Live Text

Christopher W. Fraser
Balachander Krishnamurthy

AT&T Bell Laboratories*
600 Mountain Avenue
Murray Hill, NJ 07974

May 8, 1990

Abstract

This paper describes software that allows the user to edit the output of several common software tools and to cause the changes to be written back to the input files. For example, it is possible to edit the output of a spelling checker and have the changes propagated back to the source files. This technique makes some corrections simpler and more direct. A trial implementation is embedded in `Emacs`.

Keywords: editor, shell, text processing, window manager.

Introduction

One view of screen editors is that changes to the display cause changes to a file. These editors remember where the displayed text came from, so they can write it back, either upon command or automatically at the end of the session and

*For electronic correspondence: `cwf@research.att.com` and `bala@research.att.com`.

possibly at checkpoints as well. If a program displays text, allows user edits, and then writes the result back to its source, then both the program and the displayed text will be termed *live*.

Many window managers[?, ?, ?] allow users to review, edit, and resubmit commands. Some allow editing of *all* displayed text, including output. These window managers may mimic and even share code with a text editor, but they do not implement live text. They do not know where their text came from, so they cannot write it back to its source, at least not on their own.

This paper describes a software package that makes live text from the output of several common UNIX® tools: the pattern-matcher `grep`, the file comparator `diff`, the compilation manager `make`, the spelling checker `spell`, and the file printer `cat`. When these programs present text from files, the text may be edited directly and written back to the files. These programs are often used to identify text that needs editing, so live output makes some corrections simpler and more direct. It is a first step toward unifying screen editors and window managers in a single program.

A trial implementation is embedded in `Emacs`[?]. `Emacs` is first a text editor, but it has long allowed users to execute arbitrary shell commands and to capture and display their output. This output has been editable but not live: editing the display has not affected the original source files. The prototype intercepts the commands above, post-processes their output, and propagates subsequent edits back to the sources. It requires two `Emacs` routines per shell command: the *post-processor* prepares outputs for display, and the *locater* identifies the source of any displayed line. In practice, some of these routines are null, and others share code.

The trial implementation is both useful and ad hoc. It handles only a few programs, and it cannot handle pipes at all. Correcting these shortcomings would require repackaging all programs that emit pieces of their input, to have them identify the source of each piece. It is unclear whether a full implementation would justify its cost, but the trial implementation has done so already — indeed, it went into use before it was complete. This document is published to show how live text is useful in some applications and to stimulate discussion of

® UNIX is a registered trademark of AT&T.

the merits of a full implementation.

A Live Pattern Matcher

The UNIX program `grep` searches files for lines that match a regular expression. It is often used to identify lines that need editing. For example, to rename the routine `add` in a set of C files, a user might say

```
grep add *.c
```

and observe the display

```
nums.c: int add(x, y)
main.c: int add();
main.c:  a = add(a, b);
```

Without live text, the user now invokes a text editor on the files `nums.c` and `main.c`, locates the lines shown above, and makes the changes. `grep` can help identify lines that need change, but it cannot write them back. Conversely, the text editor can write them back, but it cannot identify the files that need treatment, at least not by itself.

With live text, the necessary changes are made directly to the three output lines above. That is, the user employs ordinary editing commands to visit and change the three instances of `add` in `grep`'s output. (Editing the file name or, in general, any identifying material is an error. Such text is not live because its source is unknown.) When the edits are complete, a single command writes the changes to all source files at once. If there are other ordinary `Emacs` windows on the files, they are automatically updated at the same time. The update command is the only new command that users need learn. The prototype offers two versions: one performs all updates, and the other performs only those for the current line. The prototype can take more than a second to perform a full update, so a faster command for incremental updates seemed prudent. This optimization would be unnecessary in a free-standing implementation and could be avoided under `Emacs` as well.

Live text may also be changed using the conventional global editing commands of the editor. For example, the user could change the three lines above with a single global replacement command. As before, when the edits are complete, a single command writes the changes to all source files. A conventional stream editor may simulate such changes — for example, the change above could be performed with the UNIX stream editor `sed` — but such simulations are generally more cumbersome because they operate without the feedback provided by the display. Simulations of selective, interactive edits are trickier still.

To write text back, the live text system must record its source. In the prototype, it was easiest to assume that `grep` is run with a generally available option that inserts the line number into the text after the file name. Thus the prototype actually gives

```
nums.c:120: int add(x, y)
main.c:150: int add();
main.c:156:  a = add(a, b);
```

instead of the display above. These lines are self-identifying, but self-identification is not a general requirement. Post-processors might revert to the original, less cluttered image by recording the identifying data in a hidden, parallel buffer.

A Live File Comparator

The UNIX program `diff` displays the lines that differ between two files. It is often used to reconcile different versions of a program. For example, the user might say

```
diff main.c.bak main.c
```

and observe

```
31c31
< for (i = 0; i < 10; i++)
---
> while (i++ < 10)
55,57c55
< p = malloc(sizeof *p);
< p->value = "abc";
```

```
< p->next = 0;
---
> p = getnode("abc");
```

This output states that the old `for` loop at line 31 has been replaced with a `while` loop, and that in-line code at line 55 to allocate and initialize a structure has been replaced with a function call. Suppose that it is necessary to undo the first change but not the second. The `diff` script cannot be interpreted automatically without doing *all* edits, so this case requires human intervention.

Traditionally, the user would invoke the text editor on `main.c`, move to line 31, and retype the `for` loop (risking typos). The retyping can be avoiding by copying the `for` loop from `main.c.bak` or from the display above (some screen editors and window managers, though separate, can exchange strings), but these approaches require additional commands to open `main.c.bak` or to communicate text from the window manager to the screen editor. Another traditional alternative is offered by tools that step through `diff`'s output and prompt the user to identify the desired version of each difference.

With live text, the change is made by editing the output of `diff` just as the last section's changes were made by editing the output of `grep`. The text after each "`> `" above is live and bound to a piece of `main.c`, and the text after each "`< `" above is live and bound to a piece of `main.c.bak`. Editing the live text changes the corresponding source file. To make the change in question, the user selects all but the first two characters of the second line above and copies them on top of all but the first two characters of the fourth line above. (The first two characters are avoided because they are not live, just as the file names and line numbers in the output of `grep` are not live.) The user then completes the change by issuing the update command, which writes `main.c`.

Live text improves on the traditional ways of making such changes. There is no need to invoke (or otherwise switch to) a separate editor or to communicate text to it; integrating the screen editor and window manager obviates the need for such commands. Also, there is no need for an interactive `diff` at all. Live text replaces a `diff`-specific mechanism with a general one, and it allows the user to make changes in any order.

Live Compiler Diagnostics

Many compiler diagnostics include a source file name and line number. For example, a typical compilation might be triggered by the command `make` and yield the following output:

```
cc -c main.c
"main.c", line 2: unterminated string constant
```

Users read this output and switch to a screen editor to visit and correct the offending lines. Many `Emacs` users invoke `make` from within the editor and issue commands that parse the diagnostics and automatically cycle through the offending lines in ordinary windows on text files. Though ad hoc, this feature is particularly popular.

With live text, it is natural to change the display above. The source code does not appear there, so `make`'s post-processor fetches it. For the example above, the live text system thus actually displays

```
cc -c main.c
"main.c", line 2: printf("hello world\n);
                  unterminated string constant
```

For local errors like the one above, the user may edit this display directly. If context is needed, the user may position the cursor on the offending text and issue a command that switches to an ordinary window on that text in context. This command generalizes the long-standing `Emacs` facilities to visit errors in context. The new facilities are available for all live text, not just `make` output. For example, this command is useful to visit the context of lines from `grep` and `diff`. Alternately, the live text system could offer commands to enlarge (or reduce) the amount of context around any bit of live text.

Live compiler diagnostics complement the Berkeley UNIX `error` command[?], which turns compiler diagnostics into comments at the point of the error. With `error`, the diagnostics are inserted into the source. With live text, the source is inserted into the diagnostics.

A Live Spelling Checker

The UNIX program `spell` searches a dictionary for the words in its input and reports those that could not be found. Traditionally, the user then invokes the text editor on the input file or files and replaces each misspelled word with its correct spelling, either globally or one at a time. Global commands must take care to match only complete words: the string "tha" is not a valid word, but it is a valid substring of many valid words.

The live `spell` uses the traditional `spell` to identify a list of probable errors. Its post-processor then rescans the files and displays all lines that hold an error. The lines are presented in the order that they would appear in a concatenation of the input files, and a line appears only once, even if it holds multiple errors. The user may edit this display as live text, either line-by-line or globally. Presenting the whole line helps users identify words that are correct even though they are absent from `spell`'s dictionary.

Live text thus makes the non-interactive `spell` more like interactive spelling checkers, which step their users through the errors and may propose corrected spellings. The live `spell` allows the errors to be corrected in any order. Its post-processor could propose corrected spellings by searching the dictionary for similar words and displaying them (as non-live text) after the line with the error.

Another design is possible. The list of probable errors might be presented and edited directly, without post-processing. Each error can appear more than once in the original file or files, so an edit might have to change multiple sites. The live programs above map bits of the screen to one spot in one file, but this live program might have to map some bits to several spots in several files. This feature is beyond the prototype. It might be considered for future implementations, but unless it can take care to match only complete words, it might be too dangerous.

Live File Printers

The UNIX program `cat` displays one or more files. For example, in a directory of several short files, some users might locate a bit of desired text with the command `cat *`, which displays all of the files in the current window. If the desired text needs change, the user then switches to the screen editor. With live text, the display may be edited directly. The live `cat` also allows users to edit several small files in one window, which can save display space and window creation commands in at least some window managers. A live version of `tail`, which yields just the last few lines of the file, helps when text near the end of a file needs local editing.

It is at least theoretically possible for a live `cat` to replace the screen editor. To edit a file, users may `cat` it, use the ordinary window scrolling commands to find the spot of interest, and use live text to edit it. This application would, however, require considerable engineering to avoid losing important features of screen editors. For example, some screen editors make it easy to switch between windows on different files. This feature binds windows to files, but live text has no such natural binding — a window may contain bits of many files. For another example, consider search commands. Screen editors generally have them cycle around the file or freeze at the last instance of the search key. With live text, the window may contain commands and bits of different files. Should the search match commands? Should it cross file boundaries? Replacing screen editors with a live `cat` would require answers to many such questions, but even if they cannot be answered, it is clear that live text and even the live `cat` are useful in their current applications.

The Prototype

The current implementation is ad hoc and limited to a few tools, but it nevertheless exposes some principles that may guide future implementations. The prototype intercepts each live command, executes it, and places the command and its output in a buffer. It next calls the tool's post-processor with the buffer as the sole argument. Then it displays the result, which the user may edit with ordinary editing commands. When the user issues an update command, the live text system scans the current line (if an incremental update was issued) or all lines in the buffer. It uses the tool's locater to separate any identifying material from the text and passes the results to a generic file updater. For each line, the updater requires the final text, the file name, and the line number. Adding a new tool means supplying a post-processor and a locater. In practice, some of these routines are null, and others share code.

Some post-processors make large changes; others make none. The post-processors for `grep`, `diff`, and `cat` leave the buffer alone, because their output automatically includes both live text and an adequate indication of where it came from. The output of `make` includes no source text, so `make`'s post-processor fetches it and inserts it into the buffer. The post-processor for `spell` replaces the input buffer completely; it accepts a list of suspect words and replaces it with the source lines that hold those words.

The locaters accept a line and extract the final text, the file name, and the line number for the generic update routine. The locaters for `grep`, `make`, and `spell` find this material in the current line[1]. The locater for `diff` scans back from the given line to find the line numbers and file names. The locater for `cat` computes the file name and line number from an internal table that records the name and length in lines of each file that the tool displayed; `cat`'s post-processor creates this table.

The post-processor and locater may trade duties so long as they collaborate to provide the user and the generic updater with the information that they need. For example, `grep`'s current post-processor does nothing and leaves everything to its locater, but other trade-offs are possible. If the post-processor made all lines self-identifying, then the locater would be simpler. If all post-processors shared a common self-identification scheme, one simple locater could serve all shell commands. The prototype minimizes such visible post-processing to avoid confusing new users, but small changes in output format are probably acceptable.

Some of the lessons learned from the prototype may now be summarized.

[1] Actually, `grep` does not print the file name when it searches only one file, so in this case `grep`'s locater extracts it from the text of the `grep` command itself. If the system were changed to invoke the tool's post-processor just *before* the command is run, then it could append the name of the empty UNIX file `/dev/null`. Then all `grep` commands would name multiple files, so all lines from `grep` would share the single, self-identifying format above, even when the original command names only one file. In this case, the "post-processor" could also assume responsibility for adding `grep`'s line-numbering option.

The main parts of the live text system — the editor and the updater — are common to all the live tools, and even the tool-specific routines — the post-processors and locaters — are similar enough to share code. Using a consistent line-identification scheme would eliminate the need for tool-specific locaters. If the tools can be changed to emit such identifiers, post-processing would be needed only if the tool's output is thought to benefit from expansion (like `make` and `spell`). The identifiers need not clutter the display, for they can be hidden in an internal table like `cat`'s. These observations could be exploited in a second-generation `Emacs` implementation, but a free-standing implementation would be better.

Future Implementations

Future implementations should include some features that are beyond the prototype. For example, live text should appear in a different font, intensity, or color. Such highlighting would reduce accidental changes to files.

The system should also offer explicit commands to "deaden" live text — that is, to break the connection between the display and the file. Traditional window managers support no live text, so users cannot easily write changes back to source files. The prototype live text system makes *all* displayed file text live, so users cannot easily edit a scratch copy of a file without also changing the file. A deadening command would give the user control of what is live and thus permit work between these two extremes.

Except for `cat`, the live tools above expect lines to begin with identifying material, so only the live `cat` allows users to insert and delete lines. All the live tools allow intra-line changes, and all allow the user to switch from a bit of live text to an ordinary window on that text in context, but future implementations should address the current restriction nonetheless. Minimizing the display of identifying material should help.

`Emacs` makes it easy to propagate output edits back to ordinary windows on the underlying source files, but it is not practical to propagate edits in the opposite direction or to propagate them between different live copies of the same source in one or more windows. The best that can be done is to warn the user about such edits. A full implementation of live text should make all copies first-class citizens, at least experimentally. Thus if, say, two `grep` outputs displayed a common line, editing one display copy would change both.

Live text can be generalized to text from programs, not just files. Consider spreadsheets: their raw data comes from files and, when edited, can be transmitted back, but edits also change computed cells on the display. If live text were generalized to come from programs as well as from files, then a spreadsheet program could leave all interaction to a live text system. The spreadsheet program would store variables, evaluate equations, and emit output; the live text system would offer interactive editing and write any changed cells back to the spreadsheet program, just as it now writes changed text back to source files. The ability to write text back to programs is likely to have other applications as well.

The hardest questions about live text involve its extension to arbitrary tools. The programs already accommodated appear to be among the most useful, but in theory any program that emits pieces of its input could emit live text, so no fixed list can be complete. The prototype supports only a few live programs, so it is tolerable to determine a text's source by scanning commands and output lines for file names and line numbers, but this expedient is inelegant and extends poorly.

Many UNIX programs read and write simple text files. This convention allows a small set of general tools to collaborate to offer a wide range of functions[?, ?], which is one of the system's most valuable features. The output of these tools does not, however, record the source of the text, so it cannot support live text. A full implementation of live text would need tools that supplement the text with data that identifies its source. For example, when a traditional UNIX tool would emit a line of text, the corresponding live tool might emit a record with four components: the line of text (as before), the name of the file from which the line came, and the addresses within that file of the line's first and last characters. Such structures — sometimes called "piece tables" — have been used internally in text editors for years[?, ?]; a full implementation of live text would expose them to users.

The live text system could use piece tables to produce the traditional display and to implement updates without per-tool post-processors and locaters. Avoiding per-tool software would simplify implementation and automatically yield a consistent interface across all tools. For example, such an implementation would allow all tools, not just `cat`, to insert and delete lines.

Unfortunately, programs that emit piece tables would be incompatible with traditional UNIX tools. They could be made compatible by passing the records above through a simple filter that strips the identifying data, but an explicitly invoked filter would try the user's patience. Perhaps the least unpalatable solution would be to edit the tools to offer a live-text option, which might be the default for some users. Most changes could be confined to the i/o interface and hidden in libraries, but even this architecture would require touching tools to make them live. Thus it remains to be seen whether a full implementation of live text would justify its cost. The prototype, however, shows that a limited implementation can.

Acknowledgments

Stu Feldman suggested the live `diff` and `spell` (the unimplemented variant). Doug Blewett, Brian Kernighan, Dave Korn, Doug McIlroy, Rob Pike, Howard Trickey, Peter Weinberger, and Craig Wills suggested improvements to an early draft of this presentation.