (12)” (Netherlands) , “antillit (8)” (Antilles), “australia (8)” (Australia), “espanja (8)” (Spain), “brittiläinen (8)” (British), and “eteläinen (8)” (southern). The number in the parenthesis is the number of occurrences of the keyword in the documents.

The user may *search* for keywords in the selected components, in the current view, or in the entire collection. Components that contain the specified keywords are returned in a new view. The user may also search for fragments *similar* to the selected ones in the current view. Expert users may formulate powerful queries using the TranSID query language [JKL97] of the underlying database server (command *Evaluate TranSID query*). The result of an advanced query may be presented as a new view, an HTML page or plain text.

For instance, if the user is interested in finding material related to func-

Order#	Sel.	Components
10		Säädös: Verohallituksen päätös vuonna 1996 verosta vapaaksi katsottavien matkakustannusten korvausten perusteista ja määristä
20		Säädös: Verohallituksen päätös vuonna 1997 verosta vapaaksi katsottavien matkakustannusten korvaustenperusteista ja määristä
30		Säädös: Verohallituksen päätös vuonna 1995 verosta vapaaksi katsottavien matkakustannusten korvausten perusteista ja määristä
40		Säädös: Asetus kehitysmaista peräisin olevien tavaroiden tulli– ja tuontimaksuetuuksista
50		Säädös: Asetus maan ulkomaankaupassa sovellettavista määrällisistä rajoituksista annetun asetuksen liitteen muuttamisesta
60		Säädös: Verohallituksen päätös vuonna 1994 verosta vapaaksi katsottavien matkakustannusten korvausten perusteista ja määristä
70		Säädös: Asetus maan ulkomaankaupassa sovellettavista määrällisistä rajoituksista annetun asetuksen muuttamisesta
80		Säädös: Verohallituksen päätös vuonna 1993 verosta vapaaksi katsottavien matkakustannusten korvausten perusteista ja määristä
90		Säädös: Verohallituksen päätös vuonna 1992 verosta vapaaksi katsottavien matkakustannusten korvausten perusteista ja määristä
100		Säädös: Verohallituksen päätös työn suorittamisesta aiheutuneiden kustannusten arvioimisesta ennakonpidätystä toimitettaessa
110		Säädös: Asetus maan ulkomaankaupassa sovellettavista määrällisistä rajoituksista annetun asetuksen liitteen muuttamisesta
120		Säädös: Asetus maan ulkomaankaupassa sovellettavista määrällisistä rajoituksista annetun asetuksen liitteen muuttamisesta
130		Säädös: Asetus kehitysmaista peräisin olevien tavaroiden tulli– ja tuontimaksuetuuksista annetun asetuksen muuttamisesta
140		Säädös: Verohallituksen päätös vuodelta 1990 toimitettavassa verotuksessa verosta vapaaksi katsottavien matkakustannusten korvausten perusteista ja määristä
150		Säädös: Verohallituksen päätös vuonna 1991 verosta vapaaksi katsottavien matkakustannusten korvausten perusteista ja määristä
		15 components

Figure 6.7: The fragments containing the chosen keyword “afganistan”.

tions, he or she may execute a keyword search on the collection. In Figure 6.9 the keyword is “funktio” (function). The system finds one fragment containing the keyword (Figure 6.10) from a control engineering textbook. Because the search result included only one fragment, the user executes operation *Search similar to selected*. Now SAW finds a cluster of 8 similar fragments (Figure 6.11). Three of the most representative fragments of the cluster are enumerated in a list. By using the command *Zoom in* the user can browse the content of the fragment cluster.

When the user is satisfied with the current view he or she may save it for later use by command *Save view*. The current view may also be exported as an SGML (or XML) document. For example, in Figure 6.12 the user has selected one section from each of the documents in the original view (Figure 6.1). The original element names of the chosen sections are *chapter*, *div0*, *act*, and *hydraulics*, and all have a different element structure. If the view is the needed one, the user can save the assembly as an SGML (or XML) document by choosing the command *Export view as SGML*. The

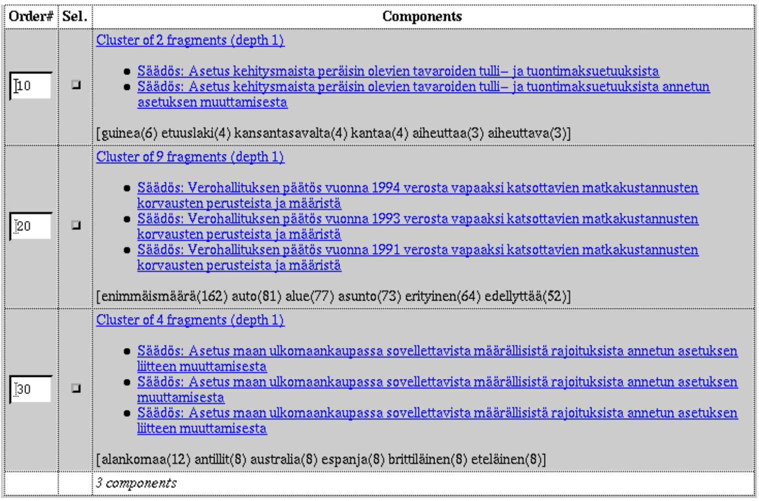


Figure 6.8: The fragments clustered into three clusters.

Figure 6.9: Search on keyword “funktio”.

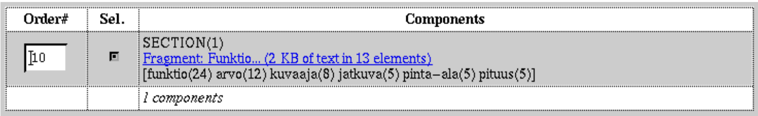


Figure 6.10: Search result using keyword “funktio”.

resulting document structure may be conforming to the generic DTD or to the HTML DTD. Although the components originally have heterogeneous structures and element names, the structure of the resulting document is guaranteed to conform to the generic DTD (or to the HTML DTD). In Figure 6.13 the assembly result is saved as an SGML document conforming to the generic DTD.

During the assembly process, the user may return to the original view of the collection, or to the saved views of the current session. The previous view may always be reached by clicking the *Back* button in the Web browser.

After saving the assembly result, the user can process the document

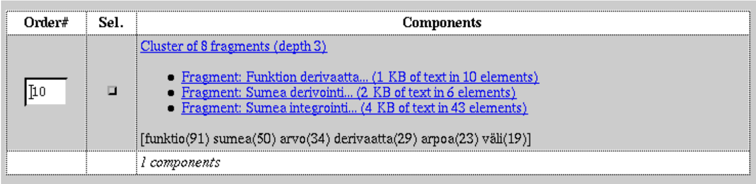


Figure 6.11: Result of the operation *Search similar.*

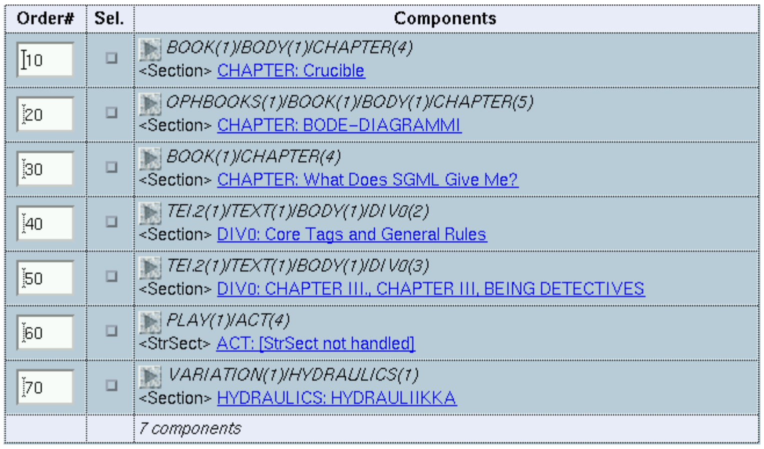


Figure 6.12: Seven chapters with heterogeneous structures.

using tools for structured documents, since the structure conforms to the generic DTD (or the HTML DTD). For instance, the user may design a stylesheet for the generic structure and import the document into a structure editor that facilitates the editing of large and complex documents and gives means to control the uniform appearance of related documents. In Figure 6.14 an assembly result is imported into a structure editor.

6.1.3 Architecture and implementation

The SID Assembly Workbench is implemented as a Web server. The architecture consists of a user interface running in a Web browser, a server interface, and a component server (Figure 6.15).

The view interface is implemented as an HTML form in a Web browser that is the client side of the SAW architecture. The form consists of a constant part containing the command buttons and a dynamic part presenting the contents of the current view. A hidden numeric identifier is attached

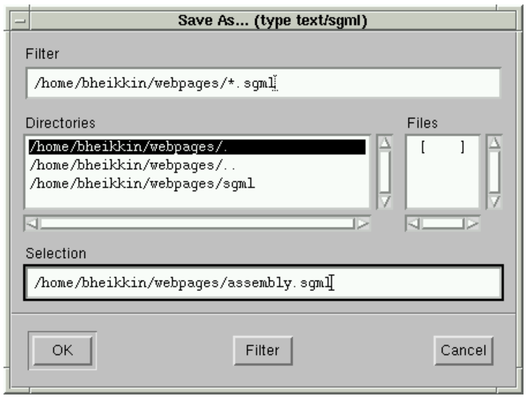


Figure 6.13: Saving the assembly result.

to each component, fragment, cluster, or view shown in the current view. The identifiers connect the assembly components of the current view to the corresponding document structures maintained by the component server. The state of the assembly process is maintained mainly by these identifiers in the form of the view. The HTTP post method is used for sending information about the modification action and the contents of the current view to the Web server. The server interface scripts parse the request and pass it as a query to the component server. The response of the component server is sent to the user as an HTML page.

The component server consists of a structured text database process. The structured text database is based on the internal data structures of the TranSID language. TranSID models documents as tree structures, and allows any part of the document tree to be accessed. The document structures are maintained in main memory, into which they are loaded only when the documents are accessed. The component server accepts queries expressed in the TranSID language, and responds by returning plain text, HTML, or partial HTML to be presented as a new view.

We chose the Web/HTML-based implementation of the user interface for several reasons. A Web browser offers a familiar look-and-feel and provides many of the user interface actions for free. HTML pages are easy to redesign, which makes maintaining and tailoring the system easier. This allows tailored initial views for different user groups. An HTML interface can also be used over an intranet or integrated in a Web site on the Internet. On the other hand, certain operations are less flexible with a form-based user interface than with a dedicated direct manipulation interface. For example,

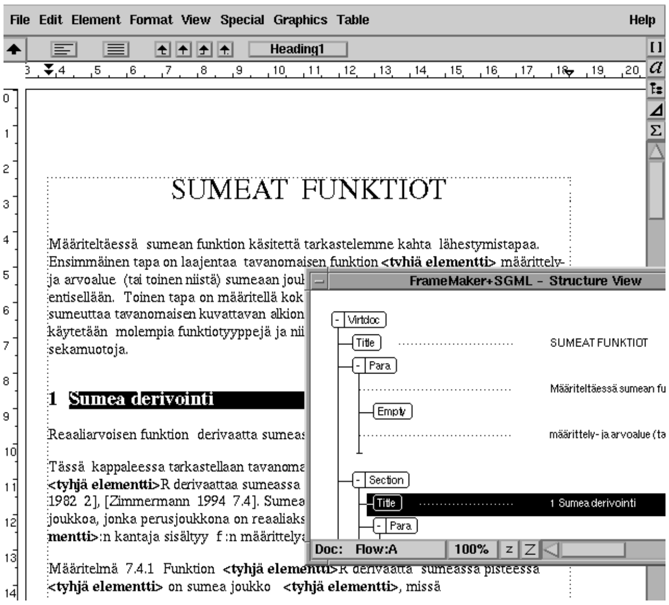


Figure 6.14: Assembly result in a structure editor.

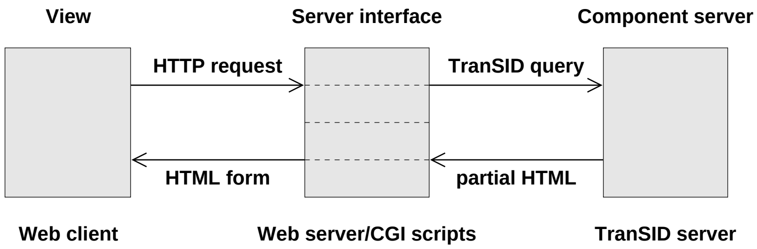


Figure 6.15: SAW architecture.

ordering of components is less flexible than it would be in a drag-and-drop environment. However, this is not an inherent limitation of the chosen architecture. A more elaborate implementation of the user interface, e.g., based on Java applets, would allow manipulation of the current view in a more direct fashion.

The system has been tested with ten document types having different DTDs. SAW has successfully processed 17 books and manuals of altogether several megabytes of text. SAW was also tested with part of our statute collection consisting of approximately 3 125 statutes[3]. The size of the statutes was about 12 megabytes, or 327 000 SGML elements.

6.2 System for tailoring textbooks via the Web

One application area we have considered is assembling educational material from digital libraries [AHHK97a]. A digital library may contain, for instance, textbooks, articles, exercise collections, and simulations. Such a digital library can be used in several ways. The publisher can print customized textbooks for various school branches and levels. Teachers can tailor compositions of WWW or CD-ROM delivered material, such as summaries, slides and tests, for their own classes. Students can also use the digital library directly as a data bank for problem-oriented learning or as an active textbook with computer-assisted learning features.

Recently, several publishers and universities have launched their own services for delivering customized textbooks [Pri97, CCW97, CP99]. Some related work was already described in Section 2.3. The approaches seem to use a fairly static set of document fragments that can be combined to form a book, and their user interfaces simply allow the user to choose a selection of these parts. We claim that dynamic compositions are needed, and to achieve this, SGML was used for the management of the material. Haake et al. have introduced the Individualized Electronic Newspaper [HHR94] which also uses SGML as the document presentation format. Their approach, although somewhat similar to ours, concentrates on gathering and presenting news-like material according to reader profiles, while our goal is more general: a coherent assembled and customized SGML document.

A new approach called The Slicing Books Technology [SIT99] is very promising. The system suggests slices of the book that most likely satisfy the needs of the reader, and delivers online excerpts of these parts. The system applies database and artificial intelligence techniques, and it can be accessed via the Web browser of the customer. The work is done in a joint project with Springer-Verlag, Slicing Information Technology, the Research Group on Artificial Intelligence in the University Koblenz-Landau, and itaw[4].

One of our SID project partners, the Finnish National Board of Education, has started to create a new digital library that will contain various educational material related to control engineering. Our project has participated in converting textbooks into SGML, and we also implemented a

[3]The statutes tested were not the same as in Table 5.6 of Section 5.1.
[4]Institut für Terminologie und angewandte Wissensforschung

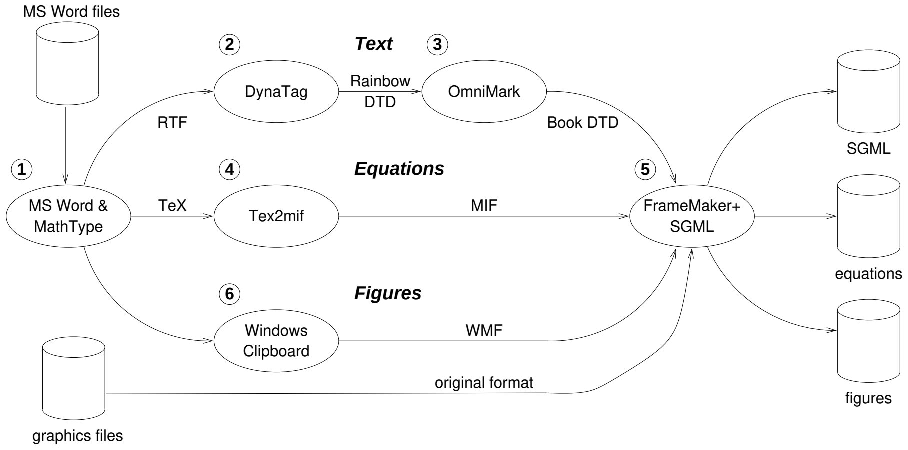


Figure 6.16: Conversion architecture.

system for tailoring specialized textbooks using the material. In general, the conversion of legacy documents into SGML (or XML) documents is a demanding process described, e.g., in [SHO$^+$96]. The following section presents the laborious conversion process from word processing documents into structured documents. Section 6.2.2 introduces the structure of the text and how it can be used in assembling. Section 6.2.3 describes the system architecture.

6.2.1 Conversion

The first components of the material pool were three text books on control engineering. The books, originally prepared using MS Word[5], contain many equations and technical pictures. Creating a structured SGML version required laborious conversion steps using tools such as MathType[6], DynaTag[7], OmniMark[8], and FrameMaker+SGML[9] (Figure 6.16). The basic structure for the books is the Book DTD of the ISO 12083 [ISO94] extended with element types for exercises, answers, examples, definitions, clauses, formulas and computer code.

In the worst case, the conversion process is as hard as writing the mate-

[5]MS Word is a trademark of Microsoft Corporation.
[6]MathType is a trademark of Design Science, Inc.
[7]DynaTag is a trademark of Electronic Book Technologies, Inc.
[8]OmniMark is a trademark of OmniMark Technologies Corporation.
[9]FrameMaker+SGML is a trademark of Adobe Systems Incorporated.

rial from scratch using a structure editor. The amount of manual editing in the conversion process depends largely on how the author uses a word processing system during the writing process. In other words, the more formal and consistent the original data, the more straightforward it is to convert. On the other hand, commercial conversion products do not offer feasible solutions for converting graphics and mathematical equations automatically. Moreover, at that time there was no widely used SGML markup language for representing equations. Finally, the richer and more complicated the required SGML structure is, the more complex and time consuming is the conversion from word processing data.

In the following, we present some problematic cases that were encountered in our material. First of all, text and graphic objects of the pictures were mixed in some Word documents. Thus, we had to re-edit hundreds of pictures in Word (Figure 6.16; (1)). The conversion of equations was also difficult. We used the MathType program with Word to convert Word equations to TEX format, and we implemented a filter called Tex2mif (Figure 6.16; (4)) to get the equations in editable format to FrameMaker+SGML. There were hundreds of pictures and thousands of equations; roughly half of the material was very hard to convert.

The word processor styles, such as heading, normal paragraph, caption, and bold, were used to recognize document structures (Figure 6.16; (2)). Unfortunately, the use of styles was inconsistent within the original documents — if they were used at all. Therefore, content recognition was used in addition to style-based recognition. It revealed that different authors had different ways to present, e.g., exercises, examples, and equation numbers. All different approaches had to be taken into account. Furthermore, it is fairly easy to notice the beginning of an example automatically, but it may be impossible to automatically decide where an example ends and a new paragraph begins. Hence, closing tags had to be added manually.

Some authors did not use the cross-reference mechanism of the word processing system, or even table and list tools; for instance, many tables were represented as picture objects within documents. Also, in some cases the words were hyphenated using hyphen signs ("-"), which had to be manually removed.

Word document styles and contents were recognized using the DynaTag program (Figure 6.16; (2)). DynaTag generated intermediate SGML documents conforming to the EBT's Rainbow DTD. Next, OmniMark transformation language was used to transform and refine the document structure according to our Book DTD (Figure 6.16; (3)). The converted SGML text, MIF equations, and pictures originally produced by graphics programs (such as Designer) were imported to FrameMaker+SGML (Figure 6.16; (5)), and finally the part of the pictures which had to be edited in Word were imported to FrameMaker+SGML via Windows Clipboard (Figure 6.16; (6)). The final editing and page layout was done in FrameMaker+SGML, which then was used to generate the SGML text, figure and equation files.

6.2.2 Structure of the documents

The major hierarchical structure elements of a book are chapters, sections, subsections and paragraphs. As a structure schema we used the extended ISO 12083 Book DTD. The books were enriched with internal supplementary information. Manual seeding based on information from the authors was necessary to classify the fragments of the material into appropriate categories. Each text paragraph was classified according to the level (vocational, college, university), the branch of specialization (metal industry, process industry, data communications, civil engineering, etc.) and the didactic form of the contents (introductory, base material, advanced, applied, summary, auxiliary, examination). Keyword elements were added automatically to the paragraphs containing occurrences of the terms that have a definition (normally recognizable as boldface words or phrases) in the book.

The categorization is represented in the text paragraphs and their superstructures as SGML attributes. Attribute values are propagated from the upper levels of the structure, if not overridden in the element. Figure 6.17 is an example of the supplementary information attached to the markup of the material.

The shown chapter was authored by Jari Savolainen (author="aJS"). It is recommended for students at the college level (level="b"). The content of Section 4.1 is base material (content="BB"). The example of the section is directed at students at the university level (level="c") and it contains advanced material (content="CC"). Keywords and their frequencies are listed explicitly. Formulas are in external entity files (attribute *entity* of element *eqbody*) in MIF format. They can also be represented in picture formats such as TIFF. Using the added markup, we can easily extract, e.g., base material for college level students, or exercises for university level students. Search features of existing commercial SGML browsers such as DynaText[10] or Panorama Pro[11] support this kind of querying.

[10]DynaText is a trademark of Inso Corporation.
[11]Panorama Pro is a trademark of SoftQuad, Inc.

```
<chapter id="c4" author="aJS" level="b">
  <title>S&Auml;&Auml;T&Ouml;LOHKOT TAAJUUSTASOSSA</title>
  ...
  <section id="c4.1" level="b" content="BB" branch="unspec">
    <title>Impedanssi-kompleksinen vastus</title>
    ...
    <example id="c4ex1" level="c" content="CC" branch="unspec">
      <figgrp id="c4fg2">
        <fig entity="c4f2">
        <caption>Impedanssin sis&auml;lt&auml;m&auml; ...</caption>
      </figgrp>
      <p><keyword freq=1>kokonaisimpedanssi</keyword>
      Laske sarjakytkenn&auml;n kokonaisimpedanssi U = 5 V</p>
      <answer>
        <p><keyword freq=1>kokonaisimpedanssi</keyword>
        Kokonaisimpedanssi on sarjaankytkettyjen impedanssien summa.
        <eq id="c4df13"><eqbody entity="c4d13"></eq></p>
        ...
```

Figure 6.17: An example of internal supplementary information.

6.2.3 System architecture

Before constructing our general assembly system SAW, we built a specialized assembly system that gives teachers a possibility to order customized control engineering textbooks. An overview of the system can be seen in Figure 6.18.

An order of a customized textbook is processed as follows.

1. **Order.** The client fills in an HTML form and submits it.

2. **Processing of order.** The order form is processed. The composition of the assembled document is created based on the input from the client and some heuristic rules. The table of contents is returned to the client. If the client confirms the order, the specification of the composition is submitted to the customization process.

3. **Assembly.** According to the input for the assembly, which is represented by a list of file names, the customization process retrieves the desired texts and images from the SGML collection and composes the new textbook. The textbook is formatted and the page layout is manually checked. Finally, a PostScript file is submitted to the printing house.

4. **Printing.** The printing house produces the textbooks and delivers them to the client.

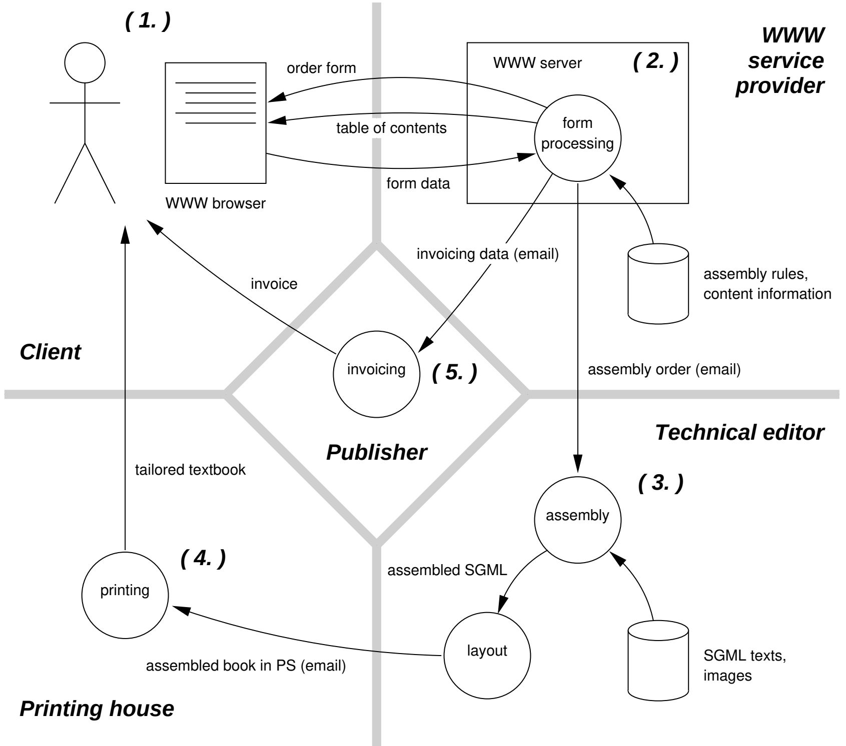


Figure 6.18: Overview of the system.

5. **Invoicing.** The publisher sends an invoice to the client.

As a user interface for the teachers we use an HTML form (Figure 6.19). The form contains ordering information and a table of contents for each of the three books. The ordering information includes the contact information of the client, price, delivery and publisher information, and instructions for using the form. In the table of Figure 6.20 the price of a book in Finnish marks is presented as a function of circulation and page count — the bigger the circulation the cheaper the book.

The fragments that can be used for selection in the table of contents are chapters, sections, subsections, and, in some cases, author-defined sequences consisting of several paragraphs. The columns in the table of contents are (from left to right) the following:

1. **Choice (“Valinta”).** By checking choice buttons the user selects corresponding structures of the books. The user can select whole

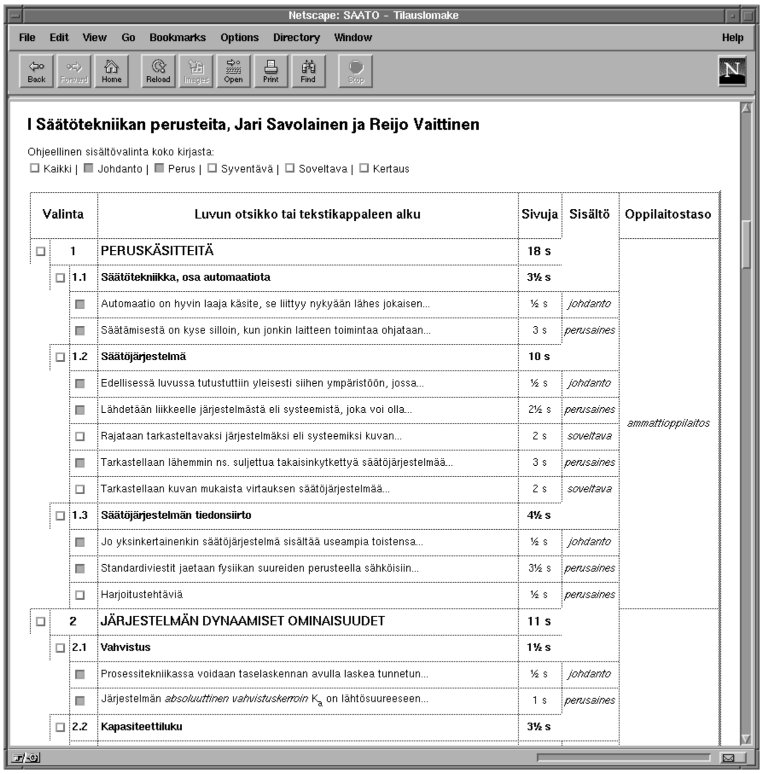


Figure 6.19: Order form.

chapters, or just some sections, subsections, or predefined paragraph sequences.

2. **Title or beginning of text ("Luvun otsikko tai...").** Original chapter, section or subsection title, or the beginning of the text of the paragraph sequence.

3. **Page count ("Sivuja").** Length of the part.

4. **Content ("Sisältö").** For example, introductory ("johdanto"), base material ("perusaines"), and applied ("soveltava").

5. **Level ("Oppilaitostaso").** For example, "ammattioppilaitos" (vocational).

The rightmost column indicates the lowest school level for which the part is recommended. At the moment this knowledge is only given to the

Book price	*Circulation*			
Page count	*30*	*60*	*120*	*200*
60	69	49	39	35
100	87	64	52	48
160	113	87	73	68
240	149	117	101	95

Figure 6.20: Price of a book in Finnish marks when circulation and page count is considered.

teacher as a hint; it is not used in any automated way. As the digital library does not contain any branch-specific material yet, we have not utilized the values of the *branch* attribute.

The six checkboxes above the table in Figure 6.19 are shortcuts. They enable the user to select multiple items by one click. The first button selects all the material, and the others select all parts of the named content, such as introductory ("johdanto"), respectively. The selection approach could also be extended to other attributes. As the selections have several dimensions, it is not at all obvious, how all the choices should be presented to the client; the order form easily becomes too complicated. One solution would be to offer predefined assemblies designed by some teachers as a starting point.

In order to achieve sensible results, the assembly has to utilize heuristics which guarantee the inclusion of all necessary fragments. For example, if only some sections of a chapter are selected, then the title of the chapter and the introductory paragraphs are also included.

If the ordered books are delivered on paper, the problem of predictability arises, because attributes and heuristics are used. Hence, the teacher should be able to check the contents of the result. After the teacher has submitted the form, the ordering system immediately returns the table of contents for the assembled textbook, as well as the number of pages and the price of the order. In addition, we assume that the teacher has a paper copy of all the original books, when he or she selects the fragments to be included. Instead of the table of contents, the whole assembled textbook could be sent to the client, e.g., by email.

After the confirmation of the order, a list of file names is submitted to the customization process. The process retrieves the required files and concatenates them to form one SGML document. As the DTD of all the assembled textbooks is the same as the original, it is possible to create the formatting layout nearly automatically using FrameMaker+SGML. Due to some problematic cases, such as improper page breaks, a manual inspection

Figure 6.21: The cover to be personalized.

is still needed.

Finally, the textbooks are produced by a printing house. In our system a preprinted cover page (Figure 6.21) is used, on which the personalized headings are printed. The cover is designed by a graphic artist. Naturally, the names of the authors of the original books appear first in the cover; author information is placed in the space reserved for fields "Tekijä 1 -" and "Tekijä 2 -" in Figure 6.21. After that, the name of the ordering client may be printed as an editor for the book, located in field "Simo Säätäjä (toim.)". The user can also give the name of the tailored book using at most three rows ("Teoksen nimi jopa kolmessakin rivissä"). Finally, the name of the ordering institution may also be printed along with the name of the publisher (field "Toimittava oppilaitos/Opetushallitus"). In the end, the printing house delivers the printed books to the client.

The writers of the material were interested in the project and participated in designing the properties of the electronic material as well as provided the supplementary information needed. They were also very enthusiastic in combining multimedia material, e.g., simulations, animations, exercises and examples, to the material pool. The market for the books considered in this experimentation is quite small, since at most a few hundred students use the books annually. Moreover, the assembly methodology is profitable in practice only if the material pool is quite large, some parts of the material will be updated quite often, and the basic material has a long life cycle. The application of this kind of a new technology needs quite large investments in time and in money from the publisher.

Our tailoring system produces high-quality textbooks, but its applicability is restricted to fairly small document collections. When large and heterogeneous digital libraries are concerned, the user should be offered more flexible ways to filter information and construct meaningful combinations. In the following list the characteristics and restrictions of our tailoring system are compared to the functions available in our general purpose assembly system SAW.

- In the textbook tailoring system the user selects chapters and sections by marking them up in the table of contents. This is not feasible if the document collection is large. However, the SAW system offers several ways to select elements to be included in the assembly.
- In the textbook tailoring system the user can only prune the existing books; mixing the sections of different books is not possible, and even within one book it is not possible to change the order of the sections. This kind of modifications are, nevertheless, often desired. In the SAW system the user can modify the set of selected document fragments to obtain a final textbook by rearranging and discarding the elements. The result of the selection phase in SAW is an ordered list of document components with the internal tree structures.
- In the textbook tailoring system all the documents share a common DTD, and the resulting textbook is of the same type. Thus, the overall structure of the new document is known, and we can also utilize the formatting declarations of the original documents. This is not true in the general case. The source documents may have differing structures, and an arbitrary composition of their fragments does not belong to any known document type. Therefore, no formatting instructions are available, i.e., the documents cannot be browsed or printed in the formatted form. However, in the SAW system these problems are solved, and the structure of the assembled document conforms to the generic DTD for which we can produce formatting instructions.

Chapter 7

Conclusions

In this thesis we have presented the element-type classification method that identifies typical logical structures of electronic documents, such as sections, paragraphs, and strings, and containers for these structures. The identification is based on element relations and on average text lengths of element instances and their subtrees. The method maps each element type of a document type into a generic element type called generic class. The definitions of the generic classes constitute the generic DTD, which can be seen as a metastructure definition for typical electronic documents.

In this work, we have shown using grammar morphisms that when a document instance is classified and when the element names are replaced with the generic class names, the resulting document conforms to our generic DTD. The element-type classification method has been tested on several public and private document types, on altogether 103 megabytes of structured text. The tests show that the method is applicable in practice.

We have also described a general document assembly model where a user can manipulate a large number of heterogeneously structured documents. The assembly process consists of the initialization of the collection, the search and selection of document fragments or components, and the construction of a coherent result. The initialization actions include the element-type classification, the text fragmentation, and the fragment clustering. In our model, interesting document components are iteratively searched by content and structure. The cluster-based browsing of the collection may also be used to facilitate searching. When the desired components are selected and reorganized, a coherent assembly result is automatically constructed. The result may be exported as an XML or SGML document conforming to our generic DTD, or the result may be automatically transformed into HTML, PDF, or PostScript format.

To test our ideas of document assembly we have developed a specialized

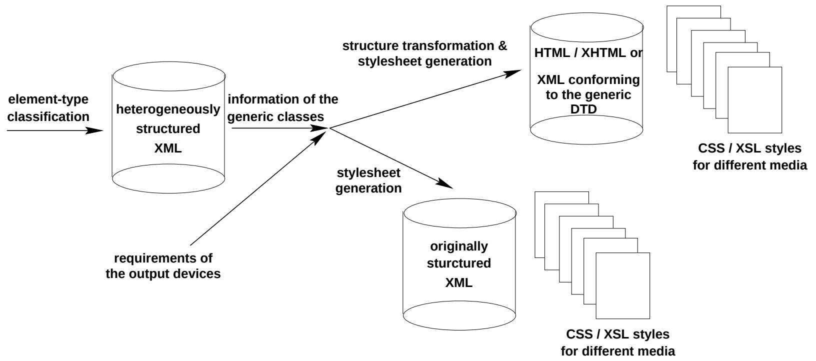


Figure 7.1: Generation of stylesheets for different media.

system for tailoring textbooks via the Web and a general assembly system called SAW that is designed to be used as an in-house publishing system. If the assembly interface of SAW is offered to the open public, concerns of copyright and security have to be taken into account. The user interface should also be re-designed to better answer the needs of an ordinary user.

Several research topics are open including extraction of useful phrases for querying, filtering of multiple text streams, intelligent text fragmentation, and producing formatted output for heterogeneously structured texts which do not have formatting instructions available [AHHK99].

The idea in producing formatted output for heterogeneously structured texts is to automatically or semi-automatically generate a stylesheet for a given document type based on the information extracted from the document instances by the element-type classification method. When the generic classes of the original element types are known, the formatting instructions designed for the generic classes can be used in the generation of the stylesheet for the given document type [AHHK00]. A common need is to produce output of the same content on different media requiring different formatting instructions. In addition to the information of the generic classes, the specific requirements of the output devices can be take into account when generating stylesheets for a document structure; see Figure 7.1. The original XML structure may be preserved, or the structure can be transformed to obey the HTML DTD, the XHTML DTD, or our generic DTD.

Another future application is to automatically or semi-automatically form fragments of a Web page to fit displays of different devices, like a cellular phone or a small portable computer. Such fragmentation could use information inferable from the known HTML structure of a web page, the basic ideas of the element-type classification (element relations and average text lengths), and the text fragmentation method in generating fragments suitable for a specific display.

References

[Abi97a] S. Abiteboul. Object database support for digital libraries. In *Proceedings of the 1st European Conference on Research and Advanced Technology for Digital Libraries (ECDL'97)*, number 1324 in Lecture Notes in Computer Science, pages 11–23, Pisa, Italy, September 1997. Springer.

[Abi97b] S. Abiteboul. Querying semi-structured data. In *Proceedings of the 6th International Conference on Database Theory (ICDT'97)*, number 1186 in Lecture Notes in Computer Science, pages 1–18, Delphi, Greece, January 1997. Springer.

[Abi99] S. Abiteboul. On views and XML. In *Proceedings of the 18th ACM Symposium on Principles of Database Systems (PODS'99)*, pages 1–9, Philadelphia, Pennsylvania, USA, May/June 1999. ACM Press.

[ACC+97] S. Abiteboul, S. Cluet, V. Christophides, T. Milo, G. Moerkotte, and J. Siméon. Querying documents in object databases. *International Journal on Digital Libraries*, 1(1):5–19, April 1997.

[ACM93] S. Abiteboul, S. Cluet, and T. Milo. Querying and updating the file. In *Proceedings of the 19th International Conference on Very Large Databases (VLDB'93)*, pages 73–84, Dublin, Ireland, August 1993. Morgan Kaufmann.

[ACM97] S. Abiteboul, S. Cluet, and T. Milo. Correspondence and translation for heterogeneous data. In *Proceedings of the 6th International Conference on Database Theory (ICDT'97)*, number 1186 in Lecture Notes in Computer Science, pages 351–363, Delphi, Greece, January 1997. Springer.

[ACM98] S. Abiteboul, S. Cluet, and T. Milo. A logical view of structured files. *The VLDB Journal*, 7(2):96–114, May 1998.

[Ade98] B. Adelberg. NoDoSe: A tool for semi-automatically extracting structure and semistructured data from text documents. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'98)*, pages 283–294, Seattle, Washington, USA, June 1998. ACM Press.

[AGMPY98] S. Abiteboul, H. Garcia-Molina, Y. Papakonstantinou, and R. Yerneni. Fusion queries over Internet databases. In *Advances in Database Technology - Proceedings of the 6th International Conference on Extending Database (EDBT'98)*, number 1377 in Lecture Notes in Computer Science, pages 57–71, Valencia, Spain, March 1998. Springer.

[AHH+96a] H. Ahonen, B. Heikkinen, O. Heinonen, J. Jaakkola, P. Kilpeläinen, G. Lindén, and H. Mannila. Intelligent assembly of structured documents. Technical Report C-1996-40, Department of Computer Science, University of Helsinki, June 1996.

[AHH+96b] H. Ahonen, B. Heikkinen, O. Heinonen, J. Jaakkola, P. Kilpeläinen, G. Lindén, and H. Mannila. Constructing tailored SGML documents. In *Proceedings of SGML Finland '96*, pages 106–116, Espoo, Finland, October 1996. SGML Users Group Finland.

[AHH+98a] H. Ahonen, B. Heikkinen, O. Heinonen, J. Jaakkola, P. Kilpeläinen, and G. Lindén. Design and implementation of a document assembly workbench. In *Electronic Publishing, Artistic Imaging, and Digital Typography, Proceedings of the 7th International Conference on Electronic Publishing (EP'98)*, number 1375 in Lecture Notes in Computer Science, pages 476–486, Saint-Malo, France, April 1998. Springer.

[AHH+98b] H. Ahonen, B. Heikkinen, O. Heinonen, J. Jaakkola, and M. Klemettinen. Analysis of document structures for element type classification. In *Proceedings of International Workshop on Principles of Digital Document Processing (PODDP'98)*, number 1481 in Lecture Notes in Computer Science, pages 24–42, Saint Malo, France, March 1998. Springer.

[AHHK97a] H. Ahonen, B. Heikkinen, O. Heinonen, and P. Kilpeläinen. Assembling documents from digital libraries. In *Proceedings* *of the 8th International Conference on Database and Expert Systems Applications (DEXA'97)*, number 1308 in Lecture Notes in Computer Science, pages 419–429, Toulouse, France, September 1997. Springer.

[AHHK97b] H. Ahonen, B. Heikkinen, O. Heinonen, and M. Klemettinen. Improving the accessibility of SGML documents — A content-analytical approach. In *SGML Europe '97*, pages 321–327, Barcelona, Spain, May 1997. GCA.

[AHHK99] H. Ahonen, B. Heikkinen, O. Heinonen, and M. Klemettinen. New tools for a knowledge worker. In *Proceedings of XML Finland '99*, pages 25–32, Helsinki, Finland, September 1999. SGML Users Group Finland.

[AHHK00] H. Ahonen, B. Heikkinen, O. Heinonen, and M. Klemettinen. Printing structured text without stylesheets. In *Proceedings of XML Scandinavia 2000*, Gothenburg, Sweden, May 2000. To be published by the XML/SGML Users Groups of Denmark, Finland, Norway, and Sweden.

[Aho96] H. Ahonen. *Generating grammars for structured documents using grammatical inference methods.* PhD thesis, Series of Publications A, Report A–1996–4, Department of Computer Science, University of Helsinki, Finland, November 1996.

[ARL95] Definition and purposes of a digital library. Association of Research Libraries, October 1995. Available at `http://sunsite.berkeley.edu/ARL/definition.html`.

[BANY97] K. Böhm, K. Aberer, E. J. Neuhold, and X. Yang. Structured document storage and refined declarative and navigational access mechanisms in HyperStorM. *The VLDB Journal*, 6(4):296–311, November 1997.

[BCGP97] M. Baldonado, C.-C. K. Chang, L. Gravano, and A. Paepcke. The Stanford Digital Library metadata architecture. *International Journal on Digital Libraries*, 1(2):108–121, September 1997.

[BCV99] S. Bergamaschi, S. Castano, and M. Vincini. Semantic integration of semistructured and structured data sources. *SIGMOD Record*, 28(1):54–59, March 1999.

[BGL+99] C. K. Baru, A. Gupta, B. Ludäscher, R. Marciano, Y. Papakonstantinou, P. Velikhov, and V. Chu. Xml-based information mediation with MIX. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'99)*, pages 597–599, Philadephia, Pennsylvania, USA, June 1999. ACM Press.

[BK93a] A. Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197–213, 1993.

[BK93b] A. Brüggemann-Klein. Unambiguity of extended regular expressions in SGML document grammars. In *Proceedings of the 1st Annual European Symposium on Algorithms (ESA '93)*, pages 73–84, Bad Honnef, Germany, September 1993. Springer.

[BKW98] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and Computation*, 140(2):229–253, 1998.

[BLMMR99a] M. S. Bonn, W. P. Lougee, J. K. MacKie-Mason, and J. F. Riveros. A report on the PEAK experiment: Context and design. *D-Lib Magazine*, June 1999. Available at `http://www.dlib.org/dlib/june99/06bonn.html`.

[BLMMR99b] M. S. Bonn, W. P. Lougee, J. K. MacKie-Mason, and J. F. Riveros. A report on the PEAK experiment: Usage and economic behavior. *D-Lib Magazine*, July/August 1999. Available at `http://www.dlib.org/dlib/july99/mackie-mason/07mackie-mason.html`.

[BMR+99] D. Bearman, E. Miller, G. Rust, J. Trant, and S. Weibel. A common model to support interoperable metadata: Progress report on reconciling metadata requirements from the Dublin Core and INDECS/DOI Communities. *D-Lib Magazine*, January 1999. Available at `http://www.dlib.org/dlib/january99/bearman/01bearman.html`.

[Bor93] R. Bordin. *Alice Freeman Palmer: The Evolution of a New Woman.* The University of Michigan Press, Ann Arbor, Michigan, USA, 1993. Available at `http://www.press.umich.edu/bookhome/bordin/`.

[Bos97] Plays of Shakespeare according to the Play XML DTD. Jon Bosak, 1997. Available at `http://www.oasis-open.org/cover/xmlSamplesBosak.html`.

[Bra98] T. Bray. RDF and metadata. *XML.com*, June 9, 1998. Available at `http://www.xml.com/xml/pub/98/06/rdf.html`.

[Bun97] P. Buneman. Semistructured data. In *Proceedings of the 16th ACM Symposium on Principles of Database Systems (PODS'97)*, pages 117–121, Tucson, Arizona, USA, May 1997. ACM Press.

[CACS94] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'94)*, pages 313–324, Minneapolis, Minnesota, May 1994. ACM Press.

[CCW97] Custom CourseWare. McMaster University Bookstore, 1997. Available at `http://bookstore.services.mcmaster.ca/home/ccw/ccw.html`.

[CDF97] V. Christophides, M. Dörr, and I. Fundulaki. A semantic network approach to semi-structured documents repositories. In *Proceedings of the 1st European Conference on Research and Advanced Technology for Digital Libraries (ECDL'97)*, number 1324 in Lecture Notes in Computer Science, pages 305–324, Pisa, Italy, September 1997. Springer.

[CGL97] D. Calvanese, G. D. Giacomo, and M. Lenzerini. Representing and reasoning on SGML documents. In *Foundations of Intelligent Systems, Proceedings of the 10th International Symposium on Methodologies for Intelligent Systems (ISMIS'97)*, number 1325 in Lectute Notes in Artificial Intelligence, pages 540–549, Charlotte, North Carolina, USA, October 1997. Springer.

[CGL98] D. Calvanese, G. D. Giacomo, and M. Lenzerini. What can knowledge representation do for semi-structured data? In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98) and 10th Innovative Applications of Artificial Intelligence Conference (IAAI'98)*, pages 205–210, Madison, Wisconsin, USA, July 1998. AAAI Press / The MIT Press.

[CGL99] D. Calvanese, G. D. Giacomo, and M. Lenzerini. Representing and reasoning on XML documents: a description logic approach. *Journal of Logic and Computation*, 9(3):295–318, 1999.

[CGLV99] D. Calvanese, G. D. Giacomo, M. Lenzerini, and M. Y. Vardi. Query answering using views for data integration over the Web. In *Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB'99)*, pages 73–78, Philadelphia, Pennsylvania, USA, June 1999. INRIA.

[CGM98] C.-C. K. Chang and H. Garcia-Molina. Conjunctive constraint mapping for data translation. In *Proceedings of the 3rd ACM International Conference on Digital Libraries (DL'98)*, pages 49–58, Pittsburgh, PA, USA, June 1998. ACM Press.

[CGM99] C.-C. K. Chang and H. Garcia-Molina. Query mapping across heterogeneous information sources. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'99)*, pages 335–346, Philadephia, Pennsylvania, USA, June 1999. ACM Press.

[Che97] D. R. Chesnutt. The Model Editions Partnership: "Smart text" and beyond. *D-Lib Magazine*, July/August 1997. Available at `http://www.dlib.org/dlib/july97/07chesnutt.html`.

[CKP93] D. R. Cutting, D. R. Karger, and J. O. Pedersen. Constant interaction-time Scatter/Gather browsing of very large document collections. In *Proceedings of the 16th ACM International Conference on Research and Development in Information Retrieval (SIGIR'93)*, pages 126–134, Pittsburgh, PA, USA, June 1993. ACM, New York.

[CP99] Custom Publishing. Thomson Learning, 1999. Available at `http://custom.thomsonlearning.com/`.

[CPKT92] D. R. Cutting, J. O. Pedersen, D. Karger, and J. W. Tukey. Scatter/Gather: A cluster-based approach to browsing large document collections. In *Proceedings of the 15th ACM International Conference on Research and Development in Information Retrieval (SIGIR'92)*, pages 318–329, Copenhagen, Denmark, June 1992. ACM, New York.

[CR94] V. Christophides and A. Rizk. Querying structured documents with hyperlinks using OODBMS. In *Proceedings of the ACM European Conference on Hypertext Technology (ECHT'94)*, pages 186–197, Edinburgh, Scotland, UK, September 1994. ACM.

[DCM99] Dublin Core Metadata Initiative. *Dublin Core Metadata Element Set, Version 1.1: Reference Description*, July 1999. Available at `http://purl.oclc.org/dc/documents/rec-dces-19990702.htm`.

[DD99] R. Domenig and K. R. Dittrich. An overview and classification of mediated query systems. *SIGMOD Record*, 28(3):63–72, September 1999.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Maier, and D. Suciu. Querying XML data. *IEEE Data Engineering Bulletin*, 22(3):27–34, September 1999.

[DFS99] A. Deutsch, M. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'99)*, pages 431–442, Philadephia, Pennsylvania, USA, June 1999. ACM Press.

[DH95] H. Davis and J. Hey. Automatic extraction of hypermedia bundles from the digital library. In *Proceedings of the 2nd Annual Conference on the Theory and Practice of Digital Libraries (DL'95)*, Austin, Texas, USA, June 1995. Hypermedia Research Lab, Computer Science Department, Texas A&M University.

[DLF98] Digital Library Federation. *The Making of America II Testbed Project: White Paper Version 2.0*, September 1998. Available at `http://sunsite.berkeley.edu/moa2/wp-v2.html`.

[DMRA97] L. M. L. Delcambre, D. Maier, R. Reddy, and L. Anderson. Structured Maps: modeling explicit semantics over a universe of information. *International Journal on Digital Libraries*, 1(1):20–35, April 1997.

[DuC99] B. DuCharme. RDF: Adding structure to the Web. *<TAG> Newsletter*, pages 1–3, May 1999.

[EJN99] D. Embley, Y. Jiang, and Y.-K. Ng. Record-boundary discovery in web documents. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'99)*, pages 467–478, Philadephia, Pennsylvania, USA, June 1999. ACM Press.

[Els96] Elsevier Science. *TULIP Final Report*, 1996. Available at `http://www.elsevier.nl/homepage/about/resproj/tulip.shtml`.

[ET94] P. M. English and R. Tenneti. Interleaf active documents. *Electronic Publishing — Origination, Dissemination and Design*, 7(2):75–87, June 1994.

[FLM98] D. Florescu, A. Levy, and A. Mendelzon. Database techniques for the World Wide Web: A survey. *SIGMOD Record*, 27(3):59–74, September 1998.

[GMPQ+97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. D. Ullman, V. Vassalos, and J. Widom. The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems (JIIS)*, 8(2):117–132, March/April 1997.

[GMW99] R. Goldman, J. McHugh, and J. Widom. From semistructured data to XML: Migrating the Lore data model and query language. In *Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB'99)*, pages 25–30, Philadelphia, Pennsylvania, USA, June 1999. INRIA.

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

[GP98a] C. F. Goldfarb and P. Prescod. *The XML Handbook*. Prentice-Hall, Inc., 1998.

[GP98b] L. Gravano and Y. Papakonstantinou. Mediating and metaseraching on the internet. *IEEE Data Engineering Bulletin*, 21(2):28–36, June 1998.

[Hei98] O. Heinonen. Optimal multi-paragraph text segmentation by dynamic programming. In *Proceedings of the 36th Annual Meeting of the Association for Computational Linguistics and the 17th International Conference on Computational*

Linguistics (COLING-ACL'98), Vol. II, pages 1484–1486, Montreal, Quebec, Canada, August 1998. ACL / Morgan Kaufmann Publishers.

[HHR94] A. Haake, C. Hüser, and K. Reichenberger. The individualized electronic newspaper: an example of an active publication. *Electronic Publishing — Origination, Dissemination and Design*, 7(2):89–111, June 1994.

[HK97] M. A. Hearst and C. Karadi. Cat-a-Cone: An interactive interface for specifying searches and viewing retrieval results using a large category hierarchy. In *Proceedings of the 20th ACM International Conference on Research and Development in Information Retrieval (SIGIR'97)*, pages 246–255, Philadelphia, PA, USA, July 1997. ACM.

[Hol99] G. K. Holman. What's the big deal with XSL? *XML.com*, April 22, 1999. Available at `http://www.xml.com/xml/pub/1999/04/holman/xsl.html`.

[HST99] T. Härder, G. Sauter, and J. Thomas. The intrinsic problems of structural heterogeneity and an approach to their solution. *The VLDB Journal*, 8(1):25–43, May 1999.

[HU69] J. E. Hopcroft and J. D. Ullman. *Formal Languages and Their Relation to Automata*. Addison Wesley, Reading, MA, 1969.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, Reading, MA, 1979.

[HY97] S. Hara and H. Yasunaga. Markup and conversion of Japanese classical texts using SGML in the National Institute of Japanese Literature. *D-Lib Magazine*, July/August 1997. Available at `http://www.dlib.org/dlib/july97/japan/07hara.html`.

[IET98a] IETF. *RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax*, August 1998. Available at `http://www.ics.uci.edu/pub/ietf/uri/rfc2396.txt`.

[IET98b] IETF. *RFC2413: Dublin Core Metadata for Resource Discovery*, September 1998. Available at `http://www.ietf.org/rfc/rfc2413.txt`.

[IFF+99] Z. G. Ives, D. Florescu, M. Friedman, A. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'99)*, pages 299–310, Philadephia, Pennsylvania, USA, June 1999. ACM Press.

[ISO86] ISO. *Information Processing — Text and Office Systems — Standard Generalized Markup Language (SGML), ISO 8879*, 1986.

[ISO92] ISO/IEC. *Information technology, Hypermedia/Time-based Structuring Language (HyTime), ISO/IEC 10744*, 1992.

[ISO94] ISO. *Information and documentation — Electronic manuscript preparation and markup, ISO 12083*, 1994. SGML DTDs are available at `http://ftp.sunet.se/pub/text-processing/sgml/ISO12083/`, XML DTDs are available at `http://www.xmlxperts.com/12083.htm`.

[ISO96] ISO/IEC. *Information technology — Processing Languages — Document Style Semantics and Specification Language (DSSSL) ISO/IEC 10179*, 1996.

[ISO97] ISO/IEC. *Information technology, Hypermedia/Time-based Structuring Language (HyTime), 2nd Edition, ISO/IEC 10744*, 1997.

[JKL97] J. Jaakkola, P. Kilpeläinen, and G. Lindén. TranSID: An SGML tree transformation language. In *Proceedings of the Fifth Symposium on Programming Languages and Software Tools*, pages 72–83, Jyväskylä, Finland, June 1997. Technical Report C-1997-37, Univ. of Helsinki, Dept. of Computer Science.

[KF97] B. Klein and P. Fankhauser. Error tolerant document structure analysis. *International Journal on Digital Libraries*, 1(4):344–357, December 1997.

[Kil99] P. Kilpeläinen. SGML & XML content models. *Markup Languages: Theory & Practice*, 1(2):53–76, 1999.

[Kim96] W. E. Kimber. Re-usable SGML: Why I demand SUBDOC. In *SGML '96*, Boston, USA, 1996. GCA.

[Kui96] E. Kuikka. *Processing of Structured Documents Using a Syntax-Directed Approach.* PhD thesis, Kuopio University Publications C, Natural and Enviromental Sciences 53, Finland, 1996.

[KW96] P. Kilpeläinen and D. Wood. SGML and exceptions. In *Proceedings of International Workshop on Principles of Document Processing (PODP'96)*, number 1293 in Lecture Notes in Computer Science, pages 39–49, Palo Alto, California, USA, September 1996. Springer.

[Lal97] M. Lalmas. Dempster-Shafer's theory of evidence applied to structured documents: Modelling uncertainty. In *Proceedings of the 20th ACM International Conference on Research and Development in Information Retrieval (SIGIR'97)*, pages 110–118, Philadelphia, PA, USA, July 1997. ACM.

[LAW99] T. Lahiri, S. Abiteboul, and J. Widom. Ozone: Integrating structured and semistructured data. In *Proceedings of the 8th International Workshop on Database Programming Languages (DBPL'99)*, Edinburgh, Scotland, UK, September 1999. Will be published by Springer.

[Lev93] D. M. Levy. Document reuse and document systems. *Electronic Publishing — Origination, Dissemination and Design*, 6(4):339–348, December 1993.

[Lev99] M. Leventhal. XSL considered harmful: Declaration of war. *XML.com*, May 20, 1999. Available at `http://xml.com/xml/pub/1999/05/xsl/xslconsidered_1.html`.

[LHKK96] K. Lagus, T. Honkela, S. Kaski, and T. Kohonen. Self-organizing maps of document collections: A new approach to interactive exploration. In *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining (KDD'96)*, pages 238–243, Portland, Oregon, USA, 1996. AAAI Press.

[Lin97] G. Lindén. *Structured Document Transformations*. PhD thesis, Series of Publications A, Report A–1997–2, Department of Computer Science, University of Helsinki, Finland, June 1997.

[LPV99] B. Ludäscher, Y. Papakonstantinou, and P. Velikhov. A framework for navigation-driven lazy mediators. In *Proceedings of the ACM SIGMOD Workshop on the Web and Databases (WebDB'99)*, pages 85–90, Philadelphia, Pennsylvania, USA, June 1999. INRIA.

[McF96] J. McFadden. Hybrid distributed database (HDDB) and the future of SGML. In *SGML Europe '96*, pages 321–327, Munich, Germany, 1996. GCA.

[Mer97] D. Merkl. Exploration of text collections with hierarchical feature maps. In *Proceedings of the 20th ACM International Conference on Research and Development in Information Retrieval (SIGIR'97)*, pages 186–195, Philadelphia, PA, USA, July 1997. ACM.

[Mil98] E. Miller. An introduction to the Resource Description Framework. *D-Lib Magazine*, May 1998. Available at `http://www.dlib.org/dlib/may98/miller/05miller.html`.

[MS99] T. Milo and D. Suciu. Type inference for queries on semistructured data. In *Proceedings of the 18th ACM Symposium on Principles of Database Systems (PODS'99)*, pages 215–226, Philadelphia, Pennsylvania, USA, May/June 1999. ACM Press.

[Mur97a] M. Murata. DTD transformation by patterns and contextual conditions. In *SGML/XML '97 Conference Proceedings*, pages 325–332, Washington, D.C., 1997. GCA.

[Mur97b] M. Murata. Transformation of documents and schemas by patterns and contextual conditions. In *Proceedings of the 3rd International Workshop on Principles of Document Processing (PODP'96)*, number 1293 in Lecture Notes in Computer Science, pages 153–169, Palo Alto, California, USA, September 1997. Springer.

[Mur98] M. Murata. Data model for document transformation and assembly. In *Proceedings of International Workshop on Principles of Digital Document Processing, PODDP'98*, number 1481 in Lecture Notes in Computer Science, pages 140–152, Saint Malo, France, March 1998. Springer.

[MW99] J. McHugh and J. Widom. Query optimization for XML. In *Proceedings of the 25th International Conference on Very Large Databases (VLDB'99)*, pages 315–326, Edinburgh, Scotland, UK, September 1999. Morgan Kaufmann.

[NAM97] S. Nestorov, S. Abiteboul, and R. Motwani. Inferring structure in semistructured data. *SIGMOD Record*, 26(4):39–43, December 1997.

[NAM98] S. Nestorov, S. Abiteboul, and R. Motwani. Extracting schema from semistructured data. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'98)*, pages 295–306, Seattle, Washington, USA, June 1998. ACM Press.

[NBY97] G. Navarro and R. Baeza-Yates. Proximal Nodes: A model to query document databases by content and structure. *ACM Transactions on Information Systems*, 15(4):400–435, October 1997.

[NSF98] NSF-EU Working Groups, P. Schäuble and A. F. Smeaton, editors. *An International Research Agenda for Digital Libraries: Summary Report of the Series of Joint NSF-EU Working Groups on Future Directions for Digital Libraries Research*, October 1998. Available at `http://www.iei.pi.cnr.it/DELOS/REPORTS/Brussrep.htm`.

[OAS91] OASIS. *The DocBook DTD*, 1991. The DocBook SGML DTD is available at `http://www.oasis-open.org/docbook/`, an unofficial XML version of the DocBook DTD is available at `http://nwalsh.com/docbook/xml/index.html`.

[OAS95] OASIS. *CALS Table Model Document Type Definition: OASIS Technical Memorandum, TM 9502:1995*, 1995. Available at `http://www.oasis-open.org/html/a502.htm`.

[Obe99] J. Ober. The California Digital Library. *D-Lib Magazine*, March 1999. Available at `http://www.dlib.org/dlib/march99/03ober.html`.

[PAGM96] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina. Object fusion in mediator systems. In *Proceedings of the 22th International Conference on Very Large Databases*

(VLDB'96), pages 413–424, Mumbai, Bombay, India, September 1996. Morgan Kaufmann.

[PBC+99] A. Paepcke, M. Baldonado, C.-C. K. Chang, S. Cousins, and H. Garcia-Molina. Using distributed objects to build the stanford digital library infobus. *IEEE Computer*, 32(2):80–87, February 1999.

[PCGMW98] A. Paepcke, C.-C. K. Chang, H. Garcia-Molina, and T. Winograd. Interoperability for digital libraries worldwide. *Communications of the ACM (CACM)*, 41(4):33–43, April 1998.

[PGMW95] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the 11th International Conference on Data Engineering (ICDE'95)*, pages 251–260, Taipei, Taiwan, March 1995. IEEE Computer Society.

[PK97] C. K. Powell and N. Kerr. SGML creation and delivery: The Humanities Text Initiative. *D-Lib Magazine*, July/August 1997. Available at `http://www.dlib.org/dlib/july97/humanities/07powell.html`.

[Pri97] Primis. Primis Custom Publishing, a Division of McGraw-Hill, 1997. Available at `http://www.mhhe.com/primis/`.

[PSHD96] P. Pirolli, P. Schank, M. Hearst, and C. Diehl. Scatter/Gather browsing communicates the topic structure of a very large text collection. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'96)*, pages 213–220, Vancouver, British Columbia, Canada, April 1996. ACM.

[PV99a] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'99)*, pages 455–466, Philadephia, Pennsylvania, USA, June 1999. ACM Press.

[PV99b] Y. Papakonstantinou and P. Velikhov. Enhancing semistructured data mediators with Document Type Definitions. In *Proceedings of the 15th International Conference on Data*

Engineering (ICDE'99), pages 136–145, Sydney, Austrialia, March 1999. IEEE Computer Society Press.

[PVH98] F. Paradis, A.-M. Vercoustre, and B. Hills. A virtual document interpreter for reuse of information. In *Electronic Publishing, Artistic Imaging, and Digital Typography, Proceedings of the 7th International Conference on Electronic Publishing (EP'98)*, number 1375 in Lecture Notes in Computer Science, pages 487–498, Saint-Malo, France, April 1998. Springer.

[PW97] J. Price-Wilkin. Just-in-time conversion, just-in-case collections: Effectively leveraging rich document formats for the WWW. *D-Lib Magazine*, May 1997. Available at `http://www.dlib.org/dlib/may97/michigan/05pricewilkin.html`.

[PWDN99] M. Papiani, J. Wason, A. Dunlop, and D. Nicole. A distributed scientific data archive using the Web, XML and SQL/MED. *SIGMOD Record*, 28(3):56–62, September 1999.

[Ran99] N. Randall. XSL: How stylish can you get? *PC Magazine*, July 6, 1999. Available at `http://www.zdnet.com/pcmag/pctech/content/18/07/ib1807.001.html`.

[Rei86] J. Reichardt. Deterministic grammars and grammar morphisms. *Acta Informatica*, 23(5):567–583, 1986.

[Rei99] L. Rein. The quest for an XML query standard. *XML.com*, March 2, 1999. Available at `http://www.xml.com/xml/pub/1999/03/quest/index.html`.

[Ren97] A. Renear. The digital library research agenda: What's missing — and how Humanities Textbase Projects can help. *D-Lib Magazine*, July/August 1997. Available at `http://www.dlib.org/dlib/july97/07renear.html`.

[Reu97] Reuters-21578, Distribution 1.0. Reuters Ltd., 1997. Available at `http://www.research.att.com/~lewis`.

[RvOHB98] L. Rutledge, J. van Ossenbruggen, L. Hardman, and D. C. Bulterman. Practical application of existing hypermedia standards and tools. In *Proceedings of the 3rd ACM International Conference on Digital Libraries (DL'98)*, pages 191–199, Pittsburgh, PA, USA, June 1998. ACM Press.

[SB97] E. J. Shaw and S. Blumson. Making of America: Online searching and page presentation at the University of Michigan. *D-Lib Magazine*, July/August 1997. Available at http://www.dlib.org/dlib/july97/america/07shaw.html.

[SHO+96] D. Stribling, T. Hunter, L. Olszewski, A. Corrigan, R. Mullis, and L. Allen. A real world conversion to SGML. In *SIGDOC'96 — Proceedings of the 14th ACM International Conference*, pages 75–86, Research Triangle Park, North Carolina, USA, October 1996. ACM.

[SIT99] The Slicing Books Technology. Slicing Information Technology GmbH, 1999. Available at http://www.slicing-infotech.com/index.html.

[SMB95] C. M. Sperberg-McQueen and L. Bournard. The Design of the TEI Encoding Scheme. In N. Ide and J. Véronis, editors, *Text Encoding Initiative, Background and Context*, pages 17–39. Kluwer Academic Publishers, Dordrecht / Boston / London, 1995.

[Sos99] M. Sosteric. ICAAP eXtended Markup Language: Exploiting XML and adding value to the journals production process. *D-Lib Magazine*, February 1999. Available at http://www.dlib.org/dlib/february99/02sosteric.html.

[ST99] A. Salminen and F. W. Tompa. Grammars++ for modelling information in text. *Information Systems*, 24(1):1–24, 1999.

[Suc97] D. Suciu. Management of semistructured data. *SIGMOD Record*, 26(4):4–7, December 1997.

[TBSN99] R. Thibadeau, J. Balderas, A. Snyder, and J. Nestor. E-commerce catalog construction: An experiment with programmable XML for dynamic documents. *D-Lib Magazine*, February 1999. Available at http://www.dlib.org/dlib/february99/thibadeau/02thibadeau.html.

[TEI94] Text Encoding Initiative (TEI), C.M. Sperberg-McQueen and L. Burnard, editors. *Guidelines for Electronic Text Encoding and Interchange*, May 1994. The TEI Guidelines, SGML and XML versions of TEI P3 DTDs and the TEI Lite DTD can be found at http://www.tei-c.org/.

[VP97] V. Vassalos and Y. Papakonstantinou. Describing and using query capabilities of heterogeneous sources. In *Proceedings of the 23th International Conference on Very Large Databases (VLDB'97)*, pages 256–265, Athens, Greece, August 1997. Morgan Kaufmann.

[VP98] A.-M. Vercoustre and F. Paradis. Reuse of linked documents through virtual document prescriptions. In *Electronic Publishing, Artistic Imaging, and Digital Typography, Proceedings of the 7th International Conference on Electronic Publishing (EP'98)*, number 1375 in Lecture Notes in Computer Science, pages 499–512, Saint-Malo, France, April 1998. Springer.

[W3C97] W3C. *Comparison of SGML and XML*, December 15, 1997. Available at `http://www.w3.org/TR/NOTE-sgml-xml-971215`.

[W3C98a] W3C. *Cascading Style Sheets, level 2 (CSS2 Specification), W3C Recommendation*, May 12, 1998. Available at `http://www.w3.org/TR/REC-CSS2`.

[W3C98b] W3C. *Document Object Model (DOM) Level 1 Specification Version 1.0, W3C Recommendation*, October 1, 1998. Available at `http://www.w3.org/TR/REC-DOM-Level-1`.

[W3C98c] W3C. *Extensible Markup Language (XML) 1.0, W3C Recommendation*, February 10, 1998. Available at `http://www.w3.org/TR/REC-xml`.

[W3C98d] W3C. *Using XSL and CSS together, W3C Note*, September 11, 1998. Available at `http://www.w3.org/TR/NOTE-XSL-and-CSS`.

[W3C99a] W3C. *Cascading Style Sheets, level 1 (CSS1 Specification), W3C Recommendation (revised from December 1996)*, January 11, 1999. Available at `http://www.w3.org/TR/REC-CSS1`.

[W3C99b] W3C. *HTML 4.01 Specification, W3C Recommendation*, December 24, 1999. Available at `http://www.w3.org/TR/html401`. Previous versions are HTML 3.2 and HTML 2.0.

[W3C99c] W3C. *Mathematical Markup Language (MathML) 1.01 Specification, W3C Recommendation (revised from April 1998)*, July 7, 1999. Available at `http://www.w3.org/TR/REC-MathML`.

[W3C99d] W3C. *Namespaces in XML, W3C Recommendation*, January 14, 1999. Available at `http://www.w3.org/TR/REC-xml-names`.

[W3C99e] W3C. *Resource Description Framework (RDF) Model and Syntax Specification, W3C Recommendation*, February 22, 1999. Available at `http://www.w3.org/TR/REC-rdf-syntax`.

[W3C99f] W3C. *Resource Description Framework (RDF) Schema Specification, W3C Proposed Recommendation (work in progress)*, March 3, 1999. Available at `http://www.w3.org/TR/PR-rdf-schema`.

[W3C99g] W3C. *XML Fragment Interchange, W3C Working Draft (work in progress)*, June 30, 1999. Available at `http://www.w3.org/TR/WD-xml-fragment`.

[W3C99h] W3C. *XML Path Language (XPath) Version 1.0, W3C Recommendation*, November 16, 1999. Available at `http://www.w3.org/TR/xpath`.

[W3C99i] W3C. *XML Pointer Language (XPointer), W3C Working Draft (work in progress)*, December 6, 1999. Available at `http://www.w3.org/TR/xptr`.

[W3C99j] W3C. *XSL Transformations (XSLT) Version 1.0, W3C Recommendation 16 1999*, November 16, 1999. Available at `http://www.w3.org/TR/xslt`.

[W3C00a] W3C. *Building XHTML Modules, W3C Working Draft (work in progress)*, January 5, 2000. Available at `http://www.w3.org/TR/xhtml-building`.

[W3C00b] W3C. *Document Object Model (DOM) Level 2 Specification Version 1.0, W3C Candidate Recommendation (work in progress)*, March 7, 2000. Available at `http://www.w3.org/TR/DOM-Level-2`.

[W3C00c] W3C. *Extensible Stylesheet Language (XSL) Specification, W3C Working Draft (work in progress)*, March 1, 2000. Available at `http://www.w3.org/TR/xsl`.

[W3C00d] W3C. *Modularization of XHTML, W3C Working Draft (work in progress)*, January 5, 2000. Available at `http://www.w3.org/TR/xhtml-modularization`.

[W3C00e] W3C. *Synchronized Multimedia Integration Language (SMIL): Boston Specification, W3C Working Draft (work in progress)*, February 25, 2000. Available at `http://www.w3.org/TR/smil-boston`.

[W3C00f] W3C. *XHTML 1.0: The Extensible HyperText Markup Language: A Reformulation of HTML 4 in XML 1.0, W3C Recommendation*, January 26, 2000. Available at `http://www.w3.org/TR/xhtml1`.

[W3C00g] W3C. *XML Linking Language (XLink), W3C Working Draft (work in progress)*, February 21, 2000. Available at `http://www.w3.org/TR/xlink`.

[W3C00h] W3C. *XML Schema Part 1: Structures, W3C Working Draft (work in progress)*, February 25, 2000. Available at `http://www.w3.org/TR/xmlschema-1`.

[W3C00i] W3C. *XML Schema Part 2: Datatypes, W3C Working Draft (work in progress)*, February 25, 2000. Available at `http://www.w3.org/TR/xmlschema-2`.

[Wal98] N. Walsh. A technical introduction to XML. *XML.com*, October 3, 1998. Available at `http://www.xml.com/xml/pub/98/10/guide0.html`.

[Wal99a] N. Walsh. The extensible style language: Styling XML documents. *XML.com*, January 19, 1999. Available at `http://xml.com/xml/pub/1999/01/walsh1.html`.

[Wal99b] N. Walsh. The XSL debate: One expert's view. *XML.com*, June 8, 1999. Available at `http://www.xml.com/xml/pub/1999/06/xsl_edit.html`.

[Wei99] S. Weibel. The state of the Dublin Core Metadata Initiative April 1999. *D-Lib Magazine*, April 1999. Available at `http://www.dlib.org/dlib/april99/04weibel.html`.

[WH98] S. Weibel and J. Hakala. DC-5: The Helsinki Metadata Workshop: A report on the workshop and subsequent developments. *D-Lib Magazine*, February 1998. Available at `http://www.dlib.org/dlib/february98/02weibel.html`.

[Wid99] J. Widom. Data mangement for XML: Research directions. *IEEE Data Engineering Bulletin*, 22(3):44–52, September 1999.

[WL97] S. Weibel and C. Lagoze. An element set to support resource discovery: The state of the Dublin Core January 1997. *International Journal on Digital Libraries*, 1(2):176–186, September 1997.

[WL98] K. Wang and H. Liu. Discovering typical structures of documents: A road map approach. In *Proceedings of the 21st ACM International Conference on Research and Development in Information Retrieval (SIGIR'98)*, pages 146–154, Melbourne, Austrailia, August 1998. ACM.

[Woo87] D. Wood. *Theory of Computation*. Harper & Row Publishers, New York, 1987.

[WS91] L. Wall and R. Schwartz. *Programming perl*. O'Reilly & Associates, Inc., 1991.

Appendix A

An example of a virtual document

In this appendix we present excerpts of two documents: the former one is structured by the Book DTD of ISO 12083 DTD [ISO94], and the latter one is a virtual document — generated from the former — structured by the generic DTD (see Sections 3.2 and 5.2.3).

```
<BOOK>
  <FRONT>
    <TITLEGRP><TITLE>Alice Freeman Palmer</TITLE>
      <SUBTITLE>The Evolution of a New Woman</SUBTITLE></TITLEGRP>
    <AUTHGRP><AUTHOR><FNAME>Ruth </FNAME>
      <SURNAME>Bordin</SURNAME></AUTHOR></AUTHGRP>
    ...
    <PREFACE><TITLE>Preface</TITLE>
      <P>The research for this biography of Alice Freeman Palmer began...</P>
      <P>Alice Freeman Palmer's career was distinguished and...</P>...
      <P>... partner in resolving my own dilemma.</P></PREFACE>
  </FRONT>
  <BODY>
    <CHAPTER ID="CH01"><NO>1</NO><TITLE>The New Woman</TITLE>
      <P>At the turn of the century, Alice Freeman Palmer, ...</P>
      <P>Her role in higher education expanded, rather than...</P>...
      <P>The phrase <EMPH TYPE="2">New Woman</EMPH> was coined originally...</P>...
    </CHAPTER>
    <CHAPTER ID="CH02"><NO>2</NO><TITLE>Genesis</TITLE>...</CHAPTER>
    <CHAPTER ID="CH03"><NO>3</NO><TITLE>Commitment</TITLE>...</CHAPTER>
    <CHAPTER ID="CH04"><NO>4</NO><TITLE>Crucible</TITLE>
      <P>When Alice Freeman graduated from the University of Michigan...</P>...
      <SECTION><NO>1</NO>
        <P>Alice began the crucible years happily enough. She...</P>...
        <P>Alice Freeman needed cash desperately. She was in debt. ...
          She described her experience at Lake Geneva in a letter to her friend
          Lucy Andrews.
          <BQ><P>The days have been very trying to me for...</P></BQ>
          She went on to describe her work:
          <BQ><P>... it isn't so very hard or wearing. ...</P></BQ>
          ...
        </P>...
        <P>... republican institutions in the eyes of Europe.</P>
      </SECTION>
      <SECTION><NO>2</NO>
        <P>In June Alice Freeman was finally free of Lake Geneva Seminary...</P>
        ...
      </SECTION>
    </CHAPTER>
    <CHAPTER ID="CH05"><NO>5</NO><TITLE>Vocation</TITLE>
      <P>In June of 1879, Alice Freeman accepted the professorship of history
        at Wellesley College. ...</P>
      <SECTION><NO>1</NO>
        <P>We do not know precisely why Henry Durant first attempted...</P>...
      </SECTION>
      <SECTION><NO>2</NO>...</SECTION>
      ...
      <SECTION><NO>5</NO>...</SECTION>
    </CHAPTER>
    <CHAPTER ID="CH06"><NO>6</NO><TITLE>Fulfillment</TITLE>...</CHAPTER>
    ...
    <CHAPTER ID="CH10"><NO>10</NO><TITLE>Resolution</TITLE>...</CHAPTER>
  </BODY>
  <BACK>
    ...
    <NOTES><TITLE>Bibliography</TITLE>
      ...
      <LIST><HEAD>Published Materials</HEAD>
        ...
        <ITEM><P>Donnan, Elizabeth. "Henry Fowle Durant."
          <EMPH TYPE="2">Dictionary of American Biography</EMPH>.
          3:541&ndash;42.</P></ITEM>...
      </LIST>...
    </NOTES>
  </BACK>
</BOOK>
```

```
<VIRTDOC>
  <SECTION>
    <TITLE><TITLE>Alice Freeman Palmer</TITLE>
      <STRING>The Evolution of a New Woman</STRING></TITLE>
    <STRGRP><STRCONT><STRING>Ruth </STRING>
      <STRING>Bordin</STRING></STRCONT></STRGRP>
    ...
    <SECTION><TITLE>Preface</TITLE>
      <PARA>The research for this biography of Alice Freeman Palmer began...</PARA>
      <PARA>Alice Freeman Palmer's career was distinguished and certainly...</PARA>...
      <PARA>... partner in resolving my own dilemma.</PARA></SECTION>
  </SECTION>
  <SECTGRP>
    <SECTION><TITLE>1</TITLE><TITLE>The New Woman</TITLE>
      <PARA>At the turn of the century, Alice Freeman Palmer, ...</PARA>
      <PARA>Her role in higher education expanded, rather than...</PARA>...
      <PARA>The phrase <STRING>New Woman</STRING> was coined originally...</PARA>...
    </SECTION>
    <SECTION><TITLE>2</TITLE><TITLE>Genesis</TITLE>...</SECTION>
    <SECTION><TITLE>3</TITLE><TITLE>Commitment</TITLE>...</SECTION>
    <SECTION><TITLE>4</TITLE><TITLE>Crucible</TITLE>
      <PARA>When Alice Freeman graduated from the University of Michigan...</PARA>...
      <SECTION><TITLE>1</TITLE>
        <PARA>Alice began the crucible years happily enough. She...</PARA>...
        <PARA>Alice Freeman needed cash desperately. She was in debt. ...
          She described her experience at Lake Geneva in a letter to her friend
          Lucy Andrews.
          <PARACONT><PARA>The days have been very trying to me for...</PARA></PARACONT>
          She went on to describe her work:
          <PARACONT><PARA>... it isn't so very hard or wearing. ...</PARA></PARACONT>
          ...
        </PARA>...
        <PARA>... republican institutions in the eyes of Europe.</PARA>
      </SECTION>
      <SECTION><TITLE>2</TITLE>
        <PARA>In June Alice Freeman was finally free of Lake Geneva Seminary...</PARA>
        ...
      </SECTION>
    </SECTION>
    <SECTION><TITLE>5</TITLE><TITLE>Vocation</TITLE>
      <PARA>In June of 1879, Alice Freeman accepted the professorship of history
        at Wellesley College. ...</PARA>
      <SECTION><TITLE>1</TITLE>
        <PARA>We do not know precisely why Henry Durant first attempted...</PARA>...
      </SECTION>
      <SECTION><TITLE>2</TITLE>...</SECTION>
      ...
      <SECTION><TITLE>5</TITLE>...</SECTION>
    </SECTION>
    <SECTION><TITLE>6</TITLE><TITLE>Fulfillment</TITLE>...</SECTION>
    ...
    <SECTION><TITLE>10</TITLE><TITLE>Resolution</TITLE>...</SECTION>
  </SECTGRP>
  <SECTGRP>
    ...
    <SECTION><TITLE>Bibliography</TITLE>
      ...
      <PARAGRP><TITLE>Published Materials</TITLE>
        ...
        <PARACONT><PARA>Donnan, Elizabeth. "Henry Fowle Durant."
          <STRING>Dictionary of American Biography</STRING>.
          3:541&ndash;42.</PARA></PARACONT>...
      </PARAGRP>...
    </SECTION>
  </SECTGRP>
</VIRTDOC>
```

Appendix B

A document conforming to the cookbook DTD

```
<?xml version="1.0"?>
<!DOCTYPE cookbook SYSTEM "cookbook.dtd">
<cookbook><title>Greek cooking</title>
   <editor>
      <fname>Barbara</fname><surname>Heikkinen</surname>
   </editor>
   <recipes course="starter">
      <author>
         <surname>Vangelis</surname>
         <url>http://www.geocities.com/Athens/Ithaca/8680/menu.htm</url>
      </author>
      <recipe><title>Greek Salad</title>
         <ingreds>
            <item>
               <measure>1/2 head</measure><ingred>romaine lettuce</ingred>
               <descr>chopped</descr>
            </item>
            <item>
               <measure>4</measure><ingred>tomatoes</ingred>
               <descr>cut into wedges</descr>
            </item>
            <item>
               <measure>1/2</measure><ingred>cucumber</ingred>
               <descr>halved lengthwise, sliced</descr>
            </item>
            <item>
               <measure>1/2</measure>
               <ingred>mild white or yellow onion</ingred>
               <descr>thinly sliced, separated into rings</descr>
            </item>
            <item>
               <measure>1</measure><ingred>green bell pepper</ingred>
               <descr>seeded, sliced</descr>
            </item>
            <item>
               <measure>6 oz</measure><ingred>feta cheese</ingred>
               <descr>cubed</descr>
            </item>
            <item>
               <measure>about 12</measure>
               <ingred>kalamatas black olives</ingred>
            </item>
```

```
          <item>
             <measure>1 teaspoon</measure><ingred>fresh oregano</ingred>
             <descr>chopped</descr>
          </item>
       </ingreds>
       <ingreds><title>Lemon Dressing</title>
          <item>
             <measure>1/4 cup</measure>
             <ingred>olive oil, extra virgin</ingred>
          </item>
          <item>
             <measure>1 tablespoon</measure><ingred>lemon juice</ingred>
          </item>
          <item>
             <ingred>salt</ingred><descr>to taste</descr>
          </item>
          <item>
             <ingred>black pepper</ingred><descr>to taste</descr>
          </item>
       </ingreds>
       <instruct>
          <p>Mix lettuce and cucamber; place in a large bowl or individual
          bowls. Add tomatoes, onion and bell pepper. Arrange cheese and
          olives on top and sprinkle with oregano.</p>
          <p>To prepare Lemon Dressing, stir together all dressing
          ingredients. Spoon dressing over salad and serve. <emph>Makes
          4 servings</emph>.</p>
       </instruct>
    </recipe>
    <recipe><title>Potato Salad</title>
       <ingreds>
          <item>
             <measure>4-6 small</measure><ingred>potatoes</ingred>
          </item>
          ...
       </ingreds>
       <instruct>
          <p>Boil the potatoes in salted water <emph>in their jackets</emph>
          until tender. Remove skins and cut into cubes, about 20 mm. Add
          onions and parsley. Season with salt and pepper. Add oil
          and lemon juice, toss and serve.</p>
       </instruct>
    </recipe>
    ...
 </recipes>
 <recipes course="maincour">
    <recipe><title>Mousaka</title>
       <ingreds>
          <item>
             <measure>6 medium</measure>
             <ingred>aubergines (egg plants)</ingred>
             <descr>sliced 5mm thick</descr>
          </item>
          ...
       </ingreds>
       <instruct>
          <p>Sprinkle aubergine slices with salt and leave for...</p>
          ...
       </instruct>
    </recipe>
    ...
 </recipes>
 ...
</cookbook>
```

Appendix C

Tables containing results of element-type classification

C.1 The Book document type of ISO 12083

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
book	*VirtDoc*	0	679513	front body back
body	*SectGrp*	0	530034	chapter
back	*SectGrp*	0	138773	notes
front	*Section*	0	10706	titlegrp authgrp pubfront ded preface
preface	*Section*	0	9783	title p
chapter	*Section*	0	53003	no title p figgrp section
section	*Section*	0	10290	no p bq figgrp
notes	*Section*	0	11564	title note list
list	*ParaGrp*	0	1893	head item
pubfront	*ParaCont*	0	801	pubname cpyrt catalog figgrp
ded	*ParaCont*	0	63	p
bq	*ParaCont*	0	440	p figgrp
note	*ParaCont*	0	114	no p
item	*ParaCont*	0	111	p list
catalog	*Para*	562	562	text
p	*Para*	337	355	text noteref emph bq poem
authgrp	*StrGrp*	0	11	author
cpyrt	*StrGrp*	0	26	date cpyrtnme
poem	*StrGrp*	0	326	poemline stanza

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
author	*StrCont*	0	11	fname surname
pubname	*StrCont*	0	131	orgname city country
cpyrtnme	*StrCont*	0	22	orgname
stanza	*StrCont*	0	330	poemline
subtitle	*String*	28	28	text
fname	*String*	5	5	text
surname	*String*	6	6	text
orgname	*String*	50	50	text
city	*String*	9	9	text
country	*String*	44	44	text
noteref	*String*	2	2	text
emph	*String*	28	28	text
titlegrp	*Title*	0	48	title subtitle
title	*Title*	58	58	text
date	*Title*	4	4	text
no	*Title*	2	2	text
poemline	*Title*	27	27	text
head	*Title*	17	17	text
figgrp	*FigCont*	0	104	title fig
fig	*Fig*	0	0	λ

Table C.1: The Book DTD of ISO 12083 DTD [ISO94].

C.2 The DocBook document type

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
book	*VirtDoc*	0	323397	bookinfo chapter glossary title toc preface appendix index
chapter	*Section*	0	24852	title para sect1 variablelist indexterm
sect1	*Section*	0	3902	title para sect2 bridgehead indexterm informaltable literallayout variablelist cmdsynopsis screen
sect2	*Section*	0	2004	title graphic para sect3 table orderedlist indexterm note
glossary	*Section*	0	4905	title glossentry
preface	*Section*	0	6852	title para sect1
appendix	*Section*	0	15385	title para sect1
bookinfo	*ParaGrp*	0	1600	bookbiblio legalnotice
authorgroup	*ParaGrp*	0	19	corpauthor othercredit author
othercredit	*ParaGrp*	0	19	authorblurb
itemizedlist	*ParaGrp*	0	478	listitem
variablelist	*ParaGrp*	0	1197	varlistentry
varlistentry	*ParaGrp*	0	233	term listitem
glossentry	*ParaGrp*	0	212	glossterm glossdef
orderedlist	*ParaGrp*	0	897	listitem
calloutlist	*ParaGrp*	0	650	callout
bookbiblio	*ParaCont*	0	516	title subtitle authorgroup pubdate abstract invpartnumber productname publisher copyright
authorblurb	*ParaCont*	0	19	para
abstract	*ParaCont*	0	344	para
listitem	*ParaCont*	0	138	para literallayout indexterm orderedlist screen
glossdef	*ParaCont*	0	205	para
legalnotice	*ParaCont*	0	361	para
note	*ParaCont*	0	231	title para
sect3	*ParaCont*	0	1200	title para sect4
callout	*ParaCont*	0	162	para
caution	*ParaCont*	0	253	title para
footnote	*ParaCont*	0	197	para
sect4	*ParaCont*	0	1693	title para
subtitle	*Para*	90	90	text
programlisting	*Para*	113	114	text literal co
literallayout	*Para*	217	241	text lineannotation
screen	*Para*	75	122	text computeroutput userinput co
comment	*Para*	144	144	text

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
para	*Para*	161	272	text quote firstterm trademark foreignphrase sgmltag programlisting itemizedlist literallayout citation citetitle xref literal note informaltable variablelist citerefentry keycap indexterm screen orderedlist table cmdsynopsis replaceable calloutlist synopsis caution figure emphasis option footnote comment
informaltable	*StrGrp*	0	380	tgroup
cmdsynopsis	*StrGrp*	0	29	command arg group
group	*StrGrp*	0	9	arg replaceable group option
author	*StrCont*	0	12	firstname surname
publisher	*StrCont*	0	50	publishername address
copyright	*StrCont*	0	38	year holder
citerefentry	*StrCont*	0	7	refentrytitle manvolnum
indexterm	*StrCont*	0	24	primary secondary tertiary see seealso
pubdate	*String*	11	11	text
quote	*String*	17	17	text
firstterm	*String*	9	9	text
trademark	*String*	11	11	text
foreignphrase	*String*	4	4	text
sgmltag	*String*	3	3	text
bridgehead	*String*	23	23	text
citation	*String*	32	32	text
productname	*String*	23	23	text
firstname	*String*	6	6	text
surname	*String*	6	6	text
invpartnumber	*String*	11	11	text
publishername	*String*	29	29	text
address	*String*	1	21	text city state
city	*String*	7	7	text
state	*String*	13	13	text
year	*String*	9	9	text
holder	*String*	29	29	text
citetitle	*String*	30	30	text
literal	*String*	8	8	text replaceable
entry	*String*	22	27	text literal replaceable footnote footnoteref para emphasis variablelist itemizedlist xref citetitle

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
replaceable	*String*	6	6	text
refentrytitle	*String*	6	6	text
manvolnum	*String*	1	1	text
keycap	*String*	5	6	text replaceable
primary	*String*	14	14	text
secondary	*String*	15	15	text
tertiary	*String*	10	10	text
see	*String*	7	7	text
computeroutput	*String*	17	17	text
userinput	*String*	21	22	text replaceable keycap
synopsis	*String*	5	16	text replaceable literal
option	*String*	1	1	text replaceable
seealso	*String*	15	15	text
emphasis	*String*	10	10	text
lineannotation	*String*	7	7	text
title	*Title*	24	25	text productname citetitle footnote
corpauthor	*Title*	15	15	text
term	*Title*	2	12	text literal replaceable citerefentry keycap option computeroutput userinput
glossterm	*Title*	7	7	text
command	*Title*	10	10	text replaceable
arg	*Title*	1	8	text replaceable arg group option
figure	*FigCont*	0	31	title graphic
graphic	*Fig*	0	0	λ
xref	*Ref*	0	0	λ
co	*Ref*	0	0	λ
footnoteref	*Ref*	0	0	λ
toc	*Empty*	0	0	λ
index	*Empty*	0	0	λ
tgroup	*tgroup*	0	880	colspec tbody thead spanspec
colspec	*colspec*	0	0	λ
tbody	*tbody*	0	856	row
row	*row*	0	68	entry
thead	*thead*	0	27	row
table	*table*	0	1311	title tgroup
spanspec	*spanspec*	0	0	λ

Table C.2: The DocBook DTD [OAS91].

C.3 The TEI and TEI Lite document types

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
tei.2	*VirtDoc*	0	185084	teiheader text
text	*SectGrp*	0	91361	front body back pb group
back	*SectGrp*	0	21822	div1 divgen div pb
group	*SectGrp*	0	101039	text pb head
front	*Section*	0	6071	titlepage div1 divgen div pb
div1	*Section*	0	4241	p head div2 classdoc entdoc tagdoc listbibl list lg pb l opener lb epigraph stage closer milestone sp note q signed argument trailer salute bibl byline docauthor
div2	*Section*	0	3470	head p div3 list eg lg l pb opener stage sp epigraph milestone note trailer q closer bibl signed salute
body	*Section*	0	85721	div0 div1 p pb head lg epigraph closer salute signed trailer opener div note bibl milestone l list
div0	*Section*	1	7053	text head div1 pb p q note byline lg opener epigraph closer l milestone stage sp trailer list lb signed salute argument bibl cit ref emph
div3	*Section*	0	2838	head p div4 list lg pb milestone q closer
div4	*Section*	0	2837	head p
div	*Section*	0	3194	p pb head list l lg signed opener salute closer epigraph note trailer q lb bibl cit milestone div listbibl
listbibl	*StrSect*	0	8806	bibl head
teiheader	*ParaGrp*	0	1782	filedesc profiledesc revisiondesc encodingdesc
filedesc	*ParaGrp*	0	1416	titlestmt editionstmt extent publicationstmt sourcedesc seriesstmt
attlist	*ParaGrp*	0	314	attdef
biblfull	*ParaGrp*	0	110	titlestmt publicationstmt notesstmt editionstmt seriesstmt extent
encodingdesc	*ParaGrp*	0	281	editorialdecl tagsdecl
lg	*ParaGrp*	0	263	l head pb lg milestone closer trailer signed epigraph

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
publicationstmt	*ParaCont*	0	499	publisher pubplace availability distributor date p idno
availability	*ParaCont*	0	881	p
sourcedesc	*ParaCont*	0	135	p bibl biblfull
titlepage	*ParaCont*	0	237	titlepart doctitle byline docdate docauthor docimprint docedition epigraph pb lb
classdoc	*ParaCont*	0	1052	class rs desc attlist remarks part classes files parents members decl ptr
remarks	*ParaCont*	0	123	p
attdef	*ParaCont*	0	296	attname rs desc datatype valdesc default eg remarks vallist
vallist	*ParaCont*	0	232	val desc
entdoc	*ParaCont*	0	287	entname rs desc files refby string ptr remarks
tagdoc	*ParaCont*	0	1397	gi rs desc attlist exemplum remarks part classes files datadesc parents children elemdecl attldecl ptr
exemplum	*ParaCont*	0	173	eg p
notesstmt	*ParaCont*	0	29	note
editorialdecl	*ParaCont*	0	281	p
epigraph	*ParaCont*	0	163	q l bibl p lg cit
sp	*ParaCont*	0	187	speaker p stage pb l lg
argument	*ParaCont*	0	676	head p l lg
p	*Para*	346	384	text q bibl list ptr gi att term socalled note tag eg emph title mentioned foreign code gloss ident index hi figure kw gap pb stage lb anchor ref name cit unclear date milestone sic corr address label abbr table add del orig
distributor	*Para*	68	68	text
note	*Para*	112	310	text q gloss title ptr ident term code att gi bibl socalled mentioned list p tag xref kw cit foreign anchor hi note abbr ref pb l emph corr milestone
eg	*Para*	331	331	text

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
desc	*Para*	77	78	text term att socalled q mentioned gi ptr code tag
parents	*Para*	205	205	text
decl	*Para*	163	163	text
valdesc	*Para*	70	74	text gi term ptr att code q title emph tag
children	*Para*	173	173	text
elemdecl	*Para*	85	85	text
attldecl	*Para*	148	148	text
docimprint	*Para*	71	124	text pubplace publisher docdate pb lb date hi address name ref
titlestmt	*StrGrp*	0	158	title sponsor editor funder author respstmt
profiledesc	*StrGrp*	0	19	langusage
revisiondesc	*StrGrp*	0	159	list change
change	*StrGrp*	0	81	date respstmt item
seriesstmt	*StrGrp*	0	90	title respstmt
table	*StrGrp*	0	182	head row pb
editionstmt	*StrCont*	0	14	edition
langusage	*StrCont*	0	19	language
list	*StrCont*	0	597	item label head pb milestone
doctitle	*StrCont*	0	33	titlepart lb
series	*StrCont*	0	32	title biblscope
respstmt	*StrCont*	0	49	resp name
xref	*StrCont*	0	47	code
address	*StrCont*	0	83	addrline
s	*StrCont*	0	344	hi
tagsdecl	*StrCont*	0	0	tagusage
cit	*StrCont*	0	392	q bibl
row	*StrCont*	0	16	cell
sponsor	*String*	51	51	text
editor	*String*	19	22	text name
funder	*String*	52	52	text
edition	*String*	21	22	text date
extent	*String*	23	23	text
pubplace	*String*	12	12	text hi lb address
language	*String*	8	8	text mentioned ident

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
byline	*String*	18	29	text name docauthor lb title
name	*String*	25	25	text abbr address
docdate	*String*	8	8	text lb
label	*String*	2	7	text gi att ident term code socalled kw ptr hi abbr
socalled	*String*	9	9	text pb
att	*String*	4	4	text
term	*String*	11	12	text kw ref pb milestone
gloss	*String*	29	30	text hi
tag	*String*	11	11	text
emph	*String*	10	10	text gi ref pb foreign
mentioned	*String*	6	6	text
foreign	*String*	14	14	text hi pb
code	*String*	6	6	text
ident	*String*	7	7	text
author	*String*	25	25	text hi
hi	*String*	17	17	text hi title lb ptr anchor q name
biblscope	*String*	19	19	text
rs	*String*	17	17	text
part	*String*	37	37	text
classes	*String*	15	15	text
members	*String*	45	48	text ident
datatype	*String*	7	7	text
default	*String*	7	7	text
refby	*String*	4	4	text
string	*String*	20	20	text
datadesc	*String*	43	44	text gi term q att
docauthor	*String*	19	19	text hi lb
resp	*String*	21	21	text
kw	*String*	5	5	text
idno	*String*	10	10	text
ref	*String*	4	4	text hi note corr ref q term abbr foreign pb
docedition	*String*	27	27	text lb abbr name
opener	*String*	47	56	text hi lb title pb date ptr ref address gloss dateline salute name num
stage	*String*	41	44	text list hi pb corr emph
signed	*String*	14	16	text hi name lb date

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
salute	*String*	19	20	text hi
closer	*String*	18	30	text salute signed pb anchor hi dateline date rs address name s lb num add
corr	*String*	6	6	text corr
sic	*String*	14	21	text sic
dateline	*String*	14	26	text address date
trailer	*String*	30	32	text anchor address date hi
addrline	*String*	22	22	text hi lb name
abbr	*String*	2	2	text hi title
unclear	*String*	3	3	text
tagusage	*String*	0	0	λ
num	*String*	7	9	text abbr
add	*String*	16	22	text name date abbr num
del	*String*	7	7	text
orig	*String*	27	27	text pb foreign
figdesc	*String*	55	56	text title
cell	*String*	2	2	text abbr ref label
title	*Title*	32	32	text pb hi date
publisher	*Title*	34	35	text hi lb address
date	*Title*	9	9	text hi lb unclear
bibl	*Title*	30	49	text title author editor biblscope publisher pubplace date series anchor hi lb name note pb corr num respstmt ref edition
titlepart	*Title*	23	24	text hi lb title name figure foreign
head	*Title*	14	15	text note list gi att lb hi ptr figure ref corr title pb abbr bibl
gi	*Title*	5	5	text
class	*Title*	8	8	text
attname	*Title*	5	5	text
val	*Title*	6	6	text
entname	*Title*	12	12	text
l	*Title*	37	37	text hi pb ptr figure bibl foreign note q corr sic anchor stage emph title gap abbr ident name ref
speaker	*Title*	7	7	text hi

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
ptr	*Ref*	0	0	λ
index	*Ref*	0	0	λ
divgen	*Ref*	0	0	λ
gap	*Ref*	0	0	λ
pb	*Ref*	0	0	λ
lb	*Ref*	0	0	λ
milestone	*Ref*	0	0	λ
files	*Empty*	0	0	λ
anchor	*Empty*	0	0	λ
figure	*Unknown*	0	7	figdesc p head text
q	*Unknown*	70	290	text mentioned bibl title foreign hi code q p term gi att tag emph pb l lg text milestone name corr list note lb address sp ref abbr
item	*Unknown*	47	81	text p note list ptr gi socalled tag term att emph ident code q mentioned index title eg kw ref lb corr hi cit bibl date text pb sic foreign abbr name

Table C.3: The TEI DTD and the TEI Lite DTD [TEI94].

C.4 The document type for paper machine manuals

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
module	*VirtDoc*	0	32347	title overview nameloc clink variation
drawings	*SectGrp*	0	3105	title hwdrawings principles
variation	*Section*	0	5053	title nameloc clink electrics structurefunct operations automation mechanics hydraulics overview pneumatics
electrics	*Section*	0	7800	title variationdef safety structure maintenance drawings spareparts functions
maintenance	*Section*	0	3139	title periodicmaint ondemandmaint troubleshoot overview
ondemandmaint	*Section*	0	2488	title overview function
troubleshoot	*Section*	0	2583	title malfunction overview
operations	*Section*	0	3741	title variationdef procedures maintenance safety functioncondit overview parameters
procedures	*Section*	0	2900	title nameloc procedure overview description
automation	*Section*	0	4409	title variationdef safety structure functions software maintenance spareparts brochures overview
software	*Section*	0	3278	title system modules maintenance userinstructions
mechanics	*Section*	0	2362	title variationdef safety maintenance overview tableslists measurereports drawings
hydraulics	*Section*	0	4015	title variationdef structure overview safety functions maintenance drawings
parameters	*Section*	0	6002	title chapter
pneumatics	*Section*	0	3335	title variationdef structure overview safety maintenance drawings
principles	*Section*	0	4126	description
topic	*Section*	0	2601	para unorderedlist
figure	*ParaGrp*	0	96	graphic title legend nameloc
variationdef	*ParaGrp*	0	78	title authornotes overview
authornotes	*ParaGrp*	0	46	name date designnotes
designnotes	*ParaGrp*	0	58	title overview
safety	*ParaGrp*	0	662	description
structure	*ParaGrp*	0	1019	description nameloc
periodicmaint	*ParaGrp*	0	805	title function periodical

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
function	*ParaGrp*	0	1126	title target overview nameloc
target	*ParaGrp*	0	724	title period step overview precondition nameloc substance postcondition reference
unorderedlist	*ParaGrp*	0	242	listitem title overview
malfunction	*ParaGrp*	0	1509	title symptom overview
causecorr	*ParaGrp*	0	227	probcause precondition correction description
precondition	*ParaGrp*	0	301	step title overview preparations
structurefunct	*ParaGrp*	0	901	title variationdef structure layout interfaces functions overview
layout	*ParaGrp*	0	121	title overview
orderedlist	*ParaGrp*	0	289	listitem title
interfaces	*ParaGrp*	0	383	title overview interface
functions	*ParaGrp*	0	494	description
procedure	*ParaGrp*	0	1162	title overview description precondition action nameloc postcondition
system	*ParaGrp*	0	1385	structure interfaces functions
modules	*ParaGrp*	0	896	title swmodule
swmodule	*ParaGrp*	0	446	title structure functions
userinstructions	*ParaGrp*	0	908	title procedure
interface	*ParaGrp*	0	230	title overview
brochures	*ParaGrp*	0	462	overview
action	*ParaGrp*	0	395	title step overview nameloc
functioncondit	*ParaGrp*	0	389	title part
part	*ParaGrp*	0	300	title operation
operation	*ParaGrp*	0	190	title condition overview
condition	*ParaGrp*	0	168	overview
postcondition	*ParaGrp*	0	176	validation title overview completion
validation	*ParaGrp*	0	180	step title
completion	*ParaGrp*	0	155	step title
tableslists	*ParaGrp*	0	1708	title overview
periodical	*ParaGrp*	0	541	title overview
overview	*ParaCont*	0	181	subtitle para figure note caution table nameloc unorderedlist
description	*ParaCont*	0	970	title para chapter figure unorderedlist nameloc caution warning orderedlist note
period	*ParaCont*	0	107	para note unorderedlist figure caution
step	*ParaCont*	0	173	para note caution unorderedlist warning title substep figure nameloc clink table orderedlist

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
listitem	*ParaCont*	0	64	para unorderedlist note figure orderedlist clink caution
symptom	*ParaCont*	0	675	title para causecorr
correction	*ParaCont*	0	116	para step unorderedlist note warning
substep	*ParaCont*	0	78	para note figure title caution unorderedlist warning nameloc substep
legend	*ParaCont*	0	92	callout explanation orderedlist unorderedlist
chapter	*ParaCont*	0	1161	title para section unorderedlist figure table orderedlist caution
section	*ParaCont*	0	686	title figure note table orderedlist unorderedlist para subject
preparations	*ParaCont*	0	130	para
para	*Para*	88	89	text clink
note	*Para*	90	90	text clink
caution	*Para*	114	114	text
warning	*Para*	70	70	text
explanation	*Para*	77	109	text clink
nameloc	*StrGrp*	0	11	nmlist nameloc
name	*StrCont*	0	17	firstname surname unit
date	*StrCont*	0	8	year month day
hwdrawings	*StrCont*	0	1	title
spareparts	*StrCont*	0	8	title
measurereports	*StrCont*	0	18	title
clink	*String*	21	21	text
firstname	*String*	4	4	text
surname	*String*	8	8	text
unit	*String*	4	4	text
year	*String*	4	4	text
month	*String*	1	1	text
day	*String*	2	2	text
entry	*String*	12	12	text
substance	*String*	31	57	text clink
reference	*String*	30	30	text

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
title	*Title*	23	23	text clink
subtitle	*Title*	41	41	text
nmlist	*Title*	11	11	text
probcause	*Title*	41	41	text
callout	*Title*	30	30	text
graphic	*Fig*	0	0	λ
table	*table*	0	369	title tgroup
tgroup	*tgroup*	0	354	thead tbody colspec
thead	*thead*	0	27	row
row	*row*	0	38	entry
tbody	*tbody*	0	328	row
colspec	*colspec*	0	0	λ
subject	*Unknown*	0	598	para title topic unorderedlist note

Table C.4: The DTD for paper machine manuals.

C.5 The HTML document type

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
html	*VirtDoc*	2	42660	text head body
body	*Section*	303	41337	text p div h1 big hr h3 h2 blockquote ul h4 dl pre address table br center ol a li img code tt em b i map
div	*Section*	11	5254	text h1 p ul hr blockquote div h2 h3 dl h4 pre ol table
ul	*ParaCont*	0	328	li
dl	*ParaCont*	0	520	dt dd
ol	*ParaCont*	0	397	li
p	*Para*	199	495	text i b a tt em cite font br dfn samp strong var img p code small hr pre h3 h2 ul li
li	*Para*	72	166	text a p blockquote pre ol ul tt i b br code samp var li strong img em h3 h2
blockquote	*Para*	209	1239	text p pre ul center h3 blockquote a h4 i hr address br
dd	*Para*	88	138	text p a ol strong blockquote em br dfn cite samp kbd var code b i u tt ul dl sup sub
pre	*Para*	300	331	text font p a b
font	*Para*	252	265	text a br h2 samp var p hr strong b
small	*Para*	98	156	text a
table	*StrGrp*	0	1432	tr
tr	*StrCont*	0	670	td th
map	*StrCont*	0	0	area
title	*String*	38	38	text
i	*String*	12	13	text a i small p
a	*String*	11	36	text img br table td tr b cite kbd h1
b	*String*	5	9	text a p i code
h2	*String*	23	25	text a br font tt
h3	*String*	24	29	text a br hr
h4	*String*	5	22	text a font

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
big	*String*	36	65	text br address
address	*String*	37	53	text a br p
tt	*String*	26	26	text br
em	*String*	11	11	text a
cite	*String*	23	33	text a
strong	*String*	11	11	text
td	*String*	58	362	text p center b pre table img a br font ul h2 h4 dl dfn samp ol blockquote address hr i strong tt
center	*String*	26	486	text a p h3 h2 hr font h1 ul br b img table tt
code	*String*	7	7	text
dfn	*String*	9	9	text
samp	*String*	6	6	text
var	*String*	8	8	text
kbd	*String*	9	9	text
th	*String*	7	9	text h4
u	*String*	9	9	text
sup	*String*	1	1	text
sub	*String*	1	1	text
head	*Title*	0	28	title link meta
h1	*Title*	25	29	text a big img br
dt	*Title*	7	17	text cite a i samp code b
img	*Fig*	0	0	λ
hr	*Empty*	0	0	λ
br	*Empty*	0	0	λ
link	*Empty*	0	0	λ
area	*Empty*	0	0	λ
meta	*meta*	0	0	λ

Table C.5: The HTML DTD [W3C99b].

C.6 The document type for the texts of Shakespeare

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
play	*VirtDoc*	0	126790	title fm personae scndescr playsubt act induct prologue
act	*StrSect*	0	25075	title scene epilogue prologue
scene	*StrSect*	0	6169	title stagedir speech subhead
induct	*StrSect*	0	7067	title stagedir speech scene
personae	*StrGrp*	0	689	title pgroup persona
epilogue	*StrGrp*	0	831	title speech stagedir subtitle
prologue	*StrGrp*	0	1515	title stagedir speech
fm	*StrCont*	0	190	p
pgroup	*StrCont*	0	52	persona grpdescr
speech	*StrCont*	0	145	speaker line stagedir subhead
p	*String*	47	47	text
persona	*String*	23	23	text
grpdescr	*String*	22	22	text
scndescr	*String*	35	35	text
playsubt	*String*	16	16	text
stagedir	*String*	24	24	text
speaker	*String*	8	8	text
line	*String*	39	39	text stagedir
subhead	*String*	6	6	text
subtitle	*String*	18	18	text
title	*Title*	29	29	text

Table C.6: The DTD for the texts of Shakespeare [Bos97].

C.7 The document type for Reuters news

$E \in N$	$E' \in N'$	T_E	R_E	$D_E \subseteq N$ in document instances
lewis	*VirtDoc*	0	905013	reuters
reuters	*ParaCont*	0	922	date topics places people orgs exchanges companies unknown text mknote
unknown	*Para*	63	63	text
body	*Para*	837	837	text
topics	*StrCont*	0	3	d
places	*StrCont*	0	4	d
people	*StrCont*	0	0	d
orgs	*StrCont*	0	0	d
exchanges	*StrCont*	0	0	d
d	*String*	4	4	text
text	*String*	17	826	text title dateline body author
title	*String*	46	46	text
dateline	*String*	27	27	text
author	*String*	28	28	text
mknote	*String*	55	55	text
date	*Title*	23	23	text
companies	*Empty*	0	0	λ

Table C.7: The DTD for Reuters news [Reu97].

ISSN 1238-8645
ISBN 951-45-9278-6
Helsinki 2000
Helsinki University Printing House