ELECTRONIC PUBLISHING, VOL. 5(4), 209–216 (DECEMBER 1992)

EP-odds and ends

Editing structured documents—problems and solutions

FRED COLE AND HEATHER BROWN

Computing Laboratory
University of Kent at Canterbury
Canterbury, Kent CT2 7NF, England

SUMMARY

Creating structured documents—where every document element belongs to a class—has many well-known advantages. Using generic document styles to define and constrain the hierarchical relationships between the different classes of element also has many advantages, but causes significant problems in interactive editing. The recent *EP-odd* paper on Rita [1] provided new insights into the possibilities and problems of editing structured documents. This 'EP-odds and ends' sketches some additional problems and suggests alternative solutions based on the idea of *fall-back classes*.

KEY WORDS Structured documents Editing Document classes

1 INTRODUCTION

The advantages of structured documents and of generic document styles are widely recognized [2]. The principles are enshrined in the international standards SGML [3] and ODA [4] and there is a clear move towards structure in many experimental [5] and production [6] document editors. However, the recent *EP-odd* paper on Rita ('Rita—an editor and user interface for manipulating structured documents' [1]) breaks new ground by presenting methods to enforce document structure and provide users with help in interactive structure editing.

The additional power of structured documents creates two main problem areas for editors

- internal structure problems:
  - — validating the structure during document creation and editing
  - — dealing with incomplete and temporarily incorrect structures
  - — identifying allowable structure edits
- user-interface problems:
  - — displaying incomplete documents
  - — providing user help for structure editing

CCC 0894–3982/92/040209–08

Received 10 November 1992
Revised 10 December 1992

The structure problems are significant, but can largely be solved by the application of well-known parsing techniques [7]. The user interface problems, as might be expected, are less clear-cut. Without simple visual help the structure becomes a hindrance rather than a help.

In addition, it is important for users to be able perform simple cut-and-paste edits without unnecessary structure problems. Providing sensible and flexible mechanisms for cut-and-paste in a structured document is a challenge for both system developers and those who design document styles.

In 1990 we proposed some methods for dealing with these problems [8]. Our methods for dealing with the structure problems were based on context-free grammars and DFAs and were very similar to (though less complete than) the methods used in Rita. On the user interface side, we also suggested presenting the user with menus of allowed edits, based on a similar analysis of the DFA controlling the document constituents. In addition, however, we proposed some methods for making structure editing simpler for the user via the use of *fall-back classes*.

The following sections outline the main points of our proposals—concentrating on the places where they differ from those in Rita—and attempt to show how they might add to the functionality provided by Rita.

2 INCOMPLETE DOCUMENTS

Structured document editors are particularly useful for documents that must conform to a company or publisher's standard. The document style can be used to ensure that essential items are all present in their correct places as well as imposing more general structures of the section/subsection type. For complex styles the best method of presenting such a structured document to a user is not clear. There are two main problems

- incomplete documents
- alternative/optional items

Some structured document editors adopt the approach of providing an outline or skeleton document that the user can manipulate by filling in empty place-holders. This works well for simple styles, but becomes tricky for complex styles with many different possibilities. Our proposals recommended that the editor should generate the 'minimum complete document' according to simple rules about selecting the first of several alternative document elements. The user's view would initially show place-holders for the elements in this minimum document. Refinements for the user interface could include indications of alternatives and of optional and repeatable elements. For complex styles there might be some restriction on the number of levels of empty elements shown in this way.

Using this method, the document is initially correct and complete. Assuming users can then only makes edits that conform to the structure rules, they will always see a correct document with clear indications of empty areas. User interface refinements, as noted above, can make many allowable edits permanently visible without the need for separate menus.

A problem with this approach is how to help users with major edits where the document becomes invalid or incomplete unless several items are deleted or replaced simultaneously. Using one of the examples from the Rita paper, if an object class X has the following allowable sub-objects

```
X = a+ ( b d c | c d )
```

a user could have difficulty in finding a series of valid edits to change from `a b d c` to `a c d` unless the user interface had some way of indicating that `b d c` and `c d` were alternative groups and of allowing the groups to be manipulated as a whole. (Even then, the document might need to be temporarily incorrect during the edit to prevent the user having to enter the contents of `c` and `d` again.)

Rita gets round these problems very neatly by adopting the interesting idea of 'subsequence incompleteness'. A document is subsequence-incomplete if its objects form a subsequence of some complete legal sequence of objects. Instead of providing skeleton documents with place-holders, Rita supports operations on subsequence-incomplete documents. Clearly, deletion is never a problem for a subsequence-incomplete document, and valid insertions can be given by separate menus. This approach is particularly good for preventing the structure getting in the way (as it can for the example above), but provides little immediate visual help, relying instead on the user calling up separate menus of valid edits.

3 CHANGING CLASSES AND CUT-AND-PASTE

There are many good reasons why a user may wish to change the class of an object during editing. Major rearrangements in the document (making chapters or sections into appendices), minor alterations in presentation (turning lists into paragraphs), and cut-and-paste shuffling may all involve changes of class.

It is relatively simple to provide menus of allowable class changes because there are generally only two main conditions that need to be checked

- the changed object and its siblings must form a valid set of subordinates for their parent
- the children of the object must form a valid set of subordinates for the new class

(The above conditions are easy to check for many structured documents, including ODA documents, because only the parent needs to be inspected to decide the validity of any set of subordinates. In other cases, such as SGML documents with inclusions and exclusions, all ancestors need to be checked.)

For more complex cut-and-paste operations on subtrees of the document structure, it is relatively easy to check whether a move is valid, but the best action to take when it is invalid is far from obvious. Users may be understandably annoyed at having to specify a tortuous set of change class, insert, and delete operations when they simply want to move a chunk of document from one place to another.

Rita's solution to this problem is to provide a 'patch area' where structure restrictions are relaxed. Anything deleted from the document is automatically copied to the patch area, and the user can subsequently copy from the patch area to anywhere in the document. If the copy is not valid (for the subsequence-incomplete document), the user is directed to the offending object in the patch area. Editing without any class restrictions can take place in the patch area, so it is up to the user to know the correct edits to enable the paste to take place.

The one place where our proposals went significantly beyond the functionality provided by Rita was in trying to automate changes of class during this type of cut-and-paste editing via the use of fall-back classes as described in the next section.

4 DOCUMENT STYLES AND FALL-BACK CLASSES

There is a significant conflict between the additional functionality provided by structure and the difficulties experienced in editing structured documents. Detailed structure is useful for information retrieval, providing different document views, allowing active or processable document content, and many other purposes. However, users will only accept structure if its benefits are significantly greater than its costs. Unless document creation and editing is made easy they will avoid structured documents altogether—or use the simplest structures available.

We believe that good document styles and better methods of helping with cut-and-paste editing are crucial factors in the acceptance of structured documents. Clearly the normal common sense rules about simplicity and consistency apply to document style design, but there may also be a case for asking the designer of the document style to think about ease of editing as well as about the overall document structure and the quality of the final result.

One possible way in which designers could provide this help is by specifying fall-back classes to be used during cut-and-paste editing. Thus, in addition to defining the various object classes and their allowed subordinates, the designer might also be asked to specify a (possibly empty) list of fall-back classes for each different object class used. When a cut-and-paste edit fails because of class problems, the editor then automatically tries changing the classes of objects to one of their fall-back classes. Such changes might cascade down the subtree being moved and might also apply to the siblings of the root of the subtree.

Clearly there is a need to be able to forbid such changes and/or to warn the user when they occur. One possible scheme is to define silent fall-backs (which may happen at any time without warning) and notified fall-backs (which trigger some kind of warning).

A simple example of the use of such fall-back classes could come from different types of paragraph. There is a good case for insisting on simple paragraphs (without embedded footnotes or equations, say) in parts of a document like its abstract, while allowing more complex paragraphs elsewhere. However, a user wishing to copy a paragraph from the abstract into the body of the document would be justifiably annoyed to have the edit declared invalid because the classes were different.

An (oversimplified) solution to this problem is as follows. A simple paragraph (`spar`) consists of ordinary text (`text`) and emphasized text (`em`), while a normal paragraph (`par`) may also contain footnotes (`fn`) and equations (`eq`), so we have

```
spar  =  ( text | em )+
par   =  ( text | em | fn | eq )+
```

and fall-back classes are defined as follows

```
spar  >  par    (silent)
par   >  spar   (silent)
fn    >  em     (notified)
eq    >  em     (notified)
```

This allows simple paragraphs to be changed automatically to ordinary paragraphs during a move. No notification is given because the subordinates will always be correct and there are no further implications. It also allows normal paragraphs to be changed automatically to simple paragraphs but, if any embedded footnotes or equations are involved, these are changed into emphasized text and the user is notified of potential problems.

Another example where automatic class changes could be useful is for a document structure allowing several levels of nested sections—sections, subsections, subsubsections, and so on. If these items were allowed to fall-back to the level above or below (subsections to fall-back to sections or subsubsections, for example), it would simplify the fairly common editing operation of inserting or deleting a section level. The details of such fall-backs, and decisions on whether they are silent or notified could become a significant part of the document style design. It is worth emphasizing that the fall-back classes have no effect on the final form of a document, they are simply a means of easing the editing problems.

Implementing fall-backs for the structure checking operations described for Rita would not create a major overhead. Reference [8] contains an extended algorithm for checking the validity of a class change for an object in an ODA document and giving details of any consequent automatic fall-back class changes needed for its siblings and subordinates. A shortened version of this is given in Appendix A.

5 CONCLUSION

Clearly there are many different levels at which fall-back classes could be used. Our original proposals envisaged the designer of the document style providing a single list of fall-backs for each separate object class, but there are many potential problems and extensions that need to be investigated, including

- might different fall-backs be needed in different contexts (for example, might a footnote need a different fall-back depending on whether it is in a chapter or an appendix)?
- could fall-backs be used to help with cut-and-paste editing between documents with incompatible styles?

The ideas described above were developed during a collaborative project to build a documentation system for software engineers called Fortune [9]. Fortune was specifically designed to handle structured documents (based on the ODA model). The structure-checking methods based on the use of DFAs were implemented in the Fortune system and menus of valid edits were provided for users. Fall-back classes for cut-and-paste editing were only tested in a separate (non-interactive) prototype, but they appeared to offer a novel addition to the functionality currently provided in Rita and other advanced systems.

ACKNOWLEDGEMENTS

We would like to thank the SERC for supporting the Kent involvement in the Fortune project, and our colleagues in the Fortune consortium, especially Douglas Mullin, for many useful discussions on document structures.

REFERENCES

1. D. D. Cowan, E. W. Mackie, G. M. Pianosi, and G. de V. Smit, 'Rita—an editor and user interface for manipulating structured documents', *Electronic Publishing: Origination, Dissemination, and Design*, **4**(3), 125–150, (1991).
2. J. André, R. Furuta, and V. Quint, *Structured Documents*, Cambridge University Press, 1989.
3. International Standards Organization, *Text and Office Systems—Standard Generalized Markup Language*, October 1986. ISO 8879.

4. International Standards Organization, *Text and Office Systems—Office Document Architecture (ODA) and Interchange Format*, 1989. ISO 8613.
5. D. D. Chamberlin, 'An adaptation of dataflow methods for WYSIWYG document processing', in *Proceedings of the ACM Conference on Document Processing Systems*, 101–109, ACM, New York, 1988.
6. P. M. English, E. S. Jacobson, R. A. Morris, K. B. Mundy, S. D. Pelletier, T. A. Polluci, and H. D. Scarbro, 'An extensible, object-oriented system for active documents', in *EP90—Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography*, Cambridge University Press, 1990.
7. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.
8. F. C. Cole and H. Brown, 'Editing a structured document with classes', Report No. 73, University of Kent Computing Laboratory, (1990). Previously available as Alvey Deliverable ALV/PRJ/SE/050 K8, 1988.
9. D. Mullin, *FORTUNE—A Documentation Support System for Software Engineers*, Springer Verlag, 1988. Lecture Notes in Computer Science 289.

APPENDIX A: CLASS CHANGE WITH FALL-BACK CLASSES

This appendix provides a brief summary of the proposed algorithm for testing possible class changes.

Figure 1 shows a subtree of a document, with each constituent labelled by its class. The arrow indicates the user has requested that the class of C be changed to P.

The lists below show the allowed sub-objects and fall-back classes defined for the objects in the document subtree (composite object classes are given by a capital letter and leaf object classes by a small letter).

```
Fall-backs     Allowed sub-objects
C  >  X        A  =  (b C D) | (b X D) | (b P Q) | (b P U)
D  >  Q U      C  =  e  F  g
F  >  R        D  =  h
g  >  s        F  =  i  j
h  >  t v      P  =  e  R  s
               Q  =  t
               R  =  i  j
               U  =  h  |  t
               X  =  e  F  s
```

The algorithm needs to check the sub-objects of A for validity after the change. Figure 2 shows a possible solution found using the fall-back classes—where the boxes indicate the objects with changed classes—and Figure 3 shows the temporary structures that might be used during the checking.

The algorithm begins by creating a table of the sub-objects of A, substituting the new class P instead of the original C, and giving all the variations allowed by fall-back classes (in this case D can become Q or U). If any of these variations are valid sets of sub-objects for A, the algorithm then effectively goes down the subtree rooted at A repeating the exercise for all composite objects until it finds a valid result—or fails to come up with any valid set of changes. (In practice the algorithm would probably only create tables until it found the first valid answer. In this example it would thus avoid the need to check the last line of the table for A or produce the table for U.)

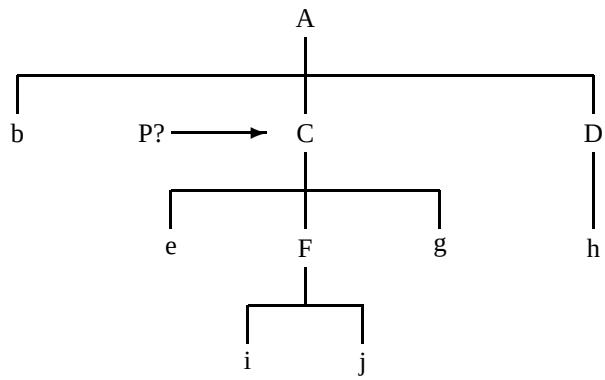

Figure 1. Original document

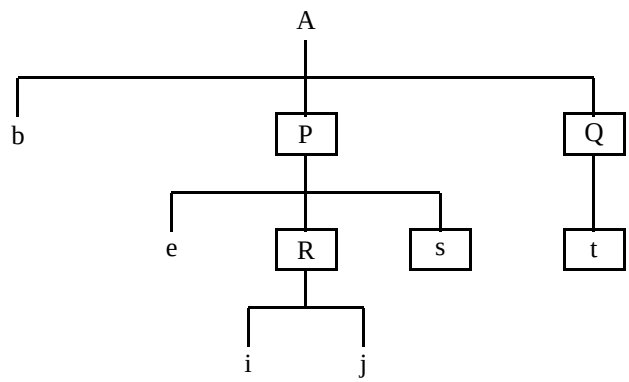

Figure 2. Document after automatic class changes

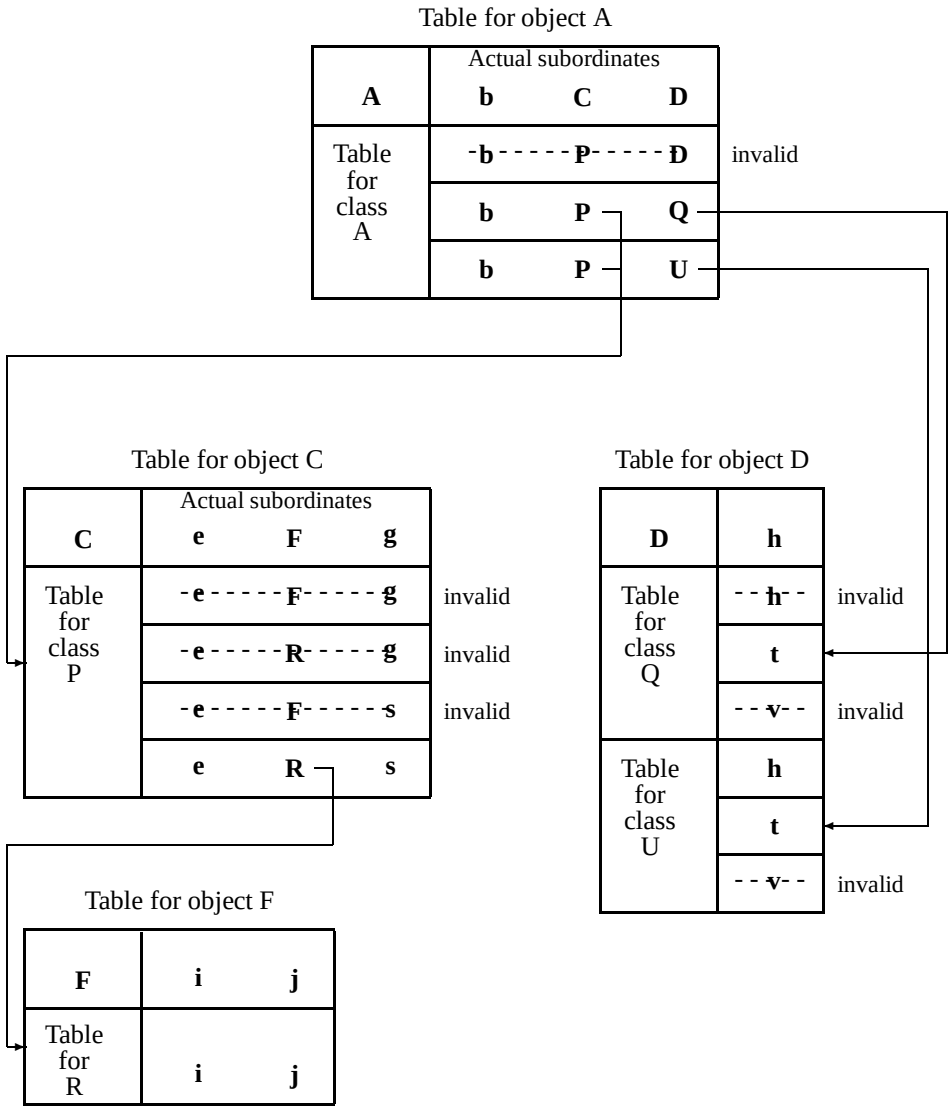

Figure 3. Temporary tables used in fall-back class algorithm