ELECTRONIC PUBLISHING, VOL. 7(2), 55–74 (JUNE 1994)

Making structured documents active

VINCENT QUINT[1] AND IRÈNE VATTON[2]

INRIA-IMAG
2, rue de Vignate
F-38610 Gières, France

SUMMARY

Active documents result from a combination of some specific features in documents and some mechanisms in a document manipulation system. In this paper we present the possibilities offered by a structured model of documents and a structured editor for making active documents. Several applications are described (annotations, electronic indexes, cooperative editing, documents as user interfaces, etc.), which show how a document's logical structure may be exploited for developing a variety of active document applications.

KEY WORDS Active documents Structured documents Editors User interfaces Grif

1 INTRODUCTION

The structured approach to document representation allows new, more powerful applications [3], but it must be admitted that the first tools developed with the representation were for the most common applications—editors and formatters such as Scribe [22] or LaTeX [16]. As an example, the first EP conference [26] was largely dedicated to structured documents, and almost all papers on that topic presented editors and/or formatters. Another example is the ODA standard [14], which, in addition to the definition of the document logical structure, considers only document formatting.[3] This is not surprising: the first use of computers for document processing was formatting, and the new approach was first tested on the same type of application. But after this initial step, other applications have been considered, taking advantage of the possibilities offered by an abstract representation of documents.

Currently, applications involving a structured model of documents can be found in many fields, such as databases, information retrieval or document recognition. In this paper, we focus on applications that have a connection with document preparation and editing, especially applications that extend the functionality offered by a document editor. More specifically, we are interested in applications that transform passive documents into active documents. There are many types of such applications, ranging from very simple tools that only consider a document as input data, to sophisticated tools that transform

[1] INRIA Rhône-Alpes.
[2] CNRS-IMAG.
[3] In that respect, SGML [10,13] is very different, as it does not consider *any* application. DSSSL [15], which is designed for specifying applications working with SGML documents, puts a strong emphasis on formatting issues, but takes also other applications into account, such as structure transformations or database feeding.

CCC 0894–3982/94/020055–20

Received 30 July 1993
Revised 22 November 1993

the documents they process as well as their environment. We present here the experience we have gained in developing active document applications based on the Grif structured document editor [19]. We also present the basic mechanisms we have developed in Grif for allowing it to handle active documents.

A number of active document systems have been described in the literature (see, for instance, [4,5,8,25,27]). In this paper, we put the emphasis on the advantages that the logical structure of documents, and specifically their generic structure, can bring to active documents. The main idea is to relate activity to documents on a generic basis. This approach implies not only that an editor be adapted to active documents, but also the document model on which it is based.

The paper is organized as follows: the next section describes some document-centred applications that have been developed with the Grif editor; it also explains why extensions to the editor were necessary. These extensions are then presented in two other sections: a section describes the extensions made in the document editor, and another section is dedicated to issues related with generic structures. Finally, these developments are discussed and compared with other active document systems.

2 DOCUMENT-CENTRED APPLICATIONS

According to the way documents interact with their environment, document-centred applications can be classified into two categories:

- The first category comprises applications that simply exploit the contents and the structure of documents for producing other documents and data. The documents they manipulate are called *passive*, as they only constitute the input of an application.
- Applications in the second category are interactive and react to user actions by modifying the documents they process and/or by acting on their computing environment. Documents handled by these applications are called *active*, as they support the interaction between the application and its user(s). Two sub-categories can be distinguished in this category:

  — Applications that only transform the documents they process, according to their structure and contents and to external parameters or programs. These applications can handle various types of documents and constitute extensions to document production tools.
  — Applications that process only documents of a given type and perform very specific functions, thus transforming a document production environment into a specialized tool acting on its computing environment.

Typical examples of applications belonging to the first category are translators or comparison tools. Tools of that kind do not transform the original document, even if comparators and translators produce new documents. Generally, they are not interactive and are implemented on the basis of a parser. We have experimented with some of them, such as a translator (from Grif to LaTeX, troff or SGML for instance), a document comparator and a static structure transformation tool [2]. They contribute to providing the user with a richer environment for handling structured documents.

The second category is very broad, ranging from applications that only update document contents to tools using the paradigm of documents as user interfaces [5]. All these

applications are based on (or include) an interactive editor; some are considered as extensions to the editor, some others as a specialization of the editor.

A spelling checker [24] is the simplest active document application we have experimented with. We have also developed an annotation tool, an electronic index [23] and a cooperative editing system [7], as active document tools extending the editor functionality.

Among the applications that specialize an editor are form editors, spreadsheets, syntax-driven editors for programming languages, etc.

Using an open-ended version of the Grif editor, we have developed both passive and active document tools, which are presented in this section for bringing out their requirements. These requirements have led to the definition of mechanisms that allow the editor to support active document applications. As these applications are all associated in some way with the Grif editor, this editor is presented first, but very briefly (more details can be found in [19] and [9]).

2.1 The Grif structured editor

Grif is a structure-driven editor. Structure-driven editing is inspired by syntax-driven editing. A *generic logical structure* (also called a *document type definition*, DTD, in SGML [13]) specifies the structure of a type of document and the editor uses that specification for helping (or sometimes obliging) the user to produce a document that is an instance of that type, i.e., with a logical structure consistent with the specification. With that approach, a document is represented as a *specific logical structure* (also called *abstract tree*) that organizes typed *elements* such as title, abstract, chapter, section, paragraph, note, etc. (element types are not predefined, but specified in the generic structure). This specific structure is mainly hierarchical, with additional relationships which represent nonhierarchical links between elements, such as cross-references. With these links, a structured document may also be considered as a hypertext [21].

The logical structure of a document not only contains elements, but also *attributes*. An attribute is information associated with a logical element of a document that adds semantics to that element. Like element types, attributes are defined in the generic structure, which specifies their type (number, name, character string, list of values, reference, etc.) and the type(s) of element with which they can be associated.

Grif generic logical structures are very close to SGML DTDs. The main differences are the following:

- Grif allows one to define generic logical structures in a modular way. Instead of writing large generic logical structures describing all types of elements that can appear in a document, the document designer can write several short generic logical structures, these generic logical structures being assembled for producing documents. This feature, discussed in [20], is the basis of the generic structure extensions mechanism presented in Section 4.
- Concurrent logical structure (Concur feature in SGML) are not allowed in Grif, but modular generic structures and dynamic sharing of documents can be used instead of this feature.
- Grif offers local and global attributes, when SGML only allows for local attributes. A global attribute is equivalent to a local attribute that can be associated with all logical elements defined in the generic logical structure. This feature is largely exploited by

active document applications for adding specific information on previously defined logical elements (for example, the access rights managed by the cooperative editing system presented in Section 2.3.3).

As the logical structure of a Grif document is always consistent with the model of a generic structure, it is possible to generically define the graphical aspect of documents: *presentation rules* are associated with element types and the editor applies them to the elements constituting the specific structure, thus producing the graphical aspect of the document. Presentation rules are expressed in a declarative language, called P. They specify such parameters as font, colour, spacing, line length, indention, justification, etc. Presentation rules are gathered in *presentation models*. A presentation model contains all rules necessary to specify the graphical aspect of all types of elements and attributes defined in a generic logical structure. Several presentation models may be associated with a generic logical structure; the graphical aspect of a type of document can be changed globally, just by changing the current presentation model.

Presentation models also define different *views*, which present in different windows the same logical structure with different graphical aspects. For instance, the table of contents of a report is a view that shows only the elements of type SectionHeading and does not use the same fonts and layout parameters as the main view. These views show a graphical representation of documents that is very similar to a printed document: the abstract (logical) structure is not directly seen by the user, who interacts with the editor through the views displayed on the screen (see Figure 1).

With these basic functions, the services offered by Grif are not fundamentally different from those offered by an editor-formatter based on a simpler document model. Nevertheless, its high-level document model allows it to produce a more abstract representation of documents, what makes it possible to process these documents in various ways. Applications presented below illustrate these capabilities offered by the document model.

2.2 Passive document applications

2.2.1 Document comparison

It is often necessary to compare two versions of a document in order to identify all changes that have been made between these two versions. In Unix, for instance, the `diff` command does that, but it can only process text files, i.e., documents that are made up of characters and lines, and it identifies changes in terms of lines. In structured documents, however, the notion of a line has no meaning: when two sections are exchanged in a document, many lines are moved, but, from the user point of view, the only interesting information concerns the logical elements of type Section. For that reason, a specific tool has been developed that compares the logical structure and the contents of two versions of a Grif document and that expresses the result in terms of the logical structure. It also links all logical elements of the second version with the corresponding elements in the first version. These links allow a user to see the corresponding elements in the two versions using the standard editing commands: by clicking any element in the second version, the corresponding element is highlighted in the first version.

The document comparator runs independently from the Grif editor: it reads two documents produced by the editor, compares their logical structures and their contents, and

produces a third document that is a copy of the second one, with specific attributes associated with all elements that are different from the first document. There is an attribute for each type of change that the comparator can detect: change of contents, element deletion, element creation, element move, etc. When the third document is printed or displayed by the Grif editor, presentation rules associated with these attributes make all changes visible to the user, under the form of revision bars, colours, etc.

Such a tool may have many applications. It can be used for comparing versions of technical documents, but it can also be useful in the humanities, for researchers interested in the evolution of electronic manuscripts [17].

2.2.2 Structure transformation

The document comparison tool takes care of changes in document instances, but generic structures also can be modified. The problem then is that document instances that have been produced with an old version of a generic structure are no longer consistent with the current version. Another, similar problem is to move a document from one generic structure (say, Report) to another (say, Book) that is not totally different, but that have different types of elements and attributes, and different structural relationships between elements.

For addressing these problems, a structure transformation tool has been developed [2], that compares two generic structures and produces transformation rules that are then applied to document instances built according to the first generic structure, thus making them consistent with the second generic structure. Due to that tool, generic structures can evolve without detriment to existing documents, and documents can move from one generic structure to another.

2.3 Active document applications

The previous tools are non-interactive applications. They accept document files as input and produce document files as output. These tools represent only a subset of document-centred applications. We present now a more representative set of document-centred applications, those based on the concept of an active document. These applications have been developed independently of each other, but some of them can use the functionality offered by others; for example the cooperative editing system can use annotations and electronic index.

2.3.1 Annotations

While preparing a document or reading a paper written by someone else, people often write comments or annotations in the margins. These comments are not really part of the document, but they are clearly related to it, or more precisely to some specific passages of it. Annotations may have an internal structure and they can improve the structure of the document by relating different parts of it.

A specific tool has been developed for the Grif editor that allows users to write such annotations and to use them for browsing through the document; in this way, the reader can build his or her own hypertext on the document. The internal structure of an annotation and the links between the main text and the annotations are represented and processed by the Grif editor like any other structured part of a document.

2.3.2 *Electronic index*

Another hypertext tool has been developed for helping users to make index tables and to exploit them. This electronic index [23] allows the author to select the passages of the document that he or she wants to be referred in the index. It also allows the user to associate a *descriptor* with each passage. A descriptor specifies in which index tables the corresponding passage must be referred to and what terms must represent that passage in the index tables. The application sorts index entries alphabetically, merges those with the same terms, and produces (or updates) the index tables, with links between the selected passages of the main text, the corresponding descriptors and the corresponding index table entries. Index tables are displayed (and printed) exactly as in a traditional book, but all these links can then be exploited by the user for browsing through the document, as in a hypertext, using the standard commands of the Grif editor. Figure 1 shows descriptors and an index table added to a manual by the index application.

The structures needed by that tool are fairly complex. They must represent and identify all entities involved in the application: selected passages of the main text, descriptors, index tables with multi-level entries, and all sorts of links. In accordance with the basic principles of the Grif editor, and for allowing the application to produce correct index tables, these entities are defined by a generic logical structure.

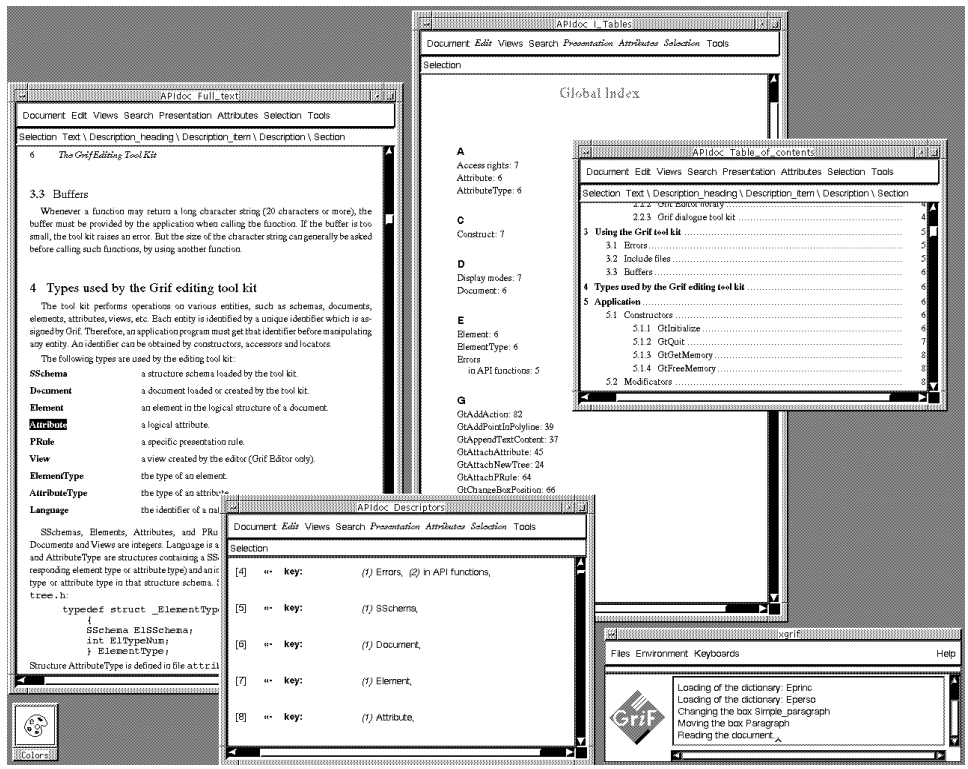

Figure 1. A manual, its descriptors and index tables

2.3.3 Cooperative editing

The structured approach to document processing proves particularly advantageous for large documents, such as technical documentation. Such documents are typically produced by several writers working simultaneously. Access to the document base must then be controlled, to avoid conflicts and inconsistencies. The problem of access control may be seen as a database problem (see the approach presented in [11]), but, as we will see, it also has something to do with editing. To address this issue, a distributed application has been developed that controls several local instances of the editor, running on different workstations, and that allows them to cooperate [7]. In this application, two main functions are performed by two categories of users:

1. A document manager builds a skeleton of the documents to be produced and decides who can access each part of the documents. Different access modes are available: some users cannot even see some parts, other users can only see them, and other users still are allowed to modify them. The document managers of a given document have all these rights, plus the right to change the rights.
2. A technical writer can access any document, provided he or she has the right to see at least one part of it. He or she can then load that document in the local editor and manipulate it, according to the rights he or she owns for each part.

It is the cooperative application's responsibility to dynamically control access rights of different users and to inform each user about actions allowed on each part. The cooperative application gives to each local editor only the parts the user can see (when the editor loads a document) and accepts to store only the parts that the user is allowed to change (when saving the document).

When a local editor allows its user to change some part of a document (which can be as large as a chapter or as small as a paragraph), that part is write-protected in other local editors as long as access rights do not change.

There is another approach to groupware, and especially to cooperative editing of documents, which considers that any user can change anything at any time: even the most elementary change of a single character by a user is immediately reflected to all other users. We decided to do it differently, because we have a high-level representation of documents, and we can take advantage of the logical structure for controlling common tasks.

Information associated with that cooperative application must be present in documents, while they are handled by a local editor: the parts on which the user has different rights must be identified, and the rights themselves must be indicated for each part. This is achieved by a special type of element that indicates the limit between two parts having different rights, and by an attribute that represents the right that the user has for each part.

2.3.4 Syntax-driven editors

Another category of interactive application that needs editing services is represented by syntax-driven editors for programming languages. The syntactic structure of a program is very similar to the abstract tree handled by Grif. The grammar of a programming language can be represented to a large extent as a generic structure, like the one used by Grif for specifying document types. It is then worth considering the use of an editor like Grif for

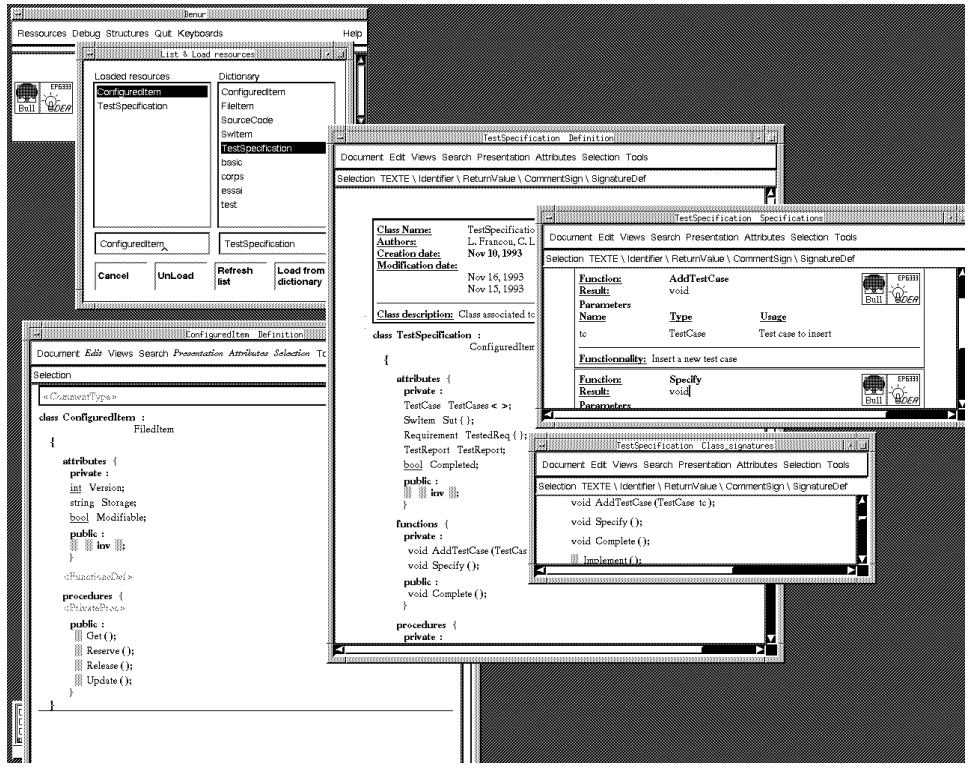

Figure 2. Several views of a Peplom program

building a program editor: by writing only a generic structure and a presentation model, a developer can make an editor for a given language.

Obviously, the semantics of a program cannot be fully represented in a structured document. Therefore, the editor should be controlled by an application that knows about the language semantics. The application can then prevent the editor from performing actions leading to incorrect programs or it can offer specific commands depending on the language semantics.

Two such editors have been built on top of the Grif editor, by two different research groups: one for the database programming language Peplom [1] (see an example in Figure 2) and another for the real-time language Argos [18].

2.3.5 Filer

Another approach to active documents consists in considering a document as a user interface tool (see [5], for instance): for controlling an application, a user manipulates a document whose elements represent the objects handled by the application. When editing actions are performed by the user on a document element, the application performs some functions on the corresponding object. Conversely, when the internal objects are changed by the

application, the document is updated accordingly, in such a way that it reflects the current state of the objects it represents.

We have used that metaphor for making a filer based on Grif. The objects managed by the application are Unix files and the functions offered by the application are those provided by the Unix commands `mv`, `cp`, `rm`, `mkdir`, etc. A special type of document, called Filer, has been designed for representing files and directories. With that document type, the Grif abstract tree represents the hierarchy of directories and files and the presentation rules show that structure in an iconic form: as in many desktops, each file or directory is displayed as an icon with a character string representing the file name.

When applied to a document of type Filer, the edit commands are interpreted as follows: modifying a filename implies changing the name of the corresponding file (`mv` command), deleting an icon implies deleting the file (`rm`), copying and pasting an icon implies copying the file (`cp`), etc.

3 EDITOR MECHANISMS

For making the applications presented above possible, several mechanisms were needed in the editor that were not available in the first version; they are presented in this section. As the Grif editor is based on the generic structure of the documents it manipulates, extensions to generic structures are also necessary; they are discussed in the next section.

The objective of the new mechanisms is to allow users of the Grif editor to develop easily such applications as those presented in the previous section. In this respect, the different types of applications presented at the beginning of this article have different requirements:

- Passive document applications (export, structure conversion, document comparison) do not require the participation of a human operator. They use only a subset of the editor's functionality: they are interested in abstract tree manipulations, but not in presentation of documents.
- Active document applications manipulate documents interactively. They need to drive the editor (that is the case with the cooperative editor and the syntax-driven editors). They need all the functionality of the document editor, but they also need to control it.

For meeting the requirements of all these applications, an editing toolkit and a special mechanism called External Call Facility (ECF) have been added to the Grif editor.

3.1 Editing toolkit

The Grif editing toolkit is a comprehensive set of editing functions that can be used for constructing applications. It allows application programs to perform the same operations as a user working with the Grif editor.

The editing toolkit is accessed by applications through an API. Typically, the API provides the application with functions for creating the skeleton of a new document of a given type, for modifying the structure or the contents of existing documents, for extracting information from documents, for opening and closing documents and views, etc. About 150 functions are available.

The toolkit takes the form of two C libraries, called Kernel and Editor. Each library is suited to a different type of application; each application uses only one of the two libraries, the second library including the first one.

3.1.1 Kernel library

The Kernel library allows an application to handle the logical structure (abstract tree) and the contents of structured documents in automatic mode. This subset of the toolkit does not produce the graphical aspect of documents, nor does it provide any service for the user interface. It is typically designed for applications performing automatic operations without direct manipulation by a human user, i.e., passive document applications.

3.1.2 Editor library

The second library, called Editor, includes all functionality provided by the Kernel and provides additional services for displaying the graphical aspect of documents. It also contains the whole Grif editor with its user interface, and some functions for extending this interface. All editing functions can be accessed both by the user and by application programs. The Editor library is typically intended for interactive applications that handle documents under the control of a user and that add new functionality to the editor, i.e., active document applications.

Functions in the Editor library that are also part of the Kernel have exactly the same interface in both libraries. The only difference is that, in the case of the Kernel, they do not display anything, and, in the case of the Editor, some of them (namely those modifying the logical structure or the content of documents) automatically modify the pictures (the views) displayed on the screen. The advantage is that applications do not worry about formatting and display; they have only to handle the logical structure and the contents of documents. Using the presentation models, the Editor library automatically takes charge of all problems related to formatting and refreshing the pictures displayed on the screen.

As a consequence, application programs that have been developed for automatic processing can be reused in the context of an interactive application without any change. This facility is also useful for debugging: a program designed for running in automatic mode shows all the actions it performs on documents, just by using a different library and making very simple changes to the program.

The application has only two functions for controlling the display: when a number of elementary changes have to be done in the same part of a document, the application asks the toolkit to stop refreshing the screen before making the first change and it resumes redisplay after the last change in the series. This avoids having each elementary change redisplayed individually.

The Editor library is used by all active document applications: spelling checker, annotation tool, electronic index, cooperative editor and syntax-driven editors.

3.2 External call facility

The API and the libraries are sufficient for passive document applications, like document comparison or structure transformation, which keep control over the toolkit: the editing

functions are called by the application and nothing happens in the toolkit that is not explicitly asked for by the application.

In active document applications, i.e., applications that are connected to the Grif editor, control is shared between the user and the application: some editing actions are triggered by the user, some other by the application. Then, an additional mechanism is required; the application not only calls the editor through the API, but it is also activated by the editor when the user performs some actions on the documents. This mechanism is called the *External Call Facility* (ECF).

As an example, in a syntax-driven editor for a programming language, the application must check the validity of identifiers: in a given scope, the same identifier should not be defined twice, and an identifier must be declared before being used. The editor cannot check that by itself, as the generic structure cannot express this kind of constraint. Then the application needs to be notified by the editor every time an identifier is entered or modified by the user. If the identifier is invalid, the application can then display a message for the user and change the document through the API (e.g., by modifying the entered identifier or by adding its declaration).

An application that extends the basic functions of the Grif editor or that controls it needs also to dialogue with the user. It needs to be able to modify the standard menus of the editor, by suppressing some commands (at least in some cases), by adding new commands, or by replacing existing commands. Therefore the ECF comprises two parts: the one is in charge of the dialogue, the other notifies the application when needed. The part concerning the user interface is not discussed here.

3.2.1 Notifying the application

The mechanism for notifying applications is based on the document logical structure and its generic specification. It allows an application to declare how and when the editor should give it control. As seen from the editor, an application is a set of procedures (called *actions*) that must be executed in certain conditions. As seen from the application, the Grif editor provides a set of editing functions and generates *events*. An event occurs each time a change is made to a document as a consequence of a user action. For instance, an event occurs when the user selects an element, modifies a character string, associates an attribute with an element, saves a document, opens a view, creates or deletes an element, etc. Each event is associated with an object: logical element, attribute, document, view.

Whenever the user performs an editing command, two events are generated: one before the command is actually performed by the editor (.Pre), the other when the editor has finished performing the command (.Post). It is important to note that a single user command may generate several pairs of events, eventually for several objects. For instance, when several paragraphs are selected and the user calls the delete command, a pair of events ElemDelete (.Pre and .Post) is generated for each selected paragraph. Another example is when the user clicks a cross-reference in an article, for displaying the referred figure. This single user action results in a pair of events ElemActivate concerning the cross-reference, a pair of events ViewOpen concerning the Figure view (supposing that the presentation model specifies that figures are presented in a different view, and that this view is not open yet), and a pair of events ElemSelect concerning the Figure element. All these events are raised separately and the corresponding actions are called one after the other.

With this mechanism, the application could receive a number of events, among which

only a few are of interest. Therefore, events are filtered for each application. For each type of document it manipulates, an application declares in what event it is interested, and more specifically in what event associated with what type of object. This declaration is made in a language called I. As an example, the abovementioned program editor is interested in the event ElemTextModified associated with elements of type Identifier (assuming that this type is defined in the generic structure of the documents processed by the application). The application also specifies what action must be executed when this event occurs. Then the Grif editor calls the desired action each time the event occurs for the type of element concerned.

Actions called on a `.Pre` event return a boolean value that indicates to the editor whether it must perform the standard function or not. This allows an action to be substituted to the normal editor command.

3.2.2 An example

The filer application presented above associates a document of type Filer to a Unix directory. It uses the following generic logical structure, which specifies that a document of type Filer contains a Title (the directory name) and a Body that is a sequence of elements of type File. A File contains only a FileName and has an attribute called FileType that specifies its type (only four types are defined in this example).

```
Filer = BEGIN
        Title = TEXT;
        Body = LIST OF (File);
        END;
File (ATTR !FileType = Directory, Script, Document, Other)=
        BEGIN
        FileName = TEXT;
        END;
```

No presentation model is given here, but a typical presentation model displays the Title element centred on top of the window and each File element after the other. The FileType attribute determines the icon representing a File element and the FileName element is displayed below the icon.

The events in which the application is interested, and the corresponding actions, are specified in language I by the following declaration, which is based on the previous generic logical structure (this is specified by the first line):

```
APPLICATION Filer;
ELEMENTS
  Filer:
      DocSave.Pre: SaveFiler;
  File:
      ElemNew.Post: NewFile;
      ElemActivate.Pre: OpenFile;
      ElemSelect.Pre: SelectFile;
      ElemPaste.Pre: CopyFile;
      ElemDelete.Pre: RemoveFile;
```

```
  FileName:
      ElemTextModified.Pre: Pre_ChangeFileName;
      ElemTextModified.Post: Post_ChangeFileName;
ATTRIBUTES
  FileType:
      AttrModify.Pre: ModifyFileType;
```

The line following the word File means that when the editor has created (event Elem-New.Post) an element of type File, it must call the function NewFile in the application. When this function is called, it receives the identifier of the new element. At that time, as specified in the generic logical structure, the editor has also created a FileName element as a child of the File element, and this element contains an empty character string (TEXT). As the FileType attribute is mandatory (this is indicated by the character "!" in the generic logical structure), the File element has this attribute with a value that has been chosen by the user among the four possible values: the editor has prompted the user for that value. Then the function NewFile can access the attribute value and generate in the empty string a default file name terminated by a suffix that depends on the FileType attribute.

The line `ElemActivate.Pre: OpenFile;` means that when the user clicks a File element, the editor must call the function OpenFile before doing anything else. This function receives the identifier of the clicked element and can then access the contents of its child, the FileName element. Then, depending on the value of the attribute FileType, it activates the corresponding script, opens the document or displays the directory. In that last case, the function creates a new document of type Filer, with one File element for each file in the directory, and it initializes the FileName elements with the names of these files; finally it opens a view for displaying that document.

As they are generated dynamically according to the state of the file system, Filer documents do not need to be saved. That is the reason why the line `DocSave.Pre: SaveFiler;` is associated with the element Filer. The action SaveFiler is called by the Grif editor when the user wants to save a Filer document. Since Filer documents do not need to be saved, this action only tells the editor not to save the document. An alternative would be to deactivate the Save command for all documents of type Filer; this can be done using the dialogue handling component of the ECF.

The two lines following the word FileName tell the editor that it must notify the application when the user begins to edit the contents of a FileName element and after the contents of a FileName element have been edited. The first action just copies the filename in a variable and the second action actually changes the name of the corresponding file, having the old name in the variable and the new one in the FileName element.

4 GENERIC STRUCTURE EXTENSIONS

Not only adequate mechanisms such as the API and the ECF are required in the Grif editor by active documents, but specific features are also needed in the document model.

Most applications presented in this paper should be able to treat any type of document: all documents are candidates for spelling correction, annotations or comparison of versions; although a letter or a memo rarely contains an index, many types of document have index tables; many may be edited simultaneously by several users. All these applications require some element types or some attributes to be defined in the generic structure of the documents

they process. That implies that almost all generic structures should be modified to contain the definition of all element types and attributes required by all applications. This is practically impossible, especially when considering that new applications can be added and new document types can be created at any time.

In fact, experience has proved that it is impossible to design a correct document type in an absolute way. The logical structure of a document is partly inherent in its type, but also partly in the manipulations intended for that document. As an example, in a letter, it is clear that an element of type Address must be defined, but if the letter has only to be edited and printed, decomposing the address into lines is enough; if it is stored in a database and if one wants to extract letters sent to a given person or to a given city, the structure of the address must be refined, with such elements as RecipientName, City or ZipCode appearing explicitly.

4.1 Principles of generic structure extensions

For solving this problem, two levels must be distinguished in a generic structure: one is independent of any application and only takes into account the intrinsic properties of the document type itself; the other is related to the applications and considers the specific requirements of each applications.

In accordance with that approach, generic structures defining document types are specified in Grif without paying attention to any particular application. These generic structures are made extensible in such a way that they can accept the pieces of generic structure needed by an application. These pieces of generic structure are called *generic structure extensions* (GSE). Such an extension can occur dynamically: an existing document that contains only the structural elements and attributes defined in the generic structure of its type may need to be processed by a new application, which then requires an extension of its generic structure (a GSE). It must also be possible to make several extensions to the same generic structure, in order to allow a document to be processed by several applications at the same time.

Dynamic GSEs make it possible to avoid associating to all documents the generic structures related to all applications: any documents should be *allowed* to use any application, but many documents do not use any application at all, or use only a few of those available.

4.2 Implementation

A GSE can define the attributes and the elements required by a given application.

4.2.1 Attributes

In Grif, there are two categories of attributes: global attributes and local attributes. A global attribute can be associated with any element belonging to an object built with the generic structure where this attribute is defined. A local attribute can be associated only with certain types of elements, explicitly indicated in the generic structure (see FileType in 3.2.2).

GSEs can define global attributes and local attributes as well. Global attributes defined in a GSE are considered as if they were defined in the main generic structure of a document. That is the case, for instance, for the attributes used by the document comparator: any element in a document can have these attributes.

In a GSE, local attributes may be associated either with the root element (whatever its name) defined by the main generic structure or to named elements. Each generic structure

defines a root element, and so local attributes for the root element can be defined in the GSE without knowledge of the main generic structure to which the GSE is applied. But if a local attribute is associated with a named element type, the GSE is not independent of the main generic structure, as that type must be present in the generic structure; this kind of extension is not as general as the other ones, but it has been considered useful for certain applications. Of course, local attributes can also be associated with the element types defined in the GSE itself.

An example of a local attribute associated with the root element of the main generic structure is the attribute DocumentDictionary that indicates the name of a dictionary that the spelling checker must use in addition to the general dictionary of each language: that attribute allows users to have their own specialized vocabulary and to indicate on the root element of a document what specialized vocabulary is used in that document.

4.2.2 Elements

A GSE can also define element types, and more specifically associated elements, units and exceptions (in the SGML sense).

An associated element is an element that belongs to a document, but is not at a fixed position in the document logical structure: it is not within the document, but it is associated with it. Typical examples of associated elements are bibliographies, notes or figures: all these elements may be interspersed in the main text, for instance at the bottom or top of pages, but they can also be gathered at the end of the document.

A GSE can define new associated elements which are considered as if they were defined in the main generic structure. That is the case of descriptors and index tables in the electronic index application. Annotations are also defined as associated elements in a GSE.

Exceptions (in the SGML sense) may be defined in GSEs. Like attributes, they can be associated with the root element type (whatever its name) or with named element types. Exceptions are divided into two categories: inclusions and exclusions. An inclusion is an element type that can occur anywhere within the subtree of an element having the type with which the inclusion is associated. An exclusion is an element type that must not occur within the subtree of an element having the type with which the exclusion is associated, even if this element type is allowed by other type definitions.

Inclusions are used in the GSEs associated with the annotation and index applications. A new type of element, a paired component (see next section), is defined in each of these GSEs as an inclusion associated with the root type of the main generic structure. Then these components can be inserted anywhere within any document that uses the GSE. They are used for marking the part with which an annotation or a descriptor (and then an indexing term) is associated (for more details, see [23]).

Inclusions are also used in the cooperative editor, in order to allow elements representing the user rights to appear at any position in the logical structure of a shared document.

Other kinds of elements may be defined in a GSE, such as units for instance (see [20] for more details about units). Only the most important are presented in this paper.

4.3 An example

The following GSE specifies all elements and attributes needed by the annotation application (for the sake of clarity, some simplifications have been made; all keywords are in uppercase):

```
STRUCTURE EXTENSION ExtAnnotation;
EXTENS
  ROOT + (Annotated_passage_delimiter);
STRUCT
  Annotated_passage_delimiter (ATTR !Link_to_annotation =
    REFERENCE(Annotation)) = PAIR;
ASSOC
  Annotations = LIST OF (Annotation = TEXT);
```

This specification extends the root element (`ROOT`) of any document with an inclusion (`+`): an element of type `Annotated_passage_delimiter` may be inserted anywhere in the abstract tree of a document that uses the GSE. This element is defined as a paired component (`PAIR`) and it has a mandatory (`!`) attribute called `Link_to_annotation`, which is a reference to an element of type `Annotation`. Annotations are associated elements (`ASSOC`): they are not part of the main document tree, but are related to it by links. Each annotation contains a character string (`TEXT`).

This definition specifies a structure such as that of Figure 3, supposing that the main generic structure of the document defines the types Section and Paragraph.

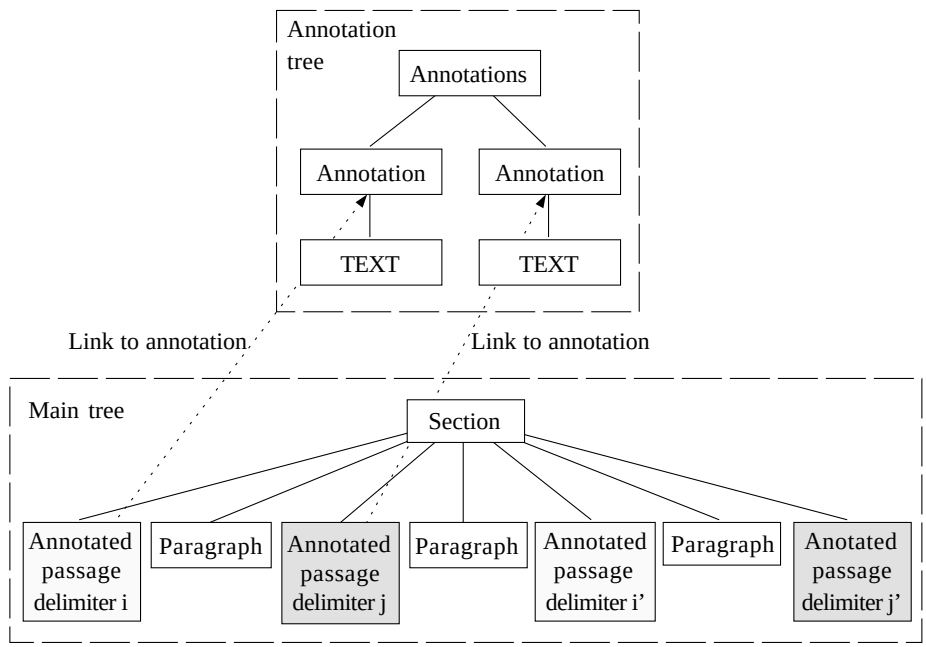

Figure 3. Structure defined by GSEs

5 DISCUSSION AND EVALUATION

The previous sections have presented the editing tool kit, the External Call Facility and the Generic Structure Extensions, which are the basic features for developing active document applications in the Grif system. In this section, we discuss these features and we compare

them with previous work in the field. To carry out this evaluation, we use the criteria proposed by Terry and Baker [25]: document model, activity specification, activity triggering, and code location.

5.1 Document model

In Grif, the document model is very rich, representing a document in a structured way, with a generic structure. Owing to generic structures, the logical structure of a document is formally defined and is predictable. As we have seen in the example of the filer, the application knows what elements are in the document and where they are. Using the concept of generic structure extensions, the application designer can also specify and organize the elements needed by the application and he or she can be sure that all documents, whatever their type, will contain these elements. This makes very powerful activities possible, because applications can work on sophisticated document structures. The index application is a good example.

The generality of the document model allows many types of applications to be developed. As the model can describe textual documents, but also equations, tables or structured graphics, activity can be associated with all these media, even when they are included in complex documents. It is then possible to activate equations in the same way as in CaminoReal [4]. Such an application is under development in the Euromath project.

Although it can represent such objects as graphics and equations, the Grif model is not really multimedia: it has no provision yet for sound or video. So, the voice annotations included in Scripted Documents [27] are not possible currently. The notion of script is not available either.

The Grif document model clearly separates presentation from logical structure. This allows application programs to handle only the logical structure and contents of documents and to take advantage of the formatting capabilities of the Grif editor and of its presentation models for handling the graphical aspect of documents.

In many systems, activity in documents is used for improving layout. That is the case in Quill [6], and to a lesser extent in active Tioga documents [25], where tables of contents are produced and maintained by active document techniques. That is not the case in Grif, because many of these layout and presentation functions are considered as standard functions and are performed by the editor itself, even for “inactive” documents. The P language allows a document designer to specify in a declarative way how elements must be numbered [12], what information should be displayed in various views (a table of contents is simply a view of the document), how cross-references must be displayed and maintained, etc.

5.2 Activity specification

Activity is specified at two different levels. At the first level, the declarative language I is used to indicate what editing events concerning what elements are interesting for the application, and what actions must be performed by the application when these events are raised. The second level is constituted by the actions, which are procedures written (often in C) by the application programmer.

There is no constraint on the computations that actions can perform. There is no restriction on the language in which actions are written, provided that actions can be

called from the C language. Actions can perform any function allowed by the surrounding computing environment and they can also act on documents. They can transform the structure and the contents of the document for which they work or any other document. This widely extends the range of applications that can be built.

Activity is specified statically, and there is no way of extending an application at run-time, as in Interleaf [8]. This is currently a limitation of active Grif documents.

5.3 Activity triggering

Activity may be triggered in a very flexible way. Activity invocation is not restricted to explicit user actions, such as clicking a button or entering a command, as in many hypertext systems. Although it is possible to add explicit commands to the Grif editor, most actions are in fact executed as side-effects of ordinary editing commands.

An action can be associated with any element type, any attribute, any view, any document, and it can be triggered on a large variety of editing events, not only when opening a document or when displaying an element. This is different from CaminoReal [4] which performs lazy evaluation of equations (equations are evaluated when they are displayed).

As all editing commands provided by the Grif editor can be called both by the user through the user interface and by the application through the API, a distinction is made between user-initiated edits and program-initiated edits, in such a way an application can avoid receiving notifications for edits that it has triggered itself. This is different from active Tioga documents [25], which notify all events, whatever their origin. The advantage of distinguishing the origin of events is that an application can mask some events according to their origin. For instance a syntax-driven editor that reacts to edits in identifiers does not want to be notified when it changes an identifier itself.

Some deficiencies of the edit notification mechanism described by Terry and Baker [25] are avoided in active Grif documents. An application receives each atomic edit separately and it has access to the current document state at any time. Thus it can more easily understand a sequence of atomic changes. This avoids the problems posed by a list of changes that are presented to the application globally and that are difficult to interpret.

5.4 Code location

The code of an application is linked to the Grif editor, not to documents. An application is simply a declaration of events and actions written in the I language and a set of functions implementing the actions. Declarations of events and actions are compiled and the compiler produces a module in C. That module and the modules implementing the actions are linked with the Editor library. Thus, actions are clearly separated from documents and generic logical structures. This allows active documents to continue to be accessed as ordinary documents by users who do not use the application. This also allows the free activation and deactivation of documents.

Adding activity to documents does not imply any change in the documents (actions are in the editor) nor in their generic structures (required elements and attributes are defined by GSEs).

Moreover, when an application such as the electronic index has transformed a document, the elements added by that application remain in the document (passage descriptors, index tables, hypertext links, etc.) and users of an ordinary Grif editor can use them, for navigating

through the document using the index tables for instance. But, without the application code, they cannot update index tables.

Multiple applications can be associated with a single document type; a declaration of events and actions is simply required for each application. A unique application can also be associated with multiple document types. Multiple applications can be linked with the Grif editor. A document can accept multiple GSEs, and then multiple activities. So, new applications can be added at any time for any document type.

6 CONCLUSION

We have presented the features that have been added to the Grif editor to allow it to support active documents. We have also described some applications built on these features. In this presentation we have tried to show how a structured model of documents can help in designing more powerful applications based on active documents.

It seems that the main differences between our system and other active document systems previously described come from the document model, and more specifically from the use of a generic structure. This is an advantage, but for taking all the benefits one can expect from generic structures, not only extensions to the Grif editor have been necessary, but also extensions to the generic structures themselves.

In associating activity with structured documents, we have explored the problems of designing generic structures. We consider that the notion of a generic structure extension (GSE) is useful for taking advantage of all possibilities offered by structured documents.

Some applications are presented in this paper. Others are under development, such as the Euromath project, whose objective is to build an environment for mathematicians that will allow a user to access several applications using Grif as the user interface. Among these applications are algebra systems, electronic mail and document composition (for producing TEX documents). This project is based on the concept of documents as user interfaces presented by Bier and Goodisman [5].

The next step will consist in designing an applicative language that will allow a user to dynamically develop new applications or to extend existing ones at run-time.

ACKNOWLEDGEMENTS

The authors would like to thank Jean Paoli, Lars Pedersen, Robert Puyol and all the people from Grif SA who have participated in the fruitful discussions that have led to the mechanisms and applications presented in this paper.

REFERENCES

1. M. Adiba, C. Collet, P. Dechamboux, and B. Defude, 'Integrated Tools for Object-Oriented Persistent Application Development', *Proc. DEXA 92 Conference*, Valencia, September 1992.
2. E. Akpotsui and V. Quint, 'Type Transformation in Structured Editing Systems', *Proceedings of Electronic Publishing 1992, EP92*, C. Vanoirbeek and G. Coray, eds, pp. 27–41, Cambridge University Press, April 1992.
3. J. André, R. Furuta, and V. Quint, *Structured Documents*, Cambridge University Press, 1989.
4. D. Arnon, R. Beach, K. McIsaac, and C. Waldspruger, 'CaminoReal: an Interactive Mathematical Notebook', *Document Manipulation and Typography, EP88*, J. C. van Vliet, ed., pp. 1–18, Cambridge University Press, April 1988.

5. E. Bier and A. Goodisman, 'Documents as User Interfaces', *EP90, Proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography,* R. Furuta, ed., pp. 249–262, Cambridge University Press, September 1990.
6. D. Chamberlin, H. Hasselmeier, and D. Paris, 'Defining Document Styles for WYSIWYG Processing', *Document Manipulation and Typography, EP88,* J. C. van Vliet, ed., pp. 121–137, Cambridge University Press, April 1988.
7. D. Decouchant, V. Quint, M. Riveill, and I. Vatton, *Griffon: A Cooperative, Structured, Distributed Document Editor,* R.R. 20, Bull-IMAG, Grenoble, June 1993.
8. P. M. English *et al.*, 'An Extensible, Object-Oriented System for Active Documents', *EP90, Proceedings of the International Conference on Electronic Publishing, Document Manipulation & Typography,* R. Furuta, ed., pp. 263–276, Cambridge University Press, September 1990.
9. R. Furuta, V. Quint, and J. André, 'Interactively Editing Structured Documents', *Electronic Publishing—Origination, Dissemination and Design,* **1**(1), 19–44, April 1988.
10. C. F. Goldfarb, *The SGML Handbook,* Clarendon Press, Oxford, 1990.
11. C. Hamon, *Conception orientée objet d'une base de données éditoriale SGML,* Thesis, University of Nancy, France, December 1992.
12. M. Harrison and E. Munson, 'Numbering document components', *Electronic Publishing—Origination, Dissemination and Design,* **4**(1), 43–60, March 1991.
13. ISO, *Information processing - Text and office systems - Standard Generalized Markup Language (SGML),* ISO 8879, 1986.
14. ISO, *Information processing - Text and office systems - Office Document Architecture (ODA),* ISO 8613, 1989.
15. ISO/IEC, *Information Technology - Text and office systems - Documents Style and Semantics and Specification Langage (DSSSL),* ISO/IEC DIS 10179, 1991.
16. L. Lamport, LaTeX*: A Document Preparation System,* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.
17. J.-L. Lebrave, 'L'hypertexte et l'avant-texte', *Proceedings of Electronic Publishing 1992, EP92,* C. Vanoirbeek and G. Coray, ed., pp. 233–246, Cambridge University Press, April 1992.
18. F. Maraninchi, 'The Argos Language: Graphical Representation of Automata and Description of Reactive Systems', *Proceedings of the IEEE Workshop on Visula Languages,* Kobe, Japan, October 1991.
19. V. Quint and I. Vatton, 'Grif: an Interactive System for Structured Document Manipulation', *Text Processing and Document Manipulation, Proceedings of the International Conference,* J. C. van Vliet, ed., pp. 200–213, Cambridge University Press, 1986.
20. V. Quint and I. Vatton, 'Modularity in structured documents', *Woodman'89,* J. André & J. Bézivin, ed., pp. 170–177, Bigre (63-64), IRISA, Rennes, May 1989.
21. V. Quint and I. Vatton, 'Combining Hypertext and Structured Documents in Grif', *Proceedings of ECHT'92,* D. Lucarella, ed., pp. 23–32, ACM Press, Milan, December 1992.
22. B. K. Reid, *A Document Specification Language and its Compiler,* PhD thesis, Carnegie-Mellon University, October 1980.
23. H. Richy, 'A Hypertext Electronic Index Based on the Structured Document Editor Grif', *Electronic Publishing—Origination, Dissemination and Design,* **7**(1), 21–34, March 1994.
24. H. Richy, P. Frison, and E. Picheral, 'Multilingual String-to-String Correction in Grif, a Structured Editor', *Proceedings of Electronic Publishing 1992, EP92,* C. Vanoirbeek and G. Coray, ed., pp. 183–198, Cambridge University Press, April 1992.
25. D. B. Terry and D. G. Baker, 'Active Tioga Documents: an Exploration of Two Paradigms', *Electronic Publishing—Origination, Dissemination and Design,* **3**(2), 105–122, May 1990.
26. J. C. van Vliet, editor, *Text Processing and Document Manipulation, Proceedings of the International Conference,* Cambridge University Press, Cambridge, 1986.
27. P. Zellweger, 'Active Paths through Multimedia Documents', *Document Manipulation and Typography,* J. C. van Vliet, ed., pp. 19–34, Cambridge University Press, April 1988.