A Flexible Object Merging Framework

Jonathan P. Munson and Prasun Dewan
University of North Carolina-Chapel Hill
Chapel Hill, NC 27599-3175
{munson, dewan}@cs.unc.edu

ABSTRACT
The need to merge different versions of an object to a common state arises in collaborative computing due to several reasons including optimistic concurrency control, asynchronous coupling, and absence of access control. We have developed a flexible object merging framework that allows definition of the merge policy based on the particular application and the context of the collaborative activity. It performs automatic, semi-automatic, and interactive merges, supports semantics-determined merges, operates on objects with arbitrary structure and semantics, and allows fine-grained specification of merge policies. It is based on an existing collaborative applications framework and consists of a merge matrix, which defines merge functions and their parameters and allows definition of multiple merge policies, and a merge algorithm, which performs the merge based on the results computed by the merge functions. In conjunction with our framework we introduce a set of merge policies for several useful kinds of merges we have identified. This paper motivates the need for a general approach to merging, identifies some important merging issues, surveys previous research in merging, identifies a list of merge requirements, describes our merging framework and illustrates it with examples, and evaluates the framework with respect to the requirements and other research efforts in merging objects.

KEYWORDS: diff, flexible coupling, optimistic concurrency control, merging, undo, versions.

INTRODUCTION

In the course of collaboratively producing a document or some other artifact, collaborators often find that they have created two versions, each containing revisions that they wish to have in a single version. It then becomes a task to take the set of revisions from one version and re-apply them to the other version of the object. A tool that points out the differences between the versions may be employed to ease the task, if such a tool is available, but merging the versions remains essentially a manual process. This situation is objectionable on two counts: first, the needed differencing tools may be unsatisfactory or unavailable, and second, the process of re-applying one set of changes to an object to another version of the object is error-prone and time-consuming. A tool that performs the merge automatically, responding appropriately to conflicting changes, would be highly useful.

0

A number of merge tools already exist. Tools for merging plain-text files include the UNIX *diff3* tool, the RCS *rcsmerge* tool [11], and the *fileresolve* tool in Sun's Network Software Environment [1]. Research efforts include the work of Horwitz, Prins, and Reps with a program merging tool that detects an inconsistent merge through the use of program dependency graphs [6], and the GINA collaborative application framework [2], which allows users to merge revised versions by merging command histories. Work related to object merging includes the PREP *flexible diff* tool [9], which gives users flexibility in finding and pinpointing differences, and the concurrency control model of Ellis and Gibbs [4], which determines a consistent ordering of operations at all sites in a distributed collaborative environment by merging concurrent operations.

Current merge tools are either limited by being based on plain text files, or are not adaptable to particular collaboration contexts. We have developed a novel, flexible object merging framework that allows definition of the merge on the basis of the particular application and the context of the collaborative activity. It performs automatic, semi-automatic, and interactive merges, supports semantics-determined merges, operates on objects with arbitrary structure and semantics, and allows fine-grained specification of merge policies. It is based on an existing collaborative applications framework and consists of of a merge matrix which defines merge functions and their parameters and allows definition of multiple merge policies, and a merge algorithm which performs the merge based on the results of the merge functions. In conjunction with our merge framework we introduce a set of merge policies for several useful kinds of merges we have identified.

The rest of the paper is organized as follows. We begin by identifying collaboration scenarios in which the problem of merging arises. We then highlight some of the issues that confront a general approach to merging, follow this with a discussion of existing merge tools and previous research in merging, and motivate and describe a set of requirements for merge tools. We then describe our merge scheme and provide examples of its use. Finally, we evaluate our merge scheme with respect to its ability to simulate the existing merge tools discussed previously.

NEED FOR OBJECT MERGING

The problems of merging do not arise in all collaborations. For instance, if the collaboration consists of brainstorming, then there is no need for producing a final merged artifact. In a large number of artifact-based collaborations, however, the need for merging potentially conflicting changes arises. In this section we identify conditions in such collaborations under which this problem arises, in terms of the functions collaboration environments may provide. We will use as examples two collaboration scenarios that we feel are typical situations requiring object merging. They will be referred to throughout the paper.

1. Two authors are collaborating on a journal article. As the deadline approaches, it becomes necessary for the authors to work on the document simultaneously, instead of the back-and-forth editing they had been doing. One corrects scattered figures and the associated references to them, and the other adds citations. They are using ordinary text editors and so are forced to work on different copies of the text, thus creating two different versions that will need to be merged before the article is submitted.
2. Two programmers collaborating on a compiler project are using the Unix parser generating tool, Yacc, to create the grammar of the compiler. They have a basic grammar completed and now work on separate versions so that they may develop different portions of the grammar. One works on all the productions that are program statements and the other works on the productions that define legal expressions in the language. At some point they will need to merge the versions before they can make further progress on the compiler.

Coupling. Merging is not useful in those collaborative activities where an artifact is always WYSIWIS (What You See Is What I See) [8] coupled. In this case concurrent edits never produce separate versions of an artifact. Merging is primarily useful where the coupling between views, or versions, of an object allow asynchronous collaboration. Concurrent editing with existing tools that do not offer coupling will of course result in separate versions, as is the case in our text document collaboration example above. The Yacc grammar collaboration provides an example of when uncoupled concurrent editing is in fact desired: during the time the programmers are working individually, the need to repeatedly edit, compile, and test requires the grammar versions to be kept in separately consistent states.

Merging may also be useful in cases where views of an object are normally coupled but become temporarily uncoupled when coupling mechanisms are unable to operate. As an example, take the case where a teleconference is interrupted by a communications failure. Conferees at both ends may continue to work, and as a result, when communications are restored, their work needs to be merged. Mobile, or "nomadic," computing, where users' environments follow them when they are out of the office, may also provide cases of this kind.

Concurrency control. Merging may be useful in an optimistic long transaction model. Under this model, collaborators who have made inconsistent changes would typically wish to merge these changes rather than abort one of their transactions. Merging in general provides "late" concurrency control for collaborations where concurrency control is required but cannot be enforced—due to lack of tool support, for example.

Access Control. In situations where collaborators work on separate versions of an object and access control mechanisms cannot enforce the access control policy desired, a suitably-defined merge procedure can provide late access control by accepting revisions in the merged object on the basis of the author of the changes. An example of this use is given later. Access control used in conjunction with partitioning—where the collaborators divide the object into sections to which one or the other has exclusive access—can prevent the need for merging, but this is not always a viable solution. Partitioning may not be practical when the object is not modularized, such as the Yacc grammar, or when users' changes need to span multiple, overlapping, modules. Moreover, the partitioning needed for a particular situation may be difficult to anticipate because it is not known beforehand who will make what changes. Furthermore, the appropriate partitioning may well change over the course of the collaboration. In the example of the journal article writers, all these objections to partitioning may be relevant.

Undo/Redo. In systems that provide collaborative undo/redo [7], the need for a merge capability may be somewhat lessened. Collaborators can merge their versions of the object by simply stepping through their edits and using a "redo" mechanism to apply one collaborator's changes to the other's version of the object. This is how merging is performed in the GINA environment. But as the edit histories grow in length, this becomes a less attractive merging procedure. The usefulness is also sensitive to the granularity of the edit histories: the finer grained the history, the more tedious merging becomes.

ISSUES IN OBJECT MERGING

In the preceding section we described how different versions of an object may arise in a computer-supported collaboration, thus engendering the need for tools to merge the versions. In this section we describe the merge problem itself, presenting informally what we regard as two different aspects of the problem. The first recognizes the fact that different collaboration contexts require different kinds of merging. The second focuses on the fact that the different kinds of objects we wish to merge have different properties, which influence the merge procedure. Our examples exhibit differences in both respects.

Merging in different collaboration contexts

In the text document example the only reason the two authors created versions of the document was so that they could work on it simultaneously. They would not have done so if their editing system had supported concurrent access to the file. The authors had different tasks and did not expect their changes to affect anything the other was doing. In the example of the compiler writers the case is somewhat different. The programmers had different tasks but their different tasks may have involved changes to the same parts of the grammar. Versions arose in this case not accidentally but intentionally. We might therefore expect the kind of merge performed for the two collaborations to differ. In the case of the two authors working on the journal article, we may expect the collaborators to want to simply include all changes in the merged document; in the case of the compiler writers, we may expect them to include some and negotiate over others—in fact, they may wish to discuss all changes. The merge procedure will operate somewhat differently in each case. We will refer to the first as a consolidation merge and the second as a reconciliation merge.

A consolidation merge assumes that each of the two revisions to be merged is a revision of a common version. There is also an assumption that while there may be some overlap or conflict between the changes, for the most part the changes are complementary. That is, a consolidation merge assumes that a meaningful version can be created that incorporates all non-overlapping changes. The merged version includes all objects that were inserted in either input version, does not include any objects that were deleted from either input version, includes all changed objects in either version, and includes all unchanged objects common to both versions. Collaborators choose between overlapping changes interactively.

A reconciliation merge does not assume complementary revisions between the two versions to be merged. It assumes that the reason for merging is to resolve conflicts. As a variation on the Yacc grammar example given above, suppose the two programmers have decided to each prepare alternative versions of that part of the grammar that defines expressions. When the two versions are completed, the programmers compare the versions side by side in order to choose the version they think better. They find, however, that they would like to take parts of both as the final grammar. In this case, they will perform a reconciliation merge of their two versions, and the merge procedure will consist mainly of making a selection of one of two choices. A definition of this kind of merge will be quite different from a definition for a consolidation merge. A reconciliation merge will not include all non-overlapping revisions as a consolidation merge will. In its most extreme form, a reconciliation merge will include only those revisions that both collaborators agree upon. For practical purposes, however, it would be desirable to allow certain kinds of revisions unchallenged—any additions would be accepted but agreement of the collaborators would be required for replacements and deletions.

Reconciliation merging and consolidation merging each represent what we call a merge "policy." A merge policy is a set of rules that determine which revisions will be included in the merged object; or a policy may provide the criteria that will used to determine which revisions will be included in the merged object. Later, when we have described our merge framework, a merge policy will have a concrete representation, allowing policies to be described more formally.

Merge policies may use mechanisms other than user interaction to determine which of a pair of conflicting revisions appear in the merged object. Suppose a salesperson keeps a file of contact information on both an office desktop machine and a laptop computer. The information is kept current by periodically merging the two files, which is done automatically whenever the computers are linked. It is possible that between merges the salesperson has modified the same piece of information on both systems, thereby causing a conflict when the two files are merged. An appropriate merge policy for this situation may be to examine the time stamp of each revision and keep the most recent one. Here we see a policy using an object attribute (time of last modification) to determine the merge result. It is also an example of a policy using automatic conflict resolution.

A merge policy may also examine the value of the objects to determine which revisions to accept. Let us suppose that two students collaborating on a short paper as a homework assignment both work on the paper the night before it is due, and although they worked on separate sections, both made changes to the Abstract section. Just before class the next day the students need to quickly merge their revisions, without time to view each other's changes. They feel that it is important to not submit anything with spelling errors so they set up their merge policy so that, in case of overlapping changes to a sentence, the sentence that passes the spell checker is chosen for the merged version. If both pass the spell checker the sentence of the first author is chosen. This is an example of a semantics-determined merge.

As a different kind of semantics-determined merge example, let us suppose an executive gave a project's proposed budget to two managers and received back their two counter proposals as versions of the original budget. To get a feel for the range of costs expected, the executive produces two merged versions of the budget: one in which for all items the lowest value of both versions is chosen, and another in which the highest value is chosen. The executive thus gets high and low estimates for the project.

Merging different kinds of objects

The two objects in our examples, a text document and a Yacc grammar, differ substantially in their structure and semantics. A text document may be defined as a sequence of paragraphs, where each paragraph is a sequence of sentences, each sentence is a sequence of words with some terminating punctuation, and each word is a sequence of characters. A Yacc grammar, on the other hand, is composed of production rules with a nonterminal on the left-hand side and a sequence of terminals and nonterminals (and perhaps semantic actions) on the right-hand side. In addition to being structurally different, the two objects are semantically quite distinct. A Yacc input object represents a grammar with explicit relationships between productions, whereas a text document has some simple dependencies such as between figures and references to figures, and some dependencies too complex to be machine-checked, e.g., “the new term was adequately defined before it was used.”

We may thus expect merge policies for the two objects to differ in those aspects of the merge policy related to the object structure. One such aspect is how the policy defines a merge conflict, by which we mean that situation when revisions from one version and revisions from the other version either overlap or, if both are accepted for the merged object, would potentially leave the object in an inconsistent state. A merge policy for a text document may define a conflict as changes to the same sentence, while a merge policy for a Yacc grammar may define a conflict as a change to the same production. Another aspect of a merge policy that is object-dependent is the conflict resolution mechanism. A tool which resolves changes to the same procedure in a program in one language will not be appropriate for programs in another language.

Collaborators may desire different policies for different particular structures, not just different types of structures as in the example of conflict definition above. For example, the compiler grammar programmers may designate certain nonterminal declarations as changeable by only one person, so that any concurrent changes to these declarations are in conflict. This is an example of a policy to enforce late access control. To accommodate these intra-object policy differences we must be able to define fine-grained merge policies.

High-level requirements for merge tools

Based on the issues in object merging that we discussed above, we have identified a set of requirements for general merge tools. A merge tool should:

- offer automatic differencing, i.e., not require users to manually find differences between two sets of changes.
- offer interactive merging, i.e., allow users to interactively select which revisions are included in the merged result.
- operate on general objects. A merge tool should not be restricted to objects represented as text files.
- provide conflict detection based on the structure of the object. This means that conflicts will be defined in terms of object structures (text sentences, program statements) instead of representation structures (lines in a text file).
- allow semantics-determined merges, i.e., provide mechanisms for the object semantics to determine the merge result. This may be through functions that evaluate which revisions to accept or functions that simply present semantic information to the users during the merge.
- let users specify the merge policy. Users should have the ability to tailor the merge policy according to their specific needs. This includes defining what sets of changes constitute a conflict.
- permit fine-grained definition of merge policies.
- offer automatic conflict resolution. A merge tool should allow users to determine beforehand how certain conflicts should be resolved, whether based on the kinds of changes made or the circumstances of the changes.

PREVIOUS RESEARCH IN MERGING

Object merging is not a new idea, not even for collaboration systems. Previous efforts have ranged from relatively simple systems for text documents to sophisticated semantics-based systems for computer programs. In this section we review these efforts and discuss them in light of the merge issues and requirements discussed above.

Software development tools

UNIX diff3. The UNIX *diff3* program compares three files and can output an *ed* script that, when applied to one of the three files, will produce a file that combines the changes in the other two, thus producing, in effect, a merged file. *diff3* is based on the *diff* program and so is strictly a text-oriented merging program.

RCS rcsmerge. The *rcsmerge* program is, like *diff3*, a text-oriented merge program, but is based on the mechanisms of the RCS system [11]. The same observations made about *diff3* above apply to *rcsmerge*. The advantages of *rcsmerge* over *diff3* are that it will warn of overlapping changes and that it is integrated with the safety and code management mechanisms of a versioning system.

Sun's NSE fileresolve. The *fileresolve* tool in Sun's Network Software Environment [1] is a differencing-based text merge tool that allows users to do side-by-side comparisons of two sets of changes to a file, and can automatically merge changes that do not overlap. Overlapping changes are flagged and presented to the user performing the merge, who may select one change or the other, or modify the file by hand. Changes may also be undone.

Semantic diff. Horwitz, Prins, and Reps [6] present an algorithm (which we refer to as "Semantic diff") to merge two different versions of a program in a semantically correct fashion. That is, if the two versions do not "interfere" with each other, Semantic diff will produce a merged program that incorporates the semantics of both versions. In brief, their merging algorithm first constructs program dependence graphs for the base version and the two revised versions, then merges the three dependence graphs into one dependence graph, checks the merged dependence graph for interferences, and finally produces the merged program from the merged dependence graph.

Collaboration tools

Flexible diff. *flexible diff* [9] is a tool in the PREP writing environment that finds and reports differences in text documents and computes scripts for automatically combining the differences in a merged document. *flexible diff* is different from the UNIX diff tool in that it allows users to choose the kind of differences that are found and how they are reported. Users first indicate to *flexible diff* the granularity of the differences they want found, with the choices being word, phrase, sentence, or paragraph. They are also offered a number of parameters which control how the differences are reported. One is granularity of the pinpointing, with the choices again being word, phrase, sentence, or paragraph. If the user chooses a pinpointing granularity of sentence, then any word differences will be shown as an old sentence deleted and a new sentence inserted. Other reporting parameters allow users to choose fine granularity finding and reporting but in cases where there are many word changes close together, have *flexible diff* report them as a phrase or sentence change—thus improving the difference report's readability.

The research on *flexible diff* has inspired our effort in this area by identifying the resolution of concurrent changes as a first-class research issue in collaboration systems and motivating the need for performing this task flexibly. The merge parameters of our work complement the differencing parameters *flexible diff* work has identified.

GINA. GINA [2] takes an approach wholly different from any of the previous merge procedures. Instead of basing the merge procedure on differences between object versions, GINA bases the merge on the command histories it keeps. The command history is central to the entire GINA system. GINA is a replicated architecture; applications maintain consistency with each other through exchanging commands. An application passes each command it receives from the user on to the other replicated applications in the form of a command object. Each application maintains a history of command objects. This history is the basis of GINA's undo/redo capabilities.

Command histories are used in the GINA merge procedure as follows. If two authors of a document begin with the same version and create different versions by executing different editing commands, the command history associated with the object forms a two-branched tree, where each author's changes form one branch of the tree. A merge of the two authors' changes is performed by taking one of the command object branches and applying it at the end of the other branch—in effect, redoing one author's changes on the other author's version of the object. This is the same principle by which *diff3* and *rcsmerge* work, except that these two programs are based on computed version differences rather than change histories. If one author has changed a part of the object that the other has deleted or has also changed, the authors are notified and will have to choose which change to keep. They may undo the affected operation of the first branch and redo the operation of the second branch, or simply not redo the operation of the second branch.

Groupware concurrency control (Ellis and Gibbs). The concurrency control model of Ellis and Gibbs [4] uses merging to ensure that each site in a distributed collaborative environment sees the same order of operations. At each site, the scheme merges the operations that are received from the other sites with the site's own operations according to a priority based on the site ID and the target object of the operation. No operations are aborted, but one operation may nullify the effects of another. The consistency achieved in this model is that each site sees the same order of operations.

Discussion of current systems

Table 1 shows how current merge tools compare with the requirements discussed discussed above. Only GINA is shown as supporting general objects. *diff3*, *rcsmerge*, and *fileresolve* merge line-oriented text objects, a class which includes a wide range of common entities such as text documents and computer programs, but does not include more structurally-complex objects such as spreadsheets and drawings. Semantic diff, because it is based on the semantics of a particular programming language, is restricted to operating on programs of that language. GINA offers true generality with respect to the object merged because it is based on command histories, which are common to all GINA applications.

Semantic diff is the only merge tool that offers object-based conflict detection; *flexible diff* offers object-based differencing. Semantic diff's conflict detection is based on the semantics of the object, i.e., whether a given statement modifies a certain variable, while *flexible diff*'s difference detection is based on the structure of the object, i.e., word or sentence or paragraph. *rcsmerge* and *fileresolve* detect when both

	automatic differencing	interactive merging	general objects	object-based conflict detection	semantics-determined merging	user-set policies	fine-grained policies	automatic conflict resolution
diff3	●							
rcsmerge	●							
fileresolve	●	●						
Semantic diff	●	●		●	●			
GINA	●	●	●	●				
flexible diff	●			●		●		

Table 1: Merge tool requirements.

versions contain changes to the same line, but we do not regard this as object-based conflict detection unless the object structure is simply "sequence of lines." For example, if the object was a text document and both versions of the document contained changes to the same sentence, these tools would notify the users if the changes were on the same line of the file, but not if the changes were on different lines. GINA conflict detection is object-based since command histories are specific to the type of object.

A FLEXIBLE OBJECT MERGING FRAMEWORK

Basis and overview

The basis of our flexible object merging framework is the Suite collaboration system [3], which provides general structured objects, fine-grained object attributes, functions for checking the semantic correctness of user changes, and flexible coupling between object views. Although Suite does not provide a version model as such, Suite's flexible coupling gives us the support for versions that we require for a basic implementation of a merging system. A basis of general structured objects allows us to take an application-independent approach to merging and to utilize the object's structure in the merge. Suite objects are constructed from basic types such as integers, reals, and strings, and aggregate types such as records and sequences. Finally, Suite's fine-grained object attributes, which may be inherited via type or structure, are used for flexible definition of merge policies.

Our merge framework consists of two elements: a *merge matrix* construct and an algorithm that uses the merge matrix to accomplish the merge of two object versions. The merge algorithm recursively traverses the structure of the object, top-down, and for each substructure takes both users' edits of the substructure and consults the merge matrix to determine which edits should be accepted for the merged result. We first describe the merge matrix in detail and then present the procedure that implements our merge algorithm.

The merge matrix

The merge matrix is a mechanism in the spirit of the access control matrix of operating systems theory [5] and the lock compatibility table developed by the research in object transactions [10]. The merge matrix has a row and column for each editing operation that can be performed on the object. The rows represent the edits of one user and the columns represent the edits of the other user. The entries of the matrix specify the action the merge procedure is to take for the pair of edits indicated by the (row, column) address of the entry. The general form of an entry is a function that returns the edit action to take. Function inputs may include anything that is relevant to choosing the action to take. We have identified a set of predefined merge functions and associated parameters, including functions that select changes on the basis of the time the changes were made or who made them. A function may invoke other functions, such as validation functions to check the consistencies of the revised objects. Typically, however, the entries of a merge matrix are the row or column edits themselves.

Table 2 shows a sample merge matrix for a Sequence object. The row and column operations in the matrix—"modify element #," "delete element #," and "insert after element #"—are for operations on the same element. If the elements of the sequence are structured objects, the "mod. elt. #" operation stands for all operations on that element; for a simple object the modify operation means "replace." We use several shorthand notations: "$\emptyset$" indicates that the user performed no operation on the element; "row" and "column" stand for the edits of the row or column user; "both" means that both edits should be accepted; "users" stands for the function that presents the users with the alternative changes and requests that they select one or the other, or another edit, then returns their choice to the merge procedure. Entries are blank where the two operations are never paired for comparison.

The value "merge edits" in the "mod. elt. #/mod. elt. #" entry of the merge matrix indicates that edits to an element that both users modified should be merged according to that element's merge matrix. This is done by invoking the merge procedure (shown in the next section) on that element. The particular policy of Table 2 is to request user selection if one user modified an element and the other deleted it; insertions of either user are always accepted.

Merge matrices are also defined for simple types such as

Sequence	ins. elt. #	del. elt. #	mod. elt. #	∅
ins. elt. #	both			row
del. elt. #		row	users	row
mod. elt. #		users	merge edits	row
∅	column	column	column	

Table 2: Merge matrix for a Sequence object.

String	modify	∅
modify	users	row
∅	column	

Table 3: Merge matrix for an atomic object.

integers and character strings (Table 3). Such merge matrices are called "atomic" merge matrices because the object is treated as an atomic unit. Atomic merge matrices are also used for structured objects when it would not make sense to merge changes to the object. For example, the article writers may feel that it never makes sense to merge changes to the same sentence. Users may want to set this atomicity property flexibly. In the early stages of their collaboration the writers allow changes to the same paragraph to be merged. But as the paper becomes more stable they decide that merging changes to the same paragraph is not safe, and wish to make the paragraphs atomic units. For this purpose, objects which have element operations, such as sequences, have two merge matrices defined: an atomic merge matrix and an element-operations merge matrix. Which is used by the algorithm is determined by a boolean attribute introduced for this purpose called the "atomic merge unit" attribute.

A merge matrix is defined for each structural level of an object and is implemented as a set of object attributes. In Suite, object attributes may be inherited from the parent structure (e.g., a record field may inherit the attributes of the record), or from type, or both (type first or structure first). Atomic merge matrices may be inherited via any of the four choices, since they are common to all types. Element-operations merge matrices are type-specific, however, having rows and columns for the operations of the structures they are defined for. Thus these matrices are inherited only from type.

Our merge matrix bears a resemblance to Schwarz's and Spector's lock compatibility table [10]. There are a number of differences, however. First, the outcome of consulting a lock compatibility table is solely the function of the pair of operations, whereas the outcome of consulting a merge matrix may be, in addition to the pair of operations, a function of the particular users doing the merge, the time the two operations were performed, or any other factor the users deem relevant. Second, a merge matrix is defined for each level of a structured object while a lock compatibility table is defined once for the whole object. Third, and following from the second, the lock compatibility table is defined for single operations, whereas the merge matrix includes a "modify" operation that may stand for multiple operations. Operations covered by a single "modify" operation are deferred to lower-level merge matrices.

The merge algorithm

We define the merge algorithm as a simple recursive procedure (in pseudo code), shown below. A change report is a list of changes to the object in terms of the operations in the object's merge matrix. Corresponding edits are edits by two users to the same element or field. If an edit in one version has no corresponding edit in another version, the entry for the version without an edit is "no edit." An edit triple is an object identifier followed by two corresponding edits to the object. Our algorithm first looks to see if the object should be treated atomically; if so, it invokes the appropriate function in the atomic merge matrix and performs the edits the function returns. Otherwise, the procedure gets the two change reports from the two revised versions of the object, computes a list of corresponding edit triples, and presents the list, element by element, to the merge matrix.

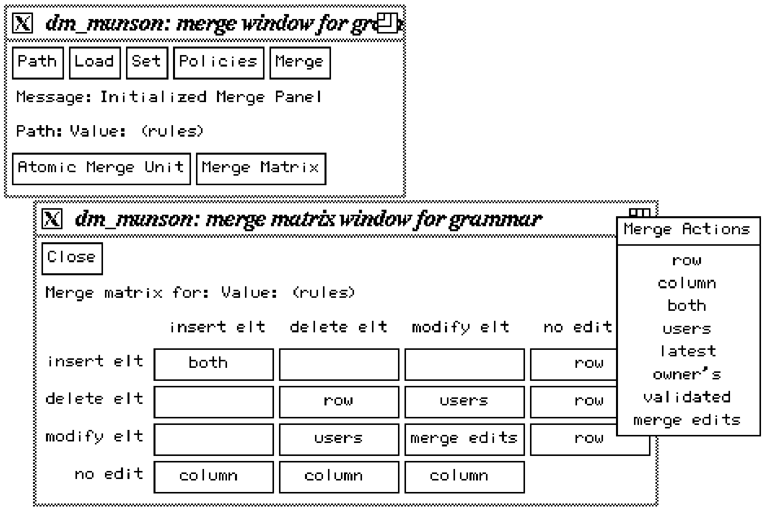
```
procedure MergeObject(X: Object)
  if X.atomic_merge_unit
    do X.atomic_merge_matrix[row edit, column edit]
  else
    get change reports for each version of object X
    compute list of corresponding edit triples
    for each triple <element Y of X, row edit, column edit>:
      if X.element_merge_matrix[row, column] = "merge edits"
        MergeObject(element Y of X)
      else
        do X.element_merge_matrix[row edit, column edit]
      end if
    end for
  end if
end MergeObject
```

AN EXAMPLE MERGING APPLICATION: A YACC EDITOR

To demonstrate our merge scheme we wrote a simple structured editing application, an editor for Yacc grammars. (The application was motivated by the first author's experience of collaboratively writing a Yacc grammar, the need to merge versions arising repeatedly throughout the collaboration.)

```
Production: Sequence of String;
Productions: Sequence of Production;
Rule: Record of {
  nonterm: String;
  prods: Productions; }
Rules: Sequence of Rule;
```

The merge matrix is presented in a Suite control panel, entitled "merge matrix window," shown in Figure 1. This window is opened from the window entitled "merge window," where users select the value or type in whose merge matrix they are interested, use the "Atomic Merge Unit" toggle button to choose the atomic or elment-operations merge matrix (if



```
Message: Initialized Merge Panel
Path: Value: (rules)
Merge matrix for: Value: (rules)
```

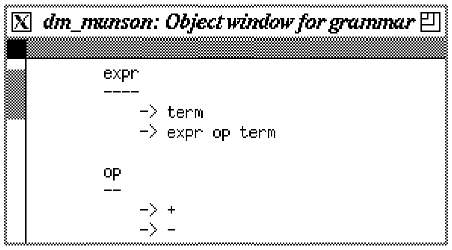
	insert elt	delete elt	modify elt	no edit
insert elt	both			row
delete elt		row	users	row
modify elt		users	merge edits	row
no edit	column	column	column	

Figure 1: Merge matrix window.

applicable) and then open the merge matrix window with the "Merge Matrix" button. The edit actions for merge matrix entries are selected from a pop-up window, also shown. The "Path" button at the top of the merge window sets the view whose merge matrix attributes will be loaded with the values shown in the window's matrix entries when the user clicks on the "Set" button. "Load" loads the existing values of the merge matrix into the window's matrix entries. The "Policies" button presents a pop-up menu of typical merge policies; these are used to set all the entries of the merge matrix with a single interaction. The "Merge" button invokes the merge procedure.

Example of object-based conflict detection

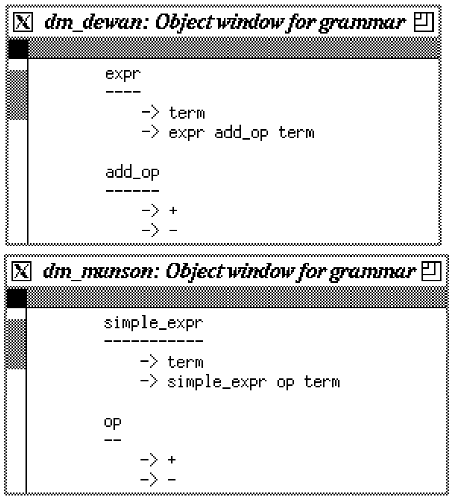
Our first example shows a case where no conflict exists but would be reported as a conflict by a non-structure-based tool such as NSE's *fileresolve*. Users Munson and Dewan are each modifying the expression section of their compiler's grammar, part of which is shown in Figure 2. Munson is adding relational operators and changes the name of the `expr` nonterminal to `simple_expr`. Dewan is altering the grammar so that multiplication and division have higher priority than addition and subtraction and changes `op` in the second production for `expr` to `add_op`. They decide to merge at this point, with their versions as shown in Figure 3.



```
expr
----
    -> term
    -> expr op term

op
--
    -> +
    -> -
```

Figure 2: `expr` and `op` before revision.


dm_dewan: Object window for grammar
dm_munson: Object window for grammar

```
expr
----
    -> term
    -> expr add_op term

add_op
------
    -> +
    -> -
```

```
simple_expr
-----------
    -> term
    -> simple_expr op term

op
--
    -> +
    -> -
```

Figure 3: Dewan's and Munson's revisions.

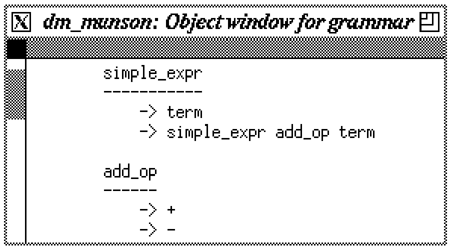
Record	modify field	∅
modify field	merge edits	row
∅	column	

Table 4: Merge matrix for `Rule`.

The merge policy for this application consists of merge matrices for each structural level of the grammar: `Rules`, `Rule`, `Productions`, and `Production`. Table 2 is used for `Production`, `Productions`, and `Rules`, and Table 4 is used for `Rule`. As mentioned before, "merge edits" in the "modify field/modify field" entry indicates that the merge matrix of the field object will be consulted. In all matrices Dewan's edits are the rows and Munson's are the columns.

An overview of the merge procedure for this example is as follows: change reports for the `Rules` sequence will indicate that both users changed the same elements (those for `expr` and `op`). The merge matrix for `Rules` indicates that the two sets of changes should be merged, so MergeObject is called on the `expr` element and change reports are generated. These indicate that only Munson changed the `nonterm` field, so his changes are accepted, but that both changed the `prods` field. The merge matrix calls for the changes to be merged, so MergeObject is called on the `prods` sequence. Change reports show that no element was modified by both users, so both sets of changes are accepted. Changes to the `op` nonterminal are merged likewise, with the result as shown in Figure 4.

At some point in the collaboration Dewan and Munson may feel that concurrent edits to the same production are likely to cause trouble and should not be merged. In this case they may set the "atomic merge unit" attribute of `Production` to True and use the default atomic merge matrix shown in Table 3. This will force them to select interactively one production or the other when both have changed the production.



```
simple_expr
-----------
    -> term
    -> simple_expr add_op term

add_op
------
    -> +
    -> -
```

Figure 4: Merged version of grammar.

Example of fine-grained merge policy

Dewan and Munson have divided responsibility for the grammar so that Dewan has responsibility for all productions stemming from the `statement` nonterminal and Munson has responsibility for productions from the `expr` nonterminal. Rather than enforcing this with access rights, they decide to set the merge policy to choose the changes of the responsible user over those of the other for overlapping changes at the nonterminal level. To do this, they set `expr`, `simple_expr`, `add_op`, and all other nonterminals underneath `expr` to be atomic units and set their atomic merge matrices to Table 5 (Dewan is row user). They do likewise for `statement` and its nonterminals but set the "modify/modify" entry to "column" so that Munson's changes are taken.

Unfortunately, a nonterminal called `subrtn` appears in productions for both `statement` and `expr` because it represents both a procedure call and a function invocation. Since both Munson and Dewan are working on sections of the grammar that use `subrtn`, they decide that any changes should be agreed to by both users before the changes go into a merged version. They thus choose Table 6 for the merge matrix of the rule that defines `subrtn`.

`expr`	modify	∅
modify	row	row
∅	column	

Table 5: Atomic merge matrix for `Rule`.

`subrtn`	modify	∅
modify	users	users
∅	users	

Table 6: Merge matrix for `subrtn` rule.

OTHER MERGE POLICIES

In this section we present seven merge policies, some of which have been discussed earlier: consolidation merging, reconciliation merging, merging when collaborators have strictly enforced roles, merging based on validation of changes, and merging based on computing changes as a function of the users' changes.

The merge matrix shown in Table 7 implements the merge policy we referred to earlier as a consolidation merge. Modifications to an element are chosen over a deletion of the element, as the conservative choice, but if both users have deleted the element the deletion is accepted. Insertions are always accepted.

The policy for reconciliation merging in Table 8 reflects a scenario where users are expecting many conflicts between their versions and will resolve these conflicts interactively. In particular, all deletions are handled interactively.

In the policy of Table 9, the collaborators are assigned strict roles. A scenario may be when one author is the primary contributor to the document and the other is a reader who is allowed to insert comments but may not change or delete anything, but the tools used do not offer this kind of access control. The rows of the matrix represent the edits of the primary contributor while the columns represent the edits of the reader. Disallowed edits of the reader are simply ignored.

The policy of Table 10 represents the case where an object temporarily exists as separate copies. This case may frequently arise in mobile computing, where a user may for a short period revise a home copy of some object (an address list for example) as well as an office copy. Since we assume that the user is aware of both sets of changes, we may assume

Sequence	ins. elt. #	del. elt. #	mod. elt. #	∅
ins. elt. #	both			row
del. elt. #		row	column	row
mod. elt. #		row	merge edits	row
∅	column	column	column	

Table 7: Policy for consolidation merge.

Sequence	ins. elt. #	del. elt. #	mod. elt. #	$\emptyset$
ins. elt. #	users			row
del. elt. #		users	users	users
mod. elt. #		users	merge edits	row
$\emptyset$	column	users	column	

Table 8: Policy for reconciliation merge.

Sequence	ins. elt. #	del. elt. #	mod. elt. #	$\emptyset$
ins. elt. #	both			row
del. elt. #		row	row	row
mod. elt. #		row	merge edits	row
$\emptyset$	column	$\emptyset$	$\emptyset$	

Table 9: Policy for merging with strict user roles.

Object	modify	$\emptyset$
modify	if Object.designated_user = row user row edit; elsif Object.designated_user = column user column edit; else merge edits;	row
$\emptyset$	column	

Table 11: Designated responsible user merge.

Object	modify	$\emptyset$
modify	if validated(row) $\wedge$ $\neg$validated(column) row edit; elsif $\neg$validated(row) $\wedge$ validated(column) column edit; elsif validated(row) $\wedge$ validated(column) users choose; (or arbitrary choice) else no edit;	row
$\emptyset$	column	

Table 12: Validation merge.

that in cases where changes overlap the most recent change is the one desired.

The policy of Table 11 formalizes the ad hoc policy that the two grammar programmers used when they assigned responsibilities for different sections of the grammar. It is similar to the one with strict user roles, except that in this case, both the identity of the user and the particular object involved are considered. Here, the users have divided responsibility for the document between themselves by attaching their ID to different sections (or in some other manner). When users' changes conflict, the changes of the user who has responsibility for the object involved are given priority. Conflicting changes to non-designated objects are selected by the users interactively. We could avoid a merge altogether in this situation with fine-grained access control, but then we would prevent the users from making helpful, non-conflicting changes to each other's sections.

Table 12 shows a merge policy which is parameterized by an application-specific validation function. The "row" and "column" arguments to the "validated" function stand for "row object version" and "column object version." The spell-checked homework example mentioned earlier would use this policy, using a spell-checker as the "validated" function.

Table 13 shows a merge policy parameterized by a function which computes the edit action for the merged object based on the edits of the object versions. For F, users may select from system-defined functions which include MIN, MAX, AVERAGE, SUM, and other arithmetic functions, string functions such as concatenation, or their own application-specific functions. The budget example with the low and high estimates would use this policy using MIN and MAX for F.

EVALUATION

In this section we evaluate the extent to which our merging scheme fulfills the requirements listed above and consider the degree to which it can simulate the merging tools discussed earlier. Following that we discuss the contributions we believe our merge scheme makes, and, finally, we identify areas of future research.

Fulfillment of requirements

Automatic differencing and interactive merging. Our scheme is fully automatic, or fully manual, or semi-automatic, according to the merge policies the collaborators select. Setting an entry of a merge matrix to the "users" function makes the selection for that pair of edits on that object interactive. Thus interactive or automatic operation may be chosen for particular substructures or particular types, giving users great flexibility in defining merge policies.

General objects. Our merge framework is based on the Suite system, which includes a type system of basic types and several type constructors. It therefore supports general,

Object	modify	$\emptyset$
modify	if row edit later than column edit row edit; else column edit;	row
$\emptyset$	column	

Table 10: Merging according to latest revision time

Object	modify	$\emptyset$
modify	replace with F(row, column)	row
$\emptyset$	column	

Table 13: Computed merge.

structured, arbitrarily-complex objects. Additionally, since our framework is based on the object structure, it merges objects independently of how they are displayed to the user. We may merge an object modified by one user with a scrollbar and edited by another user as text.

Object-based conflict detection. Our merge algorithm detects overlapping changes to objects, and so recognizes conflicts to the extent that overlapping changes represent conflicting operations. Our merge scheme also accommodates the detection of semantic inconsistencies by allowing verification functions to be called 1) before accepting one version's change to verify that the change did not bring the version to an inconsistent state, and 2) after an object or substructure versions have been merged, to verify that the merged result is consistent.

Semantics-determined merging. The validation merge policy and the computed merge policy were designed for semantics-based merging. The policies themselves do not include notions of object semantics but must be supplied with application-specific semantics functions.

User-set merge policies. Our framework provides two levels of flexibility in defining merge policies. At the lower level is the merge matrix, defined per structure, with general editing-action functions as its entries. At the higher level are the predefined, parameterized, merge policies. Some are parameterized by the value of a user-set object attribute, such as the designated responsibility policy, while others are parameterized by application-specific or user-supplied functions, such as the computed merge policy.

Fine-grained policies. Because merge policies are implemented as Suite object attributes, they are defined at each structural level of the object. Objects inherit a default merge matrix according to their type but this may be modified by users for particular structures.

Automatic conflict resolution. Our merge scheme allows conflicts to be resolved on the basis of any information users wish to encode in object attributes and base functions upon. Our framework provides predefined policies based on revision time and designated "owner," as well as policies parameterized by application-specific functions.

Coverage of other merge tools

UNIX diff3, RCS rcsmerge and NSE fileresolve. Our scheme, as presently implemented, does not utilize a shortest-edit-sequence algorithm to compute the differences between the base version of the object and the revised versions, as the *diff3*, *rcsmerge,* and (we presume) the *fileresolve* tools do. We may emulate their policies[1] by using the merge matrices of Tables 14 and 15.

[1] One feature of *diff3*'s policy we are unable to emulate. If in both versions a particular line is deleted, *diff3* deletes both that line and the line following it in the merged version.

Text	change line range	append lines	$\emptyset$
change line range	both	both	row
append lines	both	both	row
$\emptyset$	column	column	

Table 14: Policy to emulate *diff3*.

Text	change line range	append lines	$\emptyset$
change line range	users	users	row
append lines	users	users	row
$\emptyset$	column	column	

Table 15: Policy to emulate *rcsmerge* and *fileresolve*.

Semantic diff. Semantic diff may be simulated by using the computed merge policy supplied with the function used by Semantic diff to compute the merged program. This kind of use, of course, does not involve much of the generic support for merging our framework offers.

GINA. We may simulate the GINA merge procedure using the merge policy of Table 16. The difference remains, however, that the edits our merge procedure operates on are accumulated edits, whereas GINA uses actual edit histories.

Contributions

Our work towards a framework for flexible object merging has yielded the following contributions:

- a set of high-level requirements for merge tools;
- an evaluation of current merge tools with respect to these requirements;
- a new flexible object merging framework consisting of a merge matrix and a structure-based algorithm;
- several useful merge policies;
- an evaluation of our merge scheme with respect to generality and usefulness.

Future work

Our task of immediate importance is to gain more experience with merging in collaborative work. As our understanding of merging's role in collaboration grows, we hope to refine the merge tool requirements we identified and develop other

Object	any edit	$\emptyset$
any edit	users	row
$\emptyset$	column	

Table 16: Policy to emulate GINA.

important merge policies based on actual and hypothetical collaboration scenarios. We will also consider various user-interface alternatives. The user interface should not only offer various merge policies but also allow convenient customization of those policies; we do not yet have the experience to know what kind of convenience will be needed.

There are also more fundamental issues to explore. One such is n-way merging, i.e, merging n versions of an object. The merge matrix may be straightforwardly extended to n dimensions, but then what is the appropriate user interface? The number of merge matrix entries for n users and an object with m operations is $O(m^n)$, which even for two users is tedious to fill in. Applications will of course provide defaults, but users will still need to customize the matrix for particular situations.

Another interesting issue is how to handle whole-object operations. Currently we treat a sequence as either an atomic object or as a simple collection of elements, without regard to its property of being ordered. How should operations that modify a sequence with respect to this property, such as move-element operations, be treated in our merge framework? A related issue, but larger, is how to integrate user-defined abstract types into our framework.

Acknowledgments
The comments of the reviewers were helpful in improving the presentation of our work, as were discussions held with other members of the UNC Colab group. This research was supported in part by National Science Foundation grant IRI-9496184 and in part by a grant from the Software Engineering Research Center at Purdue University, a National Science Foundation Industry/University Cooperative Research Center (NSF Grant No. ECD-8913133).

REFERENCES

1. Adams, E., Honda, M., and Miller, T. Object management in a CASE environment. In *Proceedings of the 11th International Conference on Software Engineering* (May 1989), pp. 154–163.
2. Berlage, T., and Genau, A. A framework for shared applications with replicated architecture. In *Proceedings of Conference on User Interface Systems and Technology* (November 1993).
3. Dewan, P., and Choudhary, R. A high-level and flexible framework for implementing multi-user user interfaces. *ACM Transactions on Information Systems 10*, 4 (October 1992), 345–380.
4. Ellis, C. A., and Gibbs, S. J. Concurrency control in groupware systems. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (May 1989), ACM, New York, pp. 399–407.
5. Graham, G., and Denning, P. Protection—principles and practice. In *Proc. Spring Jt. Computer Conf.* (1972), pp. 417–429.
6. Horwitz, S., Prins, J., and Reps, T. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems 11*, 3 (July 1989), 345–387.
7. Knister, M. J., and Prakash, A. Undoing actions in collaborative work. In *Proceedings of the Conference on Computer Supported Cooperative Work* (Nov. 1992).
8. Lauwers, J., and Lantz, K. Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems. In *Proceedings of ACM CHI'90* (Apr. 1990), pp. 303–311.
9. Neuwirth, C. M., Chandok, R., Kaufer, D. S., Erion, P., Morris, J. H., and Miller, D. Flexible diff-ing in a collaborative writing system. In *Proceedings of ACM Conference on Computer Supported Cooperative Work* (October 1992), pp. 147–154.
10. Schwarz, P. M., and Spector, A. Z. Synchronizing shared abstract types. *ACM Transactions on Computer Systems 2*, 1 (August 1984), 223–250.
11. Tichy, W. F. RCS—a system for version control. *Software—Practice and Experience 17*, 7 (July 1985), 637–654.