UNIX Emacs: A Retrospective

Lessons for Flexible System Design

Nathaniel S. Borenstein
Information Technology Center
and Computer Science Department
Carnegie-Mellon University

James Gosling
Sun Microsystems, Inc.

Abstract

UNIX Emacs is well-known and widely used as a text editor that has been extended in a remarkable number of directions, not always wisely. Because it is programmable in a powerful yet simple programming language, Emacs has been used as a development tool for the construction of some remarkably complex user-oriented programs. Indeed, it has served as both a user interface management system and a user interface toolkit, though it was designed as neither. In this paper, we discuss the features that have made it so popular for user interface development, in an attempt to derive lessons of value for more powerful and more systematically designed systems in the future.

I. Introduction

Designing generalized tools for user interface design and user interface management systems are now subjects of widespread interest. In order to better understand what such systems should do in the future, it helps to have a clear understanding of what has come before. Although many systems have been explicitly designed to serve these purposes, few have been very widely used.

UNIX Emacs [1] is a popular text editor with a powerful extension facility, that has developed a large and extremely satisfied user community. The extension facility has allowed Emacs to be used, not merely as a text editor, but as a general testbed for user interface design, and as a user interface management system (UIMS). Of course, Emacs was designed neither as a testbed nor a UIMS, but simply as a text editor. That it was flexible enough to be extended so far afield offers us the opportunity to learn from its example, in both positive and negative ways.

In this paper, we will briefly describe Emacs and its uses. We will then try to describe the most important features of Emacs that have contributed to its success, and also to recount the problems that have been most frustrating to serious Emacs users. From this overview, we will then offer suggestions that may be of use to the developers of future extensible systems and testbeds.

This paper is an informal reflection of many years of experience with Emacs. One of the authors (Gosling) is the author of UNIX Emacs and of many extension packages for it. The other (Borenstein) is the author of several of the largest UNIX Emacs applications [2, 3]. In addition to personal experience, this paper also reflects the results of a survey, conducted over the ARPA Internet, of several dozen Emacs programmers. It is unabashedly anecdotal, and the authors make no pretense of having conducted systematic studies to prove our conclusions. Rather, we are simply attempting to convey the lessons we feel can be learned from the Emacs experience.

II. What is Emacs?

The name "Emacs" has become a generic term, referring to an entire family of powerful, extensible text editors, beginning with the original ITS/Tops-20 Emacs [4]. That family has grown and matured to the point where it now includes a wide variety of programs that resemble their predecessors to greater or lesser degree, as surveyed by Stallman [5]. The present paper discusses one of the most widely-used versions of Emacs, known commonly as "UNIX Emacs", but also available for other operating systems, and occasionally referred to as "Gosling Emacs." In this paper, the term "UNIX Emacs" will be applied rather freely to both the early non-commercial and the more recent commercial versions of the program.

The most powerful versions of Emacs are all *extensible* in some extension language. ITS Emacs was extensible in TECO, an extremely baroque but powerful language in which Emacs itself was implemented. Despite the difficulty of programming in TECO, a wide variety of extension packages, including mail and bulletin board readers, have been implemented for that version of Emacs.

UNIX Emacs offered two major innovations over its predecessors. First, because it was implemented under UNIX, it included powerful process management facilities, allowing, for example, text to be filtered through arbitrary programs, and also allowing different programs to be run in separate windows. Second, though UNIX Emacs is implemented in C, it is extensible in a LISP-like language known as *mlisp* or *Mock Lisp*. The decision to use a different language for extension than the one used for implementation was a crucial one; it appears responsible for most of the best and worst aspects of UNIX Emacs as perceived by those who have used it most extensively. A discussion of the virtues and flaws of mlisp will comprise the bulk of this paper. A few brief examples, however, will serve to introduce the unfamiliar reader to the flavor of the language.

The statement

```
(defun (change-presidents
(replace-string "Reagan"
"Mondale")))
```

defines a new function, "change-presidents" that will, when called, replace any occurrence of the word "Reagan" appearing after the cursor position in the current document into the word "Mondale". For a more complex example, the next program will also make sure that the new word "Mondale" is alone and indented on a line. Comments begin with semi-colons and are italicized here.

```
(defun
  (change-and-indent-presidents
    (save-excursion ; Save
context & restore when done
      (while
        (! (error-occurred ; Keep
looping until error
          (search-forward
"Reagan")
          (delete-previous-word)
          (delete-white-space) ;
Delete surrounding spaces
          (if (! (bolp))
(newline))
        ; Insert newline if not at
beginning
          (if (! (eolp))
(newline-and-backup))
        ; Similar for end of line
          (insert-string "
Mondale")))))))
```

Mlisp is used for programs ranging from simple but repetitive editing tasks to elaborate user interface or even database systems. Dozens of different Emacs programs have been written to manage electronic communication (mail, bulletin boards, UNIX netnews, etc.). Others have implemented spreadsheets, animation, database access, specialized program editing features, and even a BASIC interpreter. The longest Emacs program known to the authors is about 3000 lines of source code, defining approximately 150 separate functions. Clearly the user community regards mlisp as a programming language powerful enough to do a wide range of tasks beyond those related to simply editing text.

III. What is Good about UNIX Emacs and mlisp?

Why have so many application programs, particularly user-oriented ones, been written in UNIX Emacs, using mlisp, when so many other languages are available to the UNIX programmer? In an attempt to answer this question, we distributed a questionnaire to Emacs programmers via the "UNIX-EMACS" mailing list on the ARPA Internet and the UNIX Usenet. We asked only a dozen questions, mostly open-ended ones like "If you were designing an editor extension language, what might you do that is totally different from mlisp? What would you most particularly want to keep the same?" Questions like these yield very interesting and illuminating answers that are, unfortunately, extremely difficult to tabulate meaningfully. The summary below represents the aggregate of the responses in the survey, with no pretense of completeness or of meaningful quantitative data.

Extensibility

The one point on which nearly everyone seems to agree is that the best thing about UNIX Emacs is that mlisp exists. No one even suggested that Emacs would be better without the powerful facility for extension and customization, although it is clear that such facilities, if abused, can create an environment that changes so radically from day to day as to be unusable for most people. Donner and Notkin [6] discuss extension mechanisms in general and offer several convincing arguments for their desirability; the experience of Emacs users suggests that this desirability should be taken as a given. The integration of powerful editing facilities with a full-fledged programming language seems unquestionably useful, though it remains a matter of some debate whether this combination is best achieved by an extension language for a text editor or by implementing the text editor as a subroutine library in an existing programming language. Our experience suggests, in fact, that such a debate is unresolvable, because such mechanisms serve rather different purposes, and are *both* highly desirable.

Simplicity

Nearly equally unanimous is the perception that mlisp is a very simple and elegant language, well-suited to its intended application domain (editing text). Our survey revealed that mlisp is generally perceived to be an unusually easy language to learn and to use, perhaps partly because it omitted complex features crucial to system programming. The simplicity of mlisp will be a recurring theme in discussions of both its virtues and failings in the sections that follow.

A significant part of mlisp's simplicity stems from the implicit context available to a host of mlisp commands. For example, the mlisp function *(forward-character)* moves the cursor forward one character in the current window or buffer. The implicit context of this command is the name of the current window, its contents, and the current position within that window. A more powerful language might allow such commands to operate in a wider set of contexts, but might also sacrifice simplicity in favor of more general operations with more parameters. Mlisp's functions seem to be well-chosen to act reasonably in most contexts. The key principle is that there is *simple syntax for simple operations.* This does not mean that a more complex syntax should not be available for more complex operations, only that the complexity should not be forced on the programmer in a simple context.

Clearly there are other approaches to simplified syntax. Another useful approach within the LISP world is the optional argument syntax permitted by Common LISP, which also permits easy use of defaults while allowing more complex usage when necessary.

Abstraction

A major factor for many who have used mlisp is the level of abstraction it provides. The language has a built-in model of text and the operations that may be performed on text (that is, a model of text editing) that is considerably more abstract than that of most other languages.

To begin with, mlisp views text two-dimensionally, as a series of lines of arbitrary length. It thus provides primitive functions such as next-line and previous-line, as well as forward-character and backward-character. This two-dimensional view tremendously simplifies code that needs to parse small pieces of text in random parts of a file, a situation frequently encountered in such applications as spreadsheets and electronic mail.

Mlisp also provides strings as one of its basic data types. Users are entirely freed from consideration of string length and storage management, and fast primitives are provided for taking portions of strings. Of course, one can argue that strings should be taken for granted as a basic data type in any reasonable programming language, but the notion is still rather radical in the UNIX/C community.

For more complex operations, Emacs provides text "buffers" in which text can be inserted and then manipulated with the full complement of Emacs commands, including regular expression searching which can be used to implement complex parsing quite painlessly -- that is, without attention to the details of breaking words into tokens and similar low-level actions.

Also popular with users as an abstraction tool is the Emacs argument passing mechanism, which allows each procedure to be called with a variable number of arguments. A procedure calls the "nargs" function to find out how many arguments it was given, and then can ask for each argument by number. Especially popular is the feature that allows functions to take an argument if it is given, or else ask the user to supply it if the function was called interactively, without the called function ever having to know which of these two alternatives actually occurred. (It should be noted, however, that the parameter passing conventions are also one of the least-liked aspects of mlisp; it seems that the syntax and basic model are popular, but the semantics of parameter passing are almost universally despised. This is discussed in a later section.)

Other users cited the abstract notion of "word" as a major virtue of mlisp. Although mlisp does not consider words as a type distinguishable from strings in general, it does provide several primitive functions manipulating objects as "words" (e.g. forward-word and backward-word). The definition of "word" is provided by a buffer-specific syntax table, which (among other things) defines which characters can be part of words and which characters delimit words. The notion of the syntax table is extremely popular, although the interface to the syntax table is not so well-liked.

Another abstraction tool provided by Emacs is a set of several "save-state" functions. These functions allow the user to save the some portion of the current state of the editor, execute some arbitrary amount of mlisp code (which may include recursive edits, and hence an arbitrary amount of interaction with the user) and then return to the previous state. These functions are used by nearly every major application package, although there are some complaints about a few details of the implementation of the functions.

Some of those surveyed went so far as to suggest that mlisp should be made a more strongly typed language, with basic types such as "window," "file," and "paragraph". Others, the authors included, would prefer not to be troubled with type conversions between, for example, paragraph and word. Implementing some of the Emacs abstractions such as "windows" as data types would be a dangerous undertaking, because the syntactic complications might outweigh the proven value of the basic abstractions. At the other extreme, mlisp can be viewed as taking a step in the direction of *dynamic typing*. In an ideal dynamically-typed system, however, the interpreter or run-time system would detect inconsistencies in dynamically-typed objects, which mlisp does not.

Development Cycle

Another commonly-cited reason for developing user interfaces in mlisp and Emacs is the speed of the development cycle. Because mlisp is interpreted, it is simple to write and debug a procedure at a time, without ever waiting for a compiler and a linker; in essence, mlisp provides instant, integrated compilation and linking. As is well-known from other interpreted environments such as LISPs, this short write-test-debug cycle is not only more satisfying for the programmer, it also encourages the kind of small changes that tend to be so particularly important in user interface design (e.g. "Wouldn't this look better a little further to the right?")

Another key notion in the design of mlisp is that the programmer has access to the same functionality that the user has. If a user can bind a key to a function, so too a programmer can invoke that function. This creates an extremely simple model of learning to program, "programming by example". This feature (which was not included in the earlier releases) allows users to type a series of keystrokes to perform a task, and converts the keystrokes to mlisp commands automatically. Thus, entire first drafts, at least, of mlisp programs can be written simply by "walking through the program" by hand, using normal Emacs commands. Several users suggested that this process should be generalized: user interfaces could first be developed by this walking-through process, then debugged in mlisp with the interactive development cycle just mentioned, and finally translated to C for fast execution of finished programs. While it is unlikely that anyone will ever expand mlisp to do this, current work with dynamically linked languages [7,9] may yield the same advantages in the long run.

Window and Process Management

One of the reasons cited most often for the use of UNIX Emacs as a working environment is its ability to function as a window manager, even on relatively unsophisticated display terminals. This ability is enough to convince many UNIX users to do all their work within Emacs, using terminals for which other window managers are not typically available. More important, the window management features are a boon to programmers of user interface applications. The ACRONYM help system, for example [2], responded to a user typing a question mark in the shell window by placing help text in a second window and a menu of further help topics in a third window. Windowing is extremely time-consuming to code from scratch in a language such as C, but is built-in and trivial for the mlisp programmer.

Our survey suggested that the window and process management features combine to make Emacs a tool uniquely suited to the integration of systems from diverse components. Because Emacs treats both processes and windows at a high level of abstraction, it is relatively easy to write a short mlisp program that passes messages back and forth between, for example, a C program in one window and a Lisp program in another, all under the control of a user communicating with the mlisp program. Many respondents indicated that they had used mlisp for such integrating purposes. Typically, an application might have its most computationally intensive or low-level parts coded in C, with integrating code and a user front end written in mlisp.

Obviously, the window management facilities in Emacs are simply not comparable to those available in a modern window manager [10,11]. Nonetheless, most such window managers do not allow anywhere nearly as simple a manner of access to the window facilities as was made possible in Emacs via mlisp, and would benefit greatly by such a mechanism.

Miscellaneous Virtues

Several other mlisp features were frequently mentioned by respondents to our survey. Many expressed a fondness for the "execute-mlisp-line" command, which allows an mlisp program to put together a line of mlisp code itself and then execute it. Others cited the help commands, which can be used to find, for example, all defined functions including the word "string". Command completion, whereby Emacs automatically completes file names and mlisp function names, is also well-liked. The "undo" command, which can undo the effect of nearly any operation not involving file manipulation, is predictably popular. Regular expressions, which facilitate

complex conditional text searches and replacements, are extremely popular, although the syntax used is less popular.

IV. The Down Side

Despite all the useful features, UNIX Emacs is, in fact, far from paradise for the user interface designer. Most of the problems fall in the general category of "implementation failings" -- either program bugs or features that, in retrospect, should clearly work differently. Many such failings were mentioned by respondents to our survey, and will be mentioned here only briefly. A more critical set of problems pertains to inadequate integration of mlisp into the wider UNIX world; these can not fairly be categorized as Emacs implementation failings because they reflect the entire basic structure and relationship of UNIX and Emacs. Finally, many of those surveyed complained of specific features that are simply missing from the mlisp language.

Implementation Failings

Most of the implementation failings in mlisp can be traced directly to a simple misconception in its design: mlisp was "only" an editor extension language, so it was unclear that it needed to be a complete programming language. This is probably the most important lesson to be learned from mlisp: extension languages are real languages, not toys.

The most commonly cited implementation failing is simply that mlisp is too slow. Mlisp is an interpreted language; the so-called mlisp compiler merely translates tokens into a byte code for faster interpretation. The interpretive nature of mlisp makes possible the rapid development cycle discussed above, but most users would dearly love to see the interpreter supplemented by a true compiler to allow completed applications to run faster. However, it would be difficult to implement a real compiler without making the language more complex, which would be a shame.

Another problem with mlisp is its inability to run asynchronously. While an mlisp program can manipulate a large number of active asynchronous processes, none of them are mlisp processes; only one mlisp process can run at once. Worse yet, that process is uninterruptable, making an infinite loop a programming disaster. Various version of Emacs have allowed asynchronous mlisp processes, but these have typically depended on variants of UNIX and could not be supported widely.

One part of mlisp that has been almost universally condemned is its argument passing and variable scoping. These are simply poorly thought out, and can often lead to horrible bugs in which one routine resets another's variables. The variable passing and naming schemes are so bad that they make recursion almost impossible in mlisp programs. Basically, they were designed to have the same semantics as macro expansion, which turned out to be only rarely desirable in parameter passing. (However, they did permit a correct implementation of the "case" statement as an mlisp procedure.)

Mlisp also suffers from poor debugging and version control facilities. The few functions supporting mlisp debugging were added to Emacs rather late in its life, and were never completely debugged themselves. It is also difficult to insure that several parts of a complex mlisp program are mutually compatible, a significant problem for programs dependent on functions from the Emacs mlisp library.

Respondents to our survey also complained about several administrative problems, including the inadequacy of the Emacs help database and written documentation, the utter lack of a tutorial introduction, and the cluttered state of the mlisp library distributed with the program. Finally, a few genuine bugs were reported, primarily related to the overflow of certain internal Emacs parameters.

Integration Failings

Many of the respondents to our survey had complaints that can be described as failures of integration. Typically, users complained of the difficulty or slowness of getting access, in an mlisp program, to the UNIX system library, the macro preprocessor, low-level devices, or the internals of built-in mlisp primitives such as forward-character. They also complained that several aspects of Emacs' internal state are made difficult or impossible for the mlisp programmer to inspect or modify; examples of these include physical screen positions, the menu system, storage management, key binding state information, and text search state information.

These failings strongly reflect the fact that Emacs is not written in mlisp; the whole point of mlisp is to provide a simpler language for extending the editor, with the almost inevitable result that some things would not be available from within mlisp. It is not at all clear what the right solution to these problems would be; if Emacs were extensible in C, its implementation language, it would not have such problems, but would be far less convenient for most users and applications.

The integration failings also reflect the fact that Emacs suffers from the "design by accretion" syndrome. During the early stages of its life it

was possible to make wide sweeping changes when some new feature required such changes to fit smoothly. As Emacs became more popular, it also became more entrapped by its own history. Changes became impossible because there were too many users and too much mlisp code that depended on the status quo.

User Interface Failings

It should also be noted that, although Emacs has proved to be widely popular, there remains a very substantial set of people who have seen Emacs, tried using it, and violently despise it. There are many reasons for such a reaction, but the most common one is discomfort with the general user interface paradigm. The standard joke among such people is to extend one's hand to another, as if to shake hands, but to hold the hand twisted and contorted into as unlikely a position as possible, and say, "Hi, pleased to meet you, I'm an Emacs user." Such people find Emacs' reliance on the CTRL key and ESC- or ^X- prefixes to be overly confusing and difficult to learn.

Indeed, studies of text editor performance [12,13] indicate that it is not too difficult to build an editor with a more easily learned set of keyboard commands, with no sacrifice in expert performance levels. Clearly, Emacs has succeeded *in spite of* its command syntax rather than because of it, although a surprising number of people have come to regard it as *natural* to type CTRL-SHIFT-2 to set a mark in a buffer.

It didn't have to be that way. In the development of Emacs, it was far too easy to say that, since the basic interface was entirely customizable, it wasn't important to devote a lot of attention to getting the default interface to be correct. After all, anyone who didn't like the interface could change it. What this attitude ignored, however, was the old rule our mothers used to teach us about the lasting value of first impressions. Many people tried Emacs, couldn't stand the basic interface, and went back to their old standard editors, such as the vi editor on UNIX.

It is worth noting that this happened despite the fact that several people, independently, implemented more or less complete vi emulation packages for Emacs in mlisp. It seems that even if a package existed that was an absolutely perfect simulation of vi, most of the people who prefer vi to Emacs would stick with the clearly less powerful vi. While this is in part due to the smaller size of vi, it is also attributable to the unacceptable difficulty, in the minds of such people, of having to learn how to turn the vi emulation package on in the first place.

The clear lesson to be learned from this is that, while flexibility in such a system is indeed essential, great care must be paid to get the default behaviors to be simple and natural, and to provide very clear and simple ways to use a carefully chosen small set of the most commonly desired customizations. It may be, organizationally, that the people who build the flexible tools are not necessarily the right people to figure out which are the correct defaults and "standard options". The process of making such choices probably requires serious observation of users, or even controlled experiments, to compare possible system configurations.

Missing Features

Finally, a number of those surveyed suggested their favorite features that mlisp lacks. The danger of runaway featurism is amply demonstrated by languages such as PL/I and Ada. Still, it is undeniable that many of the features cited in the survey would be wonderful additions to mlisp: floating point arithmetic, complex data types, arrays, lists, case statements, graphics, fonts, underlining, highlighting, and pointers, and constants. Other, less clear-cut suggestions included vectors, infix arithmetic, support for "paragraphs" at the level now given to "words", structured document editing, alternate syntax (more like algorithmic languages such as Pascal), and multiple parenthesis types for easier parenthesis balancing.

In addition, it goes almost without saying that Emacs was written for a previous generation of computer technology; there is absolutely no support for the kind of sophisticated graphics people are coming to take for granted on a modern workstation.

V. Conclusions: Implications for Future Toolkits and User Interface Management Systems

UNIX Emacs has proven to be an extremely valuable tool for a wide variety of unintended purposes, most notably "quick-and-dirty" user interface design. The great virtue of Emacs is its extensibility; the great virtues of mlisp are its simplicity, its abstract, high-level view of text, strings, processes, and windows, and the quick development and prototyping cycle it facilitates. Its greatest failings are its lack of integration into UNIX as a whole, and specific features that reflect a design that did not recognize the need to make an editor extension language a "real" programming language. As tools are developed for rapid prototyping of user interfaces on the next generation of hardware, close attention to the success and failings of UNIX Emacs may make those tools more generally useful than they might otherwise be.

The biggest uncertainty in the minds of those surveyed, as well as the authors when the survey began, was whether or not mlisp should have been more than it is. That is, it was widely recognized that one of the great virtues of mlisp is its simplicity, and that one of the great failings of mlisp is in its lack of power for specific kinds of tasks. It is not clear whether or not these two facts can be separated. Can mlisp be made significantly more powerful without destroying its simplicity? This is an important topic for programming languages in general, but particularly so for user-oriented application programming, where rapid prototyping is most essential.

Later projects have pursued different strategies regarding the question of extension mechanisms in such systems. The original Andrew Base Editor [8], for example, retreated from the notion of a full-blown extension language like mlisp, choosing instead to provide the entire facility as a powerful subroutine library for the C programmer. This avoided problems of underpowered language and over-ambitious use of the extension mechanism, but at the cost of radically lengthening the process of prototyping and perfecting user interfaces. The most recent version of the Andrew Toolkit [9] improves this scheme substantially by allowing customization via dynamically loaded C programs, but this is still too cumbersome for casual customization purposes. It seems likely that the only "right" solution involves making the functionality of the system available at two levels: in the implementation language (e.g. C) for the most serious applications, which require high levels of reliability and performance, and in a simpler extension language for rapid prototyping and simple customization.

Acknowledgements

We would like to thank the dozens of mlisp programmers who took the time to respond to our survey. We would also like to thank Mike Kazar, Bruce Lucas, and several anonymous reviewers for their helpful comments on a previous draft of this paper.

Bibliography

[1] Gosling, James, "Unix Emacs", Carnegie-Mellon University Computer Science Department, 1981.

[2] Borenstein, Nathaniel S., "The Design and Evaluation of On-line Help Systems", Ph.D. Thesis, Carnegie-Mellon University, 1985.

[3] Borenstein, Nathaniel S., "The BAGS Message Management System", Carnegie-Mellon University Computer Science Department, 1985.

[4] Stallman, Richard M., "EMACS Manual for TOPS-20 Users", MIT AI Memo 556, 1981.

[5] Stallman, Richard M., "EMACS, the extensivle, cusotmizable self-documenting display editor", Proc. ACM SIGPLAN SIGOA symposium on text manipulation, Portland, Oregon, June, 1981.

[6] Donner, Marc, and David Notkin, 'Flexible Systems; Customization and Extension", to appear in IEEE Software.

[7] Kazar, Michael, "Camphor -- A Programming Language for Extensible Systems", Usenix Conference, 1985, Portland.

[8] Gosling, James, and David S. H. Rosenthal, "The User Interface Toolkit", in *Proceedings of PROTEXT I Conference*, 1984.

[9] Palay, et al., "The Andrew Toolkit: an Overview", Proceedings of the USENIX Technical Conference, February, 1988.

[10] NeWS Manual, Sun Microsystems, Inc., March, 1987.

[11] Scheifler, R. W., and J. Gettys, "The X Window System", ACM Transactions on Graphics 5(2) pp. 79-109, April, 1986.

[12] Roberts , Teresa, and Tom Moran, "The Evaluation of Text Editors: Methodology and Empirical Results", Communications of the ACM, April, 1983.

[13] Borenstein, Nathaniel S., "The Evaluation of Text Editors: A Critical Review of the Roberts and Moran Methodology Based on New Experiments", Proceedings of CHI '85 Conference, 1985.