ELECTRONIC PUBLISHING, VOL. 4(3), 125–150 (SEPTEMBER 1991)

Rita—an editor and user interface for manipulating structured documents

D. D. COWAN, E. W. MACKIE, and G. M. PIANOSI
Department of Computer Science and
Computer Systems Group
University of Waterloo
Waterloo, Ontario
Canada

G. de V. SMIT
Department of Computer Science
University of Cape Town
Cape Town
South Africa

SUMMARY

Structured documents such as those developed for SGML, GML or LaTeX usually contain a combination of text and tags. Since various types of documents require tags with different placement, the creator of a document must learn and retain a large amount of knowledge. Rita consists of an editor and user interface which are controlled by a grammar or description of a document type and its tags, and which guide the user in preparing a document, thus avoiding the problems of tags being used or placed incorrectly. The user interface contains a display which is almost WYSIWYG so that the appearance of the document can be examined while it is being prepared. This paper describes Rita, its user interface and some of its internal structure and algorithms, and relates anecdotal user experiences. Comparisons are also made with other commercial and experimental systems.

KEY WORDS Document manipulation Finite state automata User interfaces Incomplete documents Structured documents Syntax-directed editing Partial documents

INTRODUCTION

Document preparation can be viewed as two distinct activities: creating the content and structure of the document (editing), and generating the document using some specification of its appearance (formatting). Observations about the separation of appearance from structure and content was one of the developments which led to the creation of tagging or markup languages such as LaTeX [1], GML [2–4] and SGML [5]. In markup languages the structure of a document is usually specified by a set of tags (some of which may be implicit), which are embedded in the text. Therefore creating different types of structured documents can be an error-prone task as both the set of tags and their correct placement must be remembered. This tendency to produce incorrectly tagged documents is especially noticeable for the novice or infrequent user, and even presents problems for document processing "professionals" who must remember the "subtleties" of tagging and other similar computing esoterica.

The aim of this paper is to describe Rita [6–9], an editor and user interface for structured documents. Rita was designed for users with a wide range of expertise in document processing and has the primary goal that members of this broad class can create and edit tagged documents which are structurally correct as they are initially entered into the computer. Thus many different types of users could have the benefits of markup languages without the trouble of learning and remembering most of the details of tag types

0894–3982/91/030125–26$18.00
©1991 by John Wiley & Sons, Ltd.
Received 14 March 1991
Revised 16 November 1991
© 1998 by University of Nottingham.

and placement. The paper focuses on the details of Rita's dynamic user interface and the algorithms required for its implementation. There are several systems available from both software houses and research laboratories which have design objectives similar to the ones proposed for Rita. However the authors have not been able to locate any publications which describe both the user view and the methods used for implementation.

A tool such as Rita is invaluable as it minimizes errors, thus significantly reducing the use of computing resources, and making the creators of documents more productive. Rita has been tested successfully with different groups of users including word-processing staff, programmers, and secretarial staff with minimal computer skills. Experiences with Rita in these and other contexts have led to the conclusion that use of such a tool almost always ensures that a document will be structurally correct as it is entered into the computer. Similar experiences have been reported in References [10–15].

A number of other structured document editors have been proposed and/or developed. These include Etude [16] and its derivative the Interleaf Publishing System (IPS) [17], PEN [18], W [19], Grif [20], Quill [12], Author/Editor [21], WRITE-IT SGML Editor [22], and systems by Kimura [23], van Huu [24], Coray *et al*. [25], and Furuta [26].

The version of Rita being described in this paper is designed to operate with GML or SGML tag sets and produce output which can be processed by several different formatting systems including LaTeX [1], DCF [2], Waterloo Script [3], and WATCOM GML [4].

Rita was developed with certain views about the computing environment, the users, and the set of documents that could be handled. These three design parameters are described in the remainder of this section.

The layout or style sheet for a document is not really needed until the document is being formatted. However, while text is being prepared, it is desirable to view the approximate appearance of the document, and at least confirm that no serious structural deficiencies exist. Thus Rita was designed to accept a layout or style sheet for the terminal or computer screen being used, and can adapt itself to a particular computing and viewing environment. Documents can be prepared and viewed on displays ranging from character or simple graphics screens to high-resolution bit-mapped displays.

Many people who want to prepare documents are not necessarily interested in the "secrets" of computing; they want to produce a document which appears as if it was produced by an expert, and yet perform this task with minimum fuss and bother. This type of user was the model that guided the development of Rita. Such users were further subdivided into novices, casual users and experienced non-technical users.

Novices have minimal computing skills; they want to learn as little as possible, and yet start producing perfectly formatted documents. Casual users may or may not have substantial computer skills, but they produce documents infrequently, and hence tend to forget many of the techniques of document preparation between uses of the computer. Experienced non-technical users are those individuals who might belong to a word-processing group, but have little knowledge of markup languages. Of course as the system was developed, we tried not to forget the expert; often a feature was implemented remembering all members of this broad spectrum of users.

Different types of documents have different structures, for example, books are different from legal documents such as contracts. Hence certain tags which define entities such as titles and chapter headings might be "illegal" in a given document type or context. Rita was designed to "understand" the structure of classes of documents so that Rita can be tailored to aid the user in preparing any type of document. Rita accepts different document

types which are defined as context-free grammars consisting of a set of rules, where each rule defines a regular language. These rules can be defined so that a large document can be divided into segments such as chapters that can be merged later by the formatter to produce the complete document.

As a document is created, both its content and structure are being modified, thus editing an already-existing structure may temporarily make the document structure "incorrect" as it violates the grammar defined by the particular document type. Flexibility in editing is important; correct document structure must not be rigidly enforced at all times.

SYSTEM OVERVIEW

Rita provides the user with the following features:

- an ability to view the document as it is being prepared, in a what-you-see-is-almost-what-you-get editing environment
- a method of constraining the document to ensure that it conforms to some given document type such as a technical paper, manual, contract, letter or memorandum
- a system with retargetable output, where documents created on the system may be directed to different batch formatters and to different interactive displays
- a method of editing documents even though the editing process may make the documents temporarily incorrect

The system is illustrated in Figure 1. Rita accepts input from two different sources, a document class database and the user, and produces a specific document as output. The document class database contains two components, descriptions of the structure of the document or document grammar, and descriptions of the appearance of each structural component of the document for different media. The user supplies the text of the document and chooses tags from a menu which is constrained by the document grammar so that only structurally correct documents are entered. The output is a tagged document in standard ASCII format which can then be passed to a formatting system.

Documents such as papers, memoranda, and letters have different syntactic structures and are regarded as structured objects, with each type of document defining a document class. Thus a particular document is an instance of a specific document class. The database of document class descriptions provides information on the structure to which a document of a given class must conform, as well as how concrete representations of such a document must be generated. These class descriptions or document types are normally created by a document administrator or expert user. A compiler or class generator is provided with Rita [27] so new document types can be defined or the existing types can be extended to support additional tag definitions and concrete representations.

The structural description, or *syntax* of a document class consists of rules each of which defines a regular language, see, for example, Reference [28], and the complete set of rules is an extended context-free grammar. An example of a grammar for a simple document class is illustrated in Figure 2. Each rule defines an object class where composite object classes such as `section` are composed of other object classes, while basic object classes such as `TEXT` cannot be decomposed further and therefore have no constituents. Each basic object has a (possibly empty) string of non-structural data associated with it, referred to as the content of the object. The content of a composite object is the content of its constituents.

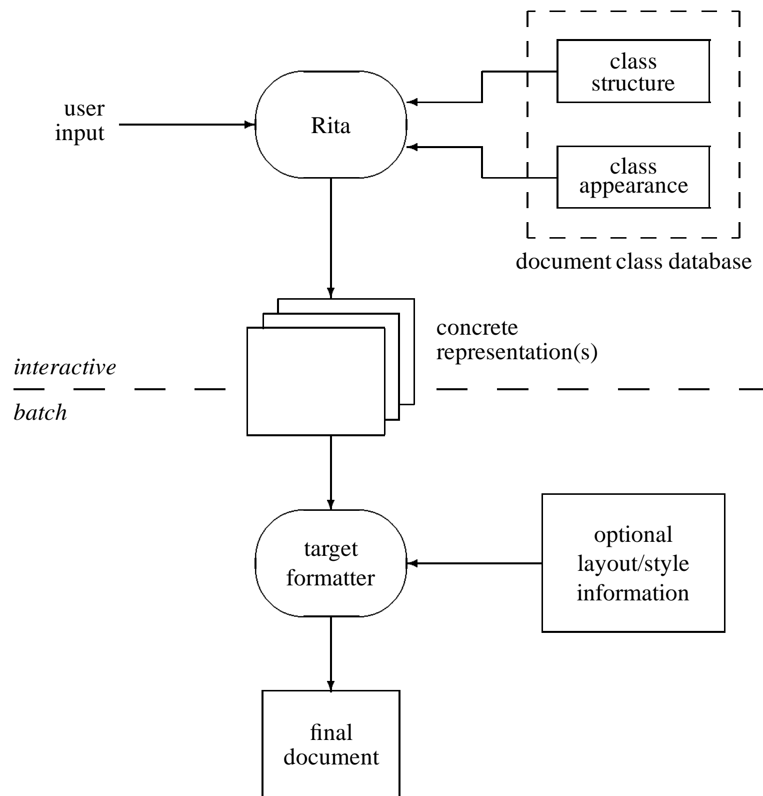

Figure 1. The Rita system

```
TechDoc = titlePage body appendices?;

titlePage = title author date abstract?;
title = TEXT;
author = TEXT;
date = TEXT?;
abstract = (paragraph | ordList | unordList)+;

body = section+;
section = heading (paragraph | ordList | unordList)* section*;
heading = TEXT;
paragraph  = TEXT+;
ordList = listItem+;
unordList = listItem+;
listItem = paragraph (paragraph | ordList | unordList)*;

appendices = section+;
TEXT;
```

Figure 2. Structure definition for a simple technical document

Although different composite objects may have similar or even identical components, their appearance on the screen or in print may not be the same. For example, the document conforming to the grammar of Figure 2 contains components `ordList` and `unordList` which both have identical definitions. However, the concrete representations of these two objects may differ. When rendering an `ordList` the system could annotate each `listItem` with a positive integer, while the display of `unordList` could show each `listItem` preceded by a bullet (•).

Objects may have optional *attributes*. Attributes are characteristics of objects other than their composition and content and may include references to other objects. Reference attributes allow non-hierarchical relationships such as cross-references to be implemented.

The concrete representation, or *semantics* of an object is defined by sequences of instructions that are executed when the object is encountered. Different sets of instructions can be used to produce different representations of the same document. One of these representations is generated incrementally and is used to display the document on the screen during the interactive editing of the document. Other representations may be used as input to existing batch formatters such as TEX [29], LATEX [1], GML [2–4], or troff [30] for direct viewing or printing by the user. More detail about specifying and generating concrete representations is provided in the section "Document semantics."

THE USER'S VIEW OF RITA

Rita was designed for users ranging from novice to expert and presents features which should appeal to this broad group. These features include the dynamic user interface, and the ability to edit both text and structure during document creation or revision. The designers also tried to apply the law of "least astonishment" [31], even accidental key depressions produce results from which the user can easily recover. Many of these aspects of Rita are described in the following subsections.

User interface

Many of the user interface features of Rita can be seen by examining Figure 3 which shows the Motif [32] version. The application window is divided into three smaller areas: a menu bar at the top of the screen, a text window and a structure window. The text and structure windows are in the middle of the application window, the text window is to the right of the vertical line and the structure window is to the left. In current versions of Rita the text window provides a what-you-see-is-almost-what-you-get (WYSIAWYG) display of the text contained in the document. The WYSIAWYG approach was to allow support of display technologies ranging from mono-font character to flexible bitmap displays. Although the screen does not exactly match the final appearance of the document, the benefits to a wide range of users are worth the relatively small degradation in appearance. Depending on the display technology, Rita supports fonts, and presentation of lists, numbered section headings and footnotes. Verbose mode allows footnotes to be exposed or hidden so the appearance of the formatted text is closer to WYSIWYG. The labels representing citations, or references to figures or sections, are displayed in-line.

The Rita user positions the cursor in the text window and starts typing. When the cursor is in the text window, word-wrap and many text-editing features found in word processors such as Microsoft Word [33] are available. The structure window to the left of the vertical

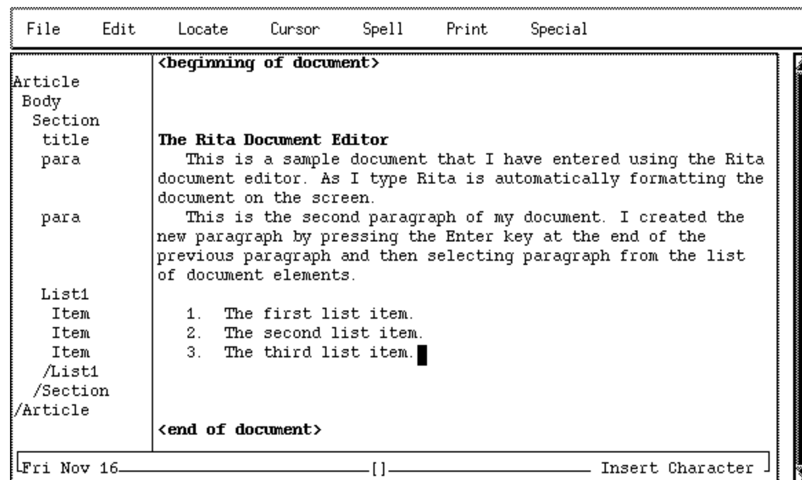

Figure 3. The user interface for Rita—the text window

line, contains a fairly readable version of the SGML tags. Some of the tags such as `Item` in Figure 3 are indented to indicate they are part of an enclosing structure indicated by `List1` and `/List1` representing an ordered list.

The menu bar across the top of the screen has a number of words or descriptors, each of which represents a pull-down menu. Each menu contains commands relevant to the specific descriptor. Thus there is no need for the user to memorize commands, although many of the commands can be activated by power users with function keys. The file commands are activated from the "File" menu and allow the storage and retrieval of text, and file-browsing. Context-sensitive help is available for all topics including the tags. The help text can be extended using a simple editor.

All operations except typing text, can be done with either mouse or keyboard; this includes operating menus, positioning the cursor, marking text and scrolling documents.

Text editing

In text-editing mode the cursor can be placed anywhere within the text window using either the keyboard or mouse, and text is automatically reformatted as you type. The cursor can be configured to be one of four sizes (character, word, sentence and block) using either the keyboard or mouse, and then can be extended to be any multiple of those four sizes. Once the text has been marked with the cursor, operations such as "cut/copy" and "paste" can be performed on the marked text.

Commands to both search for text and replace it with substitute text are available. These commands can be case sensitive or insensitive and can replace items globally or one-at-a-time. The search command can also be used to search for tags in the structure window.

There are facilities to check the spelling of words in the text window which are supported by a 96 000 word dictionary. The speller suggests corrections, and also allows the user to create multiple-user dictionaries which can be augmented during the checking phase. User dictionaries can be saved for later use or be discarded after a single session.

Structure editing

Rita so far appears to have the same functional characteristics as a comprehensive word processor. The real power of the system becomes apparent when the user is allowed to insert, delete and change the structure of the document. This structure is usually modified when the cursor is in the left-hand or structure window, although there are circumstances in which the structure can be changed while the cursor is in the text window. The cursor is shown in the structure window in Figure 4 and is placed there either with the mouse or the "Enter" key.

Rita is initialized with a standard document class, but new document class descriptions can be loaded by activating a command from the "File" menu. Each different document class or type becomes a "template" which constrains Rita and the user to create only document structures or elements which are syntactically legal for that document type.

When the user wishes to enter a new document element, the cursor is placed on a tag in the structure window and a structure menu appears at the bottom of the screen as shown in Figure 4. Only elements which can be legally inserted before the cursor are shown in the structure menu. An item in the structure menu can be chosen using either the mouse or a single keystroke. Hence it is not necessary to know the tags for a specific document type. Extensive help can be made available to describe the tags.

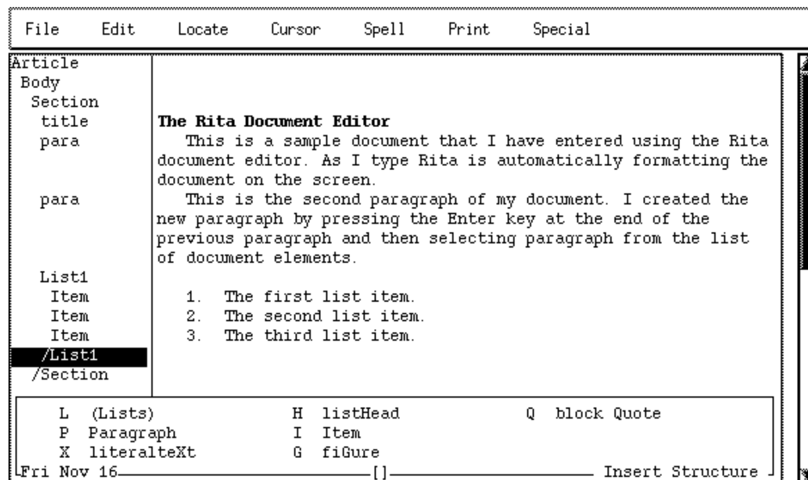

Figure 4. The user interface for Rita—the structure window

The structure of a document often changes as it is being produced. For example, the "ordered list" shown in Figure 4 could be changed to other lists such as a "simple list" or an "unordered list." Rita allows the user to transform tags into other forms. To change a tag the user selects the tag in the structure window and then chooses "Transform" from the "Edit" menu. This action causes a "Transform" menu to appear at the bottom of the screen containing the valid transformations. Text can also be marked with the cursor and then be transformed to create such entities as highlighted phrases.

Structures can be marked with the cursor, and then the structures and their accompanying text can be cut and pasted, or they can be written to a file and merged into another document.

To mark a structure the cursor is placed on any structure tag, and then extended to mark as much of the document structure as desired. When a structure is marked in this way, the cursor is automatically extended to include subsidiary parts such as the ordered list and its items shown in Figure 4.

Presentation of document elements

Three possible choices were examined for ensuring that the user of Rita would add only legal document elements: the use of syntax-checking as seen in syntax-directed programming editors such as the Cornell Program Synthesizer [34], place-holders as used by Kimura [23], and menus.

The syntax-checking approach was rejected because it required the user to have some knowledge of the tagging or markup language, and this approach was not consistent with the type of user to be served. Place-holders create a form of menu or button in the middle of the document and were excluded, because generating place-holders for document types with moderately complex grammars such as

```
x= (a b c d) | (x y z)
```

could cause visual confusion. However, simple place-holders can be useful as a reminder to the user that certain vital documents elements, such as an address in a letter, is missing.

Menus, as shown in Figure 4, were chosen because they could be placed in a position on the screen which does not conflict with the work area. Also menus are only active when the cursor is in the structure window or when a transformation is in progress. When text is being entered the menus are absent. Of course providing menus which only contain syntactically correct tags means that the menus have to be computed dynamically as a function of cursor position in the structure or text window. This need for dynamic menu construction led to the examination of the menu-construction algorithms described later under “Menu construction.”

Editing complex documents

Documents can be adjudged simple or complex on a number of criteria including: the set of tags that is used, possible attributes or modifiers for the tags, and the use of low-level formatting commands. Simple documents usually use a subset of the available tags for a document type and do not use attributes or formatting commands.

Most documents, whether simple or complex, use only basic structures for most of the content. If all the legal tags were included in the structure menu, the novice might find the number of choices presented in the menu confusing. For these reasons, the extra “complex” tags are hidden from the user until they are needed. Thus the structure editor provides both a simple (terse) and complex (verbose) mode, and the user can toggle between the two. In verbose mode all the valid tags appear in the structure menu. Verbose mode also explodes the material appearing in both the structure and the text windows. Thus in verbose mode, tags which are attached to entities such as single words appear in the structure window and their corresponding text elements are on separate lines in the text window. Tags attached to “small items” such as single words or phrases are hidden in terse mode, so as not to disrupt the flow of text.

Attributes are modifiers for many of the tags which are in the tag set. Rita allows the insertion and modification of attributes for any tag which accepts them.

There are times when the tag set is not adequate and low-level formatting commands must be introduced into the document. A document grammar can contain a "process specific" tag which allows the insertion of such formatting commands.

The document grammar uses a "user embed" tag to allow Rita to accept a document which may not completely conform to the current grammar. The "user embed" tag brackets structural components which do not parse correctly when a previously stored document, often produced using a conventional editor, is being read into Rita. The text associated with this illegal structure can then be incorporated into a correct structure using the editing commands in Rita. This facility will often allow a transformation of an "incorrect" document into a "correct" one in a small number of editing steps. Rita does support the inclusion of tables, equations and structured graphics, but the editing of these objects is not completely implemented. Current work is proceeding on improving these editing capabilities by allowing the user to access the tool appropriate to the task from within the document.

IMPLEMENTING THE USER'S VIEW

Rita contains several algorithms and data structures to manage both the internal representation of the text and structure, and the external appearance. This section of the paper makes specific reference to these algorithms and data structures, their effect on performance, on the user's view of the document and the user's ability to manipulate the document.

When the Rita editor is activated it is automatically provided with a document grammar which defines the structure of a class of documents. The Rita class compilers or generators accept Rita's own document class descriptions and those defined in an SGML Document Type Definition (DTD).

Accompanying the document grammar is a set of rules defining the appearance of this class of documents for the display device being used[1]. The user then creates a document which is an instance of that class by selecting structural components and typing in text. When the document is complete it is stored on the disk as ASCII text with embedded GML or SGML tags[2] rather than in some internal format. This both provides storage efficiency and minimizes confusion for the users, since only one representation of the document is stored on the disk. In the case of GML the file of ASCII text and the layout for that document type are then provided as input to the text formatting program. Since there is no corresponding layout language for SGML, a document in Rita with SGML tags is converted to a document which can be used as input to an existing text formatting system such as LaTeX. A few commercial systems do implement a non-standard layout mechanism and an international standard is in the draft stage [35].

The input to Rita can come from two sources, either an entire file that is being read for subsequent editing or modification, or as keystrokes from the keyboard when a user is entering structure and text. Rita supports a virtual memory model and so any length of document can be handled. If a complete file is being read, Rita acts in a manner similar to a document formatter or a batch compiler. When text and structure are entered

[1] A default set of simple display rules is used for SGML DTDs.
[2] The choice of GML or SGML tags is governed by the document grammar which was loaded with Rita.

in the document from the keyboard, Rita must operate in incremental compilation mode, since input from the keyboard can be entered anywhere in the document and performance constraints do not allow the whole document to be parsed each time the structure is modified. The approach of incrementally processing the document and providing immediate feedback through the screen presentation parallels the method described for programming languages in Reference [36].

Representing the user's view of structured documents

A tree is an appropriate data structure to use for referencing both the structure and text of a document. Each node in the tree represents an object class in a document grammar such as the one in Figure 2. Edges in the tree connect the node representing the object class on the left-hand side of a rule with its constituent object classes on the right-hand side. Such a representation is convenient for manipulating the structure, and verifying that a document obeys the syntax rules of the language.

The implementors of an early version of Rita observed that the structure of documents is primarily hierarchical and used only a tree as the underlying data structure, where each node of the tree contained a pointer to a block of text. This choice of data structure directed the implementors toward the creation of an inappropriate interface which was quite confusing to the user. Moving between two structures which were adjacent on the screen often implied a rather complex tree traversal, as these structures might only be related to each other because they shared a common ancestor. Many editing operations which appeared simple and straightforward from the user's perspective, could involve fairly complex keyboard sequences because of the nature of tree-processing operations. Although most users understand that there are rules governing the structure of a tagged document, they tend to view a document as a linear progression of text rather than as a hierarchy of document elements.

To solve this problem an additional data structure, called a field list, was introduced. The field list contains pointers into the text of the document as well as pointers to the appropriate nodes in the document tree. The field list is a sequential linked list which is in the same order as the text on the screen, and maps the internal view of the document as a tree to the linear view of the user.

The field list is the primary data structure for many of the text-manipulation algorithms used in Rita, while the tree is used for the structure-manipulation algorithms. Thus Rita amalgamates the data structures commonly used for manipulating text and programs. A field list with descriptors and pointers into a line list which contains the actual text, is a common way of representing documents for word processors. A tree is a common way of representing program source for compilers.

Presenting document structure to the user

Rita must create menus dynamically and efficiently as the user edits the document structure so that there is minimal perceived delay. The details of menu creation and editing are described in subsequent sections.

Each document class description is a context-free grammar composed of a number of object classes. Each object class is then defined by a finite state automaton (FSA), and the complete set of FSAs provides an equivalent definition of the document class

description [37]. Document class descriptions are compiled by a class generator into a set of tables which can be loaded by Rita. These tables consist of the set of transitions for each of the FSAs and formatting programs for the initial and final states and each transition of the FSAs. Rita makes use of this set of tables to support both editing of the document structure and presentation of the document to the user.

Rita must be responsive to user requests for modification of the document structure, and this is achieved through incremental compilation techniques. Each node of the document tree contains the state of its parent's FSA, and this allows an automaton to be restarted at any point in the document without processing all the predecessors of that specific object. This incremental capability is used in a number of different ways and is the key to providing fast response.

Two steps are required at a specific point in the document structure to create a menu which contains only legal document elements. First, a list of all of the potential document elements is created from the transitions of the FSA. Second, each of them is then tested to see if its inclusion causes the automaton to achieve a defined state and thus represents a valid insertion at this point in the document. This algorithm is described more fully in the section "Menu construction."

Once a document element has been selected from the menu, that element is inserted in the document tree, and this action causes the parent FSA to move to a new state. As mentioned earlier in this section this state is stored in the tree node representing the new element. Insertion of this new element may also cause states stored in the nodes representing subsequent siblings to be changed.

The FSAs are used not only to determine which objects can be added at a given location in the document, but also to check the correctness of composite objects and the legality of all structural editing operations. The correctness of any composite object in a document is checked by executing the automaton for that object, using the given object's children as input symbols. If all the input symbols have been "read" and the automaton is in a final state, the object is correct.

The FSAs are also used in incrementally formatting the document on the screen. The field list is used to record the restart points in the document; a restart point is any position where a new line is forced in the formatting programs. On every keystroke the extent of reformatting is limited; its maximum range is from the previous line up to the next restart point. After a change to the structure of the document, the FSAs are executed again. Once the text has been processed by the formatting programs it is passed to a word-wrap routine which performs the final format operations.

Menu construction

The philosophy used in Rita is to provide the user with those options which are consistent with the document grammar so that mistakes are prevented. Thus the creation menu contains only legal document elements and is calculated dynamically from the finite automaton constructed for the object class of the parent of the current object.

Suppose a correct object x of class X currently consists of a sequence of objects $a_1 \ldots a_m$, and that the cursor is currently on a_k, $0 \leq k \leq m$ ($k = 0$ implies the cursor is on the so-called phantom first child, in which case $a_1 \ldots a_k$ is the null sequence). We wish to construct a menu containing the names of all classes of objects that may be inserted immediately after a_k without violating the document class syntax. If b does not violate

the document class syntax, in other words, b is a legal menu item, then the partial object class defined by $a_1 \ldots a_k b a_{k+1} \ldots a_m$ causes the finite automaton to reach some state from which it is possible to reach a final state. Such a state is called a legal state. If b is not a legal menu item then the transition function for the sequence of objects $ba_{k+1} \ldots a_m$ is either undefined, or it leaves the automaton in a state from which no final state can be reached.

More formally let $\delta : (S,\Sigma) \mapsto S$ be the partial transition function of a finite automaton[3] that accepts all members of object class X, where S is the set of states and Σ the FSA's input alphabet. Further we extend δ in the usual way (see, for example, Reference [38]) by writing $\delta(s, a_1 \ldots a_m)$ for

$$\delta(\delta(\ldots \delta(\delta(s,a_1),a_2)\ldots,a_{m-1}),a_m)$$

Assume that the automaton is in state s_k after having read symbol a_k. The set $\mathcal{C}$ of objects b that should appear in the creation menu is then

$$\mathcal{C} = \{\ b \mid \delta(s_k, ba_{k+1} \ldots a_m) \textit{ is a legal state } \}$$

In other words the transition function for the sequence $ba_{k+1} \ldots a_m$ is defined. This set can be calculated in a straightforward manner by executing the automaton starting at state s_k with

$$ba_{k+1} \ldots a_m$$

as input string, using different values for b. The only values for b that have to be considered are those that label transitions from s_k. We will assume that the function *CONSTRUCT_MENU(X, x, k)*, whose parameters correspond to the symbols that we have used, exists and that it returns the creation menu set as just described, namely the set of acceptable bs.

Example

Let X be the class `dL` defined by

```
dL = tH? dH? (t+ d?)+
```

`dL` can be interpreted as a class of definition lists that consists of optional headings (`tH` and `dH`) for the terms (`t`) and term definitions (`d`) in the list. A sample definition list is illustrated in Figure 5, while Figure 6 shows a finite automaton that defines the class.

Let the object x of class X (the class `dL`) be

$$x = tH\ t$$

where tH is the current object (i.e. $k = 1$, and $s_k = 2$). *CONSTRUCT_MENU* returns the set {`dH, t`}, since both $\delta(2,$ "`dH t`") and $\delta(2,$ "`t t`") are legal states. Only `dH` and `t` will be considered since no other transitions are possible from state 2.

Note that if the current state of the automaton is stored with the part of the document that corresponds to the state (e.g. s_k is stored with a_k), it is not necessary to re-start

[3] We have assumed here that the automaton is a deterministic one. This does not have to be the case, however, and a similar approach can be taken for a non-deterministic automaton.

Terms	**Definitions**
term_a	A block of text, called a definition, describing the term(s) listed next to and immediately above it.
term_b$_1$	
term_b$_2$	A group of terms may share a definition.
term_c	Some terms, like the D group below might not have a definition.
term_d$_1$	
term_d$_2$	

Figure 5. A sample definition list

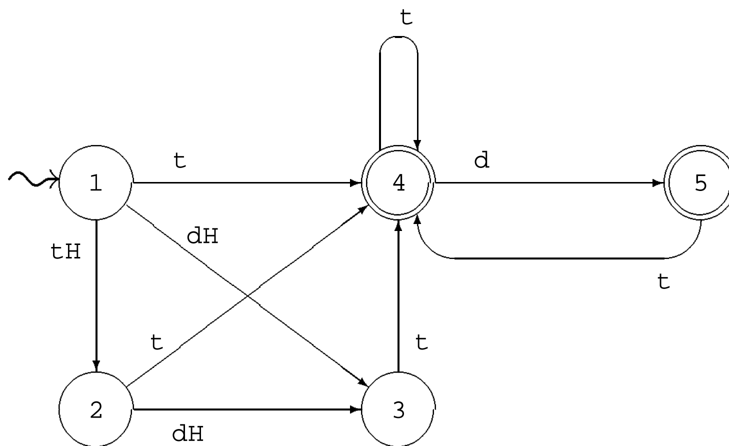

Figure 6. DFA accepting definition lists

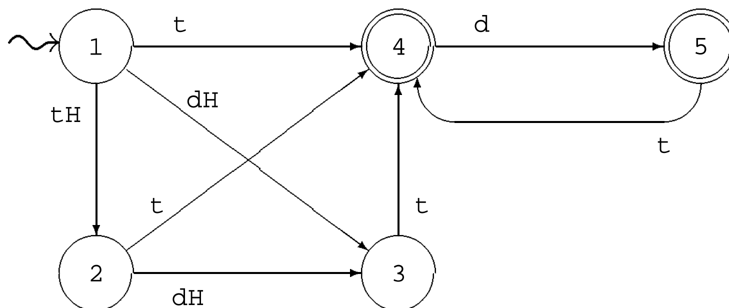

Figure 7. DFA accepting a different kind of definition list

the automaton from its start state every time, but it can be started in state s_k with input $ba_{k+1} \ldots a_m$. Nor is it always necessary to scan all of $ba_{k+1} \ldots a_m$ for each b, since for a given b, scanning can be terminated as soon as an a_j has been read for which the corresponding stored state is the same as the current scan state. It is of course necessary that the state information saved with the document be updated whenever the document structure changes through the addition or deletion of an object.

Note that it can be wrong to include all symbols that label transitions from s_k in the creation menu, in other words $\mathcal{C}' = \{ b \mid \delta(s_k,b) \textit{ is defined} \}$. For example, consider the class

```
dL = tH? dH? t (d t?)*
```

which is a slight modification of the previous example. Figure 7 illustrates a DFA for this class.

Assume the object x is an instance of this class and consists of the sequence

$$x = tH\,t$$

which is identical to the previous example. If $\mathcal{C}'$ is used as the criterion to determine symbols in the creation menu and tH is the current object (i.e. $k = 1$, and $s_k = 2$), then both `dH` and `t` would be in the menu. This is incorrect, however, since the `t` already in x "invalidates" the `t` transition from state 2 (the grammar does not allow adjacent `ts`). In general the symbol that causes the invalidation may be further away from the current object and not necessarily just its successor.

Incomplete documents

The menu construction method described earlier allows a user to create documents which are *tail-incomplete*: required objects which form the "tail" or "right end" of objects in the document could be missing. For example, a specific instance of the document component

```
titlePage = title author date abstract?
```

can appear in the document with only its sub-components `title` or `title` and `author` appearing.

The GML version of Rita uses the menu construction algorithm to produce a tail-incomplete menu for a document component. This menu shows all sub-components up to the first required one. In the example for `titlePage` the object `title` will be in the initial menu, and once `title` has been inserted, `author` and finally `date` will appear. After `date` has been inserted the optional element `abstract?` will be visible in the menu. Of course this description assumes the cursor is positioned correctly in the structure window. Rita would accept this document at any of these stages and warn the user that the document component `titlePage` is incomplete when the document is stored.

Tail-incompleteness allows documents to be created in a front-to-back sequence, and it might be concluded that this is a serious deficiency in Rita. Experience with GML and the Association of American Publishers SGML document type definitions [39] provides evidence to the contrary. Except for the immediate children of the root component, almost all document components in the GML document grammars and the AAP definition are optional. Hence creation of a document that is in fact tail-incomplete is quite uncommon,

since most users will produce an initial document which contains all the mandatory components specified in the document grammar. Even though most users will not produce a tail-incomplete document, it is still appropriate to validate the document with a parser before formatting or archiving.

Although tail-incomplete documents do not appear to be a problem for GML and the AAP definitions, this may not be the case for SGML in general. An important aspect of SGML is its use as a meta-language to allow organizations the freedom to construct grammars and associated tags for their own document types. This flexibility means that tags can be created for any language construct including single characters, and that the document grammars may contain numerous required objects.

Consider an SGML document grammar which may require both the body of a letter and the receiver's address to be present in the document. A user of this specific grammar may want to create the body of the letter before having to be concerned with the receiver's address. Since the class grammar requires both to be present, an editor which only supported tail-incompleteness would force the user to create the address first. In other words, tail-incompleteness does not allow the user to create objects in any temporal order.

A further problem occurs when an existing document is modified. During such editing, it is often necessary that one or more of the syntax rules be violated or relaxed in order to perform the edit effectively. For example, suppose an object class `X` is defined as

```
X = (a|b) c*
```

and an instance of the class was created with composition

$$\mathtt{x}_i = \mathtt{a\ c\ c\ c}$$

If the user should now decide that `a` must be deleted and replaced with `b`, there is no easy and convenient way of accomplishing this while still allowing only tail-incompleteness. A number of options do exist for handling this problem. One could define transformation rules (which might have to be elaborate—consider for example the case where `a` and `b` are non-isomorphic composite objects), or remove the grammar rules temporarily while editing the incorrect document. If the grammar rules are removed or relaxed for the entire document, it might be difficult to impose them again even if the document has only minor grammatical errors, since simple errors may cascade. Of course the incorrect portion of the document could be moved to a non-syntax checking environment for editing and later re-insertion in the syntax checking environment (see "Copying and moving" subsection). By only relaxing the rules for a portion of the document in isolation the impact of errors may be significantly reduced.

To overcome these potential problems with SGML grammars, we have developed methods which provide for the manipulation of documents that obey a different completeness criterion, namely *subsequence-incompleteness*. A document is *subsequence-incomplete* if the objects in the document form a subsequence[4] of some legal sequence of required objects. In other words, the document can be turned into a complete document using a sequence of one or more insertions. A document which is subsequence incomplete can be viewed as "almost" or *partially correct* and this term is also used.

Supporting subsequence-incomplete documents enables the user to create sibling document components in any time order. It also allows flexibility when an existing

[4] The characters in the words *seen, been, bun* and *use* are all subsequences of the characters in the word *subsequence*.

document is edited: objects can be moved/removed without causing difficulties for the syntax checker. Support for subsequence-incompleteness has not yet been incorporated into Rita, but has been tested successfully in a prototype. The discussion which follows shows how the algorithms already described could be readily extended to allow this form of incompleteness.

Subsequence-incomplete documents are supported by modifying (at appropriate times) the finite state automaton constructed for each object class. The automata are modified by adding null (ϵ) transitions: If $\delta : (S,\Sigma) \mapsto S$ is the transition function for an automaton, with S the set of states and Σ the input alphabet, then for every $r,s \in S$ for which there is some $a \in \Sigma$ such that $\delta(r,a){=}s$, the transition $\delta(r,\epsilon){=}s$ is added to the automaton. This is equivalent to relaxing the class grammar to one with the same structure as before, but all symbols are optional.

Unfortunately supporting subsequence incompleteness introduces a potential complication for the user. A user might create a document that is subsequence-incomplete and then not know how to complete it, especially a non-expert user unfamiliar with the class grammar (the user is probably not even aware that there is a grammar).

There are at least three approaches to handling subsequence incomplete documents: (i) the user could simply be allowed to create subsequence-incomplete documents, (ii) there could be a validation routine which verifies that a document is complete, or (iii) a menu-marking algorithm could be used to indicate to the user a reasonable way to make the document complete. Each approach has some merit and is discussed further in this section.

Allowing subsequence-incomplete documents implies that the user of Rita could create and attempt to format incorrect documents. This approach seems counter to the goals of Rita which are to allow the casual user to create correct documents with a minimal knowledge of tagging languages. Allowing a user to attempt to format incorrect documents could introduce a level of complexity which should be avoided for this type of user.

Running a validation routine would highlight the areas where the document is subsequence-incomplete, but would not give any indication of the correct structure to be inserted. The user would be required to run the validation routine before the document was saved, but could choose to ignore the results.

Similarly a user could be given some indication during document creation that the document is subsequence incomplete and be given a menu which not only shows all legal structural insertions but also marks a minimal set which will complete the document. This approach requires that the user survey the entire document to determine which structures produce menus with marked items.

Probably the best approach combines validation and marked menus. The validator should stop at any structure which is subsequence-incomplete, provide the user with a menu indicating the minimal insertions required, and allow the user to add the appropriate structures. Of course such a process has to be dynamic, since the user could insert a structure which would leave the document incomplete in a different way and the menu would have to change to reflect that fact.

The validation routine is a traditional parser and will not be discussed further. However, a brief description of the menu-marking algorithm is presented in the next few paragraphs. The complete details of this approach are described in References [6] and [40].

The user should at least have some indication of which sub-objects are mandatory for an object to be complete (and therefore correct). This is not a trivial problem since there

may be different combinations of mandatory sub-objects that will complete the object. For example, consider the object class

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

A subsequence-incomplete instance of this could contain `a c` and either

`a b d c` or `a c d`

would complete the object. One approach to solve this problem is for the system to mark those items in the menu that will complete the parent of the current object in the least number of insertions. If there are any required objects, at least one of them will always be marked.

There are three steps in the marking process:

1. Construct a finite automaton (call it MX_{min}) that provides all possible legal ways of completing the incomplete object.
2. Determine the shortest path(s) from the initial to final state(s) in MX_{min}.
3. Use the set of shortest paths calculated in step 2 to mark the appropriate items in the creation menu.

Allowing the user to view alternative methods for completing the parent of the current object is a possible modification of the menu-marking algorithm just presented. For example the algorithm could compute the shortest paths, the paths which are one longer than the shortest paths, two longer and so on. The number of paths could be a default parameter computed from the current grammar. In this case the user could toggle through the menus with the markings for the various path lengths looking for an appropriate document-completion strategy.

Structure editing

Creating a document requires both the insertion of new text and structure and then the modification of those two components as the document is changed. Support for editing the text is straightforward. However, editing the structure of the document can create complications particularly when the document conforms to a complex SGML grammar. The editor of a document wants to perform a number of operations including deletion of structures, moving and copying structures, transformations of one structure into a similar structure, and splitting, joining, nesting and extracting structures. In this section we describe how the finite automaton approach allows these types of editing operations to be performed in a flexible manner, while maintaining partial correctness. The description is brief, for more detailed algorithms, see Reference [6]. Not all these operations are incorporated into the current version of Rita. Implementation of many of these methods is part of an ongoing project in text-processing systems.

Deletion

Deletion is straightforward since deleting any object from any position still leaves a subsequence-incomplete document.

Copying and moving

A patch area is provided where the grammar rules are relaxed completely. Anything that is deleted from the document currently being edited is automatically copied to the patch area, from where the user can copy it again to any place in the document. Before inserting the content of the patch area into the document, the system ensures that the patch content is allowed in the new position according to the rules of subsequence-incompleteness. If the copy cannot be performed, the user can be directed to the offending object (in the patch area).

The user is allowed to edit the patch area. There is no difference between editing the patch area and editing a normal document, except that the system uses a standard, non-restrictive class grammar for the patch area. The syntax for this grammar is

$$\texttt{patch = (ob}_1 \texttt{ | ob}_2 \texttt{ | ob}_3 \texttt{ | ...ob}_n\texttt{)*}$$

where the $\texttt{ob}_i$s are all the object classes (basic and composite) that are defined in the current document class grammar. Furthermore, the syntax for each composite object class $\texttt{OB}_c$ is re-written as

$$\texttt{OB}_c \texttt{ = (ob}_1 \texttt{ | ob}_2 \texttt{ | ob}_3 \texttt{ | ...ob}_n\texttt{)*}$$

The result is that any combination and/or composition of any object is accepted in the patch area.

The patch area, together with the ability to edit it, provides a mechanism through which any change can be affected that is not provided for in the document grammar. At the same time, however, the document proper is kept partially correct.

Transformations

The current implementation allows, in a limited way, the class of an object to be changed to a class with similar structure. The algorithm mentioned provides only a limited form of transformation—for an extensive discussion of transformations see Reference [41]. A menu is constructed that lists the valid classes to which an object may be transformed. The algorithm of Figure 8 uses the following notational conventions: a_p is the current object,

CHANGE_CLASS()
begin
    class := *CLASS_OF*(*PARENT_OF*(a_p))
    Menu := *CONSTRUCT_MENU*(*class*, $a_1 \ldots a_{p-1} a_{p+1} \ldots a_m$, $p-1$)
    $b_1 \ldots b_n$:= *CHILDREN_OF*(a_p)
    for each class *class* ∈ *Menu* **do**
        if(*TEST_CORRECT*(class, $b_1 \ldots b_n$) = *FALSE*)**then**
            remove *class* from *Menu*
        endif
    endfor
    return *Menu*
end

Figure 8. Algorithm to transform the class of an object

while $a_1 \ldots a_{p-1}a_{p+1} \ldots a_m$ are the current object's siblings. First a set is constructed of all the classes of objects that may take the current object's place, should the latter be deleted. This set is then reduced by removing those members of the set for which the children of the current object do not form a legal subsequence. The function *TEST_CORRECT(X, x)* used for this reduction tests whether the sequence of symbols $x = b_1 \ldots b_n$ is accepted by the automaton for objects of class X. The resulting set contains all the elements that must appear in the transformation menu.

Split and join

It is sometimes necessary to split objects, e.g. one chapter may need to be split into two chapters. This is accomplished with the split operator: the user positions the cursor to point to the object at which the parent object must be split, such as the section at which the chapter must be split. The parent of the current object is then split at the current object, creating two new objects of the same class as the parent. The siblings to the left (or front) of the current object become the children of the leftmost of the two new objects, while the current object and siblings to the right (or rear) become the children of the other new object. This second new object becomes the new current object. The command is not executed if a partially correct document would not be obtained. Again *TEST_CORRECT* is used to ensure the split is only performed if the result would be a legal subsequence-incomplete document.

Join, the inverse operation joins the current object and its left sibling into one that has both their children. The user is prompted with a creation menu to indicate the class of the new parent if such an operation is indeed possible.

Nest and extract

It is possible to replace a sequence of objects (and their descendants) with a single new object that becomes the parent of the indicated sequence. For example, a sequence of paragraphs may be grouped into a subsection. This can be achieved with a nest function. The user would be prompted with a creation menu to indicate the class of the new parent if such an operation is indeed possible.

With the inverse function, extract, the current composite object (the cursor must be positioned on one, thus marking a subtree in the document) could be replaced by its children if it is allowed under the subsequence-incomplete rules.

Document semantics

In this section we describe the set of rules defining the appearance of a class of documents. As mentioned in the system overview, different concrete representations of the same document object can be produced, where each representation is generated by an *unparsing scheme*[5] [42]. An unparsing scheme is a set of unparsing methods. Each method describes one way of unparsing a class of objects (or class of nodes in the parse tree). More than one method may be given for a single class within one scheme, thus catering for layouts of the same object class in different contexts such as list items in different types of lists.

[5] The term *unparsing* refers to the fact that a concrete representation of a document can be obtained by "unparsing" the parse tree that represents the document.

An *unparsing method* is composed of a (possibly empty) set of variables and a set of unparsing actions. An *unparsing action* is an ordered pair (a transition and a program) that associates an unparsing program with a particular transition in a finite automaton constructed from one of the extended regular expressions that defines the class syntax. An *unparsing program* is a sequence of unparsing instructions that is executed when the particular transition is made during execution of the finite automaton[6]. *Unparsing instructions* produce output and/or change the value of the variables accessible to the method.

Figure 9 shows the semantic definition of `ordList` and `listItem` objects of the document class of Figure 2. The semantic definition, which is the part enclosed in square brackets (`[... ]`), defines two schemes named `display` and `gml`, which produce concrete representations in forms suitable for the display terminal, and as input to a GML formatter. The display scheme for a `listItem` has two methods defined, `Ordered` and `Unordered`. All other methods have been named `Standard`[7]. The actions for each method are listed in the form:

<transition> **:** *<program>*

For example,

```
listItem: @incvar @Ordered()
```

is one of the actions in the `Standard` method of the `/display/` scheme for `ordList`. It indicates that when a `listItem` is encountered within an `ordList`, the variable associated with the `listItem` is incremented and the `listItem` is then unparsed with the `Ordered` method (in the `/display/` scheme).

Unparsing methods for basic objects only have `initial` and `final` unparsing actions. It is implied that the text associated with a basic object is generated (emitted) between the `initial` and `final` unparsing actions for that object.

At least one unparsing scheme must be provided in a class description, namely one that generates a concrete representation for the display screen. The creator of the unparsing schemes must be aware that not all siblings may be present during document preparation. Fortunately, the appearance of siblings is seldom interdependent—the appearance of siblings are typically controlled by the parent which must exist before any siblings can be created.

The instructions allowed in unparsing programs include the following: adjusting margins and line length; conditional statements; emitting (outputting) string constants and values of attributes; emitting a given amount of vertical space; toggling word-wrap, centering, left-justification, right-justification, bolding, and underlining; testing for empty objects; and unparsing a constituent object with a specified method.

The current implementation supports only a limited set of variables, namely one for each object in the document. Work is in progress to extend this to provide the following:

[6] Every automaton is assumed to have at least two “transitions” that are always executed: one labelled `initial` that leads into the start state and is always executed to start the automaton, and one (or more) labelled `final` that leads from any final state to an imaginary “truly final” state.

[7] There is nothing special about the name `Standard`—any name could have been used, and each method could have had a different name. The only requirement is that different methods for unparsing the same object within a single scheme have unique names.

```
ordList = listItem+
   [ /display/
       Method Standard()
         initial:       @lm+(3) @vs(1),
         listItem:      @incvar @Ordered(),
         final:         @vs(1)
       endMethod

     /gml/
       Method Standard()
         initial:       @nl ":OL" @attrs ".",
         listItem:      @Standard(),
         final:         @nl ":eOL."
       endMethod
   ];

listItem = paragraph (paragraph | ordList | unordList)*
   [ /display/
       Method Ordered()
         initial:       @nl @emitvar ".  " ,
         paragraph:     if @firstsib then @Plain()
                        else @Standard() endif,
         ordList,unordList: @Standard(),
         final:
       endMethod

       Method Unordered()
         initial:       @nl "* " ,
         paragraph:     if @firstsib then @Plain()
                        else @Standard() endif,
         ordList,unordList: @Standard(),
         final:
       endMethod

     /gml/
       Method Standard()
         initial:       @nl ":LI" @attrs ".",
         paragraph,
         ordList,unordList: @Standard(),
         final:
       endMethod
   ];
```

Figure 9. A sample class definition. The unparsing schemes generate a concrete representation for the display and for a GML formatter

Attributes which are associated with an object and are global to all unparsing methods in the object class. Within unparsing programs attributes are read-only variables. Current support for attributes only allows limited access to them from within unparsing programs.

Method parameters which are variables that are passed to a method when the latter is invoked (from within an unparsing program) to unparse an object. Method parameters are the same as call-by-value parameters in the C programming language.

Local method variables which are variables that are local in scope to a method and are similar to local variables in a C procedure.

Additional unparsing instructions that are being implemented include: assignment to variables; string concatenation; obtaining the values of variables—this includes the value of an attribute of an object pointed to by a pointer variable; emitting values of variables (converted to strings if necessary); emitting the current date; font changes; testing if an object is the nth having a given attribute with a given value; and unparsing an arbitrary subtree of the document tree (usually determined by a pointer attribute) with a specified method.

USER EXPERIENCE

The Rita system is used extensively in a number of organizations to produce various types of documents. This section describes four specific experiences that the authors have been able to observe. These observations, in the form of brief anecdotes, support the claims of ease of use and increased productivity which were mentioned in the introduction.

A secretary in our research group who had some experience with word processors but no GML experience was asked to produce a 60-page manual from a manuscript which contained handwritten text. Document structure was implied by paragraph indention, headings for sections and titles, but there were no explicit tags in the text. Since no documentation was available for Rita the secretary was given a 10-minute demonstration. With one exception the secretary entered the document correctly the first time and that one error was easily fixed. It appears that Rita is an intuitive tool which allows novices to make effective use of text formatters.

A large company offered to test Rita in several “real world” projects. One project involved a word-processing group with the responsibility of producing letters, contracts and manuals using the Document Composition Facility. This particular group knew the GML tag set, but were not computer experts. Each member of the group indicated that Rita was a valuable tool, since they no longer had to remember all the subtle details of the GML tag set for a variety of document types. They were also able to observe the form of their document as it was being prepared, thus removing the need for many formatting runs on the mainframe just to ensure correctness of the tag set and the appearance. Although we did not make actual measurements, the manager of the department reported an increase in productivity; it was felt that more correct documents were being produced in a given time period.

The initial successes with Rita in the word-processing group prompted a test with a group of senior programmer-analysts who were obviously highly computer literate. The members of this group produced mail, reports and manuals, and did almost all their

document preparation using the Document Composition Facility and the SPF editor. They were categorized as casual expert users since they were not as knowledgeable about GML as the word-processing group. The experience again was quite positive, the group did not want to return the computers and the Rita software at the end of the experiment. Rita helped them produce correct documents the first time with minimal knowledge of the GML tag set.

A software company has produced a comprehensive software system which required a large-scale documentation effort. Specifically 12 multiple-section manuals of over 2000 pages of text and diagrams had to be prepared by 17 authors. The authors of the manuals had varying degrees of experience with GML ranging from expert to novice. The majority of the authors used Rita and found that there were almost no structural errors produced even when the sections were merged into the final versions of the manuals. A further measure of success is indicated by the fact that all the authors who used Rita did not want to return to the old methods of producing tagged documents. The experience was so successful that this group has now adopted Rita as a production tool.

RELATED WORK

The syntax-directed editing techniques used in (programming) language-oriented editors and their generators such as the ALOE system [43], the Cornell Program Synthesizer [34], the Synthesizer Generator [42,44], DOSE [45,46] and others [47,48], are to a large degree applicable to document editors, and often their authors mention that document editors have been or might be constructed, but do not go into any detail. However, programming languages are generally speaking more explicitly structured than documents, with the result that the editors generated from programming languages tend to be too rigid for use with documents. Furthermore, the incremental parsing (or recognizer) approach taken by many syntax-directed editors is not easy to adapt for use with documents, as documents often have ambiguous grammars. For example in the grammar of Figure 2, it is not possible to distinguish between the `(paragraph | ordList | unordList)*` portion of a `listItem` and the `(paragraph | ordList | unordList)*` part of a `section`.

An important aspect of any syntax-directed editing system is the trade-off between the amount of checking done to ensure correct structure and the flexibility afforded the user when editing the structure. One approach is to use templates or place-holders. Parts of the document that are required (or even optional), but still missing are represented by place-holders (“ghost windows” or “blank slots” used by Kimura [23] and “buttons” used by Furuta [26]). A penalty paid for this is the extra storage and screen space required for the place-holders. Another approach is to allow free entry of text and markup (or keywords indicating objects) and to re-parse the input at appropriate times. While this approach works well for programming languages, it is less suitable for documents since that re-introduces the dilemma of the user having to know a “formatting language.”

It is not clear from the literature how the document systems which were mentioned in the introduction and which do not use place-holders provide a flexible editing environment that uses the syntax-directed approach while also making allowance for partial documents.

Rita differs from commercially available SGML editors such as WRITE-IT [22] and Author/Editor [21] in two ways. The tags in Rita are in a separate structure window whereas the tags in these two editors are embedded in the text and may be exposed or hidden.

Also Rita dynamically produces a menu as the user traverses the structure window or moves into insert or transform mode. Author/Editor only produces menus if it is in insert or transform mode and structural checking is enabled. With automatic menu generation Rita makes it easier to enter correct documents, while with Author/Editor the user must continually select insert mode to determine what structures can be chosen at a given point in the document.

CONCLUSION

This paper contains a description of Rita, an editor and user interface for structured documents containing tags from markup languages such as SGML. Rita is provided with a grammar for a particular document class from a document class database, and thus constrains the user by only allowing the construction of documents which are syntactically correct. Rita also addresses the problem of partially correct documents, a situation which can occur as documents are being edited. The dynamic user interface presents menus of correct tags as the user constructs and edits the document. Also there is a screen layout for each document class grammar to provide an almost WYSIWYG display of the document as it is being prepared.

A number of conclusions, at least partially substantiated by observation, were drawn from the experiences with Rita. Rita has been useful across a wide range of users ranging from the novice and casual user to the expert. The user interface was certainly appropriate in that a novice could operate Rita with almost no instruction and no user manual. Using a structured editor and user interface such as Rita can contribute significantly to increased productivity because the user need have only minimal knowledge of the tag set, and none at all of the document structure and the correct position of structures in the document. Segmenting document creation into two parts, namely, structure/content and layout, minimizes the amount of typographic knowledge required by the individual user. The combination of a structured editor such as Rita and a text formatter which separates structure/content and layout provides a more productive document preparation environment.

ACKNOWLEDGEMENTS

We are grateful to Derick Wood for reading and offering helpful comments on the paper.

Research described in this paper has been partially supported by the Natural Sciences and Engineering Research Council of Canada, and the Foundation for Research Development, South Africa.

REFERENCES

1. L. Lamport, LaTeX*: A Document Preparation System*, Addison-Wesley, Reading, Mass., 1986.
2. IBM, White Plains, New York, *Document Composition Facility: General Markup Language Starter Set User's Guide*, 7th edition, August 1989.
3. Department of Computing Services, University of Waterloo, Waterloo, Ontario, Canada, *Waterloo SCRIPT GML User's Guide*, April 1988.
4. D. S. McKee and J. W. Welch, *WATCOM GML Tutorial and Reference*, WATCOM Publications Limited, Waterloo, Ontario, Canada, 3rd edition 1990.
5. C. F. Goldfarb, *The SGML Handbook*, Oxford University Press, New York, 1990.

6. G. de V. Smit, *A Formatter-Independent Structured Document Preparation System*, PhD thesis, Dept. of Computer Science, Univ. of Waterloo, Waterloo, Ontario, Canada, April 1987. Also Research Report CS-87-40.
7. G. de V. Smit and D. D. Cowan, 'Combining interactive document editing with batch document formatting'. In van Vliet [49], pp. 140–153.
8. D. D. Cowan, E. W. Mackie, and G. M. Pianosi, 'Experiences with RITA', Research Report CS-90-35, Dept. of Computer Science, Univ. of Waterloo, Waterloo, Ontario, Canada (1990).
9. E. W. Mackie and G. M. Pianosi, *Waterloo Rita — Tutorial and Reference*, WATCOM Publications Limited, Waterloo, Ontario, Canada, 4th edition, 1991.
10. D. D. Chamberlin, 'Document convergence in an interactive formatting system', *IBM Journal of Research and Development*, **31**(1), 58–72 (January 1987).
11. D. D. Chamberlin, H. F. Hasselmeier, and D. P. Paris, 'Defining document styles for WYSIWYG processing'. In van Vliet [50], pp. 121–137.
12. D. D. Chamberlin, H. F. Hasselmeier, A. W. Luniewski, D. P. Paris, B. W. Wade, and M. L. Zolliker, 'Quill: An extensible system for editing documents of mixed type', in *Proc. 21st Hawaii Int. Conf. on System Sciences*, pp. 317–326, Washington, DC (April 1988). IEEE Computer Society Press.
13. P. Chen and M. A. Harrison, 'Multiple representation document development', *Computer*, **21**(1), 15–31 (January 1988).
14. R. Furuta, V. Quint, and J. Andre, 'Interactively editing structured documents', *Electronic Publishing Origination, Dissemination and Design*, **1**(1), 19–44 (April 1988).
15. J. H. Walker, 'Supporting document development with Concordia', *Computer*, **21**(1), 48–59 (January 1988).
16. M. Hammer *et al.*, 'The implementation of Etude, an integrated and interactive document production system', *Proc. ACM SIGPLAN SIGOA Symp. Text Manipulation, SIGPLAN Notices*, **16**(6), 137–146 (June 1981).
17. R. A. Morris, 'Is what you see enough to get? A description of the Interleaf Publishing System', in *Proc. Second Int. Conf. on Text Processing Systems, PROTEXT II*, ed. J. J. H. Miller, pp. 56–81. Boole Press (October 1985).
18. T. Allen, R. Nix, and A. Perlis, 'PEN: A hierarchical document editor', *Proc. ACM SIGPLAN SIGOA Symp. Text Manipulation, SIGPLAN Notices*, **16**(6), 74–81 (June 1981).
19. P. R. King, 'An overview of the W document preparation system'. In van Vliet [49], pp. 154–170.
20. V. Quint and I. Vatton, 'Grif: an interactive system for structured document manipulation'. In van Vliet [49], pp. 200–213.
21. SoftQuad Inc., Toronto, Canada, *SoftQuad Author/Editor User's Manual*, 1st edition, April 1989.
22. Yard Software Systems, Chippenham, UK, *WRITE-IT SGML Editor*.
23. G. D. Kimura, 'A structure editor for abstract document objects', *IEEE Trans. on Software Engineering*, **SE-12**(3), 417–435 (March 1986).
24. L. van Huu, 'SGML: A standard language for text description', in *Proc. Second Int. Conf. on Text Processing Systems, PROTEXT II*, ed., J. J. H. Miller, pp. 198–212. Boole Press (October 1985).
25. G. Coray, R. Ingold, and C. Vanoirbeek, 'Formatting structured documents: batch versus interactive?'. In van Vliet [49], pp. 154–170.
26. R. Furuta, 'An integrated, but not exact-representation, editor/formatter'. In van Vliet [49], pp. 246–259.
27. G. M. Pianosi, *Waterloo Rita Document Class Generator — Reference*, WATCOM Publications Limited, Waterloo, Ontario, Cnanada, 1990.
28. D. Wood, *Theory of Computation*, John Wiley, New York, 1987.
29. D. E. Knuth, *The TEXbook*, Addison-Wesley, Reading, Mass., 1984.
30. J. F. Ossana, 'NROFF/TROFF user's manual', Research Report CS 54, Bell Laboratories, Murray Hill, New Jersey (October 1976).
31. J. Foley, A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics, Principles and Practice*, The Systems Programming Series, Addison Wesley, 2nd edition, 1990.
32. Open Software Foundation, *OSF/Motif User's Guide*, 1990.

33. Microsoft Corporation, Redmond, Washington, *Microsoft Word*, 1989.
34. T. Teitelbaum and T. Reps, 'The Cornell Program Synthesizer: a syntax-directed programming environment', *Comm. ACM*, **24**(9), 563–573 (September 1981).
35. International Organization for Standardization, New York, *Document Style Semantics and Specification Language (DSSSL) (ISO/IEC DIS10179)*, 1991.
36. M. A. Bhatti, 'Incremental execution environment', *SIGPLAN Notices*, **23**, 56–64 (April 1988).
37. D. B. Lomet, 'A formalization of transition diagram systems', *Journal of the ACM*, **20**, 235–257 (1973).
38. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Mass., 1979.
39. Association of American Publishers, *Electronic Manuscript Project, Standard for Electronic Manuscript Preparation and Markup*, Washington, DC, version 2.0, electronic manuscript series edition, November 1987.
40. G. de V. Smit and D. D. Cowan, 'Manipulating partial documents in a syntax directed environment', Research Report CS-90-02, Dept. of Computer Science, Univ. of Waterloo, Waterloo, Ontario, Canada (January 1990).
41. R. Furuta and P. D. Stotts, 'Specifying structured document transformations', In van Vliet [50], pp. 109–120.
42. T. Reps and T. Teitelbaum, 'The synthesizer generator', *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environment, SIGPLAN Notices*, **19**(5), 42–48 (May 1984).
43. R. Medina-Mora, *Syntax-directed editing: towards integrated programming environments*, PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, Pittsburg, Pennsylvania, 1982.
44. Thomas W. Reps and Tim Teitelbaum, *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer-Verlag, New York, 3rd edition, 1987.
45. P. H. Feiler and G. E. Kaiser, 'Display-oriented structure manipulation in a multi-purpose system', in *IEEE Computer Society, Seventh International Computer Software and Applications Conference, COMPSAC*, pp. 40–48, (November 1983).
46. P. H. Feiler, J. Fahimeh, and J. H. Schlichter, 'An interactive prototyping environment for language design', in *Nineteenth Annual Hawaii International Conference on System Sciences*, volume II, pp. 106–116, (1986).
47. R. H. Campbell and P. A. Kirslis, 'The SAGA project: a system for software development', *SIGPLAN Notices*, **19**(5), 73–80 (May 1984).
48. E. R. Gansner *et al.*, 'SYNED – a language-based editor for an interactive programming environment', *COMPCON, IEEE Computer Society*, 406–410, (Spring 1983).
49. *Proc. Int. Conf. Text Processing and Document Manipulation*, ed., J. C. van Vliet, Cambridge University Press, April 1986.
50. *Document Manipulation and Typography*, ed., J. C. van Vliet, Cambridge University Press, April 1988.