Differential Editors

Titus D. M. Purdin and Alan L. Wendt

Technical Report CS-93-101

February 4, 1993

Differential Editors

TITUS D. M. PURDIN
Department of Management and Information Science, The University of Arizona, Tucson, Arizona 85721, U.S.A.
ALAN L. WENDT
Department of Computer Science, Colorado State University, Fort Collins, Colorado 80521, U.S.A.

Keywords
Differential editor, Source code control system, Bank statement reconciliation

Summary

A *differential editor* is a text or file editor that excells at editing, comparison, and reconciliation of several differing versions of a file simultaneously. This paper discusses differential editors in general and gives two important examples of the paradigm. One example is an editor augmented for bank statement reconciliation. The other is a program text editor augmented to handle multiple versions of source text, using preprocessor control lines to delimit individual versions.

Introduction

Often programmers and users of computers must operate upon more than one version of a text file or a database. For example, programmers may maintain several versions of a source program and may transfer text from one version to another, compare two versions, etc. Users may need to compare two versions of a file and edit them to ensure their consistency. In general, in any system of information flow, when two sources flow together, some comparison and reconciliation of differences may be necessary. When an interactive text editor or a database file editor is augmented with the ability to compare different versions of a text, a *differential editor* is the result. Such an editor is useful when several versions of a text exist and they must be edited consistently, or their differences must be resolved.

A differential editor works with more than version of a text. It allows changes to some of the versions, and it may disallow changes to other versions. As changes are made, it may automatically compare the updated version with other versions. It may track changes to one version and generate a summary of the changes. The need to perform all of these functions can make differential editors complex, so their construction can be justified only if alternative tools for handling versions are inadequate. Many tools exist to support these tasks, but they all suffer from deficiencies.

Operating systems such as Digital Equipment's VMS explicitly provide facilities to maintain several versions of each file. In operating systems such as UNIX, control of versions is accomplished through the use of utility programs such as SCCS (Source Code Control System). Such utilities are usually large programs themselves, and they force a particular paradigm upon the programmer. Sources must be checked in and out. Only one version is instantiated at a time. Source comparison programs (`diff` et al) help but do not solve the problem. Often the difference between two versions is longer than either. The Unix `diff3` and `patch` programs are intended to allow a change to be abstracted from a pair of programs, and for the change to then be applied to a third program. These systems do not alway apply patches correctly, due to inconsistent sets of changes between the three versions. Finally, pipelines using `sort` and `cmp` can be used for simple comparisons of structured (one-line) records, but such pipelines would be too slow to run for every record update. The `sdiff` program is a primitive differential editor that compares two source files side-by-side, and allows the user to invoke a separate editor on either file. However, it is primarily a stream editor; once the output file is complete the program must be restarted. The `spiff` system [N88] includes the features of `diff` and extends it to allow approximate numerical matches. It can also compare streams of program lexemes, so as to ignore formatting and comment changes in source code. A system that uses control flow information to integrate two versions of a program is described in HPR89. An editor for source code revision control is described in FM87. None of these systems support viewing, comparison, and editing of two or more versions of a file simultaneously.

This paper describes two systems incorporating the differential editor paradigm. The first is used to reconcile bank statements with records of issued checks. The second is a source code maintenance system.

Bank Statement Reconciliation

Bank statement reconciliation is one important application for a differential editing tool. A system has been implemented to compare and reconcile bank statements with records of issued checks. Little information is available about automated systems for bank statement reconciliation. A check of some local branches of large banks reveals that reconciliations are currently done manually.

The editor described here reads a file of bank statement lines (clearances) and a file of checks written (issuances). It displays these two files side-by-side on a terminal screen. The operator can compare the two files and change records to bring the files into agreement.

The editor interprets any changes that are made as error corrections, and it builds two files of these corrections, one of updated issuances and one of updated clearances. It can then print reports to send to the bank or the source of the issuance records.

The editor has several display options. All checks can be displayed, or only unmatched checks. The corrections records can also be edited directly. All of the checks can be sorted in numeric order, or by any other field in the record such as the amount.

The system can accept input from optically-scanned bank statements. Such statements usually contain about one error per three lines. The system uses an optimal minimum-edit-distance error-correction algorithm to reduce this to about one error per page. The algorithm accepts a regular expression that describes the format of a print line on a bank statement. It also accepts a table of single-character correction costs (for example, it is cheap to correct a capital "S" to a "5"). It then reads the optically-scanned input and finds a change of minimum cost that repairs each scanned input line to an instance of the regular expression.

Bank statements are reconciled manually by comparing records of issuances against the bank's statement, which records clearances. Mismatches can be bank errors, office errors, or outstanding checks. The process is complete when all of the errors have been corrected and no unmatched clearances remain on the bank statement. Unmatched issuances can remain and be carried forward as outstanding checks to the next reconciliation.

The Editor

An invocation of the editor loads in any existing records and presents them for display. Issuances show the date of issue, while clearances show the date of clearance.

For a small example, if 5 checks were recorded as issued and the bank statement contained 3 of those, with some erroneous information, the display might appear as follows:

```
               ISSUE      CLEAR
   NUMBER       DATE       DATE        AMOUNT

      214    09/10/92                  32.16
      214               09/13/92       32.16
      215    09/10/92                  52.13
      215               09/10/92       52.18
      216    09/11/92                  24.33
      218    09/11/92                  10.00
      218               09/11/92       24.33
      219    09/12/92                  12.50
```

In this example, two of the issuances (218 and 219) were not cleared by the bank. Check number 215 was issued for $52.13 and cleared for $52.18. And check number 216, for $24.33, was apparently cleared as check 218. Some information (e.g. the payee) is actually presented by the program but omitted from this display for formatting reasons.

If a company has more than one office, issuance records may be transmitted from several different computer systems. The source of all records is tracked (which bank statement, or which register of check issuances) so that errors in the records, when corrected, can be propagated back to their source.

After loading the records, the user can hit a function key to command the editor to match up issuances and clearances. Pairs of records that match on both the amount and check number are merged into a single record. Matched records show both issue and clearance dates. The result is:

NUMBER	ISSUE DATE	CLEAR DATE	AMOUNT
214	09/10/92	09/13/92	32.16
215	09/10/92		52.13
215		09/10/92	52.18
216	09/11/92		24.33
218	09/11/92		10.00
218		09/11/92	24.33
219	09/12/92		12.50

Values in issuances and clearances can be edited directly. After examining the above display, the user might determine that the bank incorrectly cleared check number 215 at $52.18, edit the amount in the third line to match that in the second, and hit a function key to command another match. The first time an issuance or clearance record is changed, it is rewritten to an "updated" record of the same type.

Internally the editor maintains several variant record types. An "issuance" record comes from an office and contains the date that the check was issued, the amount and check number, and identification of the source of the record. An "updated issuance" contains the original version of the issuance in addition to any corrections that have been made to the issuance. A "clearance" record comes from the bank statement and contains the date that the check was cleared, the amount and check number, and identification of the source. An "updated clearance" contains the original version of the clearance in addition to any corrections that have been made to the clearance. Finally, a "merged" record is generated when issuances and clearances are matched. It contains original versions of both the issuance and clearance, as well as the current values for both (which must be the same in order for the match to occur).

The editor supports editing from several points of view. For example, the editor facilitates the examination of unmatched records by suppressing display of matched records:

NUMBER	ISSUE DATE	CLEAR DATE	AMOUNT
216	09/11/92		24.33
218	09/11/92		10.00
218		09/11/92	24.33
219	09/12/92		12.50

The editor can also sort the records on any key. To do this, the user positions on the desired column and hits a function key. Check number discrepancies may by found be examining matching amounts and vice-versa.

As errors are found and corrected on the screen, the editor generates updated clearance and issuance records. Updated clearance records (and merged records) are normally used to print a letter to the bank, requesting an update to the bank's records, but some of the discrepancies reflect errors in the optical scanning process, instead of errors printed on the bank statement. To solve this problem, the editor supports a view of updated records, showing both the original and updated versions. From this viewpoint, the user can edit either version or simply delete the update, which amounts to copying the current version onto the original version.

The goal of the reconciliation process is to eliminate all unmatched clearances. In other words, the bank should not have cleared any checks that have not been issued. Once this is done and reports printed, the matched records are purged from the file, leaving only unmatched issuances. These records are carried forward to the next reconciliation.

Correcting Scan Errors

Optical character recognition introduces many errors. The editor uses a minimum-edit-cost error correction algorithm due to Myers [M88,MM89] to correct scanned input lines to the closest instance of a regular expression that describes lines of the bank statement. Many errors are simple substitution errors, such as replacing "5" with capital "S", but other errors are more complex. The scanner sometimes inserts extra spaces into numbers, for example, changing "45.90" into "4 5.90". Such errors would be difficult to correct with an editing script and, if allowed to remain, would complicate the problem of parsing the input correctly. The correction process takes a maximum correction cost, and it rejects any line on the bank statement that requires more. Correction therefore solves three problems. It corrects most optical recognition errors. It ensures that the editor sees at least legal (if not actually correctly scanned) input, which obviates parser error recovery. Finally, it passes on the important detail lines of the bank statement and rejects other extraneous information.

The error correction system begins by constructing a state-labelled ϵ-NFA from the regular expression. A state-labelled NFA labels the states of the NFA instead of the edges, so that all edges into a given state transit on the same character.

For each input string S of length L, the system then fills in a rectangular array $V[L+1, N]$, where N is the number of states in the NFA. $V[i, p]$ gives the minimum cost of consuming i characters of input and driving the NFA into state p. The following account uses $\delta(p, c) \rightarrow q$ to mean that the NFA has a transition from state p on character c to state q. Because the NFA is state-labelled, the label on state q (denoted $L(q)$) must in this case be c.

The description below uses $C_{c,d}$ to denote the cost of correcting the character c to the character d. The cost of deleting c is given as $C_{c,\epsilon}$. The cost of inserting c is given as $C_{\epsilon,c}$. The system fills in cells in order of increasing cost. $V[i, p]$ is the least cost of the following different ways of driving the NFA into state p after consuming i characters; these costs model deletion, insertion, correction, NFA ϵ-moves, and correct matches respectively:

$$V[i,p] = min \begin{cases} V[i-1,p] + C_{S[i],\epsilon} & \\ V[i,q] + C_{\epsilon,L(p)} & \text{where } \delta(q,L(p)) \rightarrow p \\ V[i-1,q] + C_{S[i],L(p)} & \text{where } \delta(q,L(p)) \rightarrow p \\ V[i,q] & \text{where } \delta(q,\epsilon) \rightarrow p \\ V[i-1,q] & \text{where } \delta(q,S[i]) \rightarrow p \end{cases}$$

The NFA begins in state 0, having consumed no input and incurred no cost, so $V[0,0] = 0$. The NFA is constructed with one final state F, and the system uses Dijkstra's shortest-path algorithm [D59] with discrete costs to find the value of $V[L,F]$.

Correction of most scanning errors is necessary for optical scanning to be successfully used to reconcile bank statements. The system described is used routinely at a company issuing approximately three thousand checks per month.

A Differential Source Editor

Those engaged in substantial software projects find it useful to maintain copies of variations of certain software entities within the project. Software entities are files containing programs, parts of programs, or data associated with the project.

The variations on these files arise in two distinct ways: as historical *revisions* of a piece of software or as *alternatives* to a piece of software.[1] The former represents a record over time of the changing state of a program. The latter represents concurrent, competing instances of a program. The editor treats revisions and alternatives uniformly.

Variations of a program are the result of requirements to write software that is expected to operate in more than one distinct environment. This occurs, for example, if the result of a software project is expected to execute on machines running VMS and on machines running UNIX. As soon as a piece of software achieves a reasonable size or more than a little complexity, it becomes unlikely that a single, uniform source code will produce an appropriate executable image for both platforms.

In many software development environments (e.g. C/UNIX), alternative source code formulations can be accomplished through the use of *conditional compilation.* This is accomplished by including additional statements in the source that cause portions of the source to be filtered out when it is run through a "preprocessor." This is a well known paradigm in the sphere of program compilation.

It is our purpose here to look at the potential benefits of an editor that takes this same approach to conditionally included text.

[1] *Backup* copies of a piece of software could be considered a third, although they, notably, are not *variations*.

It is common practice to manage the portability of a software project through the use of conditional compilation. This may be applied to situations as simple as that of the BSD vs. SysV dispute over *strings.h* and *string.h* or as complicated as adapting a program to both *curses* and to MS-DOS. Figure 1 shows a simple example of how this is accomplished.

```
#ifdef BSD
#include <strings.h>
#else /*SYSV*/
#include <string.h>
#endif
```

Figure 1. An example of conditional compilation

The lines beginning with a '#' character are meaningful to the preprocessor program. The preprocessor determines the disposition of lines of source text bounded by such lines based on the value of the variable specified in the condition. These conditional variables are, in turn, set either by other preprocessor statements (e.g. `#define VAX`) or as a command line option to the compiler.

This is a simple and appealing method for maintaining the bulk of the code associated with an application in a single location while isolating the specifically variable features of the code. It is realized through the extension of the preprocessor phase; which offers considerable additional functionality (e.g. macro expansion and external text inclusion). Thus, while it is not obtained without cost, it does not represent a major addition to the processing or to the number of filtering steps associated with compilation.

Traditionally, use of this conditional inclusion/compilation approach has involved the use of any available editor to create and modify the source text. Such standard editors take no notice of the nature of the target file other than that it is a string (or strings) of printable characters. Thus a developer sees a, possibly, quite different view of the text of a program using an editor than that seen by a compiler. The compiler, of course, will see the "preprocessed" version of the program; which among other changes, may have conditionally bounded portions of the text omitted.

It has been the purpose of this investigation to construct and evaluate the utility of an editor that allows the developer's view of source text to, more nearly, equate to that of the compiler. Our editor incorporates a preprocessor similar to that used in the compilation process to accomplish this. Using command line specifications a user selects the view(s) of the source that are of interest. In essence, a user is allowed to edit the "VMS" version of the program, for example, without seeing "alternate" code associated with a parallel

"UNIX" or "MS-DOS" versions. This approach supplies a very different view of the source code and raises some interesting questions with regard to editing.

This section details the design of a differential editor that includes features to facilitate maintenance of multiple versions of programs written in the C language. Versions of programs are maintained within a single source file, which can be edited directly by the programmer and processed directly by the compiler. The programmer can view one version of the source file, as with other tools. She can also view several or all versions simultaneously. If two versions are identical except for formatting differences, the programmer can merge them back together easily.

Approach

The constructed system consists of three parts: a preprocessor, an editor, and a postprocessor. There is no substantial reason for the three to be separate pieces of software. In a production environment they could and would be combined into a single tool. For purposes of design and evaluation, however, some advantage is obtained from keeping them separate.

The preprocessor associated with the editor is an altered version of that associated with the compiler; in this case the *cpp* program used by the UNIX C compiler. It differs in several significant ways from the compilation preprocessor. It performs no macro subsitution or external text inclusion. Nor does it actively *filter out* any text lines as does the compilation preprocessor. Instead, the editor preprocessor prepends status information to each line before it passes the line to the editor.

The editor used in this investigation is a modified version of the `vi` editor associated with UNIX. This editor, known simply as `S` or the `S` editor, is described in [M87]. This editor was selected because of its familiar semantics, its small size, and the availability of the source code.

The potential for inducing inconsistencies through the interleaving of **`#ifdef`**'s and **`#endif`**'s or through the omission of same is very real when using a standard editor to manipulate a source text file. One of the design goals for the conditional editor was to overcome this deficiency and insure that the editor would not allow such inconsistencies to exist past a write of the target file. An independent, consistency checking postprocessor was added to the editor to satisfy this need.

The Preprocessor

The first of the three logical parts of the conditional editor is the preprocessor. In a *single tool* model of this software the preprocessor would be integrated into the *file read* facility of the editor itself. Maintaining it as a separate program greatly reduces the modification necessary to the editor code, and the interface remains relatively unchanged. The preprocessor must associate with each input line information that the editor can use when deciding which lines to display and which not to display.

The preprocessor does **not** do any macro substitution. In this regard, the conditional editor does not provide the developer with the *same* view of a source file as that seen by a compiler. Clearly, inclusion of such items would clutter the user's view and defeat the purpose of such features. It was felt that those few instances in which a developer (debugger) needs to see these things are better served by saving the compiler preprocessor output.

A separate file (`.versions`) supplies the command-line arguments to the preprocessor that yield each different version. This file can contain definitions of preprocessor variables and it can adjust the order in which directories will be searched to satisfy `#include` statements. Figure 2 gives an example of a `.versions` definition file.

```
UNIX      L -DUNIX=1 -I./unixinclude -I/usr/include
CPM       L -DCPM=1 -I./cpminclude -I/usr/include
DOS       U -DDOS=1 -DINTSIZE=16 -I./dosinclude -I/usr/include
COFFSMALL U -DUNIX=1 -DINTSIZE=16 -DPTRSIZE=16 -I./unixinclude -I/usr/include
COFFBIG   U -DUNIX=1 -DINTSIZE=16 -DPTRSIZE=32 -I./unixinclude -I/usr/include
PORT      L -DPORT=1
```

Figure 2. An example `.versions` *definition file*

Each version can be locked against updates if desired (the `L` in the second column of the figure). The figure describes versions for different operating systems. The same technique is used to describe different revisions of the same program text. The editor command line allows the user to specify which versions are to be displayed and which versions can be updated (a subset of the unlocked versions). If all versions are displayed and updated, the effect is that of a normal text editor – the user will see conditional compilation delimiters (`#ifdef`, `#else`, `#endif`, etc.) marking the lines that are unique to each different version. If the user elects to edit just one or two versions, the conditionals and unique code for the others will be tagged for suppression of display.

The editor disallows changes to read-only versions. If, for example, the user is allowed to update version 2 but not version 1, an attempt to update a line common to both will cause the editor to insert preprocessor commands to separate the two versions. In the example above, the user will not be allowed to update the UNIX, CPM, and PORT versions because they have been locked. Because all versions are available simultaneously, users can easily move code from one version to another (i.e. to fix a common bug, or to move a bug fix from an experimental version into a release). Users can easily make the same change to all versions (for example, to change the name of a global variable in all versions). Users can also compare versions and suppress display of non-pertinent versions.

To handle all versions in parallel, the preprocessor is equipped with a "parallel symbol table" that holds different values of preprocessor symbols for each version, and parallel conditional expansion logic to produce a tag for each line in the file that describes which versions that the line belongs to. The tag is a bit vector describing which versions are active at the time that the line is scanned. The parallel symbol table contains preprocessor names, definitions, and tags that list the set of versions for which the given name has the given definition.

The Editor

The `S` editor is a well written, well documented text editor that conforms very nicely to the command semantics of `vi`. It is in the public domain and the source code is readily available.

Users request incompatible capabilities from version editors and so some alternative designs were explored. Some users like to suppress all but one version of the source file but state a preference for edits applying to all versions. This capability is useful if the source is heavily `#ifdef`'ed but consists mostly of lists of alternative implementations on different platforms. The code for `uucp` and the GNU C Compiler `gcc` are examples of such. Other users feel that such updates would be unsafe as they would affect invisible code, and that any changes to visible and writable code should not affect unseen code. This alternative seems preferable for historical revisions and for any complex conditional code. The first style has a problem if the user inserts a line between two visible lines that have some hidden lines between them (Figure 3); does the new line get inserted above or below the hidden text?

```
Visible line one.
#ifdef H1 /* this conditional hidden */
Hidden line.
Hidden line.
Hidden line.
#endif
Visible line two.
```

Figure 3. Visibility problem

The second style has the same problem but in this case it doesn't matter because any new insertion does not update hidden versions. If versions `V1` and `V2` are active then any insertion will be qualified, so the result will either be that of Figure 4 or the two conditional sections could be swapped, but they are mutually exclusive so their order is immaterial. Both styles could be accomodated if desired; we chose the second style as being safer.

Little had to be changed in the editor code to allow it to accomodate the modified input provided by the preprocessor. The changes were confined to the text storage module and to the text display module.

The editor maintains text lines on a doubly-linked list. The text display module was augmented to skip over hidden lines by threading together visible lines on a separate list; this makes screen display efficient.

```
Visible line one.
#ifdef H1 /* this conditional hidden */
Hidden line.
Hidden line.
Hidden line.
#endif
#if defined(V1)|defined(V2)
New inserted line.
#endif
Visible line two.
```

Figure 4. Visibility problem

The text storage module was augmented in a similar manner to the preprocessor. The commands to insert, delete, and update lines were augmented with a bit vector telling which versions are subject to the update. The module preserves lines that apply to any version, and it merges together adjacent identical lines that apply to more than one version.

The account thus far has ignored a complication in that "versions" are not actually recognized by the preprocessor commands used (`#ifdef` et al), which instead recognize preprocessor variables. Thus there is a level of indirection between versions and the generated tests, and the editor must search for an appropriate expression to test. This can actually be impossible; nothing guarantees that two versions actually define *anything* differently, and the editor will issue an error in that cases. The editor could easily be extended to allow convenient access to all versions that define a particular preprocessor variable, and MACH maintainers could easily edit 10 `UMA` versions without messing up (or even seeing) the `NUMA` versions. If only one version is visible *all* `#ifdef`'s are suppressed and the user can edit without regard to whether the code being edited is conditional or not.

The Postprocessor

Before the file is written out it is processed by a program similar to the initial preprocessor. Instead of creating version tags, however, the postprocessor compares the existing version tags with the tests that exist in the buffer. When they are inconsistent, the postprocessor inserts and deletes preprocessor tests as necessary. This technique allows the user to supply the test that will be used, or to defer and allow the editor to create a test, and still protect un-edited sources.

Post-processing also checks the consistency of conditions. Users of the conditional editor are less prone to introduce conditional errors because they see fewer conditionals. But the consistency of matching pairs of preprocessor statements is easy to check, and adding such a check to the output module of the editor is simple. The cost of doing such checks when the target file is written by the editor is offset by the detection of simple errors at the earliest possible point, thus avoiding having to track them down at a (possibly much) later compile time.

Finally, the postprocessor attempts to simplify the conditionals contained in the buffer.

That is, it is quite possible with any editor to arrive at code that looks like the following:

```
#ifdef UMA
    ... (text A) ...
#endif UMA
#ifdef UMA
    ... (text B) ...
#endif UMA
```

where this is simple enough to detect and correct, leaving the user with a cleaner file and one less worry.

Several simple patterns fall into the same category as that shown above and can be detected and corrected inexpensively. Examples include empty conditional blocks, blocks bounded by TRUE, and nested blocks that contain identical conditionals. Note that in no case is source text removed. Only preprocessor statements are affected, always without impact to the semantics of those conditionals.

Observations

The conditional editor constructed for this investigation can be substituted directly for the `S` editor without any effect on average users. Its behavior in the absence of preprocessor conditional statements is identical to an unaltered version of the editor. This is in no way a result of the nature of the `S` editor. It would remain true for any standard editor we can imagine. This is a testiment to the simplicity and low impact of the necessary modifications.

When used in an environment in which conditional compilation is used for maintaining program variants its functionality emerges.

Experience and Summary

This paper describes a consistent editor-based paradigm (a differential editor). The paradigm is useful in any environment that requires accounting for discrepancies in a large number of records. Differential editing involves parallel display of more than one version of a dataset, automatic matching, and support of several viewpoints. Two editor implementations exemplify the utility of this paradigm.

References

[D59] E. W. Dijkstra. “A note on two problems in connexion with graphs”, Numerische Mathematik, 1:269-271, 1959.

[FM87] C. W. Fraser and E. W. Myers. “An Editor for Revision Control.” *ACM Transactions of Programming Languages and Systems* Vol. 9 No. 2 (1987), 277-295.

[HPR89] S. Horwitz, J. Prins, and T. Reps. "Integrating Noninterfering Versions of Programs." *ACM Transactions of Programming Languages and Systems* Vol. 11 No. 3 (1989), 345-387.

[MM89] W. Miller and E. W. Myers. "Approximate Matching of Regular Expressions." *Bull. Math. Biology* 51,1 (1989), 35-56.

[M87] W. Miller. "A Software Tools Sampler", Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[M88] E. W. Myers. Personal communication.

[N88] D. W. Nachbar. "SPIFF – A Program for Making Controlled Approximate Comparisons of Files" Proceedings of the USENIX 1988 Summer Technical Conference, pp. 73-84, USENIX Association, San Francisco, California, June 1988.