Micro-Versioning Tool to Support Experimentation in Exploratory Programming

Hiroaki Mikami
The University of Tokyo
Tokyo, Japan
mhiroaki@is.s.u-tokyo.ac.jp

Daisuke Sakamoto
The University of Tokyo
Tokyo, Japan
d.sakamoto@acm.org

Takeo Igarashi
The University of Tokyo
Tokyo, Japan
takeo@acm.org

ABSTRACT

Experimentation plays an essential role in exploratory programming, and programmers apply version control operations when switching the part of the source code back to the past state during experimentation. However, these operations, which we refer to as *micro-versioning*, are not well supported in current programming environments. We first examined previous studies to clarify the requirements for a micro-versioning tool. We then developed a micro-versioning tool that displays visual cues representing possible micro-versioning operations in a textual code editor. Our tool includes a history model that generates meaningful candidates by combining a regional undo model and tree-structured undo model. The history model uses code executions as a delimiter to segment text edit operations into meaning groups. A user study involving programmers indicated that our tool satisfies the above-mentioned requirements and that it is useful for exploratory programming.

ACM Classification Keywords

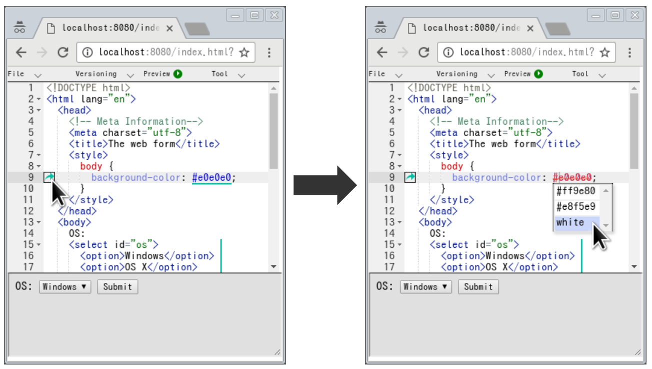
H.5.2 Information Interfaces and Presentation (e.g., HCI): User Interfaces; D.2.6 Software Engineering: Programming Environments

Author Keywords

Develpment environment; micro-versioning; version control system

INTRODUCTION

Experimentation is an essential part of exploratory programming; programmers write an experimental code fragment, test it, and discard the fragment frequently during programming [2, 13, 23]. During the iterations of this process, programmers often have to switch the part of the source code back to the past state. For example, they may sometimes restore a past code because the new code does not work as expected. Programmers use and modify different versions of source code when they revert the source code. These behaviors can be regarded as a series of small-scale version control operations, which we refer to as *micro-versioning*.



```
<!DOCTYPE html>
<html lang="en">
  <head>
    <!-- Meta Information-->
    <meta charset="utf-8">
    <title>The web form</title>
    <style>
      body {
        background-color: #e0e0e0;
      }
    </style>
  </head>
  <body>
    OS:
    <select id="os">
      <option>Windows</option>
      <option>OS X</option>
```

Figure 1. Screenshot of our prototype tool. It works in text editor to support micro-versioning. Our current implementation is designed for development of web page using HTML, CSS, and JavaScript. Bottom part shows preview of web page being developed.

A linear undo tool is widely used for micro-versioning. However, it causes various problems because it cannot undo edits if they are not the latest. Programmers need to manually maintain consistency if multiple separate parts of the code need to be undone together. For example, a programmer has to conduct linear undo operations twice to restore a deleted button widget and an associated listener code if she wants to restore the button widget. The button widget will not work if she forgets to restore the associated listener code.

Version control systems (VCSs), e.g., Git [9], are tools to support advanced operations such as commit and revert. However, they are designed to support version control operations during large-scale, collaborative software development and are not suitable for micro-versioning. These tools prevent users from conducting rapid micro-versioning operations because of the heavyweight of their user interfaces, that is, programmers have to open separate dialogs, find a specific edit, and select it to apply a versioning operation.

Yoon et al. reported that nearly 30% of micro-versioning operations are conducted by rewriting a source code directly [24]. However, a direct rewrite sometimes causes inappropriate text edits [23]. This suggests that programming environments require micro-versioning features that are better suited than the current tools. Therefore, we propose a micro-versioning tool that provides appropriate support in terms of the user interface and edit history.

We refer to version control operations *during exploratory programming* as micro-versioning. The granularity of the micro-versioning operations is between *backtracking* [23, 24] and version control operations conducted using VCSs. Backtracking focuses on only one region of a source code. On the other hand, versioning conducted using VCSs affects multiple regions and is often a large-scale, collaborative task involving multiple programmers. Micro-versioning also handles multiple regions of a source code; however, it is a small-scale, personal task. In this study, we used code executions as a delimiter to separate edit history into a meaningful set of edit operations [10]. This is based on an assumption that the user would execute a code after implementing a meaningful set of edits but would not execute it in the middle of a sequence of related edits.

We examined previous studies [21, 23, 24] to identify the following requirements for a micro-versioning tool. 1) It should have a lightweight user interface; 2) provide the user with visual information of operation; and 3) have an appropriate history model that proposes a limited amount of applicable micro-versioning features. We then developed a micro-versioning tool based on this analysis (Figure 1). The tool displays a candidate list, which is inspired by the auto-completion feature of integrated development environments (IDEs), to conduct micro-versioning. The tool visually represents locations where version control operations can be conducted. It also visualizes the operation in a code editor. We present our novel history model based on an current selective undo model [25]. Our model can extract meaningful micro-versioning operations from an edit history, eliminating inappropriate candidates. Finally, we report the results of a user study we conducted involving five participants to evaluate the usability of our tool and history model. We found that our tool satisfies the above requirements.

The main contributions of this study are as follows.

- We clarified the requirements for micro-versioning features based on previous research and developed a tool that contains the following two components.
- We developed a user interface to allow users to conduct micro-versioning operations in the main code editor without using separate dialogs.
- We constructed a history model that can extract meaningful micro-versioning operations from an edit history.
- We conducted a user study that demonstrated the usability of our tool and the effectiveness of our history model.

RELATED WORK

Linear Undo and Commenting Out

Programmers mostly conduct micro-versioning in two ways. The first involves using a linear undo tool [24], which can undo the latest edit (this is known as the shortcut-key `Ctrl+Z` in Windows and `Command-Z` in Mac OS). The second is based on commenting out [23], which is a feature of a programming language. These tools' capabilities are limited, but they and the features are widely used. For example, a linear undo tool cannot undo edits if they are not the latest. In addition, commenting out is problematic when multiple separate parts of the code need to be commented out or uncommented out together because programmers need to manually maintain consistency.

Extended Undo Tools

There are various extensions of linear undo tools because such tools are widely used and present many problems. Some text editors, such as Emacs [7], provide regional undo features that allow users to undo the latest edit in the selected region. These features solve the limitations of linear undo tools to some extent, but several problems remain. First, these tools do not provide visual representations, so they are often composed of a hidden feature. Second, they cannot apply edits to separate multiple parts of the source code at the same time. Finally, they cannot be used for all typical micro-versioning situations. For example, they cannot restore some previous text if there are undone text edits. Gundo [12] and Undo Tree [20] allow users to restore any previous texts by using tree-structured histories. However, these tools use complex user interfaces and cannot conduct regional undo operations, which are the most popular micro-versioning operations [25].

Several researchers have proposed selective undo tools that can undo any edits [18, 25], but they also use complex user interfaces. For example, when programmers selectively undo a specific edit in the simplest way, they are required to 1) open the history view, 2) find the edit they want to undo in the history view, and 3) undo the edit. Previous research has shown that many programmers tend to prefer lightweight user interfaces such as keyboard shortcuts (e.g., Quick Assist in Eclipse) rather than complex user interfaces such as dialogs and menus [21].

These tools [18, 25] sometimes generate inappropriate candidates as a result. The algorithm proposed by Yoon et al. [25] is used to attempt to generate appropriate candidates by considering regional conflicts. However, it still fails to eliminate inappropriate candidates in some cases. For example, suppose that a programmer sets a variable to `1.0`, changes the variable to `2.0`, undoes the previous edit, and finally changes the variable from `1.0` to `3.0`. If the user redoes the second operation, the algorithm generates `2.03.0`, which is inappropriate as a code. Our history model addresses these problems.

Version Control Systems

Version control systems are used often in programming. Many IDEs provide extensions that allow users to use VCSs [6] inside; however, they use complex user interfaces. Lee et al. proposed an IDE extension that embeds the information from VCSs into the code completion features [15]. This extension allows users to complete the current method name using the method name of a previous version that exists in the program. However, this extension is not suitable for micro-versioning because such tools require that users commit manually, which is a heavy-weight process for micro-versioning.

Many IDEs have local history-keeping features that save the source code when a programmer saves a file, which allows the user to restore it later [5]. However, these features involve complex user interfaces such as dialogs and menus, and most only support linear undos, so they are also not suitable for micro-versioning. Steinert et al. proposed an IDE extension that suggests version information to users based on the difference between versions of abstract syntax trees (ASTs) and the success rate of unit tests [19]. The usability for programmers was not tested, but this extension helps users to understand the versions of a program.

History Management Models

Several studies have proposed history models that allow users to manage and manipulate a history more efficiently. Nancel et al. proposed a conceptual model of interacting with a history, which uses information related to the target content of a history (e.g., texts and images), application, and commands made by the user [17]. However, this model cannot conduct regional undo operations and apply text edits to multiple separate parts of a source code at the same time, although both this model and that of our proposed tool are for simulating a tree structure within a linear history model. Cass et al. proposed a selective undo algorithm and model, which use a process-programming language to define the dependencies for a task [4]. Their model and our model are for eliminating inappropriate candidates from the result of selectively undoing, but their model cannot apply text edits to multiple separate parts of a source code at the same time. In addition, it is unrealistic to always expect the definition of a task model for programming because programming is a complex task, and there are many types of tasks in programming. Hayashi et al. proposed a tool that enables the refactoring of the edit history [14]. Both this tool and our model are for obtaining the edit history that is useful for version control operations; however, the tool requires a user to refactor the edit history manually.

Git [9] allows the user to revert a commit selectively by using the `git revert` command. The history model used by Git does not directly support regional undo operations, although it has the characteristics of both tree-structured history models and selective undo history models. This model is not suitable for micro-versioning because regional undo operations are popular in micro-versioning [25]. Git also allows the user to selectively revert the region of the source code to the previous commit by using the `git checkout -p` command. It enables a user to conduct all micro-versioning operations; however, a user has to manually maintain consistency. The history model of Git does not contain the information when the user conducts the `git checkout -p` command.

LITERATURE SURVEY TO ELICIT MICRO-VERSIONING TOOL REQUIREMENTS

We investigated previous studies to better understand the problems that programmers encounter during micro-versioning and to elucidate the requirements for efficient micro-versioning tools. First, we investigated previous studies on *backtracking*, which is a user's action during micro-versioning, to better understand some of the problems of micro-versioning. These studies show that programmers often use linear undo tools or commenting out for micro-versioning and that these tools and features have various problems [23, 24]. Programmers sometimes cannot use linear undo tools because nearly 10% of all *backtracking* operations are selective [24]. In addition, programmers often make mistakes when they comment or uncomment out multiple separate parts of a source code at the same time [23]. This indicates that a micro-versioning tool requires a history model that can propose typical micro-versioning operations.

We also investigated previous studies on automated refactoring tools. Several studies indicate that an automated refactoring tool should have a simple, minimal, and lightweight user interface, which allows users to conduct refactoring rapidly because programmers tend to prefer these user interfaces [16, 21]. In addition, several studies indicate that refactoring tools should provide information 1) to support decision making and 2) prevent the tool from being a hidden feature [16, 21].

These previous studies took into account automated refactoring tools, but we believe the same issues affect micro-versioning tools; micro-versioning tools should have the same requirements as automated refactoring tools. A micro-versioning tool should have a lightweight user interface and provide visual information.

Based on our literature survey, we established the following requirements for a micro-versioning tool.

- A tool should have a lightweight user interface that allows the user to conduct micro-versioning operations rapidly. Thus, it should not use separate dialogs or user interfaces that are similar to these dialogs.
- It should provide users with visual information to make them aware of a tool and support decision making. It should always display visual representations and information about micro-versioning operations that are applicable.
- Finally, it should use a history model that proposes applicable micro-versioning operations.

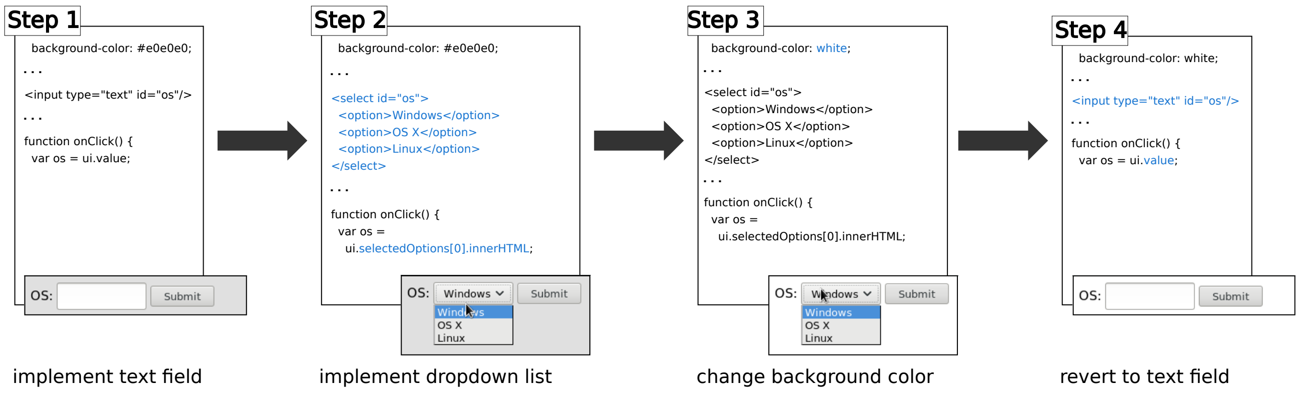
OUR MICRO-VERSIONING TOOL

We designed our micro-versioning tool based on the requirements described above. The tool's user interface is lightweight and provides the user with visual information about micro-versioning operations. As described in the next section, the history model used with the tool can conduct typical micro-versioning operations and eliminate inappropriate candidates.

Our tool works inside a text editor and can be used for any textual programming language. The user can use both our tool and traditional methods (e.g., linear undo tools and commenting-out) because our tool is integrated in a typical text editor.

Example Scenario

Suppose a programmer creates a web-form. The web-form gathers information about a user's operating system (OS). She first uses the text field to input the name of the OS (Step 1 in Figure 2). She then replaces the user interface of the web-form from the text filed to the dropdown list because there are only a few OSes (Step 2 in Figure 2). Note that this edit affects multiple locations in the code. She then modifies the background color of the web-form (Step 3 in Figure 2). She presses the preview button after each change to see the result. Our proposed tool records all these edit operations.



Step 1:

```
  background-color: #e0e0e0;
...
<input type="text" id="os"/>
...
function onClick() {
  var os = ui.value;
```

Step 2:

```
  background-color: #e0e0e0;
...
<select id="os">
  <option>Windows</option>
  <option>OS X</option>
  <option>Linux</option>
</select>
...
function onClick() {
  var os =
    ui.selectedOptions[0].innerHTML;
```

Step 3:

```
  background-color: white;
...
<select id="os">
  <option>Windows</option>
  <option>OS X</option>
  <option>Linux</option>
</select>
...
function onClick() {
  var os =
    ui.selectedOptions[0].innerHTML;
```

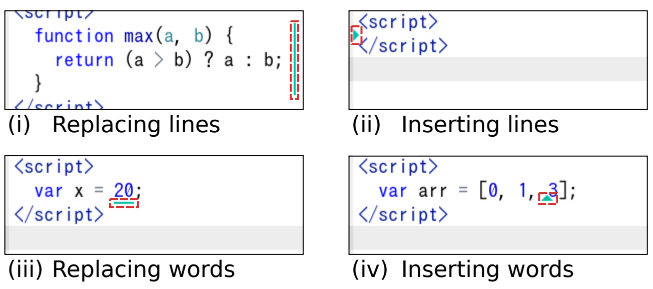
Step 4:

```
  background-color: white;
...
<input type="text" id="os"/>
...
function onClick() {
  var os = ui.value;
```

Figure 2. Example scenario of micro-versioning.

She notices that the dropdown list is unsuitable for the web-form because some users have several PCs. She wants to revert the change in the user interface. She sees *a marker* (Figure 5 A) and *indicator* (Figure 5 B), which are displayed with our tool, and notices that she can conduct micro-versioning operations related to the `select` tag by using our tool. The indicator and marker are generated by the recorded edit history.

She clicks the marker to conduct a micro-versioning operation. Our tool displays *a candidate list* (Figure 5 C) and *edit details* (Figure 5 D and E). She understands that the operation shown in the candidate list will revert the change in the user interface and the JavaScript program, which are required in order to use the text field. She clicks the operation shown in the candidate list because the operation matches her purpose. Finally, our tool conducts the micro-versioning operation, and the web-form uses the text field (Step 4 in Figure 2). Note that multiple lines including separated ones are appropriately reverted together.



```
<script>
  function max(a, b) {
    return (a > b) ? a : b;
  }
</script>
```

```
<script>
</script>
```

```
<script>
  var x = 20;
</script>
```

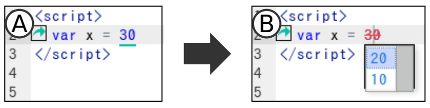
```
<script>
  var arr = [0, 1, 3];
</script>
```

Figure 3. Four types of indicators.

Lightweight User Interface

We designed the proposed tool so users can conduct all the basic operations related to micro-versioning inside the text editor without separate dialogs. The tool displays all the user interface elements in a text editor and displays indicators as visual representations. Four types of indicators (Figure 3) are used. The type (i) indicator, which is a vertical line located to the right of a text, is used when the tool must delete or replace lines of the text. The type (ii) indicator, which is a triangle located to the left of a text, is used when it is necessary to insert lines. The type (iii) indicator, which is a text underline, is used when it is necessary to delete or replace words. Finally, the type (iv) indicator, which is a triangle located at the bottom of the line, is used to insert words.

The tool displays markers to allow easy access by a user because the indicators are small and difficult to click. The location of a marker depends on the corresponding text. If the corresponding text is composed of less than one line, a marker is located near the gutter of the text editor, which displays line numbers (Figure 4 A); otherwise, the marker is located to the right of the text (Figure 5 A). The candidate list is displayed when a user clicks on a marker or indicator or uses the keyboard shortcut (`Ctrl+Space` by default). There are two types of candidate lists. The drop-down list (Figure 4 B), which is located below the text, is used when the related indicator type is (iii) or (iv); otherwise, the second (Figure 5 C) is used, which is located to the right of the text. Edit details are displayed either within the text editor or on the scrollbar (Figure 5 D and E). In addition, the user can use a mouse and keyboard to interact with our tool, such as opening a candidate list and conducting a micro-versioning operation.



```
<script>
  var x = 30
</script>
```

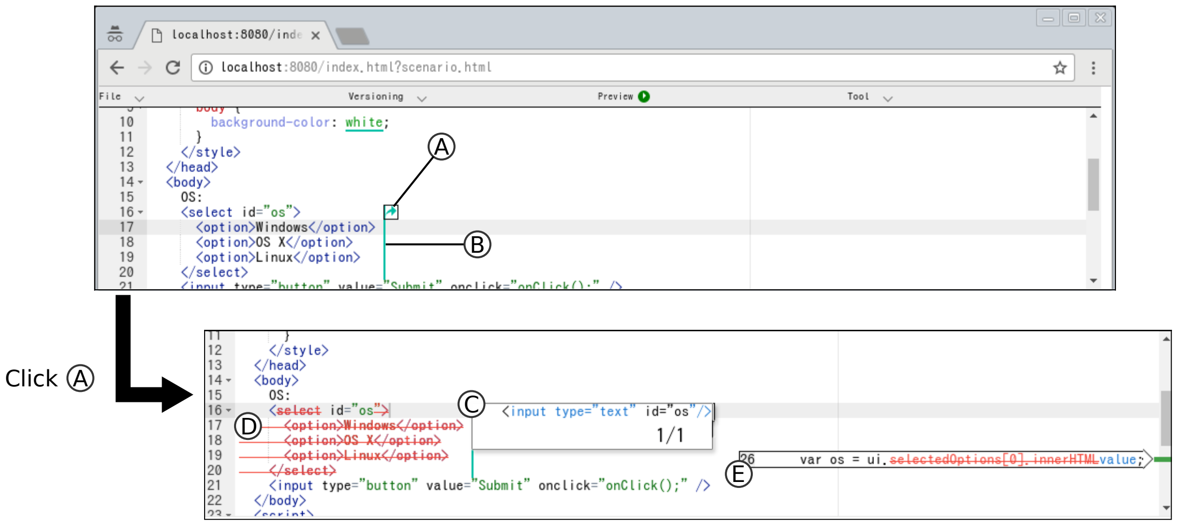
```
<script>
  var x = 30
</script>
```

Figure 4. A) Example of marker and indicator. B) Example of candidate list, which is located below text.

Visual Information

Our tool always displays indicators to enhance the user's awareness of micro-versioning operations. A marker is displayed if the caret of the text editor is positioned on the corresponding text, which provides the user with information about the micro-versioning operations that are applicable to the focused text.

In addition, edit details and candidate lists provide information about a micro-versioning operation, such as text inserted and deleted by the micro-versioning operation. Edit details in the scrollbar show information regardless of whether the micro-versioning operations are displayed onscreen. If an



```
    body {
      background-color: white;
    }
  </style>
</head>
<body>
  OS:
  <select id="os">
    <option>Windows</option>
    <option>OS X</option>
    <option>Linux</option>
  </select>
  <input type="button" value="Submit" onclick="onClick();" />
```

```
    }
  </style>
</head>
<body>
  OS:
  <~~select~~ id="os"~~>~~
    ~~<option>Windows</option>~~
    ~~<option>OS X</option>~~
    ~~<option>Linux</option>~~
  ~~</select>~~
  <input type="button" value="Submit" onclick="onClick();" />
</body>
<script>
```

```
<input type="text" id="os"/>
1/1
```

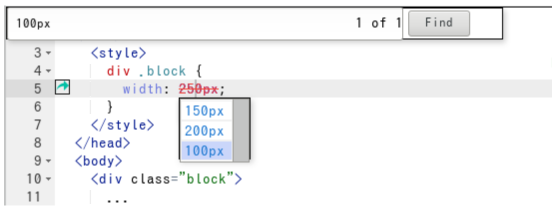
```
26    var os = ui.~~selectedOptions[0].innerHTML~~value;
```

Figure 5. Lightweight user interface for micro-versioning. (A) Marker, (B) indicator. (C) candidate list, and (D and E) editing details

operation affects a large part of the source code, some edits are not visible on the screen. Thus, users are unable to follow what occurs when some edits are not displayed onscreen. We addressed this issue by displaying edit details that show the line numbers and the difference between texts on the scrollbar. The user can understand the outcomes of a micro-versioning operation by observing the edit details, so it is expected that they can conduct more accurate micro-versioning operations by using our tool compared with using linear undo tools or commenting-out. The text deleted by the operation appears as struck-out red text, whereas any text inserted by the operations is represented as blue text in edit details (Figure 5 D and E) and candidate lists (Figure 5 C).

Search Feature

The tool implements a feature that allows the user to search through all the recorded versions using text (Figure 6). The tool shows only recent operations inside the main text editor view to prevent confusion (described in the next section), however, this search feature will search for all the versions stored in the tool. The user opens the search bar, types text in it, and clicks the find button in the search bar. The tool then opens a candidate list and focuses on the operation that includes the typed text. If the user clicks the find button once more, the tool shows another operation that includes the text. This search feature is particularly useful when the user does not remember the location of the target micro version, but remembers the text included in the version.



```
  <style>
    div .block {
      width: ~~250px~~;
    }
  </style>
</head>
<body>
  <div class="block">
    ...
```

```
150px
200px
100px
```

Figure 6. Search feature

HISTORY MODEL

Our history model extends the selective undo model [25] so that it can generate meaningful micro-versioning operations from an edit history. Specifically, our model supports the following four novel features: 1) regional redo, 2) exclusive edits, 3) grouping of separate edits, and 4) redo propagation. Before explaining these extensions, we first describe the basic building blocks of our model.

The original selective undo model aggregates collocated text edit operations (deletion, insertion, and replacement) into one and uses it as a basic unit. We call this an *atomic edit*. Our model also uses atomic edits but introduces a novel structure called *edit fragment* on top of them and uses it as a basic unit. An edit fragment wraps an atomic edit (called *representative edit*) and stores the following four additional pieces of information associated with the representative edit.

First, each edit fragment contains a flag indicating whether it is currently enabled or disabled. Second, each edit fragment stores timestamps when it was toggled (enabled or disabled) in the past. This is used to calculate its score when the system enumerates the candidates. Third, each edit fragment stores all the atomic edits caused by its last toggle. This information (called *affected edits*, AE) is necessary to undo these atomic edits when the edit fragment is toggled again. Finally, each edit fragment stores all the edit fragments that were disabled when it was disabled last time. This information (called *affected fragments*, AF) is used to support redo propagation.

Our model also stores *dependency relation* among edit fragments (this corresponds to *conflicts* among atomic edits in [25]). We define an edit fragment f2 *depending on* f1 when f2 is later than f1 and f2's representative edit overlaps that of f1 (Figure 7). Our model propagates disablement and enablement following the dependency relationship as in [25]. When f2 is enabled, f1 is automatically enabled. When f1 is disabled, then f2 is automatically disabled.

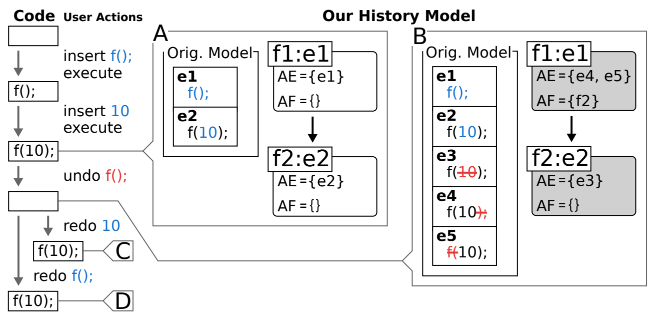


Figure 7. Regional undo and redo propagation. Our model wraps original model. e1 to e5 are atomic edits. f1 and f2 are edit fragments. Gray fragments are disabled. Black arrows represent dependency between edit fragments.

1) Regional Redo

The original model supports regional undo but does not fully support regional *redo*. Consider the situation shown in Figure 7. If the user undoes `e1` after Figure 7 A, the original model appropriately undoes `e2` as well (regional undo). However, if the user redoes `e2` after Figure 7 B (which corresponds to undoing `e3` in their model), the result becomes `10`, which is inappropriate. Ideally, the result should be `f(10)`.

Our model addresses this problem using the dependency relation among edit fragments. When the user redoes `e2` after Figure 7 B, it is recognized as enabling `f2` in our model. Since the model determines that `f2` depends on `f1`, it also enables `f1`. Enabling `f2` undoes `e3` and enabling `f1` undoes `e4` and `e5`, resulting in `f(10)` as expected (Figure 7 C).

2) Exclusive Edits

The original model fails to properly handle exclusive edits. Consider the situation shown in Figure 8. Since `e2` and `e3` replaced the same text, they should not coexist. However, if the user redoes `e2`, the original model is not aware of the relation and produces `x=2.03.0`, which is inappropriate.

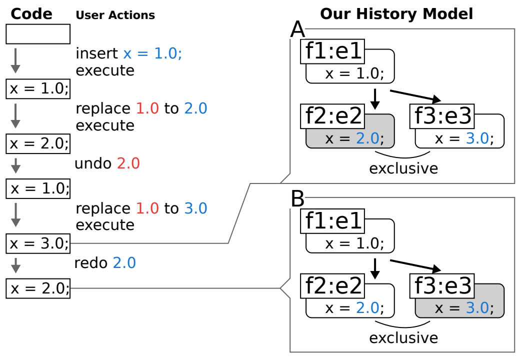


Figure 8. Example of model structure that contains *exclusive* relation.

Our model addresses this problem by simulating a tree-structured undo model by introducing an *exclusive* relation among edit fragments. We define that two edit fragments are *exclusively* related if they delete the same part of a text. When the user enables an edit fragment `f`, it automatically disables edit fragments that are *exclusively* related to `f`. For example, `f2` and `f3` in Figure 8 are *exclusively* related because both delete `1.0`. If the user enables `f2` after Figure 8 A, it disables `f3`, producing `x=2.0` as expected (Figure 8 B).

3) Grouping Separated Edits

The original model fails to handle separated, but related edits properly. Suppose that the user inserts `x=10;` and `print(x);` in distant locations in a code. If the user undoes `x=10;` then the system should also undo `print(x);` because `print(x);` becomes inappropriate without `x=10;`. However, the original model cannot support this.

Our model supports grouping of such separate edit fragments by using code executions as delimiter. The edit fragments created between the same two code executions are put into the same group [10]. If the model toggles an edit fragment `f`, it also toggles all the edit fragments in the same group. Consider the situation shown in Figure 9. If the user disables `f1`, the model disables `f2` because they are in the same group.

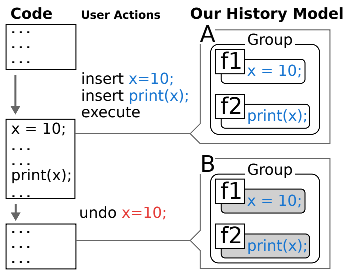


Figure 9. Example of model structure that contains separated but related edit fragments. Edit fragments `f1` and `f2` are put in same group.

4) Redo Propagation

If the user undoes an operation and redoes it later, the original code before undo should be restored. However, the original model fails to support this. Consider the situation shown in Figure 7. The original state is `f(10)` at Figure 7 A. When the user undoes `e1`, the original model also undoes `e2` because `e2` is dependent on `e1` (Figure 7 B). However, when the user redoes `e1` later, it does not redo `e2`, resulting in `f()`, which is different from the original state `f(10)`.

Our model uses the affected fragments stored in each edit fragment to address this problem. When the user enables an edit fragment, it also enables its affected fragments. Suppose that the user enables `f1` after Figure 7 B. The system automatically enables `f2` because `f2` is recorded as an affected fragment of `f1`. This successfully restores the original state `f(10)` (Figure 7 D).

Enumeration of Candidates

Our history model needs to enumerate candidate micro-versioning operations to show them in the code editor. First, the model calculates a score for each edit fragment by using an evaluation function found in an existing bug prediction algorithm [11] to generate candidates. The algorithm scores each edit fragment based on how many times the edit fragment was toggled, as well as their recentness. Second, it extracts the edit fragments with the highest N scores. We use $N = 5$ in our current implementation. It then toggles each edit fragment. Our model toggles other edit fragments related to the edit fragment as described above. The resulting code becomes a candidate of micro-versioning operations. Identical candidates are unified, so the number of candidates is less than N.

For example, the model extracts `f1` and `f2` in Figure 7 A in the case in which $N = 2$. The system generates an empty text as a result of disabling `f1` (disabling `f2` as well as a side effect), and generates `f()` as a result of disabling `f2` (no side effect).

IMPLEMENTATION

We implemented our micro-versioning tool as an extension of the Ace text editor[1], which works inside a web browser. The user interface of the micro-versioning tool was written in JavaScript, and the history model was implemented in Scala. The implemented tool is a text editor for coding HTML, CSS, and JavaScript. In addition to this micro-versioning tool, we implemented typical features of text editors, such as opening and saving a file. Our tool can also display the preview of the HTML opened in the text editor. Our current implementation is a proof-of-concept prototype and is missing important features to be used as a full-fledged coding environment. First, our current implementation only supports the edit of a single document (source code), and does not support the development of a system consisting of multiple documents. Second, our current implementation is built on a plain text editor and does not provide advanced supports seen in standard IDEs such as code completion and error highlighting.

The output of the history model is a set of candidate source codes, each of which corresponds to a candidate micro-versioning operation enumerated with the model. The system generates visual elements for each candidate source code as follows. The system first calculates the word-level differences between the current source code and the candidate source code using `dwdiff -P` command[2]. The tool also calculates the line-level difference by using the histogram diff algorithm[3], which is the algorithm used in EGit [6]. A collocated set of word-level and line-level differences is called a *delta*.

Indicators and candidate lists are created from either word-level or line-level deltas. If a word-level delta spans multiple lines, the system uses a corresponding line-level delta. Otherwise, it uses a word-level delta. Indicator and candidate lists are merged if they overlap. For example, indicator B and candidate list C in Figure 5 are created from the line-level delta because the word-level delta (replacing `">...</select>` with `"/>`) spans multiple lines. The candidate list in Figure 6 is created from the word-level delta.

Edit details are generated from the word-level difference. When the user focuses on a candidate, all the related edit details are shown, whereas other edit details are hidden. Our tool saves the edit history using the JSON format.

[1]https://ace.c9.io/

[2]http://linux.die.net/man/1/dwdiff

[3]http://download.eclipse.org/jgit/docs/jgit-2.0.0.201206130900-r/apidocs/org/eclipse/jgit/diff/HistogramDiff.html

USER STUDY

We conducted a small-scale user study to evaluate our tool and history model. The user study was composed of two parts. In the first part, participants tested our tool and evaluated its usability by completing questionnaires. The second part identified what users expect of micro-versioning candidates by showing several examples of typical micro-versioning situations to the participants.

We recruited five participants who were graduate or undergraduate students in our computer science department, all males aged 22 to 23 years. They each had over two years experience in programming and experienced with JavaScript and HTML.

Part I: Evaluation of Overall Tool

The aim of the first part of the study was to evaluate our tool. We examined whether our user interface satisfied the requirements determined in this study. First, the participants were asked to complete a training task in about 10 minutes using our proposed tool. Next, they were asked to complete a feature-adding task within about 1 hour. We observed and noted the behavior of the participants and their comments during the experiments. After finishing the tasks, we asked the participants to complete a questionnaire about our tool and micro-versioning tasks. The system usability scale (SUS) [3] was used in the questionnaire.

We prepared a training task to allow the participants to become familiar with our proposed tool so they could conduct micro-versioning operations. In the training task, a participant wrote a code to add an HTML button and its event listener first, before changing the behavior when clicking the button, then removed the button. The last change could be conducted by applying undo using our tool.

We prepared a paint program written in HTML, CSS, and JavaScript as the base code for the study, which contained 174 lines of code. Participants were asked to add a thickness control feature by using a text field and to re-implement the same by using a slider. Next, they were asked to add a preview of the thickness. Finally, they were asked to delete the preview feature. The paint program written in Java has been used in other studies [8, 23], and a thickness control feature was used to collect the *backtracking* behavior of programmers [23]. These studies showed that the Java program is sufficiently simple to understand and that is can be modified within a short amount of time, while implementing the thickness control feature requires some micro-versioning operations. Thus, we decided to use these programming tasks in our study. A previous study [23] also indicated that the behavior of most programmers was not affected by knowing that they might conduct a micro-versioning operation later. Thus, the participants were asked to do only one programming task in this study.

All the participants used a laptop computer (Core i5-5200U and 8GB RAM) on which our tool operated in Firefox 44.0.2. They used two full HD (1920 × 1080 pixels) displays and a mouse. The participants were told that they could use any resources available to write the program and for debugging. They mainly used Google to find JavaScript functions and

#	Question	P1	P2	P3	P4	P5
1	I thought it was easy to use	6	5	6	6	5
2	I thought it was suitable for programming	6	6	4	6	5
3	I felt very confident using the tool	3	3	1	6	3
4	I would imagine that most people would learn to use the tool very quickly.	6	3	2	6	3
5	I needed technical support to use it	4	2	7	2	5
6	I had to concentrate to use it	4	2	7	2	6
7	I would like to use the tool frequently	7	5	4	6	5

Table 1. Results of post questionnaire on seven-point Likert scale

	P1	P2	P3	P4	P5
Update preview	17	21	14	34	13
Open candidate lists	2	8	4	3	2
Conduct operations using our tool	1	3	3	1	2
Search operations	1	1	0	0	1
Revert operations manually	1	0	0	2	0

Table 2. Number of operations conducted during main task

HTML elements, and the Web console in Firefox for debugging.

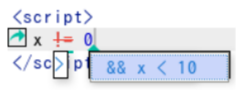
Results

All the participants successfully completed the training and main tasks. The main task required 46.2 minutes on average (SD = 19.6 minutes). The results of the post-questionnaire analysis are listed in Table 1. A score was positive when it was higher than 4 for Q1–4 and Q7, but vice versa for Q5 and Q6. Table 2 lists the participants' actions related to micro-versioning during the main task.

All the participants agreed that our tool was easy to use (Q1). Three participants also responded that they would be able to write source code faster using our tool. Four participants responded negatively to Q3 because they were not familiar with our tool. One participant (P2) said that: *"I sometimes cannot predict the results of my operation. However, I think I will feel confident because the results were intuitive in the user study."* There were no negative responses to Q2 and Q7, but one participant (P3) was neutral. This participant also responded negatively to Q4–6. His neutral or negative responses were found to be related to his programming method, in which he wanted to manually write or rewrite each source code to check changes. Thus, he felt anxiety when he used our tool because several parts of the source code were changed simultaneously.

Visual information displayed by tool: All the participants said that the information displayed by the tool was useful. One participant (P1) said that: *"By using this tool, I could understand the location where I made changes at a glance."* Another (P4) said that *"I could understand the details of the proposed candidate by observing the information displayed by the tool."* During programming, all the participants observed a candidate list and edit details to find a desired micro-versioning operation before they decided their next actions. If they found a desired micro-versioning operation, it was applied. If not, they closed the candidate list and opened another list.

However, two participants complained that the information was not sufficient in several cases. One participant (P1) noted that it was sometimes difficult to understand the outcomes when the micro-versioning operation was complex. The right figure shows the edit details and candidate list that P1 said were complex. Another (P4) said that the indicators should display more information, such as the importance of micro-versioning operations. In addition, two participants (P1 and P4) ignored indicators when they reverted an extremely small operation and manually reverted the source code. For example, P1 changed the parameter from `1` to `10` then rewrote it from `10` to `1` without using our tool.



```
<script>
x != 0
</sc>pt
&& x < 10
```

Complex visual information

Candidates proposed by tool: All the participants agreed that almost all the candidates proposed by the tool were appropriate. However, two participants (P2 and P3) said that the tool should not propose an operation for deleting the entire source code. They said that deleting all the source code was inappropriate, and it disrupted them when they tried to find a desired operation.

Other possible applications using tool: Three participants suggested that the proposed tool would also be useful for other applications. Two participants (P2 and P4) said that it might be useful for various types of creative activities, such as writing a novel, writing lyrics, composing music, and drawing pictures. One participant (P2) said *"My friend sometimes writes novels, and often conducts experiments. My friend would like to save and use the history of his experiments."* Another (P4) said that the proposed tool might be useful for programming education. He considered that novices often encounter problems while programming and that the proposed tool might be useful because it improves user experience when problems occur. One participant (P1) said that he would use the proposed tool when using document editing software such as Microsoft Word. Another (P2) considered that the tool might be useful when editing his schedule.

Performance: Our tool took several tens of milliseconds to add a new edit fragment during the experiment, and it is empirically linear time. It took 8.3 msec when the number of edit fragments was 10 on average and 54 msec when the size was 44.

Part II: Evaluation of History Model

The aim of the second part of the study was to evaluate our history model and compare it with other history models, such as linear undo and selective undo. Thus, we aimed to understand 1) whether the model can propose candidates that meet the expectations of users, and 2) whether it can eliminate inappropriate candidates. In this study, we investigated previous research [24, 25] to prepare the example situations for micro-versioning and found seven *backtracking* situations that are similar to the typical situations encountered during micro-versioning (Table 3) [23]. Second, a linear undo tool, which is the most common tool for micro-versioning, does not work well when multiple situations (e.g., refactoring and fixing small mistakes) occur simultaneously or when some text edits are undone [25].

The selective undo model may generate too many candidates when an example contains many text edits. Thus, we reduced the number of text edits by separating the example into four micro-versioning situations in which these four examples represent micro-versioning situations. Each example contained either multiple typical micro-versioning situations or text edits undone by a programmer (*A*, *B*, and *C* contained multiple situations and *D* contained an undone text edit). The programming tasks used in these examples were extracted from previous programming studies [22, 24, 25, 1, 23].

First, we showed the participants each example of an edit history and the candidate of micro-versioning operations generated using the selective undo model [25]. We used the selective undo model because the candidates it generated were composed of a superset of those generated using the extended undo models and our model. Figure 10 shows an edit history and the candidates for *D* in Table 3. We then asked the participants to select the candidates that could be considered as results after reverting a source code change. We also required explanations when a participant's selection differed from the result obtained with our model.

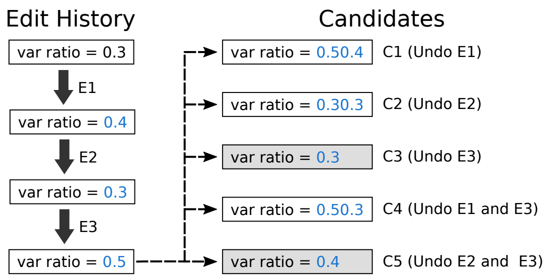


Figure 10. Edit history and candidates for *D* in Table 3. Gray candidates are those generated with our tool.

Results

Our history model generated a total of eight candidates from all edit histories. The participants correctly selected 7.4 of these on average (SD = 0.89). However, they selected 2.2 of the 14 candidates that could not be generated with our model on average (SD = 1.30). Table 4 lists the candidates that participants selected and those that our model generated for *D*. One participant (P3) selected C5, which could be generated with our model; however, he did not select C3, which could also be generated with our model.

We confirmed that all the candidates selected by the participants can be generated by combining several regional undo operations or one tree-structured undo operation. We also confirmed that none of the participants selected all the candidates generated with the selective undo model, indicating that the selective undo model generated several inappropriate candidates. These results indicate that our underlying assumptions regarding our model are reasonable. However, the sets of candidates selected by the participants were all different, except for *B*. This indicates that a history model cannot generate candidates that exactly match the expectations of all users.

We investigated the explanations given by the participants for the cases in which their answers differed from the output of our history model and identified the following two problems with our model. First, the expectations of the participants sometimes depended on the semantics of the program under development. For example, most participants (three out of five) answered that reverting several steps should be considered as a reverting operation in *A*, but all the participants answered that reverting several steps should not be considered as a reverting operation in *B*. All the participants said that *B*-2 and *B*-3 appeared to be totally different text edits and answered that they should not be reverted simultaneously. Likewise, all the participants answered that restoring an API name (*C*-2) should be considered as a reverting operation, but two participants (P1 and P5) said that restoring a typo (*A*-3) should not be considered as a reverting operation. They said that: *"I do not think that fixing a typo should be part of the edit history. Thus, I did not select restoring typo as a reverting operation."*

The second problem is concerned with reverting the last text edit, which is similar to a linear undo operation, and it differed according to the participant. Three participants (P1, P3, and P4) responded that reverting the last text edit should be considered as a reverting operation regardless of whether the reverted program could be compiled. However, two participants responded that a reverted program should be capable of being compiled.

DISCUSSION

The results of the first part of the user study indicate that our tool was useful and easy to operate. The participants agreed that our tool satisfied the requirements described above. In addition, we found that the user interface of the proposed tool might be useful for several tasks in addition to programming, such as creative activities and programming education. The results of the second part indicate that our history model could combine two extended undo models, show that appropriate individual reverting operations are different and that no history model can support all micro-versioning operations in all situations.

The possible extensions proposed by the participants were related to various targets such as writing novels, drawing pictures, and writing music, although our tool was designed for text-based programming. This shows that trial and error is a common task because a user encounters many problems during these iterative processes. Our findings also indicate that the defined requirements are important in various domains in addition to programming. However, providing micro-versioning tools in various domains would present many challenges, but identifying the specific requirements and using the proposed tool could help address these challenges.

User Interface of Micro-Versioning Tool

Our tool successfully displayed useful information. However, our results suggest that a more sophisticated visualization method is required, which should 1) be capable of generating

Case	Situation	Edit Summary	#Candidates	References
A	• refactoring • fix typo or small mistakes	(1) Write program with magic number (2) Introduce new variable to remove magic number. (3) Fix typo in name of variable.	7 (2)	[1, 22]
B	• try to find appropriate algorithm • fix code that was just added because it is not working	(1) Implement `factorial` method using recursion (2) Fix method (3) Implement `factorial` method using tail recursion	3 (2)	[1]
C	• understand use of new API • test various user interface designs	(1) Modify HTML using `innerText` variable (2) Change `innerText` to `innerHTML`. (3) Modify user interface from slider to text field	7 (2)	[23, 25]
D	• tune parameters	(1) Change parameter from 0.3 to 0.4 (2) Undo it (3) Change it to 0.5	5 (2)	[24]

Table 3. Examples of typical micro-versioning situations. Numbers in parentheses are those of candidates that can be generated with our history model.

Case *D*	C1	C2	C3	C4	C5
P1			✓		✓
P2			✓		✓
P3					✓
P4			✓		✓
P5			✓		✓

Table 4. Results of *D*. Checked cells were selected as reverting operations by participant. Gray cells can be generated with our model.

an understandable visualization of a complex operation and 2) provide all of the information related to a micro-versioning operation. Our tool limited the number of candidates, however, it can easily show more candidates by adding a "see more candidates" button, for example.

Displaying the behavior of the source code after the micro-versioning operation in a separate dialog sometimes provides a user with useful information, but it makes the user interface complex. We did not implement this feature because a micro-versioning tool should have a lightweight user interface. However, one participant reported that he experienced anxiety when he used the proposed tool and previewing the behavior of software might help reduce anxiety. However, a previous study indicated that many programmers did not observe a preview of the source code [21]. Thus, we cannot be certain whether previewing the behavior of software is helpful.

Extracting Appropriate Candidates

Another issue is that the appropriate candidates differ according to the programmers. There are several approaches to solve this issue. One approach is displaying many indicators so that all users can find their appropriate candidates by using the micro-versioning tool. However, this approach increases the number of inappropriate candidates and makes the user interface complex. Another approach is using a more sophisticated history model. Such a model may decrease the number of candidates that the user wrongly wants to revert to (e.g., the user overlooks the syntax error of the candidate). Yet another approach is changing the behavior of the history model according to the user. How to address this issue will be addressed in future research.

Limitations

Our history model uses code executions as delimiters, but this does not always work. For example, a programmer makes a button and the event listener of the button (edit *A*). She then executes the program. Finally, she adds the extra listener of the button (edit *B*). The relation between edits *A* and *B* cannot be extracted with our current implementation.

Using an AST can relax this restriction because it contains the syntax relation between edit fragments. However, there are several types of relations between fragments that cannot be extracted using an AST. For example, the font size of a title and that of a subtitle are related; thus, a programmer sometimes wants to update two parameters in sync. However, there is no relation between two parameters at the AST level. Our tool worked well in the user study; however, it and an AST should be combined to support more efficient micro-versioning.

We used the artificial programming tasks and edit histories in the user study without the related context of exploratory programming. We designed the tasks and histories based on common backtracking situations and previous programming studies to prevent them from being too artificial. All the participants were recruited from our university. However, previous studies showed that both university students and professional programmers frequently conduct exploratory programming and backtracking [23]. Our user study was small-scale and it is insufficient to quantitatively prove the practical usability of the system. Formal studies with a sufficient number of participants and comparison are necessary for this purpose.

CONCLUSION

We referred to the version control operations during exploratory programming as micro-versioning. We analyzed the problems that programmers encounter during micro-versioning processes and defined the requirements for a tool to support micro-versioning tasks. We also proposed a tool based on these requirements. The interface of the tool displays markers, indicators, candidate lists, and edit details in a text editor. Users can select an option from the candidate list to do micro-versioning tasks. The results of our user study indicate that our tool satisfies the requirements we identified, and that it is useful for programming and other tasks.

REFERENCES

1. Kent Beck. 2003. *Test-driven development: by example*. Addison-Wesley Professional.
2. Joel Brandt, Philip J Guo, Joel Lewenstein, Scott R Klemmer, and Mira Dontcheva. 2009. Writing Code to Prototype, Ideate, and Discover. *Software, IEEE* 26, 5 (2009), 18–24. DOI: http://dx.doi.org/10.1109/MS.2009.147
3. John Brooke. 1996. SUS-A quick and dirty usability scale. *Usability evaluation in industry* 189, 194 (1996), 4–7.
4. Aaron G Cass and Chris ST Fernandes. 2007. Using task models for cascading selective undo. In *Task Models and Diagrams for Users Interface Design*. Springer, Hasselt, Belgium, 186–201. DOI: http://dx.doi.org/10.1007/978-3-540-70816-2_14
5. Eclipse 2016. Eclipse - The Eclipse Foundation open source community website. (31 July 2016). Retrieved July 31, 2016 from http://www.eclipse.org/.
6. EGit 2016. EGit. (31 July 2016). Retrieved July 31, 2016 from http://www.eclipse.org/egit/.
7. emacs 2016. Emacs. (31 July 2016). Retrieved July 31, 2016 from https://www.gnu.org/software/emacs/.
8. James Fogarty, Andrew J Ko, Htet Htet Aung, Elspeth Golden, Karen P Tang, and Scott E Hudson. 2005. Examining task engagement in sensor-based statistical models of human interruptibility. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems (CHI 2005)*. ACM, Portland, Oregon, USA, 331–340. DOI: http://dx.doi.org/10.1145/1054972.1055018
9. git 2016. Git. (31 July 2016). Retrieved July 31, 2016 from https://git-scm.com/.
10. Max Goldman, Greg Little, and Robert C Miller. 2011. Collabode: collaborative coding in the browser. In *Proceedings of the 4th international workshop on Cooperative and human aspects of software engineering (CHASE 2011)*. ACM, Waikiki, Honolulu, HI, USA, 65–68. DOI:http://dx.doi.org/10.1145/1984642.1984658
11. Google Bug Prediction 2016. Google Bug Prediction. (24 July 2016). Retrieved July 31, 2016 from http://google-engtools.blogspot.jp/2011/12/bug-prediction-at-google.html.
12. gundo 2016. Gundo - Visualize your Vim Undo Tree. (29 March 2016). Retrieved July 31, 2016 from http://sjl.bitbucket.org/gundo.vim/.
13. Björn Hartmann, Loren Yu, Abel Allison, Yeonsoo Yang, and Scott R Klemmer. 2008. Design as exploration: creating interface alternatives through parallel authoring and runtime tuning. In *Proceedings of the 21st annual ACM symposium on User interface software and technology (UIST 2008)*. ACM, Monterey, CA, USA, 91–100. DOI: http://dx.doi.org/10.1145/1449715.1449732
14. Shinpei Hayashi, Daiki Hoshino, Jumpei Matsuda, Motoshi Saeki, Takayuki Omori, and Katsuhisa Maruyama. 2015. Historef: A tool for edit history refactoring. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015)*. IEEE Computer Society, Montreal, QC, Canada, 469–473. DOI: http://dx.doi.org/10.1109/SANER.2015.7081858
15. Yun Young Lee, Sam Harwell, Sarfraz Khurshid, and Darko Marinov. 2013. Temporal code completion and navigation. In *35th International Conference on Software Engineering (ICSE 2013)*. IEEE Computer Society, San Francisco, CA, USA, 1181–1184. DOI: http://dx.doi.org/10.1109/ICSE.2013.6606673
16. Erica Mealy, David Carrington, Paul Strooper, and Peta Wyeth. 2007. Improving usability of software refactoring tools. In *Australian Software Engineering Conference (ASWEC 2007)*. IEEE Computer Society, Melbourne, Australia, 307–318. DOI: http://dx.doi.org/10.1109/ASWEC.2007.24
17. Mathieu Nancel and Andy Cockburn. 2014. Causality: A conceptual model of interaction history. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, Toronto, ON, Canada, 1777–1786. DOI: http://dx.doi.org/10.1145/2556288.2556990
18. Atul Prakash and Michael J Knister. 1994. A framework for undoing actions in collaborative systems. *ACM Transactions on Computer-Human Interaction (TOCHI)* 1, 4 (1994), 295–330. DOI: http://dx.doi.org/10.1145/198425.198427
19. Bastian Steinert, Damien Cassou, and Robert Hirschfeld. 2012. Coexist: Overcoming aversion to change. *ACM SIGPLAN Notices* 48, 2 (2012), 107–118. DOI: http://dx.doi.org/10.1145/2384577.2384591
20. undotree 2016. EmacsWiki: Undo Tree. (31 July 2016). Retrieved July 31, 2016 from http://www.emacswiki.org/emacs/UndoTree.
21. Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P Bailey, and Ralph E Johnson. 2012. Use, disuse, and misuse of automated refactorings. In *34th International Conference on Software Engineering (ICSE 2012)*. IEEE Computer Society, Zurich, Switzerland, 233–243. DOI: http://dx.doi.org/10.1109/ICSE.2012.6227190
22. YoungSeok Yoon and Brad A Myers. 2011. Capturing and analyzing low-level events from the code editor. In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools (PLATEAU 2011)*. ACM, Portland, OR, USA, 25–30. DOI:http://dx.doi.org/10.1145/2089155.2089163
23. YoungSeok Yoon and Brad A Myers. 2012. An exploratory study of backtracking strategies used by developers. In *Proceedings of the 5th International Workshop on Co-operative and Human Aspects of*

Software Engineering (CHASE 2012). IEEE Computer Society, Zurich, Switzerland, 138–144. DOI: http://dx.doi.org/10.1109/CHASE.2012.6223012

24. Young Seok Yoon and Brad A Myers. 2014. A longitudinal study of programmers' backtracking. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2014)*. IEEE Computer Society, Melbourne, VIC, Australia, 101–108. DOI:http://dx.doi.org/10.1109/VLHCC.2014.6883030

25. Young Seok Yoon and Brad A Myers. 2015. Supporting selective undo in a code editor. In *the 37th International Conference on Software Engineering-Volume 1 (ICSE 2015)*. IEEE Computer Society, Florence, Italy, 223–233. DOI:http://dx.doi.org/10.1109/ICSE.2015.43