ELECTRONIC PUBLISHING, VOL . 3(2), 105–122 (MAY 1990)

Active Tioga documents: an exploration of two paradigms

DOUGLAS B. TERRY AND DONALD G. BAKER[1]

Xerox Palo Alto Research Center
Palo Alto
CA 94304, USA

SUMMARY

The advent of electronic media has changed the way we think about documents. Documents with illustrations, spreadsheets, and mathematical formulae have become commonplace, but documents with active components have been rare. This paper focuses on our extensions to the Tioga editor to support two very different styles of active documents. One paradigm involves dynamically computing, or at least transforming, the contents of a document as it is displayed. A second paradigm uses notifications of edits to a document to trigger activities. Document activities can include database queries, which are evaluated and placed in the document upon opening the document, or constraints between portions of a document, which are maintained as the user edits the document. The resulting active documents can be viewed, edited, filed, and mailed in the same way as regular documents, while retaining their activities.

KEY WORDS Active documents Structured document editors User interfaces Databases

1. INTRODUCTION

Electronic documents are not limited to formatted text to be rendered on paper. The concept has expanded to include multimedia items such as graphics, voice annotations, and video presentations. Reactive components, such as derived cells in spreadsheets, embedded menus in hypertext systems [1], and constrained fields in forms, are also becoming more common. Such reactive components are useful, for example, in building user interfaces based on the document metaphor. Adding behaviors to documents yields what has been called *active documents* [2].

An active document has an associated set of behaviors, hereafter referred to as *activities*. For the active documents discussed in this paper, activities are arbitrary actions performed as a result of a user opening, scrolling, editing, or closing a document. Many activities have no visible manifestation; hence, the user's model of documents need not change. Although a document's activities are actually triggered by the document editor, they are associated with the document and are preserved when the document is filed, copied, or even electronically mailed.

The uses for active documents are broad and varied. Active documents provide a general platform for extending an editor's capabilities for specialized types of documents without modifying the editor for each new one. Specialized documents might include

[1] Current affiliation: Rice University.

0894–3982/90/020105–18$09.00
Received 1 February 1990
Revised 3 August 1990

© 1990 by John Wiley & Sons, Ltd.

© 1998 by University of Nottingham.

forms, programs, or spreadsheets. Within an active document, a set of one or more activities can cooperate to perform a specific function. Perhaps most importantly, active documents can serve as a user interface for document-based applications. Applications that are embodied as active documents may be called *active document applications*. Such applications may be easier to build than their traditional counterparts because they can take advantage of the capabilities of a document editor. In addition, they may be easier for users to learn because of the user's familiarity with the editor.

The document editor plays a vital role in supporting active documents. An active document has two major components: its stored contents and its associated activities. A document editor is responsible for displaying the document and for invoking its activities at appropriate times. This paper focuses on our extensions to the Tioga editor to support two very different styles of active documents. One paradigm involves dynamically computing, or at least transforming, the contents of a document as it is displayed. A second paradigm uses notifications of edits to a document to trigger activities. These activities, for instance, might maintain constraints within the document or between the document and some external database.

Pertinent details about Tioga and the Cedar Programming Environment are discussed in section 2. Sections 3 and 4 describe the node transformation and edit notification facilities added to Tioga as well as several active document applications that were built using these mechanisms. Of particular significance is the application that motivated our interest in active documents: using documents as editable, customized views of databases. Section 5 discusses general active document issues and how they are handled in related systems. These issues include the appropriate document model to support active documents, the rules for how and when activities are triggered, where the code for activities is stored, and special user interface considerations for active documents.

2. CEDAR AND TIOGA

Before launching into a discussion of our mechanisms for supporting active documents, it is helpful to understand the environment in which they were built. The Cedar programming environment [3,4] is a single-user environment that runs on a powerful workstation. Interaction is done primarily with a mouse and keyboard on a large bitmapped screen. The environment supports multiple lightweight processes residing in a single address space. Programs are dynamically loaded as they are needed. This type of environment is ideal for experimentation because the single address space allows easy cooperation of components and dynamic loading allows the edit-compile-load loop to be performed on individual modules rather than on entire programs.

Tioga is the always-resident document editor of the Cedar environment. Tioga maintains a distinction between the logical structure of a document and the layout structure. This distinction is becoming common in editors for structured documents [5–8]. Tioga documents are logically structured as a tree of nodes. Typically, a node of the document tree contains the text for a paragraph, heading, or table. Text within a node is represented as a linear sequence of characters. Each node of the document can have an arbitrary number of persistent properties, represented as name–value pairs. Characters can also have persistent properties, again, represented as name–value pairs. The major purpose of node and character properties is to allow experimentation without modification to the document model.

root

Format: Title
Text: Active Tioga Documents
An Exploration of Two Paradigms

Format: Authors
Text: Douglas B. Terry
and Donald G. Baker

Format: Affiliation
Text: Xerox Palo Alto
Research Center

Format: Heading
Text: 1. Introduction

• • •

Format: Heading
Text: References

Format: Paragraph
Text: Electronic documents
are not limited to formatted
text to be rendered on . . .

• • •

Format: Reference
Text: 15. Apple Computer
Inc., 'HyperCard', . . .

• • •

Figure 1(a). The logical structure of a sample Tioga document

Active Tioga Documents:
An Exploration of Two Paradigms

Douglas B. Terry and Donald G. Baker

Xerox Palo Alto Research Center

1. Introduction

Electronic documents are not limited to formatted text to be rendered on paper. The concept has expanded to include multimedia items such as graphics, voice annotations, and video presentations. Reactive components, such as derived cells in spreadsheets, embedded menus in hypertext systems [8], and constrained fields in forms, are also becoming more common. Such reactive components are useful,for example,in building user interfaces based on a document metaphor. Adding behaviors to documents yields what has been called *active documents* [10].

An active document has an associated set of behaviors, hereafter referred

Figure 1(b). The layout of the sample document depicted in Figure 1(a).

When displaying a document, Tioga generates the layout structure of the document from its logical structure by walking the document tree in prefix order. Two special node properties, 'Style' and 'Format', are recognized by the editor and are used to format individual nodes of the tree according to specifications written in an external style language. Within a node, the formatting of individual characters is affected by the 'Look' character property. Tioga only generates enough of the layout structure to fill a window on the screen. As a user scrolls this window, or thumbs to a new part of the document, more of the document's layout structure is generated on demand. Figures 1(a) and 1(b) illustrate the logical structure and the corresponding layout structure for a sample document, a version of this paper. (The document tree depicted in Figure 1(a) is simplified, in that a typical Tioga document is likely to have additional properties on its nodes. Figure 1(b) is also a simplified portrayal of a Tioga window.)

Tioga is a WYSIWYG editor; that is, users edit a document by directly editing the document's layout structure displayed in a window. Direct editing operations include adding, deleting, or modifying a node's text, splitting and joining nodes, and changing the nesting of nodes. As these edits are made, a lock on the logical document data structure is acquired and the logical document is appropriately modified.

As each modification is made, interested processes are notified, via procedure call, with details about the editing operation being performed. (Our second active document mechanism makes extensive use of this edit notification facility.) Processes associated with each window in which a document is displayed also monitor changes to the document and make appropriate updates to the screen as a result of changes in the underlying logical structure. In doing so, these processes must acquire a read lock on the document and a write lock on a portion of the screen. All of the processes act instantaneously, which gives users the feeling that they are editing the document directly.

3. NODE TRANSFORMATIONS AS ACTIVITIES

3.1. Motivation

Our interest in active documents arose from the desire to have documents whose contents are derived from external data sources. These sources might include conventional databases as well as files, other documents, and even continually changing sources like clocks. As the information provided by these sources changes over time, so should any documents containing this information. Moreover, updates to these documents should appear to occur instantaneously without human intervention. Requiring someone to explicitly edit a document whenever the base information changes, or to explicitly run a command as in Towner's auto-updating system [9], is considered an unacceptable responsibility. In short, we wanted active documents that dynamically compute some or all of their contents by reading external information.

We soon observed that replacing stale data in a document with up-to-date data is but a special instance of more general activities that can perform arbitrary transformations on a document's contents. Thus, our first active document paradigm is based on the idea of transforming the text of a node as it is fetched for display. While more powerful mechanisms may be desirable for many active documents, this mechanism has the advantage of simplicity. Activities need only concern themselves with textual transformations, not the manipulation of the entire document model. Another advantage

of this approach is that transformations are not performed until they are needed for display. This lazy evaluation can be exploited by active document applications.

3.2. Implementing node transformations

The node transformation mechanism works as follows. At the lowest level of the Tioga implementation is a module called TextNode that manages the tree of nodes for a document. It provides operations such as ‘return the *N*th child of a given node’, ‘return the next node in tree-walk order’, or ‘return the distance in characters between two given nodes’. Higher levels of Tioga call upon these operations to retrieve parts of a document for display. We modified the TextNode module so that activities associated with nodes are triggered when the contents of those nodes are requested. The result of the request is the transformed node, which can be displayed in the usual fashion.

To associate an activity with a node, a property named ‘Active’ is attached to that node; the value of this property indicates the ‘class’ of the activity, but does not contain the activity’s actual code. The implementor of a particular activity class registers with TextNode two procedures: a Transform procedure and a Size procedure. Figure 2 gives programming-language declarations for these procedures. Normally, the Transform procedure is called when the contents of the node are being requested; the Size procedure may be called at any time. However, there is also a simple tool to temporarily inhibit a registered activity class.

```
Transform: PROCEDURE [current: Node] RETURNS [new: Node];
Size: PROCEDURE [current: Node] RETURNS [size: INTEGER];
```

Figure 2. Procedures constituting a node transformation activity

The Transform procedure for an activity class takes as input the current textual contents of a node and produces the node’s new contents. It is called only when a node is being retrieved, for instance to display its contents on the screen or to search for a given text string. This procedure need not base the new contents on the old text. In particular, the Transform procedure is free to read the entire document and even to make arbitrary calls on the Cedar programming environment. This procedure can also produce side-effects, such as modifying the document tree, though there are restrictions on the modifications to the document tree that are allowed at this level.

The Size procedure estimates the size of an active node after it is transformed without actually performing the transformation. This estimate does not have to be very exact since it is used solely to paint the scroll bar (the shaded region on the left of the window depicted in Figure 1(b)). The Size procedure is necessary so that the actual transformations need not be performed to estimate their transformed size; without it, lazy evaluation would be lost.

3.3. Sample active documents

The node transformation paradigm has proved remarkably useful for active documents. To experiment with this mechanism, we have built a variety of node transformation activities. For instance, the ‘Command’ activity class treats a node’s contents as a

command line. It sends the line to a command shell for execution and then formats the command's results. As an example of an activity that does not depend on the node's contents, the 'TimeOfDay' Transform procedure simply returns the current date and time.

As an example of a transformation that depends on more than a single node, we built the 'TableOfContents' activity class. The Transform procedure for this class finds all of the section headings in the document and formats a table of contents. A simple version of this procedure returns the table of contents as a single character string, which becomes the node's new contents. A more complex Transform procedure could produce a node-structured table of contents. Figure 3(a) depicts the document shown in Figure 1(a) with an additional active node that represents a table of contents. Figure 3(b) shows how this document might look when formatted and displayed in a window.

Surprisingly, node transformation activities can even be used to give interactive feedback to users. For instance, one sample active document is a true/false examination that grades itself as the examinee types in answers. Following each question in the document is an active node in which a user types 'true' or 'false'. If the user types 'true' and the answer is false, or vice versa, then the Transform procedure appends '-- wrong' to the node's contents and updates the running tally of right and wrong answers. Correct responses are processed similarly. This 'TrueFalse' Transform procedure also prevents the examinee from changing an answer once it has been graded by removing the 'Active' property from the answer node. This particular activity relies on the fact that the contents of a node must be re-fetched for display (and hence, re-transformed) each time the user types text into the node.

Of the active document applications written to date using node transformations, the one that best takes advantage of lazy evaluation is an incremental database query evaluator. This activity allows database queries written in a high level query language to be given as the contents of an active node with the property 'Active: DBQuery'. The Transform procedure for this class of active nodes executes the query and replaces the query node with the query's results. If the query returns multiple database entries, then each of these is formatted and added to the document as a new node. Someone browsing through an active document containing query nodes need not be aware that some of the document's contents are derived from a database. However, if the person opens the exact same document sometime later, its contents will reflect changes that have taken place in the database.

An online electronic telephone directory, which we use daily, makes extensive use of active query nodes. The telephone directory is formatted as an electronic document so that users can read it and scroll around in it in the same way that they browse any electronic document. The directory entries are stored in a database so that queries can be run on the data. The document is useful for casual browsing, such as looking for a name that you don't quite know how to spell. The database is useful for performing more exact queries, such as finding the names and phone numbers of everyone whose first name is 'Mortimer'. To ensure that the document and the database remain mutually consistent, i.e. have the exact same contents, the document is an active view on the database. That is, the telephone directory document contains database queries that are executed on demand to retrieve its contents.

Figure 4(a) shows an example of a telephone directory document containing queries as active nodes. Figure 4(b) shows this document as it would look to a user after the node

root

Format : Title
Text : Active Tioga Documents
An Exploration of Two Paradigms

Format : Authors
Text : Douglas B. Terry
and Donald G. Baker

Format : Affiliation
Text : Xerox Palo Alto
Research Center

Active : TableOf Contents
Format : Contents

Format : Heading
Text : 1. Introduction

• • •

Format : Paragraph
Text : Electronic documents
are not limited to formatted
text to be rendered on . . .

• • •

Figure 3(a). The logical structure of the sample document with an active table of contents node

Active Tioga Documents:
An Exploration of Two Paradigms

Douglas B. Terry and Donald G. Baker
Xerox Palo Alto Research Center

Table of Contents

1. Introduction

2. Cedar and Tioga

3. Node Transformations as Activities

3.1. Motivation

3.2. Implementing node transformations

3.3. Sample active documents

3.4. Experiences and evaluation

Figure 3(b). The layout of the sample document depicted in Figure 3(a); note that the table of contents has been automatically computed

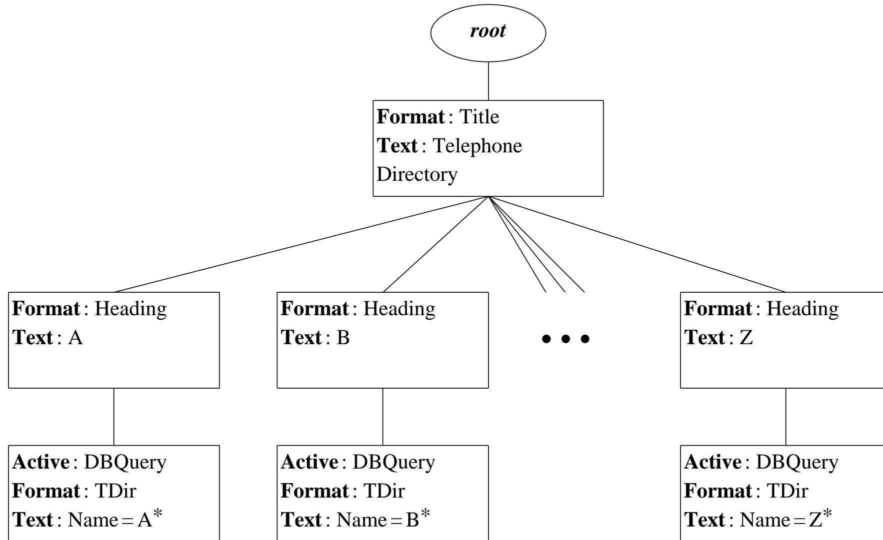

Figure 4(a). Logical structure of a telephone directory document with active query nodes

Telephone Directory

A

Adam, John	555-3920
Alberts, Frank	555-1294
Anthony, Mark	555-2939

B

Barret, Fred	555-3829
Bowers, Sue	555-2039
Brown, Ted	555-2048
Bryant, Mike	555-7862

C

Collins, Joyce	555-3902

Figure 4(b). Layout of telephone directory document including results of executed queries

transformations have been triggered. In practice, the user need not be aware that the document depicted in Figure 4(b) is an active document.

Because node transformations are executed in a lazy fashion, only those portions of the telephone directory that the user sees are actually retrieved from the database. For example, if the user only browses through the 'M's, then none of the entries for 'A' through 'L' or 'N' through 'Z' need be retrieved. This is a substantial performance benefit when the database is very large.

3.4. Experiences and evaluation

The node transformation mechanism has the following features:

- *ease of implementation*. Very few changes were made to the Tioga editor to support node transformations. In particular, none of the complex code for opening, formatting, displaying, scrolling, or locking documents was modified.
- *ease of creating active documents*. It is very easy to implement new types of activities; one must simply write a procedure that maps a text string to a new text string. A simple Transform procedure requires only a few lines of code. Moreover, its effects on the document can be readily understood.
- *efficiency*. The simple notion of an activity results in very low overhead for the mechanism itself.
- *lazy evaluation*. Having nodes transform on demand as the contents of those nodes are requested by higher levels of Tioga means that only nodes that are needed to fill a window get transformed. As the user scrolls the document, more node transformations may be activated as needed.

This simple mechanism also has several drawbacks:

- *lack of control over execution of activities*. Nodes are evaluated in the order that they are required for display and are only re-evaluated when they need to be redisplayed. This may yield incorrect values for an active node whose transformation depends on the contents of other active nodes that appear later in the document.
- *debugging is difficult*. Although Transform procedures are easy to write, they are hard to debug since they are activated while the screen (or a portion thereof) is locked, thus preventing debugging output.
- *editing and saving active documents is cumbersome*. Filing a document once its nodes have been transformed saves the new contents of active nodes, not the pre-transformed contents. This is usually inappropriate for active documents such as those containing database queries since the original queries are lost. Thus, editing a query, for instance, requires the user to inhibit the DBQuery activity, re-load the document, edit the query, and save the document before reactivating the DBQuery activity.

These drawbacks primarily stem from our reluctance to make substantial modifications to the Tioga editor. A document editor designed from the start to support active documents could avoid these deficiencies. The Quill document editor, for instance, keeps track of data-sharing dependencies between nodes and automatically reformats nodes when necessary [6]. These mechanisms in Quill were intended to support figure

references and other document formatting constraints, but they could also prove useful for active document applications. The problems with saving and editing documents that contain active nodes are addressed by the edit notification paradigm described in the next section.

4. EDIT NOTIFICATIONS FOR MAINTAINING CONSTRAINTS

4.1. Motivation

Node transformations work acceptably well for active documents that are read-only views on external databases. As long as users simply read one of these documents, they need not be aware that the document is active. That is, the activities are transparent. However, once a user attempts to edit the document, this transparency breaks down. Ideally, edits to the document should be propagated to the database from which the document was derived. In other words, there are important constraints between the document and the database that should be maintained regardless of which is updated. These constraints could be as simple as ensuring that a portion of the document is identical to a string stored in the database, or they could be much more complex. Constraints may also be desirable within a single document. For example, a document's table of contents should accurately reflect the document's section headings.

While exploring ways to maintain general constraints in editable documents, we developed a new active document paradigm based on notification of edits. Importantly, it allows applications to be notified about user actions that open, close, edit or save active documents. In response to these notifications, an application can take arbitrary action, including modification of the document.

4.2. Implementing the edit notification mechanism

In this second active document mechanism, activities are triggered via an *edit notification dispatcher*. This dispatcher receives all edit notifications generated by the Tioga editor, sorts them according to the document involved, and forwards them to interested applications. Active documents built using these edit notifications are standard Tioga documents, except that their root nodes contain a special property. The name of this property is the name of a registered activity class. Multiple activities can be associated with a single document by simply associating multiple properties with the root node of the document.

An edit notification activity consists of three procedures, which are registered with the edit notification dispatcher as a named activity class. Figure 5 gives declarations for these procedures. The dispatcher calls on these procedures at various times as a document is being displayed or edited.

```
Startup: PROCEDURE [doc: Document];
Notify: PROCEDURE [doc: Document, changes: LIST OF Edits];
Shutdown: PROCEDURE [doc: Document];
```

Figure 5. Procedures constituting an edit notification activity

The first registered procedure, the Startup procedure, is called once for each associated active document that is open at the time of the registration. It is also called whenever the user opens an active document possessing the appropriate root property. The Startup procedure sets up application-specific storage for the document, performs any initial edits to the document, and registers the potential edits to the document for which it wishes to receive notification. The Shutdown procedure, analogously, is called on all associated open active documents at the time the activity class is unregistered, or whenever an associated active document is closed by the user.

The Notify procedure, the most significant of the three procedures, is called as a result of edits to an active document. Editing operations that generate notifications include changes to the text of a node, changes to a node's properties, insertion of a new node, deletion of a node, and changes to the nesting of nodes in the document tree. The Notify procedure is passed edit notifications containing specific information about the edits that have occurred. For instance, when a user edits some text, the edit notification includes both the old and new contents of the affected node. The dispatcher batches edit notifications rather than calling the Notify procedure on every keystroke.

Any of the three registered procedures can edit the document during their execution. No distinction is made between user-initiated edits and program-initiated edits. That is, all edits produce edit notifications with one notable exception. The edit notification dispatcher avoids a potential infinite loop by inhibiting notification to an active document application of the document edits made by that application. Infinite loops are still possible when two or more edit notification activities are associated with the same document, because each activity may make edits about which the other is notified.

Several aspects of the actual edit notification mechanism merit further discussion. Instead of being notified of all edits to an active document, an application can inform the dispatcher of the specific kinds of edits for which it is to be notified. The application can also restrict notifications to those edits affecting particular nodes. In addition to document edit notifications, the application can indicate its interest in receiving notification of the 'save' operation (both before and after the file is written) and of a special event that triggers notification procedures without a user actually editing the document. The special event can be activated by any notification from outside the document, such as a timed pulse or a database update.

4.3. Applications

Section 3.3 discussed how the node transformation mechanism can be used to automatically build a table of contents for a document. This application has also been constructed using the edit notification mechanism. Moreover, using edit notifications, the table of contents is kept up-to-date as the document is being edited. The table of contents is a subtree of the document with each node of the table corresponding to a heading of the document. Textual changes to either the table of contents or the document's headings result in the corresponding changes to the other. Similarly, structural changes to either the table of contents or the heading tree result in corresponding changes to the other.

Maintaining an up-to-date table of contents is but one example of tracking dependencies that can occur between nodes of a document. Another typical example involves managing references to figures in a document; if a figure is renumbered, then

any references to the figure should be updated. Figure citations, and a host of other document constraints, can be readily maintained using the edit notification mechanism.

Using edit notifications, we were also able to build a database query mechanism similar to the one described in section 3.3 but with the added feature that the database can be directly edited in much the flavor of Query by Example [10]. In other words, as a user edits a document, such as the online electronic telephone directory shown in Figure 4(b), the underlying database is updated to reflect the user edits. Having fully editable documents as views onto databases allows users to take advantage of the formatting, structuring, browsing, and editing facilities provided by the document editor while avoiding specifics of the database's schema, physical organization, or query language.

Editing a document whose contents are derived from a database is accomplished as follows. In the saved document, nodes are tagged with a query of a specific database. On opening the document, the queries are evaluated and formatted into entry nodes. When a user edits one of these nodes, the application's Notify procedure determines the appropriate database update and adds it to a list of pending updates. A list of deleted nodes is also kept. These updates are not actually applied to the database until the user 'saves' his edits. If the user decides to 'reset' the document, then it is restored to its original contents and the database is not updated. When the document is saved, the set of entry nodes in the document is checked for collisions and inconsistencies; if there are any, the save is aborted with an error message. If there are no inconsistencies, the database is updated and the document is again collapsed to contain just its query nodes before writing the document to the file system. After a successful save, the queries are re-evaluated.

This active document application, as first implemented and described above, has the drawback that it requires complete evaluation of all queries before the document is available for user editing. Using a mixture of edit notification activities and node transformation activities (as described in section 3.3) we were able to build an electronic telephone directory that both preserves lazy evaluation and permits editing of the database view. That is, the two paradigms work together to yield an active document containing the best properties of each.

4.4. Experiences and evaluation

The edit notification mechanism has the following features:

- *complete document manipulation*. The entire document can be modified by an active document application in response to an edit notification. Such modifications can include both textual and structural changes to the document.
- *applications can react to most user actions*. Active document applications can react to the opening, closing, editing and saving of an active document. For example, a Notify procedure can react to a save notification by turning database query evaluations back into queries in the saved version.
- *transparent use by concurrent active document applications*. An active document application need not be aware of other applications on a document, and multiple activities at the same time are possible.

Some deficiencies of the edit notification mechanism have also been observed:

- *complex activity model*. Implementing active document applications that use edit notifications is not a simple task. First, a Notify procedure is not well insulated from the internals of the Tioga editor. For example, the type of information provided in an edit notification is strongly based on the programmers' interface to the Tioga editor. Second, the application's Notify procedure is given a list of recent changes to the document. The procedure must determine what actions are required based on this list of incremental changes and the current document state, a difficult chore.
- *no support for application-specific editing constraints*. Some structural changes to a document may not make sense in the presence of an application. For example, what does it mean to move a document heading into a contents outline given that the heading tree and the outline are constrained to be the same? Ideally, an application should have a way to inform the document editor about what edits are allowed and what are not.
- *no notification of selection change*. A significant user action that does not generate an edit notification, due to limitations in Tioga, is changing the selection. Notification of changes in the current selection is useful in building a rudimentary button interface.
- *debugging is difficult*. Startup, Notify, and Shutdown procedures are difficult to debug because document locks are held during their invocation so that they can make edits to the document.

The first two of these drawbacks cannot be easily remedied by changes to the document editor. They are fundamental problems that must be solved by additional research. We have not yet succeeded at designing a satisfactory application interface for supporting editable active documents. The remaining two problems are quite solvable. In fact, recent changes to the Tioga editor have made it possible to add active buttons to Tioga documents [11].

5. ACTIVE DOCUMENT ISSUES

Our experience with these two implementations has provided insights into several active document issues. These issues include such questions as:

- What features are necessary in the document model to support active documents?
- How and when are a document's activities triggered?
- How does an active document creator implement activities?
- Where is the code for the activities stored?
- Can users browse and edit active documents using familiar document editors?

This section discusses each of these issues in turn and tries to indicate how other related systems have addressed them. Table 1 gives a comparative summary of several systems.

5.1. Document model

Only one feature of the document model seems to be strictly necessary for a successful active document implementation: the ability to define and tag a region of the document. Tagging a region allows the specification of behaviors that will act on the region. Region

Table 1. Active document issues in various systems

System	Activity	Triggering	Code location	Document model
Active Tioga: Node Trans-formations	unrestricted transformation of text, some tree manipulation	on display of node	separate registered procedures	node structure WYSIWYG, rich extensions
Active Tioga: Edit Notifications	general computation	normal user actions; opening, scrolling, editing document	separate registered procedures	node structure WYSIWYG, rich extensions
Scripted Documents [12]	sequences of general actions	explicit start script operation	scripts stored in separate database	meta-document editor
CaminoReal [13]	mathematical evaluation	on display	in document	mathematical structure editor
EDS [14]	navigation among pages, animations	explicit mouse clicks	in database with documents	artwork, fields, buttons, not WYSIWYG
HyperCard [15]	navigation among cards, application manipulation function calls	explicit mouse clicks, other coarse-grained user-generated HyperTalk events	in 'stack'	artwork, fields buttons on
Interleaf [7]	general computation	on display, edits selection, menu clicks	in document or dynamically bound methods	hierarchical, WYSIWYG extensible
Quill [6]	complex SGML semantic routines	on display, on update to shared data	in design file (document style)	hierarchical SGML, WYSIWYG
Active Mail [16]	limited questions and mail routing	explicit run command	in message	message as a unit

identification seems to be most useful when regions correspond to logical units of the document such as paragraphs, headings, or fields in a form. Smaller granularities, such as a group of contiguous characters, can also be useful. Tioga allows the tagging of nodes of the document tree, which, by convention, correspond to logical units of the document. Tioga also allows the tagging of individual characters. By comparison, both Brown University's Electronic Document System (EDS) [14] and Apple's HyperCard [15] allow the tagging of regions of the document, but these regions correspond to physical areas of a page (or card) and do not typically correspond to logical units of the document.

Additions to the document model add to the richness of potential active document applications. Both of our active document mechanisms were able to take advantage of Tioga's formatting capabilities to improve the appearance and user interface of active documents. Two other types of active documents, Zellweger's Scripted Documents [12] and Arnon's CaminoReal documents [13], have also been built using the Tioga editor.

Several recent document editors, such as Interleaf [7] and Quill [6] have a document model very similar to Tioga's.

5.2. Activity triggering

In most existing systems, activities are explicitly triggered by the user. That is, the user clicks on buttons with a mouse in order to invoke activities. Scripted Documents [12] contain sequences of activities that are executed in temporal order, but nevertheless, scripts are initiated by explicit user action. Several systems allow activities to be initiated based on opening a document or opening a page or card in the document. Our edit notification mechanism, EDS [14], and HyperCard [15] all have this ability. Both our node transformation mechanism and CaminoReal [13] allow transformations of the document as it is being rendered on the screen or printer.

Some systems also allow activity triggering as a result of user edits. HyperCard allows activity triggering as the user's focus leaves a field that may have been edited. Active Tioga's edit notification mechanism is more general in that it reacts to all edits performed on the active document. The Quill [6] and Interleaf [7] document editors, because of their extensibility, allow programmers to write procedures that are triggered by display and edits as well. Thus, these editors can be used to achieve many of the same effects as our node transformations and edit notifications. However, these systems lack the fine-grain dispatching of edit notifications performed by Active Tioga, that is, the ability to trigger an activity based on certain types of edits to a particular node of a particular document.

5.3. Activity specification

Active Tioga allows any procedure that can be written in the Cedar programming language to be invoked as an activity associated with a document, including those that would modify the document itself. This generality allowed us to experiment with many diverse types of active documents. However, it raises important concerns about security. For instance, malicious, or even buggy, programs embedded in an electronic mail message could cause great harm to the recipients. Scripted Documents can also contain arbitrary Cedar executive commands. The Interleaf editor permits arbitrary Lisp procedures to be written and dynamically linked to the editor. Quill allows routines written in the REXX programming language to define the semantics of complex SGML elements; to our knowledge, these semantic routines have not been used to build active documents but could be. EDS allows general actions, but document authors do not typically write the code for the actions.

Other systems restrict the activities associated with a document, often by providing a special-purpose language in which activities are programmed. Most hypertext systems provide only interdocument navigation activated from tagged regions of documents [17]. HyperCard provides a language, HyperTalk, for specification of actions (though there is a way to invoke separately compiled procedures from within the HyperTalk language). Spreadsheets focus on numerical computations. CaminoReal limits activities to symbolic mathematical manipulation. Hogg and Gamvroulas's active mail messages [16] are written in a very constrained language that allows them to ask certain questions of users and route themselves around a network.

5.4. Code location

Storing the code for activities in the document itself guarantees that they remain with the document as it is moved or copied. HyperCard uses this strategy. Version control problems arise as documents proliferate with activities that later need to be improved or debugged. Alternatively, activities may be stored externally to a document and simply associated with locations in the document. The Scripted Document system, for instance, maintains scripts in a separate database. We have adopted a registration scheme whereby activity invocations are stored in a document, but the procedures implementing those activities are registered with the document editor by external programs. This has the drawback that the proper activities may not be registered at the time they are to be invoked (in which case no activity takes place); it has the benefit that individual workstation owners can control what activities they want to permit on their machines (thus alleviating the Trojan horse problem alluded to above). In practice, all users in our environment are expected to include a set of trusted and debugged activities in their running workstation software.

5.5. User interface

One of the most interesting qualities of active documents is that the documents themselves become a user interface to the activities. Users are already familiar with browsing, editing, and filing standard documents. We believe that active documents should not be treated any differently. Our two mechanisms do not require separate tools to manipulate active documents. Both HyperCard and EDS require the use of a special purpose application with its own user interface. Mathematical expressions in CaminoReal are required to be edited in a special structure editor. Scripted Documents also require a special editor. Live spreadsheets are, however, slowly finding their way into documents in the context of newer integrated document environments.

The consequences of using a document as a user interface have not been fully explored. Several issues did arise during the development of our experimental active documents. These same issues would also need to be addressed in the user interface of any high-quality active document.

- *Predictability and display inertia*. Two time-honored user interface design maxims [18] can be paraphrased as ‘the computer’s behavior should be predictable to the user’ and ‘the display should change as little as possible to carry out a user’s request’. An active document can easily violate these maxims, potentially confusing or annoying the user. We were able to avoid this in most cases by creating active documents that are completely transparent to the user. Even for non-transparent activities, we have found in practice that users who know they are working with an active document, and who know the intent of the active document, are rarely confused by it.
- *Handling errors*. It may not be possible to make sense out of every conceivable edit a user can make to a document. For example, what does it mean to rearrange a form? In practice, experienced users do not often make the kinds of edits that would be damaging to an active document. Our mechanisms, therefore, do not try to guard

against the most unusual of damaging edits. Nevertheless, means for inhibiting certain edits or tolerating damaged documents would be required of a system that catered to novice users.

- *Document user interface techniques*. The document is something of a new medium for user interfaces. Some user interface techniques, such as buttons and menus, can be adapted from traditional, window-based user interfaces. Others will be completely new to the medium. New techniques will involve typography and document layout to carry meaning. We have just begun to explore this interesting issue.

6. CONCLUSIONS

An interesting class of active documents results from associating activities with parts of a document and having those activities automatically triggered as the document is manipulated in various ways. We have designed and experimented with two different paradigms for creating active Tioga documents. One mechanism is based on transforming a document's contents as it is being displayed. The other mechanism generates notifications of edits to a document, which can be used to maintain constraints between parts of the document. To demonstrate the utility of these Tioga extensions, we built a wide variety of active documents.

For some applications, such as computing a document's table of contents, either mechanism could be used with certain advantages and disadvantages. More importantly, the two mechanisms can operate together to build an application that is not possible by either mechanism alone, for example to maintain an online electronic telephone directory whose contents are obtained by querying a database. Using on-demand node transformations to execute queries and format their results permits lazy evaluation. Using edit notifications to detect updates and perform them on the database allows users to edit directory entries using the Tioga editor rather than a special database update program. Both mechanisms combine to maintain the illusion that the document is a plain Tioga document.

The active documents supported by our extensions to the Tioga document editor are unique in that general activities are performed in response to normal user actions such as opening, scrolling, or editing a document. In particular, users do not need to push buttons or type commands to invoke activities. In most cases, users need not even be aware that they are dealing with active documents.

Adding support for node transformations and edit notifications to the Tioga editor required surprisingly little effort. Both mechanisms were implemented with little or no modification to the complex formatting, scrolling, painting, and editing machinery in Tioga. We suspect that these mechanisms would fit easily into many existing document editors.

While these efforts have taught us much about the nature of active documents and what is necessary to implement a general active document system, more work is necessary in the areas of editor support, error handling, code placement, and user interface issues. In the future, we would like to try implementing more complex active document applications such as a command executive, a spelling checker, or an incremental program syntax checker.

7. ACKNOWLEDGMENTS

We thank Dan Swinehart for some interesting brainstorming sessions that led Doug into the world of active documents, Michael Plass for helping us understand the internals of the Tioga editor, and Polle Zellweger for her insights into the nature of active documents and her support in our efforts. Dennis Arnon, Rick Beach, Dan Swinehart, and Polle Zellweger provided comments on earlier drafts of the paper, for which we are grateful.

REFERENCES

1. Larry Koved and Ben Shneiderman, 'Embedded Menus: Selecting Items in Context', *Communications of the ACM*, **29** (4), 312–318 (1986).
2. Robert Spinrad, 'Dynamic Documents', *Harvard University Information Technology Quarterly*, **VII** (1), 15–18 (1988).
3. Daniel Swinehart, Polle Zellweger, Richard Beach, and Robert Hagmann, 'A Structural View of the Cedar Programming Environment', *ACM Transactions on Programming Languages and Systems*, **8** (4), 419–490 (1986).
4. Warren Teitelman, 'A Tour Through Cedar', *IEEE Software*, **1** (2), 44–73 (1984).
5. Norman Meyrowitz and Andries van Dam, 'Interactive Editing Systems: Part 1', *Computing Surveys*, **14** (3), 321–352 (1982).
6. Donald D. Chamberlin, Helmut F. Hasselmeier, and Dieter P. Paris, 'Defining Document Styles for WYSIWYG Processing', in *Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography (EP88)*, ed. J. C. van Vliet, Cambridge University Press, pp. 121–137, (April 1988).
7. Paul M. English, Ethan S. Jacobson, Robert A. Morris, Kimbo B. Mundy, Stephen D. Pelletier, Thomas A. Polucci, and H. David Scarbro, 'An Extensible, Object-oriented System for Active Documents', in *Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography (EP90)*, ed. Richard Furuta, (September 1990).
8. Richard Furuta, Vincent Quint, and Jacques André, 'Interactively Editing Structured Documents', *Electronic Publishing*, **1** (1), 19–44 (1988).
9. George Towner, 'Auto-updating as a Technical Documentation Tool', in *Proceedings ACM Conference on Document Processing Systems*, Sante Fe, New Mexico, pp. 31–36, (December 1988).
10. Moshe M. Zloof, 'QBE/OBE: A Language for Office and Business Automation', *IEEE Computer*, **14** (5), 13–22 (1981).
11. Eric A. Bier and Aaron Goodisman, 'Documents as User Interfaces', in *Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography (EP90)*, ed. Richard Furuta, Cambridge University Press, (September 1990).
12. Polle T. Zellweger, 'Active Paths Through Multimedia Documents', in *Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography (EP88)*, ed. J. C. van Vliet, Cambridge University Press, pp. 19–34, (April 1988).
13. Dennis Arnon, Richard Beach, Kevin McIsaac, and Carl Waldspurger, 'CaminoReal: An Interactive Mathematical Notebook', in *Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography (EP88)*, ed. J. C. van Vliet, Cambridge University Press, (April 1988).
14. Steven Feiner, Sandor Nagy, and Andries van Dam, 'An Experimental System for Creating and Presenting Interactive Graphical Documents', *ACM Transactions on Graphics*, **1** (1), 59–77 (1982).
15. Apple Computer Inc., *HyperCard*, Apple Computer Inc., 20525 Mariani Ave., Cupertino, CA, 95014 (1987).
16. J. Hogg and S. Gamvroulas, 'An Active Mail System', in *Proceedings SIGMOD'84, Boston, Mass.*, pp. 215–222, (1984).
17. Jeff Conklin, 'Hypertext: An Introduction and Survey', *IEEE Computer*, **19** (9), 17–41 (1987).
18. Wilfred J. Hansen, 'User Engineering Principles for Interactive Systems', in *Proceedings of the Fall Joint Computer Conference*, pp. 523–532, (1971).