Vincent Kruskal

Managing Multi-Version Programs with an Editor

When more than one version of a program must be maintained, generally much of the code is repeated unchanged in many versions. Techniques such as "deltas" and conditional compilation are commonly used to avoid duplicating these common parts. In addition to saving storage, these methods aid the programmer greatly in managing updates to the versions. Unfortunately, these representations of multi-version programs can appear very unlike a program, making them difficult to edit. Described here is a new method of automating much of the bookkeeping involved in dealing with multi-version programs. It entails use of a special editor that enables a multi-version program to be seen and modified in a fashion that is far closer to that normally permitted for a single-version program.

1. Introduction

A *multi-version program* is a data structure that contains fragments of code as well as control information to determine, for each version, which of the fragments are needed. An automatic process can produce, on demand, any desired version upon presentation of a version name or equivalent identifier.

Two techniques are in common use for dealing with the many cases in which multi-version programs are encountered. So-called "deltas" represent reasonably well sequences of versions over time but represent concurrently existing versions poorly. Conversely, conditional compilation allows concurrent versions to be represented well but in practice not sequences of versions. Moreover, both representations can seem very unnatural to the programmer.

A special editor is described in this paper to deal with both concurrent and sequential multi-version programs in a uniform manner. The code for only the version the programmer is working on is displayed and no version control information is displayed. If he is editing more than one version, one of them is displayed as an example and the sections that differ between those versions are highlighted. Multi-version documentation, as well as code, is often maintained as well.

In this paper, we first review some typical instances of multi-version programs. We then describe the traditional delta and conditional compilation methods for representing multi-version programs. In the next section, a method is reviewed in which an editor is used to overcome some of the human factors problems associated with these traditional methods. Finally, some specifics about the editor, P-EDIT, are discussed.

2. The need for multi-version program control

One use of multi-version programs is as part of an application customizer, such as MACS [1]. Here a prospective user fills out a questionnaire to make his needs known to the customizer program, which in response produces an appropriate version of the application program, along with customized documentation. By far the most expensive part of a customizer is the multi-version program (and documentation) within it that defines the possible versions that can be produced. Operating systems are also often written as multi-version programs, with management at each installation choosing an appropriate version for their system.

Another common use for multi-version programs results from the need to maintain a record of the changes that were made to a program. Here a suitable time period is chosen (daily, edit session, etc.), and a new version is created each time the program is changed within the period; alternatively, a method of naming versions is chosen (version 2.4.3, etc.) and a new version created when deemed appropriate. In

either case, a sequence of versions is generated, usually with a clearly latest version of primary interest. Thus, sequences of versions seem quite different from the concurrent versions of an application customizer. In fact, they are sometimes given different names, as in SVCE's versions and revisions [2], and some multi-version techniques only handle one or the other. However, the distinction is sometimes not so clear, as when there is a released version being maintained and a next version being developed. Here a redundancy of effort can be avoided by making the latest two versions concurrent to the maintainers, the developers, or both.

When a large application program is being constructed by a group, there is often need for one programmer to work on a temporary version of one module without imposing his transitory problems on the others in the group. Complex relationships can develop among such versions when, say, one programmer wishes to make the module he is writing conform with another module that is not yet ready to be turned over to the group as a whole. A sufficiently powerful version control system can solve many of the problems inherent in this situation so that it is even possible to relax the common module ownership rules that forbid more than one programmer from modifying a particular piece of a program. Ownership of versions, rather than modules, can provide the same control with far greater freedom.

Another instance of the need for a more powerful version control system results from the requirement of some customers of certain kinds of programs to make local modifications. One way that software manufacturers can satisfy this need is to sell the needed modification to the customer as a customized product. However, the cost of such customized products tends to be quite high, primarily because of the high cost of maintaining them while making changes to the general product. Since the general product with the modifications needed by a particular user is simply an example of another version of the product, any method that substantially reduces the cost of maintaining an additional version would permit software manufacturers to quote far lower prices for customized products.

A kind of version control not otherwise discussed here is really the same concept at a macroscopic level: control over which modules should be selected to generate a particular version of a large system. Here the control is exercised at the module level, rather than at the code fragment level. However, the typical procedure for building a large system from modules is governed by a control file for each version to be built. Such control files themselves can be thought of as programs written in a very special purpose language. Thus, code fragment version control might well prove useful even at the module level if one multi-version control file were maintained, rather than many separate ones. This would be particularly true when a large amount of duplication existed in the otherwise separate control files.

3. Representations of multi-version programs

Two general ways are used to represent multi-version programs, deltas and internal Boolean expressions.

- *Deltas*

Deltas are instructions on how to modify a previous version to produce the next version. They use line numbers or similar identification to name the points in the previous version where the modifications are to be made. Deltas are really packages of edit commands and were, in fact, originally invented to provide batch editing for tapes at least as early as the FAP assembler for the IBM 7090. They are still used for purposes other than version control, i.e., to send instructions on how to modify a program at a remote location. However, they look nothing like the edit commands common today in interactive editors and are quite difficult to deal with. Each delta is usually represented as a separate file. (The SCCS system, described in a paper which gives an excellent description of the problems of deltas [3], is an exception.) Thus, there is typically a base file and a sequence of delta files. Additional mechanisms are often introduced to permit cataloging of the order in which the deltas are to be applied for various versions.

Deltas work fairly well to represent sequences of versions, but they represent concurrent versions poorly. Concurrent versions are represented as a tree of deltas. Take a simple case of a multi-version file with three concurrent versions, X, Y, and Z. A typical representation might assign to X the role of being the base, and one delta would represent the transformation of X into Y and a separate delta would represent the transformation of X into Z. But if a particular place in X had to be changed in the same way for both Y and Z, that change would have to be represented in both the X to Y delta and the X to Z delta. But the duplication of identical code is exactly what is being avoided by having multi-version programs. When a programmer is changing that section of the X to Y delta, he is given no help in making the corresponding change to the X to Z delta, not even help in knowing that it should be done. It cannot be argued that the original delta representation was wrong, i.e., that Y should have been the base and that Y to X and Y to Z should have been the deltas. For in some other place in the program, X and Z might be the same while Y is different.

Actual attempts to use deltas to represent concurrent versions usually are a bit more complex. Often an attempt is made to have the deltas represent a sequence over time, as well as concurrent versions. It is sometimes useful to do this, but it is extremely limiting. Deltas are of primary importance here, because an interesting technique is in current use to

deal with their obscure data representation, which is not generally available for their alternative, described next.

• *Internal Boolean expressions*

The alternative to deltas is to have meta-linguistic conditional statements placed within an otherwise normal-looking program. This is often called conditional assembly or compilation. The interpretation of these conditional statements is sometimes done during compilation [4] and sometimes by a pre-processor [5]. But in either case there are meta-conditional statements that contain Boolean expressions which cause the compiler to ignore or to compile the fragments of code referred to by the conditional statements. When editing such a multi-version program, these meta-conditional statements are visible, and alternative fragments of code that would not coexist in any version are seen close together. If overused, the result can be impossible to deal with.

Representing sequences of versions over time by this method is unworkable, since the number of changes are great and not normally of interest to the programmer. Thus, we often see two different version-control systems being used (and having to be learned) within the same project: a delta system for recording sequences over time and an internal Boolean expression system for concurrent versions. Worse, more than one Boolean expression system is often used, one for each programming language being used. However, human factors aside, there is no multi-version program that cannot be represented by the internal Boolean expression method, a fact exploited by the approach to be described in this paper.

• *Support by an editor*

In the cases of both deltas and internal Boolean expression control, the raw data representation is quite unnatural, and the programmer may have to deal with an object that is little like a source program. But in the case of deltas, a method has been developed to avoid this problem. Editors, such as XEDIT [6], have been built to interpret the delta language and can thus be instructed to edit some version, usually the latest, of a multi-version program represented as deltas. The user is then placed in an environment where it appears to him that he is editing this version, even though it does not exist as such outside the editor. He uses normal editing commands, but when he declares that he is finished editing, only the changes are recorded, in the form of a delta, rather than the entire version. This delta would optionally be a modification of the last delta applied when he started editing, or it might be a new delta to be added to the delta sequence.

This method was a significant improvement over the direct editing of deltas. It solved all of the human factor problems, but left the problems of dealing with concurrent versions, which are inherent with the delta representation. This paper presents a way to extend this idea to the internal Boolean expression representation.

It is interesting to observe how the limitations of the delta representation show through such a delta editor. Take again the example of the three-version program with version X, the base, and deltas Y and Z. When the programmer changes version Y with this editor, he is not told which parts of it are the same in version Z. Even if he knew this from experience, he could not change both versions at once. So we see that he needs two facilities: notification of commonality and the ability to change more than one version at a time. The following discussion shows how this can be provided by permitting him to edit both versions at once and defining appropriate methods of displaying them to inform him which sections are common to both. This has not been attempted using a delta representation; any such attempt would result in generating an unmanageably large number of deltas. This technique is only realistically possible with the internal Boolean expression representation.

4. Generalized support by an editor

• *The beginning*

When the editing of a file begins using the method described in this paper, the entire multi-version file is brought into storage so that the user can quickly choose the versions that are to be edited at the moment. All code fragments and controlling Boolean expressions are immediately available to the editor, so that it can easily create whatever environment the user requests. The user can define a default environment so that he is not necessarily put into the position of editing all versions initially. For example, if there were a sequence of versions, he would probably want to specify that he be automatically editing only the latest version. He might also want a new latest version to be created for him. From that point on in this simple case, he could use the editor normally with no special knowledge that he is editing a multi-version program.

Another kind of automatic initialization might be required for an application program that consists of many related files. Here, the user may specify that no matter which of these files he is editing, the corresponding versions of them should be available. Thus, for example, if he is editing the latest version of a program and he also wishes to edit the documentation file, the latest version of the documentation file will be edited as well. Conversely, it is important to be able to edit other files under independent version control.

• *Sequences of versions*

With nothing more said, this support for sequences of versions is the same as that provided by delta-oriented

editors, which can apply and generate deltas. But since the editor can permit the simultaneous editing of more than one version, the user could request to edit, say, both last month's version and the current version, or even all the versions that have existed since the beginning of the year. The details of how this works are left to the next section, but it can now be seen that since the editor is responsible for keeping the user informed of the commonality among the versions being edited, it must in this case highlight the fragments of code that differ among the points of time selected (these being the parts not common among the versions). A command is provided to restore some code from the past into the present (RESTORE). Even such a change as that is properly recorded in the sense that the restored code will be controlled by a Boolean expression that specifies that this code existed for a time, then disappeared, and then existed once again (1981 <=YEAR<1982|YEAR>=1983). This is an implicit way to introduce an OR into Boolean expressions. As will be seen, most complex Boolean expressions are introduced in such implicit ways, thus freeing the user from having to deal with them explicitly.

• *Concurrent versions*

Since the underlying data structure of these multi-version files is the traditional one of Boolean expressions associated with code fragments, it is clear that the innovation of this editor is the way in which this material is presented to the user. As previously mentioned, it is important that he not be confused with complex Boolean expressions or code fragments that do not correspond to a normal source program. To achieve this, only one version of the program is actually displayed, even when he is editing more than one version. He controls which version is displayed, or if he fails to exert such control, it is done by the editor. In the latter case, the editor uses heuristics to avoid gratuitous changing of the version displayed. For example, no change is made while the user scrolls through the file. Also, if the user changes his environment, in the sense of which versions he is editing, and returns to a previous environment, the editor remembers which version it was previously displaying in that environment.

This control over which version is displayed can be seen as a multi-dimensional extension of the normal two-dimensional scrolling that editors provide (vertical and horizontal). While this can be thought of as adding a third dimension, it is most usefully thought of as adding many dimensions. If there is a sequence of versions over time, there is a TIME dimension. If there are concurrent versions to support the program running under different operating systems, there is a SYSTEM dimension. If there are concurrent versions to support single- and double-precision computation, there is a PRECISION dimension. Each of these dimensions varies independently from the others. The total number of versions in this case is the number of checkpoints times the number of systems supported times two (the number of precisions).

The version to be displayed can be specified by requesting the editor to HIDE some code that is displayed or UNHIDE some that is not. These commands often leave a choice for the editor to arbitrate if there are more than two versions of that code. In order to make sure that all are seen, the user can request a VIEWSHOW that starts a loop, displaying each in turn when the user requests STEP. The loop is terminated by UNSHOW. Such a loop can be imbedded within other loops. All editing is permitted even when such a loop is active. The final and least often used alternative is to specify an explicit Boolean expression (VIEW PRECISION=DOUBLE). This might or might not leave a choice for the editor to arbitrate, depending on the complexity of the Boolean expression specified with respect to the Boolean expressions controlling the code looked at.

The user must be warned that other versions of the code he is looking at exist, just as editors often tell him that he is missing something above, below, to the left of, or to the right of the screen. This is done by highlighting (e.g., brightening) displayed code that is in some, but not all, versions being edited. However, this leaves nothing to highlight if a section of code is absent in the displayed version but not in all versions being edited. In this case, an adjacent section is highlighted, perhaps in a different color, if available. By this method the user sees nothing but a normal source program, yet is warned of which code is common and which varies among the versions being edited.

• *Selecting the versions to be edited*

The versions to be edited are ultimately selected by specifying a Boolean expression, although this is usually done implicitly. This Boolean expression is called the *mask*, since it masks out the unwanted versions. The editor simulates that the masked-out versions do not exist by comparing the Boolean expressions controlling the code fragments with the mask. If a particular Boolean expression is inconsistent with the mask, the editor operates as though its code did not exist. If that Boolean expression must be true given the mask (the mask implies the Boolean expression), the corresponding code is in all versions being edited. Such code is called *fixed*, in the sense of "determined"—it is determined that the code is in the versions being edited. If that Boolean expression might or might not be true given the mask, the corresponding code is in some, but not all, of the versions edited and is displayed bright if it is in the version being displayed. Such code is called *unfixed*.

In order to permit parts of the mask to be manipulated independently, the editor actually provides any number of masks. But operationally, there is only one, the AND of all

the masks. Multiple masks are just a technique of referring to parts of the real operational mask. Some of these masks have user-chosen names, often denoting the restriction they define, say, TIME. Others form a push-down list to permit a user to save one version of an environment and enter a more restrictive one, a common operation (PUSHMASK and POPMASK). Masks can be turned off temporarily (TURN TIME OFF) to permit the editor to remember its correct value, but to ignore it for now.

As pointed out before, masks are often set in response to a user's declaration of the default environment desired when an editing session begins. A typical command that might be issued automatically is MASK V VERSION>=2.3.4 to set the V mask so that only the latest version is being edited. Note that sequences are defined by a greater-than-or-equal relation so that the next version, established by MASK V VERSION>=2.3.5, say, will incorporate the earlier version except where explicitly changed in the new version.

Users often zoom into a more restrictive environment, make a few changes, and return. Often this is done as a result of the user's understanding of the meaning of some unfixed code. He might say that he wants to make changes only to the versions that contain that code by using the FIX command to set a mask so that the code ceases to be unfixed. Or, alternatively, he might say he wants to make changes only to the versions that do not have that code by using the EXCLUDE command, which similarly sets a mask. After making the changes he had in mind, he would use the UNMASK command to clear the mask just set. In order to make a change to some unfixed code itself, it is not necessary to go through the trouble of fixing it, making the change, and then clearing the mask. In that case, normal editing commands can be used, since no editing operation will modify versions not currently being edited. Similarly, there is also a shorthand for inserting some code after some unfixed code only in the versions that contain it (ANNEX).

There is an analogy between the HIDE command and the EXCLUDE command. HIDE is a multi-dimensional scrolling command that gets rid of a line only in the sense of what is being displayed. EXCLUDE is a version control command that gets rid of code in the sense of restricting which versions are being edited. (Contrast these with DELETE, which gets rid of code in the sense that it no longer exists in the versions being edited.) Similarly, the VIEWSHOW command is analogous to the SHOW command. VIEWSHOW guarantees that each version of a section of code will be displayed, and SHOW guarantees that a mask will be set in turn so that each version of a section of code will be the version edited. As with VIEWSHOW, SHOW is used to assure that no versions are missed. Note that "each version of a section of code" does not mean each version of the entire program, since that section may not vary in some dimension. So although the code against which the SHOW was issued will necessarily be fixed during the loop, other sections of code might well remain unfixed. A typical use of SHOW is to make corresponding, but different, changes to the one section of code according to which version of another will coexist with it.

- *Internal operation*

Since the current mask (the AND of all masks turned on) determines which versions are being edited, a request to insert new code into the file is done so that the controlling Boolean expression for it is precisely the current mask. Similarly, the request to delete code from the file modifies that code's Boolean expression so that it becomes the AND of what it used to be and the NOT of the mask. If the result of this AND is FALSE, the code is in no version and can be physically deleted. Modifying existing code is just a combination of deleting it relative to the current mask and inserting its new version as before. But here if the modified code was unfixed, the inserted code would be further restricted by making its controlling Boolean expression be the AND of the original controlling Boolean expression and the mask.

- *Dealing explicitly with Boolean expressions*

The above method of presenting multi-version files was designed with the intention that the user need never deal with Boolean expressions explicitly, except for simple relations when a new version is created. But as in most high-level approaches, this is only imperfectly the case.

A number of commands are provided to display the Boolean expressions controlling fragments of code. BOOL does that for a small segment of code. SHOWPARM displays the dimension names (VERSION) and the values to which they are compared in relations (2.4.3 and so forth). SHOWUNFIXED displays all alternatives at once, along with the controlling Boolean expressions for a small segment of code.

Similarly, commands are provided to display the masks. MASKS displays the names of the masks in use. MASK displays the value of one of them. SHOWMASKS displays all the masks, along with their values, and which are turned on or off.

Also provided is a way to modify the controlling Boolean expressions using the same editing commands as are used for the text (EDITMODE BOOLEAN). Such direct modification could result in an attempt to introduce syntax that is not representable internally. Such errors are caught and reported. The editor must assure that any such direct editing be done only to the versions being edited, as usual. This is accomplished by making the new Boolean expression be $(B\&\neg M)|(B'\&M)$, where B is the previous Boolean expres-

sion, M is the current mask, and B' is the new Boolean expression. Because of this, even though the modification was specified using normal text editing commands, the result will generally be more complex than a simple textual change.

More common than direct editing of Boolean expressions is the use of the MAKE command. This is not a textual editing command and does not depend on the EDITMODE feature. It edits the Boolean expressions in a semantic fashion and is, therefore, much easier and safer to use. It acts on a segment of code, typically the entire program, and takes as arguments an equal relation and an arbitrary Boolean expression. It searches the Boolean expressions that control the specified segment of code for semantic occurrences of the equal relation (VERSION=2.4.3). Any relation based on the same dimension as in the equal relation (VERSION) is deemed a "semantic occurrence." This is because such a relation can be thought of as having that equal relation as part of it [VERSION>2.4.3 can be thought of as VERSION>2.4.3 & ¬(VERSION=2.4.3)]. MAKE replaces the equal relation part of each such relation with the specified Boolean expression (see Table 1). The result may be simply to change the names being used in the Boolean expressions (MAKE GENDER=MALE SEX=MALE). Or it might be to introduce a new version that is to be initially the same as an existing version (MAKE GENDER=MALE GENDER=MALE | GENDER=NEUTER). It may even be used to remove a version altogether (MAKE GENDER=NEUTER FALSE). But it never will have the effect of creating a version of the segment to which it is applied that did not exist before. That is, it cannot place two mutually exclusive segments of code in the same version or remove part of one version of the segment without removing all of that version. Some of its uses are a bit obscure but can be nicely packaged in an editor that has macros or other means of extensibility. For example, the PURGE macro can be used to remove all versions prior to 2.4.3 (MAKE VERSION=2.4.3 VERSION<=2.4.3).

Of course, as with EDITMODE BOOLEAN, the editor assures that the modifications made by the MAKE command are done only to the versions being edited $[(B\&\neg M|(B'\&M)]$. Therefore, the previous example to remove old versions from the file would have to be preceded by a command to cause these versions to be edited (TURN V OFF).

To change an equal relation E to a Boolean expression B in a semantically consistent fashion, all relations R that make up the Boolean expressions involved that act over the same dimension as E must be replaced as in Table 1. The value to which a dimension name is compared in a relation X is shown in the table as $V(X)[V(\text{VERSION}<2.4.3)$ is 2.4.3].

Table 1 Replacements for the MAKE command.

Type of relation, R	$V(E)<V(R)$	$V(E)=V(R)$	$V(E)>V(R)$
<=	$R\|B$	$R\|B$	$R\&\neg B$
=	$R\&\neg B$	B	$R\&\neg B$
>=	$R\&\neg B$	$(R\&\neg E)\|B$	$R\|B$
<	$R\|B$	$(R\|E)\&\neg B$	$R\&\neg B$
¬=	$R\|B$	$\neg B$	$R\|B$
>	$R\&\neg B$	$R\&\neg B$	$R\|B$

5. Related work

All techniques of dealing with multi-version programs mentioned here have a common deficiency: a particular segment of code can occur only once (zero or one time) in a given version, and the relative order of code cannot be different between versions. This forces an artificial duplication of code in certain unusual cases. It appears that Theodor Nelson plans to address these cases in his XANADU system, proposed but not yet implemented [7].

James King has proposed a related editor that would permit the user to view a program under various simplifying assumptions. This would differ from the editor discussed here in that the purpose is to clarify a single-version program. The Boolean expressions acted upon would be those coded in the program itself. This editor would use King's program reduction techniques [8].

6. P-EDIT

The ideas presented here have been implemented by the author and Paul Kosinski [9, 10] using a Boolean expression simplifier written by Peter Sheridan [11] at the IBM Thomas J. Watson Research Center at Yorktown Heights, New York. It has a small community of users within IBM, and the next step is to arrange a realistic field trial in a substantial program development project.

The normal edit commands, not the ones introduced for multi-version programs, are like the ones typically supported by editors that run under the IBM Virtual Machine/System Product (VM/SP) [12]. Since P-EDIT runs under VM/SP, there is little new for users of that system to learn in order to edit a single-version program or the latest version of a multi-version program representing a sequence of versions.

To provide support for any kind of file, none of the existing representations of internal Boolean expression control were chosen, since each supports only the programming language for which it was designed. Rather, each physical line of a multi-version file contains two fields, the normal text for the line and the Boolean expression that controls in which

versions that text should appear. Since variable-length records with more than one terminal blank are rare in the VM/SP environment, the existence of exactly two terminal blanks is used to signify that the preceding nonblank characters are a Boolean expression. The absence of a Boolean expression is a shorthand for TRUE and permits a single-version file to be treated as a multi-version file that happens to have only one version.

This compromise to achieve language independence works quite well in practice. However, objections have sometimes been raised in the case of structured languages (where indentation is typically used to illustrate the structure) when identical code occurs at different levels in two versions. For example, in one version a fragment of code might be executed unconditionally and in another conditionally, which would make it indented more. This would make the entire fragment formally different in the two versions, not just the indentation of it. Also, with files representing documentation, the user will get better results if lines are broken at natural syntactic points, rather than when there is no more room on a line. The latter will tend to generate too many unimportant differences between versions when, say, a paragraph is reformatted following the removal of only a few words.

But all the above comments apply only to P-EDIT *per se,* not to this method of dealing with multi-version files in general. The general method would fit quite well in syntax-directed editors, which have some knowledge of the programming language [13, 14]. In fact, it was first implemented for such an editor [15, 16]. It also would fit quite well into a true document editor that continually keeps the document formatted [17].

Any editor is greatly improved by having facilities by which it can be extended, especially a multi-version editor. P-EDIT permits this by using a general-purpose macro interpreter, EXEC 2 [18], which is part of VM/SP. In addition to macros, which extend the command set, P-EDIT has quite a few places where it calls such programs. Examples are whenever a specified number of changes are made to a file, whenever the edit session starts, and whenever editing of a particular file starts. The latter is done according to the type of file being edited, usually the name of the compiler that will process it. This is how the aforementioned default environment is specified, by writing an EXEC 2 program to issue the P-EDIT commands that should be initially done. Thus, multi-version files typically have a different file type for each application being maintained, and by convention files of the same type share the same masks. In addition, EXEC 2 is used to permit the user to write extended Boolean expressions [MEMBER(X, (A, B, C)) would become X=A | X=B | X=C] and the values within them [TIME>=CURRENT(TIME) would become TIME>=1983.4.5.30.4.1]. These are also used as an interface to the file system to permit, for example, convenient access to stored Boolean expressions [READ(YORKTOWN, SYSGEN) would become the Boolean expression stored in the SYSGEN file under the name YORKTOWN].

P-EDIT also supports an UNDO command, which permits recent commands to be undone entirely. This, clearly, is an important feature for any editor, but is particularly important for editors such as P-EDIT that record changes rather than simply make them. This is because trivial mistakes are often immediately detected and repaired. Without an UNDO command, the repairing would involve normal edit commands that just happened to restore the code. This cannot easily be detected by the editor and would still be recorded as a change, thus generating spurious differences between versions.

In cases where the availability of UNDO fails to avoid spurious differences between versions, the user can use the MERGE command to combine mutually exclusive, identical text. A syntax-directed editor could do this merging operation automatically.

7. Summary

Many programming projects involve multi-version programs in one way or another. The programmers have to suffer either dealing explicitly with a confusing data structure or dealing with one that often proves inadequate to their needs. A technique has been outlined that permits more than one version to be edited at once without confusing the user by displaying version control information or the code for more than one version. This offers a new high-level way to interact with multi-version files that promises to make such programs more reliable, less expensive, and useful to a wider number of users.

It is hoped that widespread use of these techniques will not only solve problems recognized today, but open up new opportunities. While it is always difficult to anticipate such gained opportunities, it seems likely that these techniques will permit software manufacturers to respond more flexibly to their customer's individual needs and to remove much of the formal interaction required between programmers working in large groups.

Acknowledgments

The author is indebted to those who worked with him to implement P-EDIT, Paul Kosinski and Peter Sheridan. Support and encouragement have been provided at different times by various IBM Research Division managers: Patricia Goldberg, Archie McKellar, and Herbert Schorr. Also due mention is Leroy Junker, who as an inquisitive and aggres-

sive user, *par excellence*, has made getting the bugs and quirks out of P-EDIT far easier than it would have been otherwise.

References

1. R. D. Gordon, "The Modular Application Customizing System," *IBM Syst. J.* **19,** 4, 521–541 (1980).
2. G. E. Kaiser and A. N. Hapermann, "An Environment for System Version Control," Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, February 2, 1982.
3. M. J. Rochkind, "The Source Code Control System (SCCS)," *IEEE Trans. Software Eng.* **SE-1,** 370–376 (December 1974).
4. *IBM Document Composition Facility: User's Guide (SCRIPT/VS)*, Order No. SH20-9161, available through IBM branch offices.
5. *IBM OS PL/I Optimizing Compiler: Programmer's Guide*, Order No. SC33-0006, available through IBM branch offices.
6. *IBM Virtual Machine/System Product: System Product Editor Command and Macro Reference (XEDIT)*, Order No. SC24-5221, available through IBM branch offices.
7. T. Nelson, "A New Home for the Mind," *Datamation* **28,** 3, 168–180 (March 1982).
8. J. C. King, "Program Reduction using Symbolic Execution," *ACM-SIGSOFT Software Eng. Notes* **6,** 1, 9–14 (January 1981).
9. V. Kruskal, "Applications of Parametric Files Using P-EDIT," *Research Report RC-8628*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1980.
10. V. Kruskal, "P-EDIT Reference Manual," available from the author.
11. P. B. Sheridan, "A Formula Decision/Simplification Program," *Research Report RC-7132*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1978.
12. *IBM Virtual Machine/System Product: Introduction (VM/SP)*, Order No. SR-20-6200, available from IBM branch offices.
13. C. N. Alberga, A. L. Brown, G. B. Leeman, M. Mikelsons, and M. N. Wegman, "A Program Development Tool," *IBM J. Res. Develop.* **28,** 1, 60–73 (1984, this issue).
14. T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Commun. ACM* **24,** 563–573 (September 1981).
15. W. G. Howe, V. Kruskal, and I. Wladawsky, "A New Approach for Customizing Business Applications, *Research Report RC-5474*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1975.
16. V. Kruskal, "An Editor for Parametric Programs," *Research Report RC-6070*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, 1976.
17. J. Prager and S. Borkin, "POLITE Progress Report," *Scientific Center Report G320-2140*, IBM Scientific Center, Cambridge, MA, April 1982.
18. *IBM Virtual Machine/System Product: Reference Manual*, Order No. SC24-5215, available from IBM branch offices.

Received May 5, 1983; revised August 24, 1983

Vincent Kruskal *IBM Research Division, P.O. Box 218, Yorktown Heights, New York 10598.* Mr. Kruskal was manager of systems development at the University of Chicago Computation Center prior to joining the IBM Research Division in 1971. There he has worked in the areas of operating systems and program development tools. He is currently working in the office systems group in the Computer Sciences Department.