An Extensible, Object-Oriented System for Active Documents

Paul M. English, Ethan S. Jacobson, Robert A. Morris[*], Kimbo B. Mundy, Stephen D. Pelletier, Thomas A. Polucci, and H. David Scarbro

Interleaf, Inc.
Ten Canal Park
Cambridge, MA 02141
[*]*University of Massachusetts at Boston*
Harbor Campus
Boston, MA 02125

ABSTRACT: An extensible, object-oriented system for describing and executing active documents is discussed. An existing commercial, structured document processing system was extended with a run-time bindable object system and Lisp interpreter.

1 Introduction

Object-oriented programming paradigms are now well understood to be highly productive and to contribute to re-usable code. Extensible document processing systems, even as "simple" as Emacs (Sta81), the original and one of the most widely-used extensible editors, are known to be especially powerful. This is because they permit the entire community of users to mold the underlying editor for tasks sometimes not imagined by the software architects.

The earliest computerized document processing systems reflect an understanding that portions of documents interact with each other and with outside data, events, and programs. Even such primitive internal relations as automatic number streams, cross references, tables of contents, and indices represent document objects that must be computed by the processing system based on more or less complex relations within parts of the document.

Walker (Wal81) describes mechanisms for using Emacs as both the editor and invoker for documents formatted by Scribe, as well as the invoker of related documentation tools. Recently, explorations of complex document interactions have appeared from many directions. Towner (Tow88) describes documents with sufficient indirection that they support auto-updating from an external database whose form is known in advance to the software. Chamberlin, et al. (Cha88) describe a mechanism for providing external formatting modules based on properties of the document object being formatted. Arnon, et al.(Arn88) report about a system that can send a mathematical object from a document to a symbol manipulation system and reformat the returned object, so that editing a mathematical object can cause related objects to have their content changed before reformatting. Zellweger (Zel88) has built a mechanism for documents that invokes external scripts supporting such things as voice annotation.

A theme has emerged that the document pieces might be described not by what they *are*, but rather by what they *do*. That notion may best be captured by the name "active documents," a term that seems to have originated with Brian Reid during the EP-88 conference. We are unaware of any precise definition of active documents, but we take it to mean structured documents and their processors in which the objects in the documents can be acted upon by, and can themselves act upon, other objects in the document or the outside world.

This paper describes work at Interleaf, Inc. to produce an extensible, object-oriented active document processing system. In its most static state, this system corresponds to the standard commercial software. In its most dynamic state, the software provides facilities whereby third parties, including end users, can radically change the manner in which document objects behave, as well as change the user interface, the methods by which objects are treated (e.g., the pagination and composition methods), and even what kinds of objects can be regarded as part of a document. Unlike the approaches mentioned above, what we describe requires no special advance cooperation between the system architects and the document architects. Each document object can carry its own method for interacting with its environment or for computing its content or that of other document objects.

2 Objects and Methods

The Interleaf object software is a message-passing, object-oriented system built on a Lisp interpreter embedded in the distributed software. Standard class creation and inheritance mechanisms are provided (Mey88).

Lisp programs can have access to all objects in documents, ranging from characters and graphics to the higher-level objects that give them structure, such as document *components* and other document hierarchy elements. At a minimum, each such object responds to messages that request the object to set or report its properties. Classes can be requested to create new instances and these instances can be given properties, be linked into documents, or even have their methods changed. For example, although equation objects are externally modeled on *eqn*, it is possible to introduce a new class called doc-tex-equation-class whose composition methods cause an invocation not of Interleaf composition modules, but of an external TeX engine. Sub-classes that inherit the methods of the parent class can be created and can have their methods rebound or have new methods added.

What are sometimes called "live" or "hot" links in personal computing environments are a simple kind of active document object that is easy to implement with the system described here. Two instances that have been demonstrated involve binding the edit method for particular kinds of graphics elements to an invocation of external software. In one case, selecting a CAD-CAM picture produced by AutoCad™ software causes AutoCad to begin running, permitting the recomputation of the graphics object. That object is then automatically updated by receiving messages to set its properties based on the AutoCad computation (e.g., as recorded on the file system or sent via interprocess communication mechanisms) and a message to redraw itself. In a similar application, selecting a table or data-driven chart object can cause the invocation of Lotus 1-2-3™. The resulting manipulations are reflected—without further user intervention—in the document object. The actors (AutoCad or 1-2-3 in these examples) do not need to run on the same host as the document processor, as long as they can communicate with the host either through the network file system or via various supported interprocess communication facilities that can be used by the Lisp programs. Such a capability is similar to that of CaminoReal (Arn88), but the dynamic bindings of Lisp make it possible to implement such capabilities without any special advance compiled-in arrangements in the document processor. The particular choice of AutoCad and Lotus is incidental. Any network-accessible programs that support compatible data interchange could be used just as easily; indeed, such choices could be made at run-time.

Another simple application supports document objects by indirection. At various events, such as opening, printing, or display, objects can be sent messages causing them to query external data or processes to get the current state of the data from which they are produced. Several mechanisms are in place to facilitate this. In the most straightforward, the open method for a document can examine a list of document objects that require messages to be sent to them. These objects can be located by document navigation facilities described below.

We note that the above constructions are simple but powerful instances in which the object content is computed much as, say, cross references might be in traditional systems. The lack of distinction between program and data in Lisp provides the ability to have content be the result of procedure, not data. In addition, the run-time binding of Lisp eliminates the requirement that the object architecture be determined at compile-time, as is the case for other object-oriented languages such as C++.

3 Document Objects

From its initial release, Interleaf publishing software has been object-oriented from the user's point of view. However, its objects had a fixed collection of properties chosen by the system architects. For example, components have margins, default font, leading, and similar properties; graphics objects have size and color properties, etc. Initially, this object orientation was reflected more in the user interface than in document structure.

Recently there has been further evolution of structural object orientation. By release 4.0, the current commercial release, a style sheet mechanism built on the document substrate was in place. In these style sheets, called "catalogs," master templates for components and graphics objects can contain shared content as well as form. Catalogs do not appear to be different from any other kind of document, but editing them causes corresponding changes to all associated objects in all associated documents. These associations were not programmatic but rather were based on the name space to determine object affiliation. All objects of the same name as the master object were changed when the master was changed. Geometric relationships on the desktop further restricted the scope of conflicting names in multiple masters. (An object is an instance of a master only if the corresponding catalog is in the same instance of a special container, called a "book," and if the catalog is the nearest one with that master in the geometric ordering of the documents in the book.)

There was no other class mechanism than that described above, but a mechanism was added that provided a limited way to add properties. This was called "effectivity control" (Ils88) and permitted components to be marked with numerical or textual attributes, on whose values certain actions could be modified. For example, in an aircraft maintenance manual, a particular component could be tagged with an attribute specifying the customer for the aircraft, and a specialized document printed that contained all the common maintenance documentation plus that which applied only to the planes of that customer. This kind of structure tagging mechanism was powerful enough to support an SGML-based system compliant with the CALS requirements (Cal88).

The new document object ("doc-obj") system is built on top of a Lisp-based general object system. All the "traditional" Interleaf objects (both in the document and in the user interface) are available in this system, and new ones are easily added. The pre-existing doc-objs have a hierarchy that encompasses objects as large as a directory full of documents and as small as a marker between any two characters in a document. Individuals characters may be addressed via markers, but the characters are not full-fledged objects. However, strings may be wrapped in "inline component" objects, which can be recursively nested to provide objects with which to manipulate text, much as we provide a "group" to manipulate graphics.

Doc-objs come equipped with default methods that, in some cases, are simply Lisp invocations of the underlying C code that supported them in previous versions of the system. The inheritance mechanism permits an application to add the specific intelligence it needs. Sub-classing document objects leverages on the fact that documents are a very general paradigm of communication.

A document is a heterogeneous collection of doc-objs. Due to the logical complexity needed to support a structured, revisable, WYSIWYG, formatted collection of objects, the traversal between doc-objs differs depending on the perspective of interest to the programmer. These perspectives are referred to by optional keywords in the navigation methods. Two important perspectives include *structure* and *layout.*

In the structure perspective, objects are related according to their logical dependencies: headings precede paragraphs, graphics are contained within the token that anchor them in the text flow. This is similar to how these kinds of structures are described in declarative markup languages like Scribe or LaTeX.

In the layout perspective, the relationships reflect the appearance of the printed page: pages contain columns, and columns contain lines of text or floating graphics. Normally, these objects are created and linked automatically by the composition rules built into the software. However, by giving programmers access to this information, it is possible to add composition-sensitive extensions to the editor or to the layout itself.

More than one perspective is needed because some doc-objs have meaning in multiple perspectives. For example, markers can be viewed either as being contained within a paragraph or within a line on the page; floating graphics are treated both as containers for graphics and as layout elements on a page. Thus, it is possible for Lisp programs to concern themselves as little or as much as needed with the structure or the layout, separately or in concert.

4 User Interface Configurability

Interleaf software usually provides a uniform user interface (UI) across multiple platforms, but this interface is not suitable for all users (for example, occasional MacIntosh users might prefer a more familiar "Mac-like" UI). Further, since the document processing system itself is extensible, we need to allow the UI to extend to new objects. Finally, where we seek to improve the standard user interface, we need to provide experienced users with the ability to retain a familiar interface if they wish, or to otherwise tailor it to their own needs.

These requirements are met by exposing the user interface to Lisp programs in much the same way as are document objects. In fact, document objects can *become* part of the user interface. This permits the full power of the graphics and text editing facility already in the software to be invoked to do the graphics and text parts of the interface. Lisp permits the run-time reconfiguration of the user interface, from its "look" (the graphical representation of its paradigms) to its "feel" (the actions it supports) to the text of its messages.

More than any other extension features described in this paper, Interleaf UI extensibility may be seen as evolutionary from earlier versions of the system. Release 3.0 provided a desktop "Create" popup menu. Adding files to the desktop Create cabinet caused these file names to be added to the menu. Selecting one of these names caused the file to be copied to form a new instantiation. Release 4.0 added a similar desktop-based extensible "Custom" menu. But files in the Custom cabinet contained Lisp programs to be executed upon selection. These programs had access to some internal state, but no access to objects within a document. Their effect was limited to interaction with the desktop and the underlying operating system, and to choosing from among fixed UI alternatives.

We next describe how Lisp programs access the user interface. The UI of any contemporary interactive document processing system may be said to have several components: the input event handlers, window managers, and display managers. At the lowest level, these are often handled by general software that is not of concern to us here. At the next level of abstraction, the user interface comprises things such as keyboard mapping and the menu system. It is these that we have opened to Lisp.

Keystrokes are mapped to Lisp actions, thus supporting an arbitrary collection of "keyboard macros" much as Emacs or many PC software products do. As with Emacs, these actions can be arbitrary procedures and potentially have access to entire documents and more.

For example, Figure 1 contains a key binding, the invocation of which "tells" the current insertion point to insert the text output of a date routine:

```
(kbd–define–key doc–kbd–map (fcn–key "F1")
  '(tell (doc–point–marker) mid:insert (get–date :usa–format)) )
```

Figure 1: Defining a key to insert the current date

The extensibility of the menu system provides more novel possibilities. For example, property sheets can be implemented as a special case of active documents. An ordinary document with graphics looking like "buttons," "sliders" or other "widgets" can have Lisp procedures tied to the graphics objects. Selecting these objects invokes the procedure and the desired action takes place. Even dialog in such sheets can be parsed by the Lisp program.

Interrupt messages, which post a dialog box with which the user must deal before continuing, have simple access from Lisp. The corresponding objects have text properties (the message text) and a list of buttons. When a button is selected, the button object is returned to the Lisp caller, which then executes the procedure that is mapped to that button. This procedure need not be specified at compile time, although for pre-existing dialogs the default binding is simply the C code that previously was called when the process was controlled in C (we say more about our C to Lisp interface below).

We plan ultimately to recode the entire UI using doc-objects, including graphics objects. Some of this has already been done. In release 5.0, the entire UI will be exposed to Lisp to facilitate our, or third-party, replacement of the UI with doc-obj based interfaces. This could be used to change the UI to one that behaves and looks like any of a number of popular user interface paradigms.

5 Lisp Environment

With one exception, the Lisp interpreter we implemented has no features unique to document processing environments. The exception arises from the requirement to deal with the 16-bit characters required in Interleaf documents to support large multi-language character sets. These characters are externally represented with 8-bit character sequences, but within Lisp strings they are not treated differently from 8-bit characters. Lisp symbols are restricted to 8 bits to promote portability of the interpreter (the software runs on a wide variety of platforms and operating systems, including several variants of Unix, DOS, VAX VMS, and IBM mainframe operating systems).

To assist migration from C to Lisp implementations, the Lisp system provides in-house programmers with the ability to call C from Lisp and Lisp from C. The principal goal of this arrangement is to facilitate an object-oriented interface to existing C code and data structures. Lisp programs need not distinguish between data and procedures defined in the underlying C code and those defined purely in Lisp, and thus they may be freely mixed. We do not provide arrangements whereby Lisp programs can call C code not defined at compile time because not all platforms support the required dynamic linking of the resulting executable code. However, below we describe supported interprocess communication methods that permit our system to become part of composite environments with no advance arrangements required between Interleaf and the authors of the other pieces of the environment. In such environments, Lisp code can be used to encapsulate IPC calls to external code written in any language environment.

As with many other Lisp systems, our Lisp objects have documentation strings among their properties, intended as on-line documentation for Lisp programmers. These strings form the data for an interactive inquiry system about the definitions and values of Lisp objects, classes, methods, functions, and symbols. Thus, arbitrary Lisp programs can be self-documenting.

6 Active Documents as Generalizations

Carrying methods with the document is very powerful. Permitting these methods to be *defined* at document execution time potentially decouples the constraints on the document architecture from those decreed by the system architects, or even by the original document architect.

Hypertext is easily implemented on such a substrate. Links are simply the execution of programs based on events, such as selection, that cause the editor to move to other objects. Since Lisp programs are data, it is straightforward to make a link editing mechanism that has the effect of generating (invisibly to the user) those links. Indeed, *hypermedia*, not merely hypertext, can be supported with no further requirements. For example, if the underlying environment supports voice I/O, objects in a suitably designed document can invoke this mechanism as desired (see "Interprocess Communication" below).

The notion of an independent filter or other document conversion program is not inherently necessary. An active document that contains a reference to a document in an alien format can carry with it as much or as little as needed of the method for interpreting that format. The user of such a document need have no concern for the existence of conversion software, nor, if the document design itself is adequate, even be aware that the object *is* in an alien format. Further, as we describe below, the document objects need not carry the actual interpretation software. Instead, they can carry a method for *accepting* or *requesting* the interpretation procedures when they are needed. Or document objects can identify themselves as a certain class of object for which methods are already known.

An object-oriented system with procedures as data and with run-time binding makes it feasible to use a document processing substrate to solve problems for which advance knowledge is unnecessary on the part of the system designers. Indeed, *document* architects can even free themselves of some of these details by making arrangements to receive them at execution time.

7 Interprocess Communication

Some applications require a document to communicate with external processes. We have provided three methods for this. The simplest of these—file system event handling—is generally useful on all supported platforms.

The Lisp *load* primitive permits an arbitrary ASCII file containing Lisp to be read and interpreted. *read* and *write* primitives permit uninterpreted stream I/O. These might be done on events such as opening or printing the document, selecting an object, or any thing else that causes a Lisp procedure to execute. The modification time of a file (as well as all other file primitives useful in a programming environment) is available, so a procedure that wished to implement an invisible update or change could do so.

When the system runs interactively, the outermost event control mechanisms examine timer objects, which can be created by a Lisp procedure with arguments detailing a timeout period, and a Lisp expression to be evaluated at the timeout. Using this mechanism, document objects can schedule their own responses to external events by creating such timers.

On platforms that support it, we have also demonstrated communication with sub-processes through their own return mechanisms and communications elsewhere on the network with TCP/IP sockets. Lisp code can use IPC to communicate with a remote process, thereby adding its intelligence to the document. In the opposite direction, a Lisp *read-eval-print* loop is offered as the top-level IPC protocol, allowing the active document system to be programmatically directed from the outside. A particularly simple, but quite useful application of this mechanism is to induce the remote process to produce Encapsulated PostScript, which our underlying software knows how to display.

8 Applications

A number of interesting and important applications have been developed both in-house and externally, and we describe some of them now. Two things are especially noteworthy. First, the architecture described here supports a very broad notion of documents and even permits use of the software as *both* a back end (i.e., as a report generator) and a front end (i.e., as the user interface) to virtually any kind of software, such as data base, spreadsheet, CAD-CAM systems, or even on-line help for the system itself. We describe below a very powerful production quality, forms-management system written by the Amoco Corporation with Interleaf's assistance. Second, Lisp turns out to be a language that is easily learned by non-programmers. We describe a document-based UI editor written in Lisp by a user with minimal previous programming experience, and also some smaller but interesting applications written in-house. Finally, we sketch a document-based online help system for the end user.

Forms management. The Amoco application is a combined inventory, ordering, and documentation system for handling oil exploration equipment. It is time and cost critical that these complex equipment orders are correct for a given project. Amoco could find excellent forms-generation software to support their input design, database software to verify their constraints, and publishing software to publish purchase orders and reports. But there was no way to make these pieces cooperate invisibly to the user. Since the constraints could be programmatically specified, the solution possible with active documents is to make the form itself do the constraint computation and generate the report. This system was prototyped in three weeks and is now in production use. Because much oil exploration is cooperative, Amoco is now seeking to make the system an industry-wide solution.

User Interfaces. The popup menu system is a hierarchical collection of context-dependent menus that appear in response to particular mouse or key events. Access to this by Lisp programs permits each node in the tree to be represented by a Lisp object which, as always, can be either procedure or data—there being no essential difference in Lisp. It is possible, however, to know whether the Lisp object represents a method or a doc-obj, and when the selection is made (typically by releasing the mouse button), the popup system traverses the selection path back up the hierarchy until it encounters a method. That method is then applied with the selected child list as argument. For example, suppose the selection yielded the list "Create→Character→Math→∫." In this case, the calling Lisp program would find a method only when it reached the "Create" Lisp object, and would invoke that method on a list containing "Character," "Math," and "∫," resulting in the insertion of the integral sign at the current point.

It is incidental that the application of the create method is induced by a popup event. If it is also desired to have a keyboard event insert an integral sign, no different program is required, only a different invocation event. More importantly, in an active document such programs can be stored *with the document objects*, and these programs can be altered at run-time. Imagine this kind of program attached to a document object only a little more complex, say an entire integral expression. That program could be replaced *at run time* with, for example, a program that sends the expression to a symbolic mathematics processor for symbolic integration in much the same way that the CaminoReal software might (Arn88).

An example of the utility of traditional documents as a front end can be mentioned here. Using the popup interface described above, an Interleaf quality assurance engineer constructed a simple but useful popup editor that is manipulated as an ordinary document. The user specifies a portion of the popup tree for editing simply by inducing but not invoking it (in this case, pressing a mouse button and making appropriate mouse motion, but not releasing the button). Then, the user presses a keyboard function key, temporarily bound to "record-the-popup-tree." This causes the program to construct a text representation of the current popup tree in an ordinary document. Then a special subset of editing capabilities can be used to manipulate that tree graphically to add, delete, or rearrange nodes. Finally, the Lisp program interprets the new text tree as a popup tree and re-inserts it into *the running user interface.* In this case, the text tree document is the active document. Its content is far less important than its interpretation as a representation of a user interface component. While this scheme is roughly limited to rearrangement of existing UI pieces, such rearrangement is a task one might want to provide to users who want some UI control but do not want to learn to program to make new paradigms. Moreover, implementing this editor in Lisp took the non-programmer about a week. (We estimate that experienced Lisp programmers could accomplish it in a few hours.) This level of productivity was possible because the editing substrate was already in place and all that was necessary was to interpret a document as a UI specification in ways that *produced* the UI. Finally, this example also shows the commonality between menu arrangements and keyboard arrangements. Rearranging the popup tree is not essentially different from re-mapping keyboards.

Miscellaneous applications. Another inexperienced in-house user—a marketing writer—wrote several simple but useful programs for demonstration and personal use. One of these permits the filling in of expense forms, with appropriate minor arithmetic done by the form. In addition, filling in such a form has the side effect of updating *another* document that keeps running expenses and can generate an explanatory memo if the expenses go over budget.

A more utilitarian application is a primitive outline processor that rebinds the key normally bound to "create component of the same type." If the current component is given a particular attribute, this binding instead creates an arbitrary sequence of components of types specified by its argument list. This twenty-line program was written in about fifteen minutes, using a small library of general string manipulation procedures the user had already written to deal with attribute values (see above discussion of effectivity control).

Using the same string handling facility, the user has demonstrated documents that examine the environment for the name of the user attempting to open them, then compare that name to a "security list" in another document. Depending on the data found in the security list, the document hides some or perhaps all of its sub-objects by comparing attributes in the document to security clearances the user has in the security list.

Complex content from external objects. Using a Lisp program that interprets Lotus 1-2-3 spreadsheet files and returns lists of cell values, a demonstration program was written that correctly maintains, in a single document, three different views of the data. The demonstration displays the manufacturing costs of a bicycle. In one view, the data are presented in a table, in another in a pie chart, and in a third as line drawings of the bicycle parts, with their sizes proportional to their costs. The objects can be told to update themselves at document open time, on a menu event, or when the spreadsheet changes. Simplified key fragments of this program are shown below, in Figure 2.

```
(defun import–contents–of–diagram (diagram)
  (let (  (data–editor (doc–get–attr–value diagram "data–editor"))
          (data–file (doc–get–attr–value diagram "data–file"))
          (data–filter (doc–get–attr–value diagram "data–filter"))
          cell–list cost–list )

    ;; The data editor, file name and filter name have been
    ;;    read from the object. Now run the program
    ;;    named by data–editor on the file named by data–file...
    (proc–wait (proc–create data–editor data–file))

    ;; External edit completed and data–file rewritten,
    ;;  so parse it with function named by data–filter...
    (setq cell–list (funcall (find–symbol data–filter) data–file))

    ;; Now cell–list reflects the spreadsheet.
    ;; Get the data from the row "TOTAL:"...
    (setq cost–list (make–cost–list cell–list "TOTAL:"))

    ;; Update the various doc–obj representations of data–file...
    (rebuild–chart (get–first–chart diagram) cost–list)
    (rebuild–table (get–first–table diagram) cost–list)
    (rebuild–graphic (get–first–graphic diagram) cost–list)
    (tell diagram mid:draw)))  ;Redisplay revised diagram

(defun get–first–chart (diagram)
  (let ((chart (tell diagram mid:get–child :along :structure)))
    (while (and chart (not (typep chart 'dg–chart)))
      (setq chart (tell chart mid:get–next :along :structure)))
    chart))

(defun rebuild–chart (chart cost–list)
  (let ((row 0) value)
    (while cost–list
      (setq value (cost–of–row (pop cost–list)))
      (tell chart mid:set–prop :data (list (list row 0 value )))
      (setq row (1+ row)))))
```

Figure 2: Lisp document object program to import a diagram

In this example, *data–editor* was the program name "123," *data–file* was "bicycle.wk3," the name of the spreadsheet file to be associated with this diagram, and *data–filter* was "wk3–to–list," the name of a Lisp function used to parse a Lotus wk3 file into a Lisp list. All are stored in the document as the values of the corresponding user-defined attributes of the diagram. The edit method for the diagram is bound to this procedure, so that the normal process of selecting and editing it actually results in the external software being invoked, followed by an interpretation in the document of the result. In the navigation procedure *get–first–chart*, the keywords *:along :structure* in fact denote the default perspective for diagram objects and could be omitted.

9 Implementation Experiences

We feel that most of the engineering challenges we encountered were not specific to document processing systems. They included extending the definition of important C data structures such that they can be manipulated by Lisp, and making the object system work with C, with Lisp, and with combinations of C and Lisp.

Enabling programs to change the behavior of time-critical functionality without incurring interpreter-related performance penalties was an important goal. Decisions about the granularity with which to expose internal methods and objects to Lisp were driven by engineering time constraints and the requirement to maintain current performance levels. We may need to reduce this granularity (giving more detailed access) as users' experiences give us more insight into what their Lisp programs require.

The problem most centrally related to documents is how to store Lisp data in documents that survives editing sessions. Because some of our document objects are ephemeral, only living at document edit time, the associated Lisp cannot be saved, but must be regenerated at load time.

10 Summary

We have implemented a runtime-extensible, object-oriented system for describing and executing active documents. This system sits on a substrate comprising a powerful, high performance, structured document editor and an open object architecture with an application programming interface with sufficient power to permit easy participation in cooperative software environments. The programming interface can be invoked as a side-effect of opening a document or selecting an object or almost any other event. The object-orientation facilitates application programming in which mechanisms are hidden from end users, who need only know what the objects are—the application takes care of what they do. Because document construction, editing, and interpretation are, first of all, symbol manipulation problems, Lisp is an especially suitable language for rapid, powerful implementation of a wide range of facilities.

References

(Arno88) Dennis Arnon, Richard Beach, Kevin McIsaac, and Carl Waldspurger, "Camino-Real: An Interactive Mathematical Notebook," *Document Manipulation and Typography,* Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography, Nice (France) April 20-22, 1988, Cambridge University Press, 1988, pp. 1-18.

(Cal88) *Markup Requirements and Generic Style Specification for Electronic Printed Output and Exchange of Text,* Military Specification MIL-D-28000, CALS Policy Office, DASD(S) CALS, Pentagon, Room 2B322, Washington, D.C., 1988.

(Cha88) Donald D. Chamberlin, Helmut F. Hasselmeier, and Dieter P. Paris, "Defining Document Styles For WYSIWYG Processing," *Document Manipulation and Typography,* Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography, Nice (France) April 20-22, 1988, Cambridge University Press, 1988, pp. 121-138.

(Ils88) Richard Ilson, "Interactive Effectivity Control: Design and Applications,"*Proceedings of the ACM Conference on Document Processing Systems, December 5-9, 1988 Santa Fe, New Mexico,* ACM Press, 1988, pp. 85-92.

(Mey88) Betrand Meyer, *Object-oriented Software Construction,* Hemel Hempstead, Herts., England, and Englewood Cliffs, N.J, U.S.A., 1988.

(Sta81) Richard M. Stallman, "Emacs, The Extensible, Customizable Self-Documenting Display Editor," *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, Portland Oregon, June 8-10, 1981,* SIGPLAN Notices v. 16, no. 6, June, 1981, pp. 147-56.

(Tow88) George Towner, "Auto-Updating as a Technical Documentation Tool," *Proceedings of the ACM Conference on Document Processing Systems, December 5-9, 1988 Santa Fe, New Mexico,* ACM Press, 1988, pp. 31-36.

(Wal81) Janet H. Walker, "The Document Editor: A Support Environment for Preparing Technical Documents," *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation, Portland Oregon, June 8-10, 1981,* SIGPLAN Notices v. 16, no. 6, June, 1981, pp. 44-50.

(Zel88) Polle T. Zellweger, "Active Paths Through Multimedia Documents," *Document Manipulation and Typography,* Proceedings of the International Conference on Electronic Publishing, Document Manipulation and Typography, Nice (France) April 20-22, 1988, Cambridge University Press, 1988, pp.19-34.