Interactive Simultaneous Editing of Multiple Text Regions

Robert C. Miller and Brad A. Myers
Carnegie Mellon University
http://www.cs.cmu.edu/~rcm/lapis/
{rcm,bam}@cs.cmu.edu

Abstract

Simultaneous editing is a new method for automating repetitive text editing. After describing a set of regions to edit (the *records*), the user can edit any one record and see equivalent edits applied simultaneously to all other records. The essence of simultaneous editing is generalizing the user's selection in one record to equivalent selections in the other records. We describe a generalization method that is fast (suitable for interactive use), domain-specific (capable of using high-level knowledge such as Java and HTML syntax), and under user control (generalizations can be corrected or overridden). Simultaneous editing is useful for source code editing, HTML editing, and scripting, as well as many other applications.

1 Introduction

Text editing is full of small repetitive tasks. Examples include:

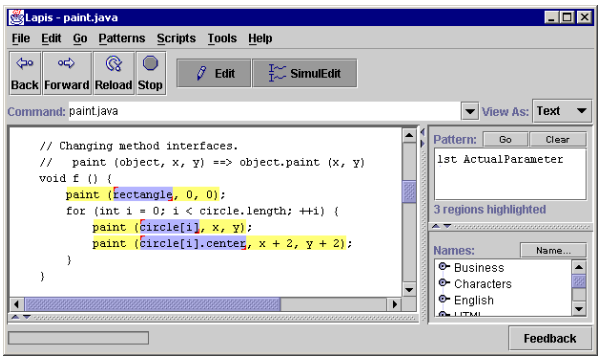
- Replace the string "Hashtable" with "Map" throughout a program.
- Reformat a list of phone numbers from "(xxx) yyy-zzzz" to "+1 xxx yyy zzzz".
- Insert print statements to trace entry and exit from each of a set of functions.
- Generate get/set methods for the instance variables of a class.
- Generate a mailing list from the From headers of a large file of email messages.

Appeared in *Proceedings of USENIX 2001 Annual Technical Conference*, Boston, MA, June 2001, pp 161-174.

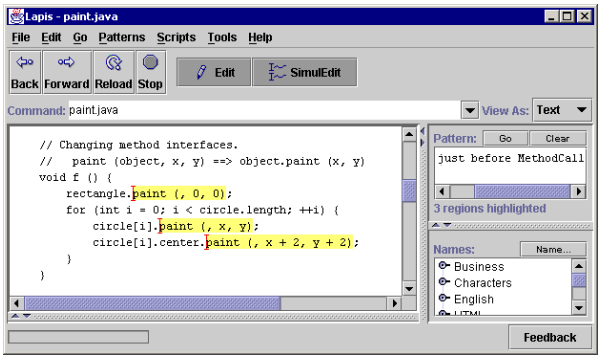
Users have a rich basket of tools for automating tasks like these. *Search-and-replace*, in which the user specifies a pattern to search for and replacement text to be substituted, is good enough for simple tasks. *Keyboard macros* are another technique, in which the user records a sequence of keystrokes (or editing commands) and binds the sequence to a single command for easy re-execution. Most keyboard macro systems also support simple loops using tail recursion, where the last step in the macro reinvokes the macro. For more complicated tasks, however, users may resort to a *custom program*, often written in a text-processing language such as Perl, awk, or Emacs Lisp.

This paper proposes a new technique to add to this basket of repetitive text editing tools: *simultaneous editing*. In simultaneous editing, the user first describes a set of regions to edit, called the *records*. This record set can be defined by a pattern, direct selection, or some combination of the two. After defining the records, the user makes a selection in one record using the mouse or keyboard. In response, the system makes an equivalent selection in all other records. Subsequent editing operations – such as typed text, deletions, or cut-and-paste – affect all records simultaneously, as if the user had applied the operations to each record individually. Figure 1 shows simultaneous editing in action.

Simultaneous editing has several advantages over other techniques for repetitive text editing. First, simultaneous editing is interactive. No programming is required. Second, simultaneous editing uses familiar editing commands, including mouse selection. Macro recorders generally ignore or disable mouse selection. Third, the effect of a simultaneous editing operation on any record is readily apparent from the selection. If there is a tricky step in a transformation, the user can check it beforehand by scanning through all records and verifying the location of the selection. Finally, mistakes made in the middle of a simultaneous editing transformation can be immediately detected and corrected with undo. Other techniques may require undoing, debugging, and reexecuting the entire transformation.



(a)



(b)

Figure 1: Simultaneous editing on Java code. The records (highlighted lightly) are calls to the `paint()` function, which is being transformed into an object-oriented method. (a) User selects "rectangle", and the system generalizes the selection across all records. (b) User cuts the selection, pastes it before `paint`, and inserts a dot. The same operation affects every record.

The greatest challenge to an implementation of simultaneous editing is determining the equivalent selection where editing should occur in other records. Given a cursor position or selection in one record, the system must generalize it to a description which can be applied to all other records. Simultaneous editing puts several demands on the generalization algorithm:

- Generalization should be fast, so that the system is responsive enough for interactive editing. We solve this problem by preprocessing the records to discover useful features in advance, so that the generalization search for each selection is relatively cheap.
- Generalization should be domain-specific. For example, a user's selection might best be described in terms of Java syntax. Our solution to this problem is a knowledge base, represented by a library of patterns and parsers that detect structure in text. Users can extend the library on the fly by specifying new patterns, which can be either regular expressions or high-level patterns called *text constraints* [9].
- Generalization should be able to guess accurately from only one example. When multiple generalizations are consistent with the user's selection, the generalizer must make its best guess, which hopefully will often be the description the user intended.
- Generalization should be correctable. If the generalizer's best guess is wrong, the user must have a way to correct it. In our system, the user can select or deselect regions in other records, providing additional positive and negative examples that the generalizer uses to improve its guess. The user can also override the generalizer completely, making a selection by hand or by a pattern.

The rest of this paper is organized as follows. Section 2 surveys related work. Section 3 describes the user interface to simultaneous editing, in the context of an extended example. Section 4 describes some more examples of simultaneous editing. Section 5 delves into the details of our implementation, and Section 6 evaluates its performance. Section 7 outlines some future directions, and Section 8 makes some conclusions.

2 Related Work

Simultaneous editing is similar in concept to Visual Awk [6], a system for developing awk-like file transformers interactively. Like awk, Visual Awk's default structure

consists only of lines and words. When the user selects one or more words in a line, the system highlights the words at the same position in all other lines. For other kinds of selections, the user must select the appropriate tool. For example, Visual Awk's Cutter tool makes selections by character offset, and its Matcher tool uses regular expressions provided by the user. In contrast, simultaneous editing is built into a conventional text editor, operates on arbitrary records (not just lines), uses standard text editing operations, and automatically infers general, domain-specific descriptions from a user's selections.

Another closely-related approach to the problem of repetitive text editing is *programming by example*, also called *programming by demonstration* (PBD). In PBD, the user demonstrates one or more examples of the transformation in a text editor, and the system generalizes this demonstration into a program that can be applied to the rest of the examples. PBD systems for text editing have included EBE [12], Tourmaline [11], TELS [13], Eager [1], Cima [7], and DEED [3].

Simultaneous editing is similar to PBD in many ways. Both approaches allow the user to edit with familiar operations, including mouse selection. Both approaches generalize the user's actions on one example into a description that can be applied to other examples. Both approaches must be able to incorporate multiple examples into the generalization.

However, simultaneous editing has a dramatically different user interface from PBD. In simultaneous editing, the user's demonstration affects all records simultaneously. After demonstrating part of a transformation, the user can scan through the file and see how the other records were affected by the partial transformation. In PBD, on the other hand, each demonstration affects only a single example. In order to see what the inferred program will do to other examples, the user must run the program on other examples. One consequence of this is a lack of trust [1][3]. Users do not trust the inferred program to work correctly on other examples. Although simultaneous editing also does inference, and thus is also susceptible to mistrust, the additional feedback provided by simultaneous selections across all records makes the system's operation more visible, hopefully inspiring more confidence.

The inference used in simultaneous editing is actually *less* powerful than in some PBD systems. TELS, for example, can infer programs containing conditionals and loops. Simultaneous editing assumes just one implicit loop (over the records) and no conditionals (every editing action must be applied to every record). These assumptions permit fast, predictable inference, and allow inference to be applied only to *selections* and not to the sequence of *actions* performed.

Simultaneous editing also requires the user to describe the set of records. The record description is often simple (e.g. lines, or paragraphs, or functions), but some record sets may be hard to describe. By contrast, in most PBD systems, and keyboard macros too, the record set is implicit in the user's demonstration. For example, the demonstration may end with the cursor at the start of the next record.

3 User Interface

This section describes the user interface of simultaneous editing implemented in our prototype system. Features of the user interface will be introduced by presenting an example of the system in operation.

Our implementation of simultaneous editing is built into LAPIS, a text processing system which has been described previously [9][10]. LAPIS has several unusual features that make it well-suited to this effort. First, LAPIS supports multiple simultaneous text selections; most text editors allow only one contiguous selection. Multiple selections make it easy to display the corresponding selection in every record. Second, LAPIS includes an integrated text pattern language, *text constraints*. Text constraint patterns are convenient not only for the user to describe the record set, but also for the system to describe how it has generalized the user's selection. Finally, LAPIS has a library of built-in parsers and patterns for various kinds of text structure, including HTML and Java source code. The domain knowledge represented by this pattern library enables the system to make its generalizations more accurate and domain-specific, as we will see in the example to follow.

3.1 Example: Get/Set Methods

The example is a common task in Java and C++ programming: for each field `x` of a class (member variable in C++ terminology), create a pair of accessor methods `getX` and `setX` that respectively get and set the value of `x`. Figure 2a shows the original Java class. We want to transform each field declaration so that the variable declaration is followed by its accessor methods, as shown in Figure 2g.

To enter simultaneous editing mode, the user first selects the records to be edited, using multiple selection. A multiple selection can be made two ways in LAPIS: by entering a pattern, which selects all regions that match

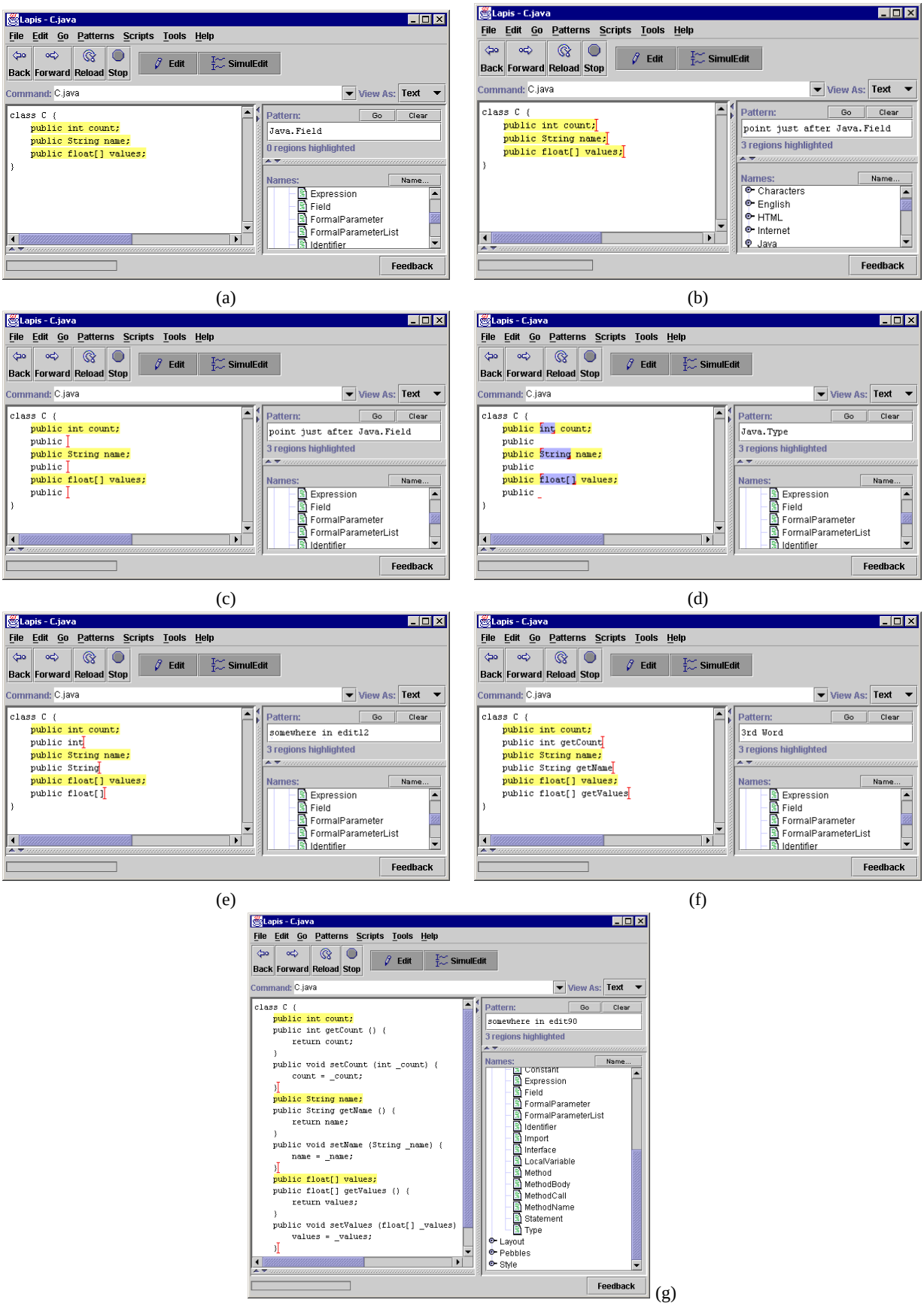


Figure 2: Simultaneous editing used to transform Java field declarations into get/set methods.

the pattern; or by holding down the Shift key and selecting text regions with the mouse. In this case, the user chooses `Java.Field` from the pattern library, which runs a Java parser and highlights all field declarations in the current file. If only some of these fields need accessor methods, then the user can either specialize the pattern (e.g. `Java.Field starting with "public"`) or manually deselect the undesired fields.

Having selected the records, the user enters simultaneous editing mode by pressing the SimulEdit button on the toolbar. The system then does some preprocessing, which involves running all appropriate parsers (such as the Java parser) and searching for interesting features in the selected records. Preprocessing is described in Section 5. The preprocessing delay depends on the number and length of the records. In this simple example, preprocessing takes less than one second. After preprocessing, the editor shows that simultaneous editing is enabled by highlighting the records in yellow.

The user now starts to edit. First, the user clicks at the end of one of the records. The system immediately generalizes this click to the other records, displaying an insertion point at the end of each record (Figure 2b). At the same time, the Pattern box displays a description of the generalization that was made: `point just after Java.Field`. In this case, the description is actually a text constraint pattern, which could be evaluated to select the same insertion points. The description is not always a valid pattern, because of some design decisions made in our prototype, discussed later. Regardless, the description provides an additional cue for the user to check that the system is properly generalizing the selection.

Having placed the insertion point, the user starts to type in the `getX` method, first pressing Enter to insert a newline, then indenting a few spaces, then typing "public " to start the method declaration. The typed characters appear in every record (Figure 2c). If typos are made, the user can back up and correct them, using all familiar editing operations. Maintaining the simultaneous insertion points during text entry is trivial, since all records receive the same typed text. No generalization occurs until the user makes a selection somewhere else.

Now the user is ready to enter the return type of the `getX` method. The type is different for each variable `x`, so the user can't simply enter it at the keyboard. Instead, copy-and-paste is used. The user selects the type of one of the fields, in this case, the "int" of "public int x". The system generalizes this selection into the description `Java.Type`, and selects the types of all the other fields (Figure 2d). Note that other generalizations of this selection are possible: `"int"`, `2nd Word`, `2nd from last Word`, etc. Some of these generalizations can be discarded immediately because of assumptions of simultaneous editing. For example, "int" does not appear in every record, and so it cannot be selected in every record. Other generalizations are less preferable because they are more complicated than `Java.Type`. In this case, the system's best guess is the right one.

The user then copies the selection to the clipboard, places the insertion point back after "public ", and pastes the clipboard. In response to the copy command, the system copies a *list* of strings to its clipboard, one for each record. When the paste occurs, the system pastes the appropriate string back to each record (Figure 2e).

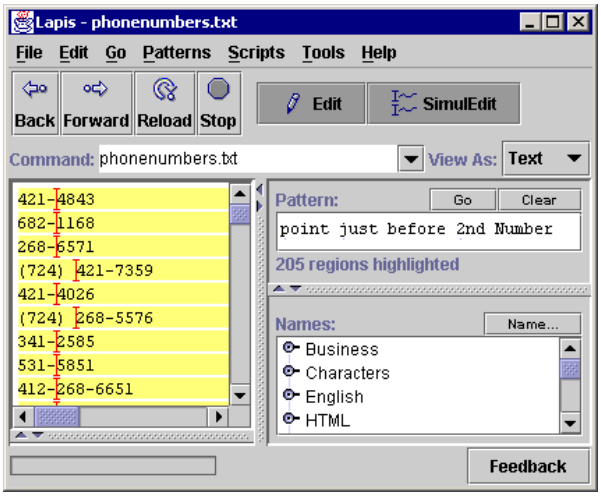
Similarly, the user copies and pastes the name of variable to create the method name. The lowercase variable name `x` is converted into capitalized `X` by applying an editor command that capitalizes the current selection (Figure 2f). Any editor command that applies to a selection or cursor position can be used in simultaneous editing mode.

The rest of `getX` and `setX` are defined by more typing and copy-and-paste commands, until the desired result is achieved (Figure 2g). The user exits simultaneous editing mode by clicking again on the SimulEdit toolbar button, releasing it from the depressed state.

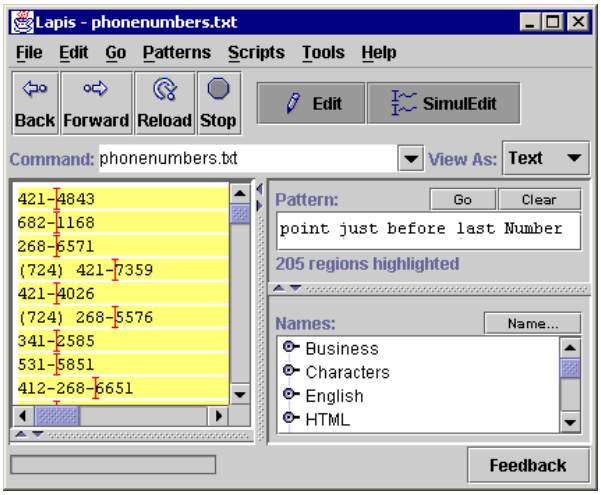
3.2 Correcting Generalizations

The example above raises an important issue: what if the system's generalization was incorrect at some point in the simultaneous editing session? How can the user correct it? Several techniques are available in our system: switching to a counterexample, giving multiple examples, and naming landmarks. These techniques are explained next.

The first correction technique is illustrated in Figure 3. While editing a list of phone numbers, the user tries to place the cursor just before the 4-digit component of each phone number. The first attempt (Figure 3a) is a click before "4843" in the first phone number. This click is incorrectly generalized to `point just before 2nd Number`. An easy way to correct the generalization is to pick one of the records where the generalization failed – for example, a phone number with an area code such as "(724) 421-7359" – and make the selection in that record instead. This selection results in a satisfactory generalization (Figure 3b). This strategy, which we call *switching to a counterexample*, corrects the system by providing a more generic example of the desired selection. The system is still generalizing from only one example; the more generic example replaces the earlier example. An expert user may even

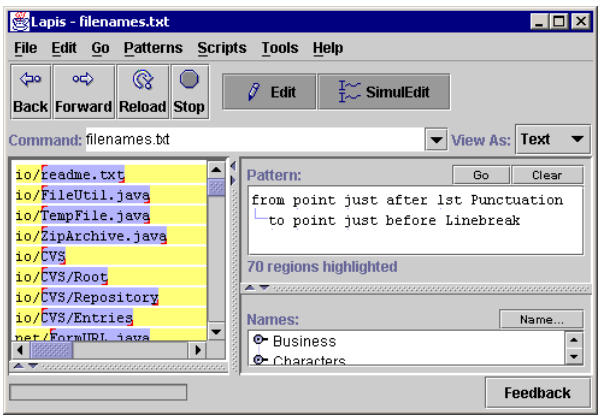


(a)

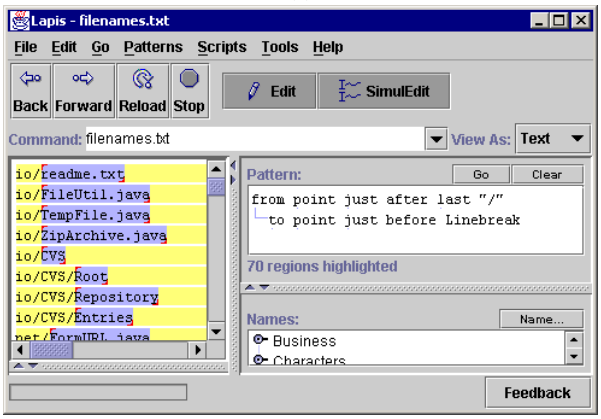


(b)

Figure 3: Correcting generalization by switching to a counterexample.



(a)



(b)

Figure 4: Correcting generalization by providing multiple examples.

avoid the incorrect generalization entirely by selecting the most generic example first.

Sometimes an incorrect generalization cannot be fixed by switching to a counterexample. For example, in Figure 4, the user is trying to select just the filenames, without any directory prefix. Selecting “readme.txt” in the first record generalizes to an incorrect description referring to the `point just after 1st Punctuation` (Figure 4a). Switching to a counterexample doesn’t work either. For example, selecting “Root” in the sixth record would generalize to `last Word`, which is also wrong, because it would select only “txt” in the first record instead of “readme.txt”. To get the desired selection, the user must provide at least two examples of the selection. This is done by holding down the Shift key while selecting the additional example. Alternatively, the user can specify a negative example by deselecting an incorrect selection in another record. Deselecting is done by right-clicking on a selection and picking Unselect from the popup menu that appears. Any number of positive or negative examples can be given.

After receiving a new positive or negative example, the system searches for a generalization that selects exactly one region in every record and is consistent with all positive and negative examples. In this case, two positive examples suffice to select the last filename component correctly (Figure 4b).

The user can also assist generalization by making a selection some other way, either by hand or by a pattern, and then assigning it a unique name. The named selection becomes part of the pattern library, where the system can use it as a *landmark* for generalizing other selections. For example, the user might specify a regular expression for the product codes used in his company, and name it ProductCode. Subsequent selections of product codes, or of regions adjacent to product codes, will be generalized much faster and more accurately. This strategy adds more domain knowledge to the system.

Generalization may sometimes fail. There may be insufficient domain knowledge, or the selection may require a more complicated description than the generalizer is designed to generate. For example, our generalizer does not form disjunctions, such as `either "gif" or "jpg".` If no generalization can be found that is consistent with the user's positive and negative examples, then the system gives up, beeps, and leaves only the positive examples selected. No further generalization attempts are made until the user clears the selection and starts a new selection. The user can finish the desired selection by hand, either by selecting the appropriate regions in the other records, or by entering a pattern.

4 Applications

This section describes some applications of simultaneous editing. Two common themes run through these examples. First is the power of *domain knowledge*, such as HTML and Java syntax. Domain knowledge allows the user to specify patterns more concisely and enables the generalizer to make more accurate generalizations with fewer examples. Most text editors either eschew domain knowledge, understanding only low-level concepts like words and characters, or else embed knowledge for only one domain, such as C++. LAPIS strives for a middle ground by centralizing domain knowledge into a pattern library that simply generates region sets. Other parts of LAPIS, such as the generalizer, are domain-independent, and new domain knowledge is easy to add by installing new patterns and parsers in the library.

The second important theme is *interactivity*. Whereas other solutions to these tasks would involve specifying a program and then running it in batch mode, simultaneous editing allows the task to be performed interactively.

4.1 HTML

The following tasks take advantage of the HTML parser included in the pattern library. The HTML parser defines named region sets for each kind of HTML object (e.g. Element, Tag, Attribute) as well as specific HTML tags and attributes (e.g. , href).

Change all elements into elements. The user runs the pattern `Bold` to select all bold elements (which look like bold text), then enters simultaneous editing mode. The user selects the first b with the mouse (which the system generalizes to `"b" in "<b>"`), deletes it, and types in "strong". The user then selects the last b (which generalizes to `"b" in "</b>"`), deletes it, and types in "strong" again.

Convert HTML to XHTML. One difference between HTML and XHTML (an XML format) is the treatment of tags with no content, such as <img>,
, or <hr>. In XHTML, elements with no end tag should be written as <img/> so that an XML parser can parse them without access to the XHTML document type definition. Making this conversion with simultaneous editing is straightforward. To select the empty tags, the user runs the pattern `Tag = Element`, which matches all regions that the HTML parser identified as both tags and complete elements. Entering simultaneous editing mode, the user places the cursor at the end of the tag (which generalizes to `point just before ">"`) and inserts a slash to finish the task.

4.2 Source Code

Programming is full of tasks where simultaneous editing is useful, particularly when given knowledge of the language syntax. The examples below are in Java because LAPIS has a Java parser in its library. Other languages could be edited in similar fashion by adding an appropriate parser to the library.

Change access permissions. Suppose a class contains a number of fields or methods that currently have default access permission, and the programmer wants to change their permissions to `private`. The programmer selects the relevant fields and methods (using, for instance, `(Field or Method) not starting with "public" or "private"`), enters simultaneous editing mode, and types `private` at the beginning of a field, changing all the others simultaneously.

Change a method interface. If a method's parameters change, then simultaneous editing can be used to

rewrite all the calls to that method at once. For example, suppose a method `copy(src, dest)` must be changed to `copy(dest, src)` for consistency with similar interfaces in the program. The programmer selects all calls to `copy` (perhaps using the pattern `MethodCall starting with Identifier equal to "copy"`), enters simultaneous editing mode, selects the first argument (which generalizes to `first ActualParameter`), copies it to the clipboard, and then pastes it after the second argument. A little more editing fixes the comma separators, and the change is done. This example demonstrates how simultaneous editing with domain knowledge can deliver the power of syntax-directed editing inside a freeform text editor.

Wrap every method with entry and exit code. While debugging a class, the programmer wants to run some code whenever any method of the class is entered or exited. This code might do tracing (printing the method name to a log), performance timing, or validation (checking that the method preserves class invariants). To add this code, the programmer selects all the methods using the pattern `Method` and starts simultaneous editing. Next, the programmer types in the entry code at the start of the method, wraps the rest of the method body with a `try-finally` construct, and types the exit code inside the `finally` clause. All the methods change identically. This kind of modification is an example of *aspect-oriented programming* [5], where code is "woven" into a program at program locations described by a pattern.

4.3 Scripting

To understand the next set of examples, the reader should be aware that LAPIS is also a shell [10]. An external command can be executed in the LAPIS command box, drawing its standard input from the current contents of the editor and sending its output back to the editor.

Disposable scripts. Suppose the user wants to make a group of GIF files transparent using `giftrans`. Simultaneous editing offers a solution based on the idea of creating a one-time script directly from data. The user first runs `ls *.gif` to list the relevant filenames in the editor. Using simultaneous editing, the user edits each line into a command, such as `giftrans -T X.gif > X-transparent.gif`. Then the user runs the resulting script with `sh`. Disposable scripts are a more interactive way to achieve the effect of the Unix commands `foreach` or `xargs`.

Impedance matching. Data obtained from the output of one program must often be massaged before it can be fed into another program. Simultaneous editing offers the opportunity to perform this massaging interactively, which is particularly sensible for one-shot tasks. For example, suppose a user is testing network connectivity with `traceroute`, and wants to pass the network latencies computed by `traceroute` into `gnuplot` to generate a graph. The user first runs `traceroute` to generate a trace. Using simultaneous editing, the user edits each line of the trace, leaving only the hop number (1, 2, ...) and the latency time. After exiting simultaneous editing mode, the user inserts a `gnuplot` plot instruction before the first line (`plot "-" with lines`) and finally runs `gnuplot -persist` to plot the data.

5 Implementation

This section describes the algorithm used to generalize the user's selection to a description that can be applied to all records. The input to the generalizer is a set of positive examples, a set of negative examples, and the set of records. The output is a selection consistent with the positive and negative examples that selects exactly one region in every record, plus a human-readable description of the selection.

Like other PBD systems, the generalizer basically searches through a space of hypotheses for a hypothesis consistent with the examples. The details of the implementation are novel, however. Our generalizer is actually split into three parts: preprocessing, search, and updating. Preprocessing takes the set of records as input and generates a list of useful features as output. Preprocessing takes place only once, when the user first enters simultaneous editing mode. The search phase takes the positive and negative examples and the feature list generated by preprocessing, and computes a selection consistent with the examples. Search happens whenever the user makes a selection with the mouse or keyboard, or adds a new positive or negative example to the current selection. Finally, updating occurs when the user edits the records by inserting, deleting, or copying and pasting text. Updating takes the user's edit action as input and modifies the feature list appropriately. Each of these phases is described in more detail below.

5.1 Region Sets

Before describing the generalizer, we first briefly describe the representations used for selections in a text file. More detail can be found in an earlier paper about LAPIS [9]. A *region* $[s, e]$ is a substring of a text file, described by its start offset s and end offset e relative to the start of the text file. A *region set* is a set of regions.

LAPIS has two novel representations for region sets. First, a *fuzzy region* is a four-tuple $[s_1, s_2; e_1, e_2]$ that represents the set of all regions $[s, e]$ such that $s_1 \leq s \leq s_2$ and $e_1 \leq e \leq e_2$. Note that any region $[s, e]$ can be represented as the fuzzy region $[s, s; e, e]$. Fuzzy regions are particularly useful for representing relations between regions. For example, the set of all regions that are inside $[s, e]$ can be compactly represented by the fuzzy region $[s, e; s, e]$. Similar fuzzy region representations exist for other relations, including *contains, before, after, just before, just after, starting* (i.e. having coincident start points), and *ending*. These relations are fundamental operators in the text constraints pattern language, and are also used in generalization.

The second novel representation is the *region tree*, a union of fuzzy regions stored in an R-tree in lexicographic order [9]. A region tree can represent an arbitrary set of regions, even if the regions nest or overlap each other. A region tree containing N fuzzy regions takes $O(N)$ space, $O(N \log N)$ time to build, and $O(\log N)$ time to test a region for membership in the set.

These two representations are used by the preprocessing phase to construct a list of features that the search phase can use to quickly test positive and negative examples. The selection returned by the search phase is also represented as a region set.

5.2 Feature Generation

Preprocessing takes the set of records and generates a list of useful features. A feature is a region set, containing at least one region in each record, where the regions are in some sense equivalent. For example, the feature `Java.Type` represents the set of all regions that were recognized by the Java parser as type names. The preprocessor generates two kinds of features: *pattern features* derived from the pattern library, and *literal features* discovered by examining the text of the records.

Pattern features are found by applying every parser and every named pattern in the pattern library. LAPIS has a considerable library of built-in parsers and patterns, including Java, HTML, character classes (e.g. digits, punctuation, letters), English structure (words, sentences, paragraphs), and various codes (e.g., URLs, email addresses, hostnames, IP addresses, phone numbers). The user can readily add new named patterns and new parsers. The result of applying a pattern is the set of all regions matching the pattern. The result of applying a parser is a collection of named region sets. For example, the Java parser generates region sets for Statement, Expression, Type, Method, and so on.

After applying all library patterns, the preprocessor discards any patterns that do not have at least one match in every record. This is justified by two assumptions made by the generalizer: first, that a generalization must have at least one match in every record; and second, that a generalization can be represented without disjunction. Given these two assumptions, only features that match somewhere in every record will be useful for constructing generalizations.

By the same reasoning, the only useful literal features are common substrings, i.e. substrings that occur at least once in every record. Common substrings can be found efficiently using a *suffix tree* [4]. A suffix tree is a path-compressed trie into which all suffixes of a string have been inserted. With a suffix tree for a string s, we can test whether a substring p occurs in s in only $O(|p|)$ time. Naive suffix tree construction (inserting every suffix explicitly) takes $O(|s|^2)$ time, which is sufficient for our prototype since records tend to be short. Several algorithms exist for building a suffix tree in linear time, however [4], and extending the algorithm below to accommodate them would be straightforward.

The common substring algorithm works as follows. The algorithm starts by building a suffix tree from the shortest record, in order to minimize the size of the initial suffix tree. This suffix tree represents the set of common substrings of all records examined so far. For each of the remaining records, the suffixes of the record are matched against the suffix tree one by one. Each tree node keeps a count of the number of times it was visited during the processing of the current record. This count represents the number of occurrences (possibly overlapping) of the substring represented by the node. After processing each record, all unvisited nodes are pruned from the tree, since the corresponding substrings never occurred in the record. After processing every record in this fashion, the only substrings left in the suffix tree must be common to all the records. These common substrings are used as literal features. The operation of the common substring algorithm is illustrated in Figure 5.

5.3 Feature Ordering

After generating useful features from the set of records, the preprocessor sorts them into a list in order of preference. Placing the most-preferred features first makes the search phase simpler. The search can just scan down the list of features and stop as soon as it finds the first feature consistent with the examples, since this feature is guaranteed to be the most preferred consistent feature.

Features are classified into three groups for the purpose of preference ordering: *unique features*, which occur exactly once in each record; *regular features*, which occur exactly n times in each record, for some $n > 1$; and *varying features*, which occur a varying number of times

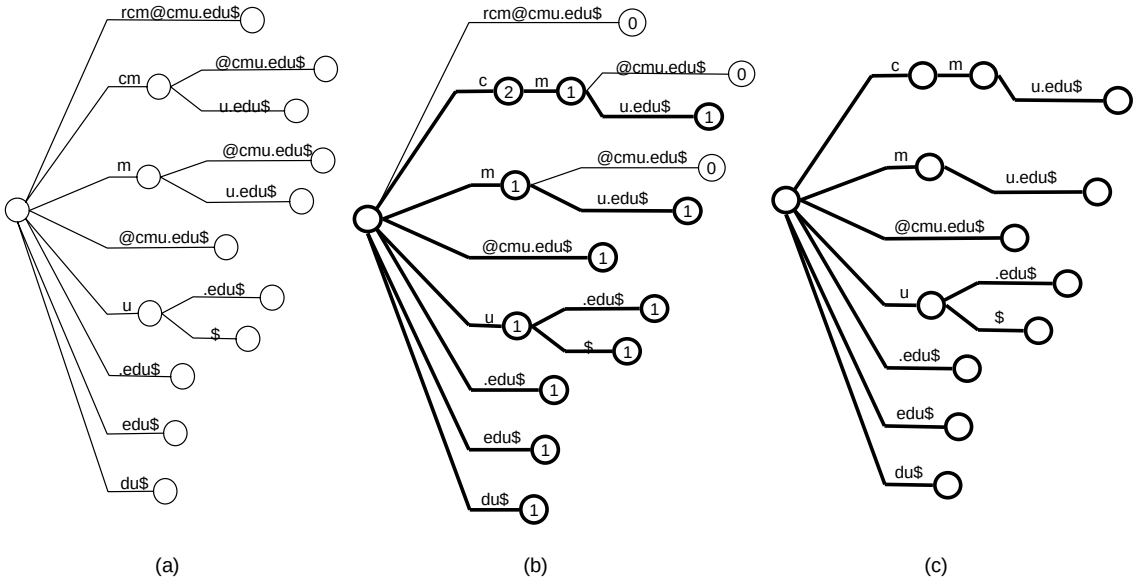


Figure 5: Finding common substrings using a suffix tree. (a) Suffix tree constructed from first record, `rcm@cmu.edu`; `$` represents a unique end-of-string character. (b) Suffix tree after matching against second record, `ljc@cmu.edu`. Each node is labeled by its visit count. (c) Suffix tree after pruning nodes which are not found in `ljc@cmu.edu`. The remaining nodes represent the common substrings of the two records.

in each record. A feature's classification is not predetermined. Instead, it is found by actually counting occurrences in the records being edited. For example, in Figure 2, `Java.Type` is a unique feature, since it occurs exactly once in every variable declaration. Regular features are commonly found as delimiters. For example, if the records are IP addresses like 127.0.0.1, then "." is a regular feature. Varying features are typically tokens, like words and numbers, which are general enough to occur in every record but do not necessarily follow any regular pattern.

Unique features are preferred over the other two kinds. A unique feature has the simplest description: the feature name itself, such as `Java.Type`. By contrast, using a regular or varying feature in a generalization requires specifying the index of a particular occurrence, such as `5th Word`. Regular features are preferred over varying features, because regularity of occurrence is a strong indication that the feature is relevant to the internal structure of a record.

Within each group, pattern features are preferred over literal features. We also plan to let the user specify preferences between pattern features, so that, for instance, Java features can be preferred over character-class features. We are still designing the user interface for this, however, so the prototype currently leaves pattern features in an arbitrary order. Among literal features, longer literals are preferred to shorter ones.

To summarize, the preprocessor orders the feature list in the following order, with most preferred features listed first: unique patterns, unique literals, regular patterns, regular literals, varying patterns, varying literals. Within each group of patterns, the order is arbitrary. Within each group of literals, longer literals are preferred to shorter.

5.4 Search

The search algorithm takes the user's positive and negative examples and the feature list generated by preprocessing, and attempts to generate a description consistent with the examples.

The basic search process works as follows. The system chooses the first positive example, called the *seed example*, and scans through the feature list, testing the seed example for membership in each feature. Since each feature is represented by a region tree, this membership test is very fast. When a matching feature is found, the system constructs one or more candidate descriptions representing the particular occurrence that matched. For example, if the seed example matches the (varying) feature `Word`, the system might construct the candidate descriptions `5th Word` and `2nd from last Word` by counting words in the seed example's record. These

candidate descriptions are tested against the other positive and negative examples, if any. The first consistent description found is returned as the generalization.

The output of the search process depends on whether the user is selecting an insertion point (e.g. by clicking the mouse) or selecting a region (e.g. by dragging). If all the positive examples are zero-length regions, then the system assumes that the user is placing an insertion point, and searches for a point description. Otherwise, the system searches for a region description.

To search for a point description, the system transforms the seed example, which is just a character offset b, into two fuzzy regions: $[b, b; b, +\infty]$, which represents all regions that start at b, and $[-\infty, b; b, b]$, which represents all regions that end at b. The search tests these fuzzy regions for intersection with each feature in the feature list, which is just as fast as a simple region membership test. Candidate descriptions generated by the search are transformed into point descriptions by prefixing `point just before` or `point just after`, depending on which fuzzy region matched the feature, and then the descriptions are tested for consistency with the other positive and negative examples.

To search for a region description, the system first searches for the seed example using the basic search process described above. If no matching feature is found – because the seed example does not correspond precisely to a feature on the feature list – then the system splits the seed example into its start point and end point, and recursively searches for point descriptions for each point. Candidate descriptions for the start point and end point are transformed into a description of the entire region by wrapping with `from...to...`, and then tested for consistency with the other examples.

This search algorithm is capable of generalizing a selection only if it starts and ends on a feature boundary. For literal features, this is not constraining at all. Since a literal feature is a string that occurs in all records, every substring of a literal feature is *also* a literal feature. Thus every position in a literal feature lies on a feature boundary. To save space, the preprocessor only stores maximal literal features in the feature list, and the search phase tests whether the seed example falls anywhere inside a maximal literal feature.

5.5 Updating

In simultaneous editing, the user is not only making selections, but also editing the file. Editing has two effects on generalization. First, every edit changes the start and end offsets of regions. As a result, the region sets used to represent features become invalid. Second, editing changes the file content, so the precomputed features may become incomplete or wrong. For example, if the user types some new words, then the precomputed `Word` feature becomes incomplete, since it doesn't include the new words the user typed. The updating algorithm addresses these two problems.

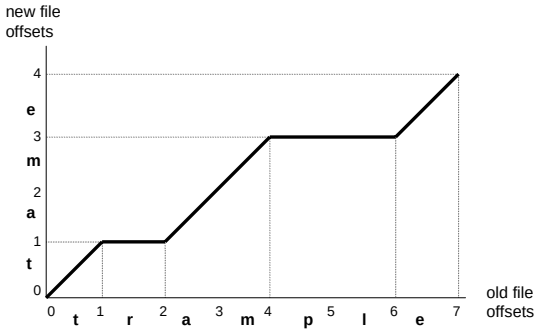


Figure 6: Coordinate map translating offsets between two versions of a file. The old version is the word `trample`. Two regions are deleted to get the new version, `tame`.

From the locations and length of text inserted or deleted, the updating algorithm computes a *coordinate map*, a relation that translates a file offset prior to the change into the equivalent file offset after the change. The coordinate map can translate coordinates in either direction. For example, Figure 6 shows the coordinate map for a simple edit. Offset 3 in `trample` corresponds to offset 2 in `tame`, and vice versa. Offsets with more than one possible mapping in the other version, such as offset 1 in `tame`, are resolved arbitrarily. Our prototype picks the largest value.

Since the coordinate map for a group of insertions or deletions is always piecewise linear, it can be represented as a sorted list of the (x,y) endpoints of each line segment. If a single edit consists of m insertions or deletions (one for each record), then this representation takes $O(m)$ space. Evaluating the coordinate map function for a single offset takes $O(\log m)$ time using binary search.

A straightforward way to use the coordinate map is to scan down the feature list and update the start and end points of every feature to reflect the change. If the feature list is long, however, and some feature sets are large (such as Word or Whitespace), the cost of updating every feature after every edit may be prohibitive. Our generalizer takes the opposite strategy: instead of translating all features up to the present, we translate the user's positive and negative examples *back to the past*. The system maintains a global coordinate map representing the translation between original file coordinates (when simultaneous editing mode was entered and the feature

list generated) and the current file coordinates. When an edit occurs, the updating algorithm computes a coordinate map for the edit and composes it with this global coordinate map. When the user provides positive and negative examples to generalize into a selection, the examples are translated back to the original file coordinates using the inverse of the global coordinate map. The search algorithm generates a consistent description for the translated examples. The generated description is then translated forward to current file coordinates before being displayed as a selection.

An important design decision in a simultaneous editing system that uses domain knowledge, such as Java syntax, is whether the system should attempt to reparse the file while the user is editing it. On one hand, reparsing allows the generalizer to track all the user's changes and reflect those changes in its descriptions. On the other hand, reparsing is expensive and may fail if the file is in an intermediate, syntactically incorrect state. Our generalizer never reparses automatically in simultaneous editing mode. The user can explicitly request reparsing with a menu command, which effectively restarts simultaneous editing using the same set of records. Otherwise, the feature list remains frozen in the original version of the file. One consequence of this decision is that the generalizer's human-readable descriptions may be misleading because they refer to an earlier version.

This design decision raises an important question. If the feature list is frozen, how can the user make selections in newly-inserted text, which didn't exist when the feature list was built? This problem is handled by the update algorithm. Every typed insert in simultaneous editing mode adds a new literal feature to the feature list, since the typed characters are guaranteed to be identical in all the records. Similarly, pasting text from the clipboard creates a special feature that translates coordinates back to the source of the paste and tries to find a description there. When the generalizer uses one of these features created by editing, the feature is described as "somewhere in edit$N$", which can be seen in Figures 2e and 2g.

A disadvantage of this scheme is that the housekeeping structures – the global coordinate map and the new features added for edits – grow steadily as the user edits. This growth can be slowed significantly by coalescing adjacent insertions and deletions, although we have not yet implemented this. Another solution might be to reparse when the number of edits reaches some threshold, doing the reparsing in the background on a copy of the file in order to avoid interfering with the user's editing. In practice, however, we don't expect space growth to be a serious problem. In all the applications we have imagined, the user spends only a few minutes in a simultaneous editing session, not the hours that are typical of general text editing. After leaving simultaneous editing mode, the global coordinate map and the feature list can be discarded.

6 Evaluation

Simultaneous editing was evaluated by a small user study. Eight users were found by soliciting campus newsgroups. All were college undergraduates with substantial text-editing experience and varying levels of programming experience (5 described their programming experience as "little" or "none," and 3 as "some" or "lots"). Users were paid for participation. Users first learned about simultaneous editing by reading a tutorial and trying the examples. The tutorial took less than 10 minutes for all but one user (who spent 30 minutes exploring the system). After the tutorial, each user performed the following three tasks:

1. Put the author name and publication year in front of each citation.
   Before:
   1. Aha, D.W. and Kibler, D. Noise-tolerant instance-based learning algorithms. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence. Morgan Kaufmann, 1989, pp. 794-799
   2. Hayes-Roth, B. Pfleger, K. Morignot, P. and Lalanda, P. Plans and Behavior in Intelligent Agents, Technical Report KSL-95-35, Stanford University, 1995.
   ... (7 more) ...
   After:
   [Aha 89] Aha, D.W. and Kibler, D. Noise-tolerant instance-based learning algorithms. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence. Morgan Kaufmann, 1989, pp. 794-799.
   [Hayes-Roth 95] Hayes-Roth, B. Pfleger, K. Morignot, P. and Lalanda, P. Plans and Behavior in Intelligent Agents, Technical Report KSL-95-35, Stanford University, 1995.
   ... (7 more) ...
2. Reformat a list of mail aliases from HTML to text.
   Before:
   <DT><A HREF="mailto:cg@cs.umn.edu" NICKNAME="congra"> Conceptual Graphs</A>
   <DT><A HREF="mailto:kif@cs.stanford.edu" NICKNAME="kif"> KIF</A>
   ... (5 more) ...
   After:
   ;; Conceptual Graphs
   congra: mailto:cg@cs.umn.edu
   ;; KIF
   kif: mailto:kif@cs.stanford.edu
   ... (5 more) ...
3. Reformat a list of baseball scores into a tagged format (7 records).
   Before:
   Cardinals 5, Pirates 2.
   Red Sox 12, Orioles 4.
   ... (5 more) ...
   After:
   GameScore[winner 'Cardinals'; loser 'Pirates'; scores[5, 2]].
   GameScore[winner 'Red Sox'; loser 'Orioles'; scores[12,4]].
   ... (5 more) ...

All tasks were obtained from other authors (tasks 1 and 2 from Fujishima [3] and task 3 from Nix[12]). After performing a task with simultaneous editing, users

repeated the task with manual editing, but only on the first three records to avoid unnecessary tedium. Users were instructed to work carefully and accurately at their own pace. All users were satisfied that they had completed all tasks, although the finished product sometimes contained undetected errors, a problem discussed further below. No performance differences were seen between programmers and nonprogrammers. Aggregate times for each task are shown in Table 1.

Following the analysis used by Fujishima [3], we estimate the leverage obtained with simultaneous editing by dividing the time to edit all records with simultaneous editing by the time to edit just one record manually. This ratio, which we call *equivalent task size*, represents the number of records for which simultaneous editing time would be equal to manual editing time for a given user. Since manual editing time increases linearly with record number and simultaneous editing time is roughly constant (or only slowly increasing), simultaneous editing will be faster whenever the number of records is greater than the equivalent task size. (Note that the average equivalent task size is not necessarily equal to the ratio of the average editing times, since $E[S/M] \neq E[S]/E[M]$.)

As Table 1 shows, the average equivalent task sizes are small. In other words, the average novice user works faster with simultaneous editing if there are more than 8.4 records in the first task, more than 3.6 records in the second task, or more than 4 records in the third task.[1] Thus simultaneous editing is an improvement over manual editing even for very small repetitive editing tasks, and even for users with as little as 10 minutes of experience. Some users were so slow at manual editing that their equivalent task size is smaller than the expert's, so simultaneous editing benefits them even more. Simultaneous editing also compares favorably to another PBD system, DEED [3]. When DEED was evaluated with novice users on tasks 1 and 2, the reported equivalent task sizes averaged 42 and ranged from 5 to 200, which is worse on average and considerably more variable than simultaneous editing.

Another important part of system performance is generalization accuracy. Each incorrect generalization forces the user to make at least one additional action, such as selecting a counterexample or providing an additional positive or negative example. In the user study, users made a total of 188 selections that were used for editing. Of these, 158 selections (84%) were correct immediately, requiring no further examples. The remaining selections needed either 1 or 2 extra examples to generalize correctly. On average, only 0.26 additional examples were needed per selection. Unfortunately, users tended to overlook slightly-incorrect generalizations, particularly generalizations that selected only half of the hyphenated author "Hayes-Roth" or the two-word baseball team "Red Sox". As a result, the overall error rate for simultaneous editing was slightly worse than for manual editing: 8 of the 24 simultaneous editing sessions ended with at least one uncorrected error, whereas 5 of 24 manual editing sessions ended with uncorrected errors. If the two most common selection errors had been noticed by users, the error rate for simultaneous editing would have dropped to only 2 of 24. We are currently studying ways to call the user's attention to possible selection errors [8].

After doing the tasks, users were asked to evaluate the system's ease-of-use, trustworthiness, and usefulness on a 5-point Likert scale. These questions were also borrowed from Fujishima [3]. The results, shown in Figure 7, are generally positive.

7 Status and Future Work

Simultaneous editing has been implemented in LAPIS, a browser/editor designed for processing structured text. LAPIS is written in Java 1.1, extending the JFC text editor component JEditorPane. Directions for obtaining LAPIS are found at the end of this paper.

We have many ideas for future work. First and perhaps most challenging is the problem of scaling up to large tasks. Although our prototype is far from a toy, since it can handle 100KB files with relative ease, many interesting tasks involve megabytes of data spread across multiple files. Large data sets pose several problems for simultaneous editing. The first problem is system responsiveness. Making a million edits with every keystroke may slow the system down to a crawl, particularly if the text editor uses a *gap buffer* to store the text [2]. Gap buffers are used by many editors, among them Emacs and JEditorPane, the Java class on which our prototype is based. With a gap buffer and a record set that spans the entire file, typing a single character forces the editor to move nearly every byte in the buffer. One way to address this problem is to delay edits to the rest of the file until the user scrolls. Another solution would be to have multiple gaps in the buffer, one for each record.

Another problem with large files is checking for incorrect generalizations. When editing a small file, the user can just scan through the entire file to ensure that a selection has been generalized properly. With a large file,

[1] These estimates are actually conservative. Simultaneous editing always preceded manual editing for each task, so the measured time for simultaneous editing includes time spent thinking about and understanding the task. For the manual editing part, users had already learned the task, and were able to edit at full speed.

Task	Records in task	Simultaneous editing	Manual editing	Equivalent task size novices	expert
1	9	142.9 s [63-236 s]	21.6 s/rec[7.7-65 s/rec]	8.4 recs [2.1-12.2 recs]	4.5 recs
2	7	119.1 s [64-209 s]	32.3 s/rec [19-40 s/rec]	3.6 recs [1.9-5.8 recs]	1.6 recs
3	7	159.6 s [84-370 s]	41.3 s/rec [16-77 s/rec]	4.0 recs [1.9-6.2 recs]	2.4 recs

Table 1: Time taken by users to perform each task (mean [min-max]). *Simultaneous editing* is the time to do the entire task with simultaneous editing. *Manual editing* is the time to edit 3 records of the task by hand, divided by 3 to get a per-record estimate. *Equivalent task size* is the ratio between simultaneous editing time and manual editing time for each user; *novices* are users in the user study, and *expert* is one of the authors, provided for comparison. A task with more records than *equivalent task size* would be faster with simultaneous editing than manual editing.

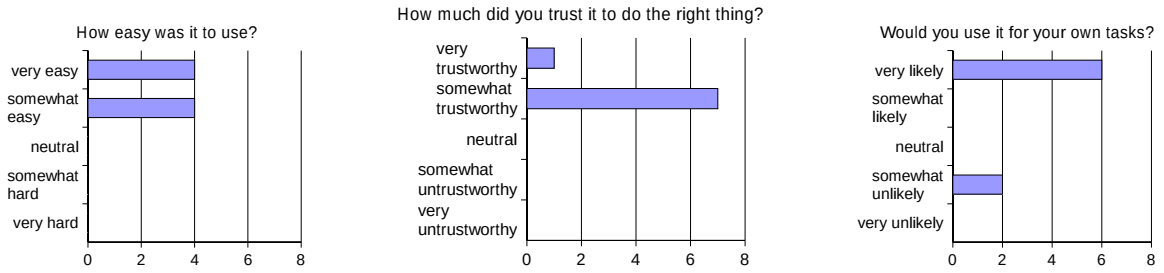


Figure 7: User responses to questions about simultaneous editing.

scanning becomes infeasible. We have several ideas for secondary visualizations that might help with this problem. One is a "bird's-eye view" showing the entire file (in greeked text), so that deviations in an otherwise regular highlight can be noticed at a glance. Another is an abbreviated context view, showing only the selected lines from each record. A third view is an "unusual matches" view, showing only the most unusual examples of the generalization, found by clustering the matches [8].

A third problem with large data sets is where the data resides. For interactive simultaneous editing, the data must fit in RAM, with some additional overhead for parsing and storing feature lists. For large data sets, this is impractical. However, it is easy to imagine interactively editing a small sample of the data to record a macro which is applied in batch mode to the rest of the data. The batch mode could minimize its memory requirements by reading and processing one record at a time (or one translation unit at a time, if it depends on a Java or HTML parser). Macros recorded from simultaneous editing would most likely be more reliable than keyboard macros recorded from single-cursor editing, since simultaneous editing finds general patterns representing each selection. The larger and more representative the sample used to demonstrate the macro, the more correct the patterns would be. The macro could also be saved for later reuse.

8 Conclusions

Simultaneous editing is an effective way for users to perform repetitive text editing tasks interactively, using familiar editing commands. Its combination of interactivity and domain specificity makes simultaneous editing a useful addition to our basket of tools for text processing, which is practical for inclusion in a wide variety of editors.

The LAPIS browser/editor, which includes an implementation of simultaneous editing with Java source code, may be downloaded from

`http://www.cs.cmu.edu/~rcm/lapis/`

Acknowledgements

The authors are indebted to Yuzo Fujishima for providing the materials to reproduce the DEED user study. We would also like to thank Laura Cassenti, Sarit Sotangkur, Dorothy Zaborowski, Brice Cassenti, and Jean Cassenti for enduring early versions of simultaneous editing, and Sheila Harnett and the anonymous referees for their helpful comments. This research was funded in part by USENIX Student Research Grants.

References

[1] A. Cypher. Eager: Programming repetitive tasks by demonstration. In A. Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 205–218. MIT Press, 1993.

[2] C. A. Finseth. Theory and practice of text editors, or, a cookbook for an EMACS. Technical Memo 165, MIT Lab for Computer Science, May 1980.

[3] Y. Fujishima. Demonstrational automation of text editing tasks involving multiple focus points and conversions. In *Proceedings of the International Conference on Intelligent User Interfaces (IUI '98)*, pages 101–108, 1998.

[4] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[6] J. Landauer and M. Hirakawa. Visual AWK: a model for text processing by demonstration. In *Proceedings of the 11th International IEEE Symposium on Visual Languages '95*, pages 267–274, 1995.

[7] D. Maulsby. *Instructible Agents*. PhD thesis, Univ. of Calgary, 1994.

[8] R. C. Miller. *Lightweight Structured Text Processing*. PhD thesis, Carnegie Mellon University, 2001. In preparation.

[9] R. C. Miller and B. A. Myers. Lightweight structured text processing. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 131–144, June 1999.

[10] R. C. Miller and B. A. Myers. Integrating a command shell into a web browser. In *USENIX 2000 Annual Technical Conference*, pages 171–182, June 2000.

[11] B. A. Myers. Tourmaline: Text formatting by demonstration. In A. Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 309–322. MIT Press, 1993.

[12] R. Nix. Editing by example. *ACM Transactions on Programming Languages and Systems*, 7(4):600–621, October 1985.

[13] I. H. Witten and D. Mo. TELS: Learning text editing tasks from examples. In A. Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 183–204. MIT Press, 1993.