Abstract

Modern WWW applications are highly integrated, rapidly changing, distributed multi-user systems which must take into account system and enterprise critical non-functional requirements. We present a twofold approach to web engineering: the static part addresses all aspects that belong to the contents of pages, text and images, while the dynamic part deals with interaction with databases and other application systems. We will focus on the static part, called document engineering. It involves the tasks of document design, authoring, and document production. Our approach to web engineering will suggest the reusable components content, structure, navigation, and layout.

We will present SchemaText, an integrated software tool, that provides a schema-based approach to web engineering. SchemaText implements the basic ideas of our methodology.

1. Introduction

Estimates on the effort needed to implement a web application will usually be scattered over a wide range. On the one hand, it is a common misconception that a web site can be created by untrained personnel using standard office products. On the other hand, companies spend millions of dollars for the implementation and maintenance of sophisticated and sparkling WWW applications.

Considering the large investment, a systematic and organized approach to web engineering is increasingly necessary. Web applications combine aspects of software and document engineering. Usual characteristics of current web based hypertext systems are:

- rapidly changing, heterogeneous technical environment
- rapidly changing contents
- distributed system architecture (client browsers and HTTP servers)
- high significance of information structuring, hypertext navigation, and layout
- hypertext structure reflecting the semantics of the contents
  - multi-user capability
  - 24-hour availability

Additionally, the distributed nature of web applications imposes numerous compatibility requirements. Elaborate browser and task oriented software (e.g. applets) has to be integrated with existing environments, which often include databases, PPS (Production Planning Systems), and workflow systems. For such applications, maintenance efforts will be high, particularly if large-scale changes are required at runtime. Furthermore, dynamic aspects of structure and content depending on user interaction have to be considered.

Basically, our approach to web engineering is twofold, distinguishing a static part and a dynamic one. The static part comprises all aspects that consider the contents of pages, text and images, while the dynamic part includes interaction with databases and application systems. For the dynamic part, like server or client side extensions or electronic commerce applications [7], methods from software engineering are applied.

In the following we will concentrate on the static aspects named document engineering. Document engineering involves three major tasks: Document Design defines the quality, writing standards, and modular text structures of the desired document. Authoring composes text, tables, and images. Document Production deals with the final publishing process, mapping the results of authoring and design onto a concrete layout.

The methodology of web engineering presented in Section 4 meets the specific requirements of document engineering. As a result, we will consider the reusable components content, structure, navigation, and layout as solutions for the hard problems of hypertext engineering. SchemaText, an integrated software tool, provides a schematic approach to web-engineering, based partly on the object-oriented paradigm found in software engineering. SchemaText implements the basic ideas of our methodology to generate complete hypertexts, including navigation and layout.

After summarizing the problems most commonly impeding hypertext production in Section 2, Section 3 will...
identify the components, paving the way for efficient web publishing. In Section 4 a methodology of web engineering is developed by analysis of the hypertext process model. Techniques from linguistics and software engineering are combined into the emerging discipline of document engineering. Subsection 5.1 will introduce the software SchemaText and offer appropriate solutions to the present problems by enumerating the necessary working steps when using the software.

2. Problems addressed

Since the advent of hypertext research many systems have been built and sometimes had to proof usability under heavy conditions including large, heterogeneous, and untrained groups of authors or consumers. Particularly the success of the World Wide Web was a major break through for the concept of hypertext and the hypertext markup language HTML. Unfortunately, users of hypertext systems are fighting with several problems that became clear only by taking part in large scale documentation projects:

1. A hypertext consists of nodes, containing portions of text, and links, connecting these nodes. Necessarily links should preserve referential integrity, which means that each link must point to an existing node. Additionally the link should run from a to b if and only if the intended semantic relation between a and b is still valid.

2. Many hypertext systems do not provide a consistent and reliable navigation. So users feel uncomfortable and have to concentrate on operating the system rather than understanding the text.

3. When first creating a hypertext that is assumed to have a long life expectancy or the final size, structure or content are unknown in advance, system designers lack of techniques and tools to plan, test, and develop such a complex system.

4. Problem 3 also comes in, when maintaining a hypertext of considerable size. Sometimes huge structures have to be changed very frequently. If too much of a hypertext is ‘hard-coded’, such changes demand enormous effort if they are not impossible at all.

5. Another problem arises if two or more hypertexts need to share common properties. An obvious example is a multilingual hypertext that has to pertain the same structure while containing different languages in all places where text is shown. This also holds if different groups of readers are expecting to be addressed individually with regard to their languages, cultural aspects, the level of presumed knowledge, or the level of detail of the provided information. In all these cases several ‘versions’ or ‘perspectives’ of local node contents have to be managed while the overall structure as well as semantic links should be preserved for all productions.

6. Last but not least, most hypertexts cannot be viewed as isolated documents. They are embedded in larger structures and strategies. A hypertext is in most cases only one facet of the output produced by a person, project, or company. The surrounding environment is formed by other information repositories, formal, and informal guidelines. These are for example guidelines compiled to reflect a consistent style of corporate design. Authors of any material addressed to public are supposed to follow the rules in these guidelines including typographic conventions, certain fonts, dimensions and proportions of images and text, and, of course, usage and proper placement of special logos, identifying the corporation.

This last problem has the following aspects:

- The layout may have to conform to corporate guidelines.
- The structure often corresponds to an organizational or geographical structure.
- The content of a hypertext may have to be published in several ways or formats. However the content of the hypertext may stem from multiple external sources. It may be changed frequently or gathered dynamically.
- When publishing the hypertext to different targets, not all of the content might be included in the resulting product or publication (flyer, website, technical documentation, book, ...). The undesired material should be masked out or hidden temporarily for each target produced.

Our experience, gained in numerous projects implementing large hypertext systems, shows that significant effort was necessary to find satisfying solutions.

Boosted by commonly available internet technology, the usage of hypertexts will continue to spread and rise in fields that have been dominated by conventional, linear texts and documents up to now. Based on the experience with existing systems, we are interested in streamlining the production, reducing the efforts and costs, and increasing efficiency of building new hypertext systems. Effort can be reduced to a minimum when we succeed in identifying components of hypertext systems that can be stored in libraries, recalled, and reused in individual combinations in order to create a new hypertext out of existing elements.
3. Identifying components for web publishing

Figure 1 shows the central components of web publishing: content, structure, navigation, and layout. Content-related work in our terminology is called authoring, whereas the activity of structuring is marked as design.

As content and structure are intrinsic elements of a hypertext, representing the author’s message, they are located at the top of the image, serving as input to navigation and layout, which we consider mappings. The output of navigation is labeled as ‘Concrete Navigation Structure’ (CNS), which is incorporated into the final hypertext by layout mapping (see Subsection 4.3).

When creating a large hypertext, it is a good starting point to build a hypertext pattern first. The use of a pattern avoids dealing with format-related stuff too early. Emerging from the hypertext pattern created by authoring and design activities (see top of Figure 1), the final hypertext is generated by an automated production process. The result of the production is a format-dependent collection of hypertext pages with hypertext links as shown at the bottom of Figure 1. For the web, mainly HTML is used as a formatting language, however linear document formats like RTF (Rich Text Format) and MIF (Maker Interchange Format for FrameMaker) can be created using the same approach.

In order to examine the dependencies during the production of hypertexts, the elements of the hypertext pattern have to be named properly and the production process has to be described in detail. We distinguish the following types of objects:

- **Nodes (N)** are primitive entities of the hypertext pattern.
- **Links (L)** are directed connections between two nodes.
- **Node types (NT)** are classes of nodes with common properties.
- **Link types (LT)** are classes of links with common properties.

These concepts are lent from the object-oriented programming paradigm in order to enable the reuse of content patterns. With this distinction, authoring can be divided into authoring-in-the-large, when dealing with content patterns (NT and LT), and authoring-in-the-small, when dealing with instances (N and L). By contrast, we speak of design-in-the-large, when defining structural patterns (LT and NT) and design-in-the-small when manipulating struc-
ture (L and N).

We have found it unavoidable to discriminate structural nodes and links from those expressing content. However authoring and design are represented using the same technical elements (NT, LT, N, and L). An example will clarify, how content and structure can be distinguished, though:

An instruction manual for a technical device may contain a description of all significant physical parts as content nodes. The aggregation of parts (e.g. a computer consists of a CPU, Memory and several input and output devices) could be modeled by links of a particular type (e.g. "consists of"). Independently, the manual is usually built up from sections as structural nodes and a table of contents, collecting structural links.

Usually structural nodes embed content nodes by way of structural links, hence connecting both concepts. Content carries terminological relationships between concepts, whereas structure serves to arrange and divide information into connected units. A particular link can express a terminological relationship as well as being merely a structural connection. A node (and its type) can carry some concept intrinsically bound to the content of the hypertext but it also might serve as a structural unit. Hence, it is not possible to separate content from structure simply by distinguishing nodes and links. In general, separation of both processes, authoring and design, and the grades of detail, in-the-large and in-the-small, is sometimes ambiguous. The appropriate decision depends a lot on the authors’ experience and personal style. Consequently the supplied process model (Section 4) will make no assumption on ordering these actions.

Above, we have described navigation and layout as mappings. In a software environment such mappings are realized as programs or scripts. Each project requires a special set of these scripts to map its input patterns onto the final hypertext elements. In order to achieve reusability of patterns on the one hand and scripts on the other hand, we have introduced an interface layer. With this prerequisites it is now possible to build libraries of normalized scripts acting on normalized sets of parameters.

The objective of dealing with hypertext components is to make hypertexts modular and modules reusable. Components, if fully separate, can be stored into a repository and recombined later. Separation and recombination of content and structure including their patterns involve typical graph algorithms.

4. The methodology of web engineering

Describing a methodology of web engineering demands an appropriate process model for the development of web applications [12], [11]. We are identifying methods and techniques to effectively process the development steps, and indicate tools to support particular techniques. The presented process model is based upon standard process models applied to the software engineering task (e.g. [18]). These models usually comprise analysis, design, implementation and test steps and result in a final maintenance phase. As introduced, web engineering integrates both dynamic and static aspects. In the following, we will focus on the static aspects.

But what exactly are the static aspects of web applications? It is that part of an application, that behaves reproducible, i.e. not dependent on previous interaction, and involves nothing but standard system infrastructure (web server, internet connection, file system, etc.). Thus it is hypertext [3], for the web usually formatted with HTML. A detailed description of functional dependencies concerned with the production of hypertext has been given in Section 3. This knowledge will now be used as the base for a hypertext process model. In our opinion, every document is a hypertext, since most often there are links, either structural or semantic, used to organize text. To be compliant with the terminology applied elsewhere [6] we will use the term document engineering as a synonym for hypertext engineering in the following.

4.1. The hypertext process model

The hypertext process model in Figure 2 centers around the creation of components which have been defined in Section 3 in order to model the functional dependencies of hypertext production (Figure 1). From our experience, the according major steps authoring and design are inseparably tied together and are thus dealt with in unspecific order or alternately in several steps. Usually, large-scale authoring and design precede small-scale authoring and design, but that depends for the most part on personal style.

The particular process steps comprise sets of tasks and results. During the analysis, - especially in large projects - editorial guidelines may be written in order to define the quality, writing standards, the target groups, the target formatting language, and dynamical aspects of the intended hypertext. These guidelines can be compared to requirements analysis documents in software engineering. A separation of the hypertext into handy modules and the coordination of cooperative work groups, divided vertically (module-oriented) or horizontally (process-oriented), are required to handle larger projects. Often, multiple variants of a hypertext must be planned according to different target groups, for instance multiple languages or multiple security requirements implementing restricted access. A time-limited validity of information may require the con-
sideration of multiple text versions. Often pre-existing hypertexts have to be integrated, eventually wrapped into a new layout and navigation. Layout itself, possibly provided by graphic designers, has to be considered.

Figure 2: The hypertext process model

Based upon the analysis phase, the design-in-the-large produces types of structure-building nodes and links for the hypertext pattern, used as templates during the following design-in-the-small. Simultaneously content patterns are defined by authoring-in-the-large. Then nodes and links are created, representing the document’s content. The classical authoring task of formulating text also belongs to the authoring-in-the-small. Of course, it cannot be underestimated. Since navigation and layout are mappings, their final definition is refined along all of these steps.

In order to validate the properties of the resulting hypertext, achieved by automated production, all steps of the production process should be tested frequently and at any phase. Once the specification of hypertext components is complete, the selection of an appropriate version, variant, and format should suffice to generate the final hypertext. No further creative act is involved.

Finally, the maintenance of hypertexts requires a sophisticated change management for all of the process steps. The quality of a solution for hypertext evolution is in fact the yardstick a supporting software tool will be assessed with.

4.2. Techniques of document engineering

Document engineering involves both techniques from linguistics and computer sciences, in our opinion. According to [5] and [6], among the first phases of document engineering the quality of the final product has to be defined. Agreement in the desired quality is the precondition for forming authoring teams and scheduling the process of production. The quality definition should include the typical readers addressed by the document, the function of the text, the domain and limits of the content, situations and the media the content will be presented. These conditions form the base when designing macroscopic textual units during authoring-in-the-large.

After that, modules of minimum dependencies can be assigned to working groups. Rules of wording and a set of common terms can be handed to the authors to support them in the task of authoring-in-the-small to achieve an optimum of comprehensibility and productivity.

Besides the ordinary waterfall model reused in our hypertext process model in Subsection 4.1, document engineering can profit from methods and techniques developed by object-oriented programming (OOP) [2], [4], [16].

Analogous to structured data types going along with OOP, it is useful to guide authoring activities by templates or forms to ensure comparable structures for contents written by different authors and to enable automated processing in the future. This is equivalent to assigning types to certain textual elements. Even more benefits can be gained from a consequent and clear class structure, if changes applied to the classes are inherited by all existing instances [10].

With increasing production of written information the need for reuse and sharing of existing information seems to become as obvious as it is in software engineering already. An adequate model of modularity is required before separating modules in order to reuse them independently.

4.3. Navigation structures

In order to reuse and recombine the parts of large hypertexts into new documents efficiently, several preconditions seem to be necessary:

- Separation of content, structure, layout, and navigation, in order to generate a new document with different layout and navigation structures.
- Consistent tagging of the textual building blocks according to the editorial guidelines.

We focus here on navigation structures. Since navigation is normally an integral part of a document, there is no obvious and standardized interface between navigation structures and content respectively layout. Nevertheless, we characterize a feasible separation and means for combination. Navigation structures in hypertext only appear in several typical instances:

- Tree: A classical table of contents (TOC) is organized as a tree. A document may have various tables of contents for reader groups or application scenarios.
- A directed acyclic graph (DAG) may be used as an alternative representation for a table of contents, a structured index, and of course bibliographic references.
- Linear sequences are used for ‘guided tours’ which may appear multiply for various topics in one document.
5. Components in practice

Authoring like painting most often requires not only technical skills but depends on creativity and experience as well. The order of processing belongs to personal style, thus a convenient and useful tool cannot restrict its users’ activities by providing a stringent flow of work. It might be seen as a drawback that training efforts are usually rather high in order to become productive, but it is our opinion that flexibility will pay in the long run. Hence a guidance of intellectual activity by the presented tool for hypertext design, authoring, and production, SchemaText, should rightly not be expected.

5.1. SchemaText

The graphical user interface of SchemaText 1) allows to create and manipulate hypertext patterns. Figure 3 shows the class editor and the instance editor representing this article, provided to work on node and link types or node and link instances, respectively. Since all objects of the hypertext are under control of SchemaText, it can guarantee referential integrity. The object-oriented paradigm allows to efficiently manipulate an arbitrary number of hypertext elements by schema evolution and application of user-definable metadata. Schema evolution includes merging nodes or link types, splitting link types, changing types of nodes and links, and rearranging and ordering nodes or links. Additionally a linearization mechanism is offered to produce on print media.

As introduced, SchemaText concentrates on the static aspects of web engineering. This is reflected by the architecture of hypertext production, that is based on a compilation process, converting a hypertext pattern into the final hypertext. This yields the ability to generate different formats (as demanded in Section 3), since the authoring process takes place on a certain level of abstraction. The hypertext pattern itself is stored using the hypertext structure definition language (HSDL), the basic file format of SchemaText.

The compilation is based on small scripts, called expanders, serving to ‘expand’ nodes and their content into the final format (e.g. HTML). These expanders, written in IEEE Scheme 1178 [1], allow to completely customize SchemaText’s output. Information scattered over several nodes can be extracted automatically and huge collections of documents can be generated by batch-processing.

---

1) A demo of the single-user version of SchemaText can be downloaded from the URL http://www.schema.de.
One of the parts defining a SchemaText production is the so-called perspective, which is an isomorphic structure of the hypertext representing different text languages. Figure 4 shows the (E)nglish perspective, cutting off the layer of contents from nodes A and B containing only English language. Other productions could use the (G)erman or (F)rench perspectives. Structure is common to all perspectives but can be varied among productions by using different masks.

5.2. Implementing components with SchemaText

In this subsection we will examine the presence and properties of components in SchemaText by answering the following questions:

- **Representation**: Where are the components in SchemaText?
- **Manipulation**: How can they be edited?
- **Discrimination**: How can they be extracted, isolated, and stored separately.
- **Recombination**: How can they be recalled and reused?

As introduced in Subsection 5.1, schematic and efficient authoring is supported by offering the ability to define patterns. Thus SchemaText follows a technical approach, i.e. the software deals with types of nodes and links, whereas the human author has to distinguish between structure and content. Content and structure are tied together in many aspects and so it is quite reasonable to implement both using the same technical abstraction of types on the one side and instances on the other. So, considering authoring-in-the-large, content and structure cannot, and often must not, cleanly be isolated from each other.

Regarding authoring-in-the-small, separation is quite easy: Contents of a node can be edited in the integrated node editor or using one of the supported external editors, such as Microsoft Word for instance.

Structure can be manipulated by using the graphical structure editor of SchemaText, e.g. by drawing a link with the mouse and eventually choosing the link type. A variety of sophisticated and programmable import mechanisms exist to extract existing structure from monolithic files, tagged in some manner. Integrating new elements and rearranging huge structures can be accomplished by schema evolution, which allows to modify sets of objects by changing the structure and the properties of their classes.
Since the final hypertext is generated, modification, separation, and reuse of layout can be handled very easily. Layout is generated by expanders that are always specific to the format of the current production. So layout can naturally be tuned to meet the special abilities and limitations of the file format to be generated for different media (e.g., paper, WWW).

Certain navigation structures are preferably supported by SchemaText (ref. Subsection 4.3). Semantic navigation patterns can be modeled by using links of corresponding link types. Arbitrary navigation patterns can be extracted from the set of all links by accessing the full programmable expander interface to SchemaText’s object model (STOM) [14].

As introduced in Section 3, layout and navigation are mappings from the domain of structure and content to gain the final hypertext. A mapping always depends on the parameters currently supplied, so we cannot hope to successfully combine content and structure of one hypertext with layout and navigation of another. To make layout and navigation more independent and reusable, we introduce an interface layer as shown in Figure 1 formed by user-defined markup and rules. Markup can be done using special metadata types and assigning metadata values to selected objects (types or instances). The implementation of mapping rules is a predestined task for AI-languages like Scheme as used for SchemaText expanders (see Table 1).

<table>
<thead>
<tr>
<th></th>
<th>Representation</th>
<th>Manipulation</th>
<th>Discrimination</th>
<th>Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content</td>
<td>large: special types of nodes and links; small: inside nodes; metadata</td>
<td>metadata editor, import; large: creating nodes and links of certain types in structure editor; small: typing (internal or external editors);</td>
<td>large: clean design and sound usage of types, metadata and sgml-tags based on functional aspects; perspectives, grouping certain nodes in separate repositories; small: trivial</td>
<td>large: perspectives, interschema-links, various import mechanisms, masks; small: conditional text; use of embedding links &lt;IGNORE&gt;-Tags for conditional text</td>
</tr>
<tr>
<td>Structure</td>
<td>large: structured instances; small: links and nodes</td>
<td>large: creating nodes and links of certain types in structure editor; schema and structure editor: mouse, schema evolution; import, applying scripts</td>
<td>large: clean design and sound usage of types, metadata and sgml-tags based on functional aspects;</td>
<td>large: interschema-links, various import mechanisms, masks</td>
</tr>
<tr>
<td>Navigation</td>
<td>expanders, metadata</td>
<td>GUI-options, expander programming, metadata editor</td>
<td>clean design and sound usage of expanders and metadata for appropriate aspects</td>
<td>Select expander library, define interface by setting metadata or modifying expander parameters for the library’s navigation rules</td>
</tr>
<tr>
<td>Layout</td>
<td>expanders, metadata; certain types of nodes and links</td>
<td>GUI-options, expander programming, metadata editor</td>
<td>clean design and sound usage of expanders and metadata for appropriate aspects</td>
<td>Select expander library, define interface by setting metadata or modifying expander parameters for the library’s layout rules</td>
</tr>
</tbody>
</table>

Table 1: Components in SchemaText
5.3. Other approaches

Comparing our approach to others is cumbersome, since all of the existing systems and approaches vary regarding their functionality and emphasis. In order to gain some insight, we start from the discussion of approaches presented by Lowe and Hall [12]. Their comparison covers system and process features. In order to evaluate hypermedia systems and development processes several crucial questions have to be answered. Our focus is on the capabilities of the processes:

1. How is the issue of application maintenance addressed?
2. How is the issue of reuse addressed?
3. How is the full development lifecycle supported?
4. How is process management supported or promoted?
5. How is the issue of cognitive management during development addressed?
6. In what ways is support for enhancing development productivity provided?

Lowe and Hall present five approaches, focusing around academic prototypes as well as commercial systems. The first approach is the Matilda hypermedia framework [13] which emphasizes scalability of information presentations. Although Matilda is still under development, its prominent features are the organization, representation, and reuse of large information spaces. The second system is Microcosm [8] which is a commercial product based on a powerful linking server and an extensive set of filters for media import and export. The server-based approach allows flexible usage of the linking structures, however it requires proprietary browsers in order to exploit all the functionality. The third system is Hyper-G [15] respectively Hyperwave, a distributed hypermedia information management system. While Hyper-G is an academic prototype system, Hyperwave is a commercial product which differs significantly in functionality. Like Microcosm, this is a server based approach to support the multi-user access and management of multiple media documents.

The Relationship Management Methodology (RMM) [9] is a complete design methodology which covers the process from analysis to construction and testing. However, there exist only prototypical implementations yet. For this reason, RMM is not directly comparable to the approaches above with respect to system specific requirements like application maintenance and component reuse.

Our approach, as presented above, covers the basic concepts of RMM like entity-relationship, entity, navigation, user interface, and screen design. The user interface of SchemaText is designed to structure and handle large information spaces comprehensively. It allows the reuse of components (node types) and links (link types) and the configuration of a particular application. Furthermore, the reuse of generic navigational patterns becomes possible and a complete lifecycle model for an application is provided.

Table 2 summarizes the strength of the mentioned approaches with respect to a selection of criteria. We do not claim that these criteria provide a complete description of processes, however, the most important features are highlighted.

<table>
<thead>
<tr>
<th></th>
<th>Matilda</th>
<th>RMM</th>
<th>Microcosm</th>
<th>Hyper-G</th>
<th>Hyperwave</th>
<th>SchemaText</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application maintenance</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Information reuse</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Development lifecycle support</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Development process management</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cognitive management during</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>development</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Comparison of approaches
6. Conclusions

We have presented an approach to web engineering focussing on the document engineering part which comprises the activities document design, authoring, and document production. In order to identify appropriate steps and techniques for document engineering a detailed model of functional dependencies during hypertext production was developed. A large-scale process model was suggested depending on some software engineering standards. The importance of both linguistic and computer science techniques for document engineering purposes has been explained. In this context we have emphasized the role of navigational structures for automated hypertext production. The presented methodology centers around the identified reusable components contents, structure, navigation, and layout, which - as we state - are distinct parts of a document repository.

Finally, we described the software system SchemaText which supports our document engineering approach by several useful features, meeting the following particular requirements of

- referential integrity
- reliable and consistent navigation
- support of the overall document lifecycle, including planning, testing, and maintenance
- handling even huge hypertexts
- incorporating rapid prototyping facilities
- intuitive graphical representation and manipulation of hypertext structures
- sophisticated change management by schema evolution
- different levels of abstraction
- multilinguality

Finally five hypertext authoring tools have been compared against a set of requirements implied by the document engineering approach, showing that the derived methodology of document engineering is not restricted to SchemaText.

7. Acknowledgements

We would like to thank our colleagues Stefan Freisler, Sebastian Göttel, and Marcus Kesseler for lively and stimulating discussions on the issues of this paper and for reading and commenting drafts.

8. Bibliography