Making Complex Document Structures Accessible Through Templates

Felix H. Gatzemeier and Oliver Meyer*

RWTH Aachen
Lehrstuhl für Informatik III
Ahornstraße 55

52074 Aachen

phone: +49 (2 41) 4 40-2 13 13
fax: +49 (2 41) 88 88-2 18

{fxg,omeyer}@i3.informatik.rwth-aachen.de

Abstract We address two problems of technical authors in structured environments: **(1)** Structure definitions of the SGML school are limiting: they require one primary hierarchy and do not cater for link types and **(2)** Real-life structure definitions are too large to be comprehended easily. As solutions, we propose graph types and usage templates.
The edge types and inheritance of the proposed graph type model are useful modeling tools. We give examples for structures that can be expressed more precisely and with gain for the author using graph structures. There are also graphical tools available to define graph types and to specify operations on graphs.
Templates can be used as a simple parameterization mechanism. A template illustrates the usual usage of a substructure, as opposed to the minimal one required by a structure definition, or the maximal one allowed by it.
We also present a prototype authoring application based on these ideas.

1 Introduction

While markup languages support authors in producing documents that adhere to a certain structure, there is still the serious problem of the authors knowing only parts of the structure definition. This leads to

- cognitive overhead for the authors who cannot create the documents the way they want to and
- formally correct documents that use the structures in semantically inappropriate ways.

* This work has been funded by the Deutsche Forschungsgemeinschaft (DFG) in its "Schwerpunktprogramm V3D2" (Verteilte Verarbeitung und Vermittlung digitaler Dokumente, Distributed Processing and Exchange of Digital Documents), http://www.cg.cs.tu-bs.de/dfgspp.VVVDD.

The problem is to a significant part due to the complexity of structure definitions used in technical writing. For example, the reference description of the DocBook DTD extends over more than 400 pages (WALSH AND MUELLNER, 1999, part II, pp. 113–566). Only a few of the structures allowed by these definitions are frequently used in actual document construction.
While the problem itself cannot be solved through tools, we address the symptoms mentioned above. One reason for the volume and unwieldlyness of DTDs is their limited expressive power and the inherent ordered hierarchy. As an alternative, we discuss graph types as more expressive descriptions of allowed structures in section 2.1. As a direct aid for the author, we present templates of related structural entities that are documented and can be instantiated together in section 2.2. In section 3, we report on the prototype authoring environment `rwe` that has been implemented based on these ideas. References to related work are made at the end of the relevant sections.

2 Solution Concept

In this paper, our goal is to offer authors generic structured editing applications that are parameterized by structure definitions and simple additional data to provide a customized user interface that supports authors in creating the appropriate information, while the corresponding application logic places it in the document and links it to the relevant places. Today, parameterization is mostly based on macros, requiring programming skills.
The DTD of a class of documents is used to parameterize generic XML/SGML-editors (such as FrameMaker+SGML, Adept Editor, or XMLSpy). Given only the information from the DTD and additional layout information in style sheets, little support for the author can be provided.
We claim that using graph types as a way to describe document types increases knowledge in the parameterization data and provides better support for the author. We further claim that templates support the author significantly, while still being lightweight parameterization. Even if templates can be used with today's hierarchical structured documents, they fit even better into our graph-based approach, as they model authoring actions spanning distributed parts of the document hierarchy more directly.
As a running example, we will use an author who wants to extend the manual for a software modeling tool. As a general Basis, the DocBook DTD has been set. By contrast, a graph type using concepts and names from this DTD is discussed. For the architecture notation, we'll use UML (RUMBAUGH ET AL. (1999)), while our imaginary tool bears some resemblance to Rational Rose (RATIONAL (1996)).

2.1 Improved Expressiveness

The modeling tool provides views and operations on models based on specific modeling concepts. In the manual the descriptions of the highly interrelated concepts (e. g. classes, attributes, objects, and inheritance) are used to describe the views (e. g.

class diagrams). To describe the commands, their invocation from the visualization and their effect on the model should be documented.

Some further constraints hold for the various descriptions: For every concept introduced there exists some visualization and commands to modify the model. Inversely, every visualization represents one concept. Each command may operate on any number of concepts or visualizations.[1]

There is no *natural* mapping of these relationships and constraints to a hierarchical document. One choice would be to divide into subject areas (e. g. class diagrams vs. collaboration diagrams). In each subject area, each relevant concept with its visualization and commands is described. This keeps the information on a concept together and groups some interrelated concepts close by. But there is no overall description of the involved concepts, their relationships across subject area boundaries are artificially broken.

A more comprehendible structure would divide concepts, commands and visualization as in figure 1. This provides the declarative information needed by the reader, as well as an indexed description of the commands. The dependencies of the content can be used in the single sections to produce a coherent and readable text.

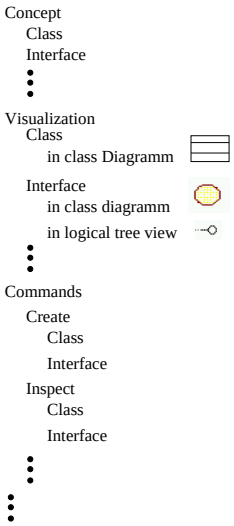

Figure 1. Abstract contents of a software handbook.

[1] For now we use a restrictive view. In almost every given text, there are exceptions from this rule and some visualization entities might show more than one concept, e. g. a class in a class diagram contains its attributes and methods.

Customizing DocBook. Given the expressiveness of a regular DTD, however, the formal relationships of concepts, visualizations, and commands can not easily be expressed. A concept element in a DocBook customization layer can be used to ensure the presence of the descriptive parts required for a concept as follows:

```
<!ELEMENT Concept (Introduction, Visualization+,
                   Command+)>
```

This new element would be incorporated in the hierarchy underneath “sect1”. Later processing would reorder the “Visualization” and “Command” elements into the output sections. The author’s view on the document should bear more resemblance to the result, though. Therefore “ConceptSection”, “VisualizationSection” and “CommandSection” elements will be introduced replacing “sect1”, each containing their corresponding description elements.
To still enforce the description of the visualization for a concept or vice versa, required references between these elements can be introduced. Thus every “VisualizationDescription” has an IDREF attribute that has to reference a “ConceptDescription”. This element in reverse carries an IDREFS attribute to refer back to its “VisualizationDescription” elements. As ID/IDREF[S] are untyped, this constraint can not be expressed in the DTD. An author has to conclude from the attribute name (e. g. displayedConcept) that it should reference a “ConceptDescription”.
The authoring application, however, cannot draw this conclusion. When the author is asked to supply a value for the required attribute, all the application can offer him is a list of all elements with defined ID attributes. Furthermore, a generic validator cannot validate the document to the intended extent.

Modeling the content in a graph. A graph as the underlying data structure of a document can reflect this nonhierarchical, interconnected nature, leading to better author support. For the following discussion, let us assume the simple translation of a subset of the DocBook section hierarchy to a graph type as shown in figure 2. The sectioning elements “sect1” and “sect2” inherit from a common “section” class that provides the title, as an example. Sections of the same level follow each other and lower levels can be attached to higher ones.[2] Edges may be traversed in both directions, so there is no need to define back-links.
Further customization using inheritance yields the concept substructure as shown in figure 3. “Concept Section” etc. are derived from “sect1”, inheriting the title and embedding of “sect2” elements. The description elements in turn inherit from this class.
With the required “visualized as” and “modified through” edges, the document modeler demands that each “Concept Description” be connected to at least one “Visualization Description” and at least one “Command Description”. The hierarchical

[2] The relationship of graphs and ordered hierarchies is treated in more depth in GATZEMEIER AND MEYER (2000). In this document, we assume a library of basic visualization and query features for ordered hierarchies we implemented in our prototype `rwe` (section 3).

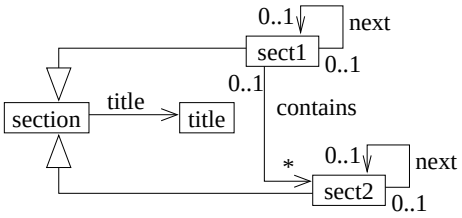

Figure 2. Translation of DocBook section hierarchy to a graph type.

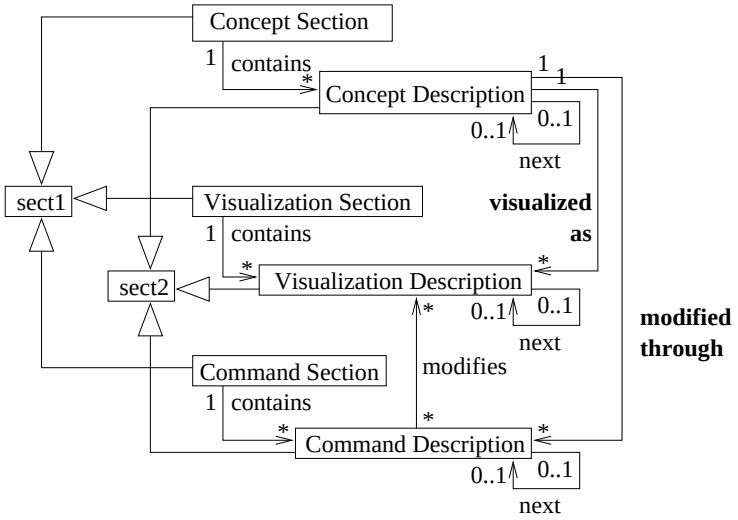

Figure 3. Graph type for the concept substructure.

structure, however, is still based on the resulting document. The content structure does not interfere with it.
The information about the required context is also available to the authoring application in the graph type. It can use that information to offer *automatic context creation*. When the author requests creation of a new “Concept Description” element, so-called *placeholders* for the elements required by edges with minimal cardinality of at least one are also created, together with those edges. The author may thus create elements of any type while the document still conforms to its graph type.
Placeholders are not fully valid elements. The only operations allowed for them are **(1)** Instantiation: convert the placeholder into a regular element, with automatic context creation or **(2)** Unification: merge a placeholder with an existing element, preserving edges where possible. The author is free to take these actions whenever he wants, so the application does not enforce working styles as structured editing applications without placeholders do.
As the targets of edges are typed, the author can be presented with elements of allowed types only. For example, when drawing another “visualized as” edge from a “Concept Description”, he would get a list of the “Visualization Description” elements.
The use of inheritance tells the author that the “title” edge really describes the same connection in “Concept Section”, “Visualization Section” and “Command Section”. In a DTD, he has to assume the same use from attribute or names or content models, relying on consistent naming by the document modeler. Moreover, the document modeler does not have to replicate all properties from an element type to another that can now inherit them.
Inheritance can also help the author in finding a special element type in the document model. Well-used inheritance provides the author with a hierarchical view on the element types. Today’s tools can provide the author only with a flat list of all or the currently valid element types.

Comparison. Figure 4 shows the meta-schema of the PROGRES graph types (SCHÜRR (1991)) used here as a UML class diagram. The most noticeable extensions are inheritance between node classes (a node type is a leaf in the inheritance lattice) and typed edges. The differentiated edge cardinalities are another extension, but they offer little benefit to the author or reader. The graph meta-schema does not support “contains” or “follows” as special relationships as a DTD implicitly does (see footnote 2).
There are also uses for elements with truly unordered contents. If the “Command Section” of our example is expressed with markup languages, the linear nature of the document instance would always impose an order on the “Command Descriptions”. Now suppose that the command descriptions are also used to create an additional alphabetical list of all commands and their descriptions. In this example, the command list is handled as a command set, the order is arbitrary. As there is no means to express this formally, the author of the document might not be aware of this. As a result the author might bind the “create Object” command description to the “create Class” description by e. g. describing only the differences in the successor. While this deictic reference works for the continuous manual, it breaks in the alphabetical reference.

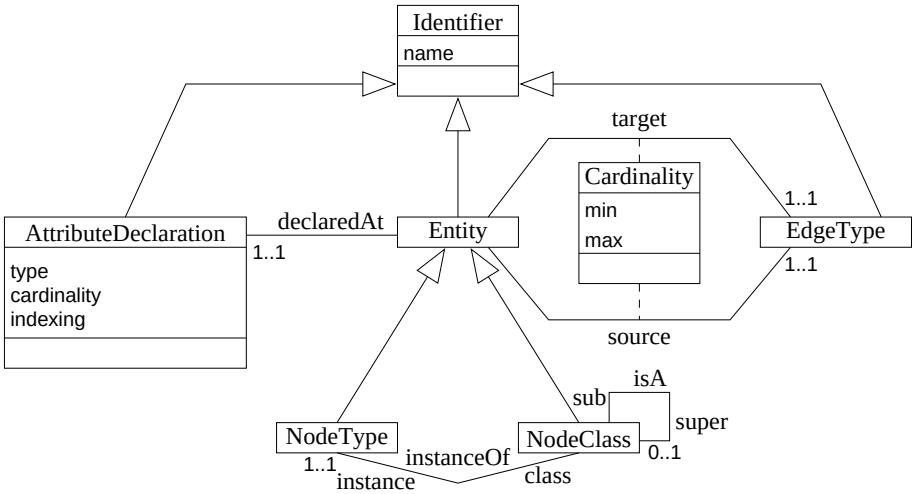

Figure 4. Graph meta-schema.

In the graph type this unorderedness can be expressed by not allowing “next” relations. A generic tool can then use this information to present the author with an user interface that does not suggest a sequential order and thus encourages self-contained sections.
It might be argued that most of these benefits can be achieved through minor extensions in the DTD and special attributes in the elements. The graph model is the more general approach and results in a consistent, well understood model. Graph rewriting systems (for example Grrr (RODGERS AND KING (1997)) and PROGRES (SCHÜRR (1991))) allow complex interactive transformations. They can also be used to define very detailed further constraints on the resulting graph structure (BEHLE ET AL. (1999)).
The main benefits result form the non-planarity of the graph model. document hierarchy along sections and the concept hierarchy are really of equal importance. In markup languages, the document modeler has to choose one of these hierarchies creating an artifical asymmetry. This might surprise and confuse the author if he expects to use the other hierarchy that is expressed only through ID/IDREF[S] attributes. The graph model and its type allow an equal treatment of both hierarchies and thus a symmetric document.

2.2 Templates

In the previous section, the document type information was enhanced to further constrain document instances and to better parameterize a generic editor. In this section, additional parameterization by templates is used to guide the author. This is an extension of the widely used practice of supplying sample documents with a DTD that authors can study and modify.

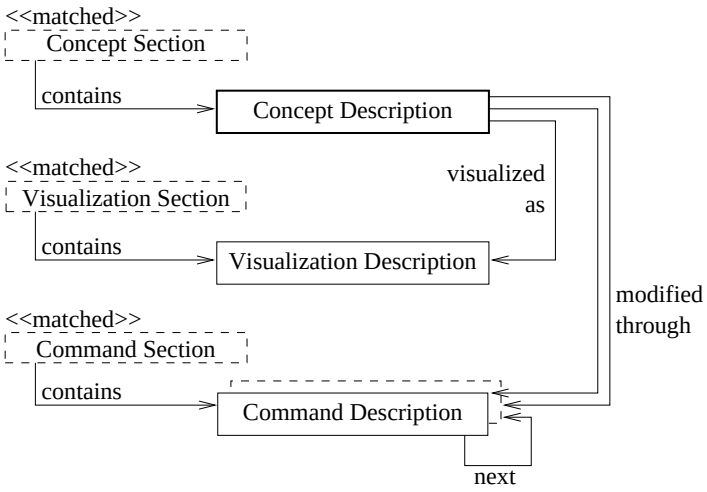

Figure 5. Template Example.

The “visualized as” and “modified through” relations of figure 3 enforce at least a single visualization description and command description for every concept. Still, there are the abovementioned exceptions (see footnote 1) from this rule. To cater for these, the cardinalities for these relations have to be loosened; both descriptions become optional. With this modified type, the automatic context creation does no longer create a visualization description. Yet it is the *usual case*. On the other hand, there may be optional connections to historical notes that are rarely used. Solely on the basis of the document type no differentiation can be made between these kinds of options.
The information about the usual case is given to the authoring tool in form of a template. The template for this application looks like figure 5. In the context of a “Concept Description”, exactly one “Visualization Description” and at least one “Command Description” are expected.
A *template* describes a subgraph of the document structure together with embedding information. It contains elements to be created shown as solid boxes, placeholders shown as dashed boxes and embedding connectors marked by the “matched” stereotype. New elements are to be filled with contents. Placeholders can be manipulated as described above. Embedding connectors are, when possible, automatically unified with an unambiguous element of correct type. If e. g. the document contains exactly one “Concept Section” the new “Concept Description” is automatically placed there.
The template also indicates to the tool that usually multiple “Command Descriptions” belong to a single concept. Therefore additional placeholders are created for this element, conveying this to the author.
The tool support for templates should include the history of template instantiation. When instantiating e. g. the “Command Description” placeholder another one is created automatically. The tool keeps the connection from the placeholder instance to

the used template. In the template, the “Command Description” is defined as being required and set-valued, so removing the last “Command Description” linked to the “Concept Description” would result in a warning instead of being attributed to generic structure-conforming operations.
The templates serve three purposes: **(1)** They guide the author in how to use the given element types. In our example, they show that a Concept usually is visualized in one way and operated on through multiple commands. When presented with a new DTD, example documents are frequently used to learn the common usage of defined element types. Yet no further tool support can be provided for that approach. **(2)** Templates also help to create documents faster, as they allow to create complex structures en bloc. **(3)** In our graph-based approach, templates make it cheap enough to create connecting edges between related elements. In the more text-oriented view, templates become multiple text fragments that need to be inserted at multiple locations at once. The interconnectedness can only be provided through the weak, untyped, unidirectional ID/IDREF[S] attributes.
Templates as proposed here resemble design patterns (ALEXANDER ET AL. (1977)) or conceptual graph schemata (SOWA (1999)). The former concentrate on capturing proven knowledge, which corresponds to the notion of usualness. The latter are used as feasible, not rigid structures (called conceptual graph types). In a similar manner, our graph templates are instantiated in a group, but later structural checks report at most warnings about deteriorating usages.

3 Implementation

We have implemented a prototype `rwe` that uses the graph database GRAS3 (BAUMANN (1999)) directly and offers general context-sensitive editing, but also specialized commands for argumentation structures. The general edit commands make `rwe` adopt to any given graph type just like generic SGML/XML editors do to a given DTD, but on the more expressive level described here. Through automatic context creation the graph always conforms to the required structure.
Current development focuses on the CHASID (Consistent High-level Authoring and Studying with Integrated Documents) prototype, which is implemented as an executable system of PROGRES graph transformations that may be invoked interactively. Content editing takes place in external applications. Here we extended the automatic context creation to templates offering optional context.

4 Conclusion and Plans

Graph types provide a formalism to describe structure definitions more accurately, but will have to be adopted before they can readily be used for publishing purposes. One promising approach is to leave hierarchical core parts of the document in XML documents and connect it to a content graph. Graph-aware tools can use it to offer further functionality. The content graph can be dumped with loss of readability, but not information, into the XML document.

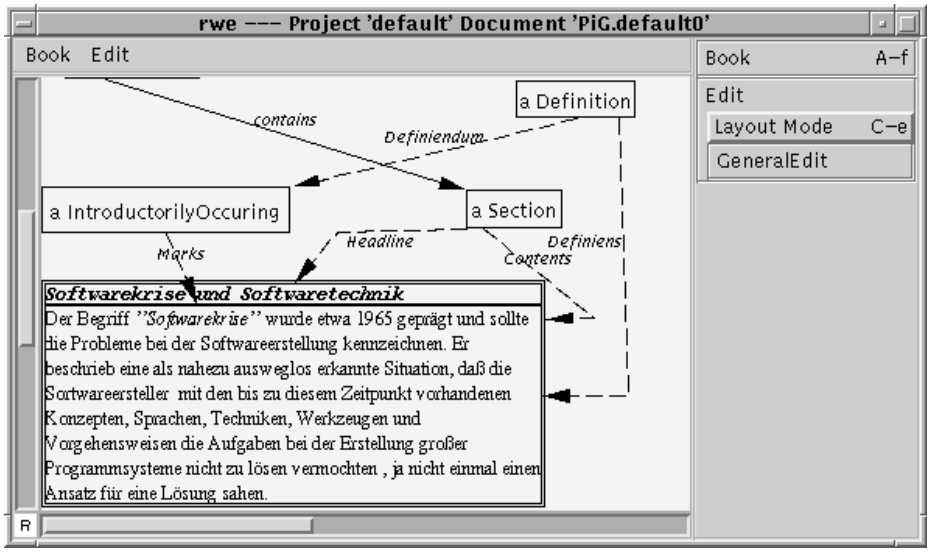

Figure 6. Screen-shot of Reader Writer Environment `rwe`.

Structural templates are an easy-to-understand technique to provide access to complex structure definitions. As no programming knowledge is required for this kind of parameterization, it is a lightweight technology for on-site parameterization of editing applications.

References

ALEXANDER, C., S. ISHIKAWA, M. SILVERSTEIN, M. JACOBSON, I. FIKSDAHL-KING, AND S. ANGEL, *A Pattern Language*. Oxford University Press, New York, 1977. 2.2

BAUMANN, R., *Ein Datenbankmanagementsystem für verteilte, integrierte Software-Entwicklungsumgebungen (A Database management system for distributed, integrated software development environments)*. Ph.D. thesis, RWTH Aachen, Wissenschaftsverlag Mainz, Aachen, 1999. 3

BEHLE, A., F. GATZEMEIER, O. MEYER, AND M. NAGL, Feingranulare Konsistenzsicherung multimedialer Dokumentkomplexe (Ensuring fine-grained consistency of multimedia document compounds). In *Abschlußberichte des Verbundprojekts „Virtuelle Wissensfabrik“ (Final reports of the joint project “Virtual knowledge factory”)*, to appear as a book, 1999. 2

GATZEMEIER, F. AND O. MEYER, Improving the Publication Chain through High-Level Authoring Support. In M. Nagl, A. Schürr, and M. Münch (editors), *Applications of Graph Transformation with Industrial Relevance (AGTIVE)*, volume 1779 of *LNCS*, Springer, Heidelberg, 2000. 2

RATIONAL, *Using Rational Rose 4.0*. Rational Software Corporation, Santa Clara, CA, 1996. 2

RODGERS, P. AND P. KING, A Graph Rewriting Visual Language for Database Programming. *The Journal of Visual Languages and Computing*, 8(6): 641–674, 1997. 2

RUMBAUGH, J., I. JACOBSON, AND G. BOOCH, *The Unified Modeling Language Reference Manual*. Object Technology Series, Addison-Wesley, Reading, MA, 1999, ISBN 0-201-30998-X. 2

SCHÜRR, A., *Operationelles Spezifizieren mit programmierten Graphersetzungssystemen*. Ph.D. thesis, RWTH Aachen, Deutscher Universitätsverlag, Wiesbaden, 1991. 2, 2

SOWA, J., Conceptual graphs: Draft proposed american national standard. In W. Cyre and W. Tepfenhart (editors), *Conceptual Structures: Standards and Practices*, volume 1640 of *LNAI/LNCS*, 1–65, Springer, Heidelberg, 1999, ISBN 3-540-66223-5. 2.2

WALSH, N. AND L. MUELLNER, *DocBook: The Definitive Guide*. O'Reilly, 1999, ISBN 1-56592-580-7. 1

Biographies

Felix Gatzemeier has received his M. Sc. in computer science from the RWTH with a thesis on frameworks of editing applications coping with changeable schemata. He is working in the area of structured editing applications at the department III of computer science of the RWTH. His main research interest is the tradeoff between markup cost for the author and markup benefit for publisher and readers. He expects higher-level authoring applications to provide net benefits for authors.

Oliver Meyer received his M. Sc. from the RWTH. In his thesis he developed tools to create and manipulate the semantical structure of scientific texts. In his research at the department III of computer science of the RWTH he used graph rewriting systems to define, check and restore consistency constraints in natural language texts. His current research focuses on the parameterization of structured text to create specialized variants of one source document for different reader groups.