# Supporting Selective Undo in a Code Editor

YoungSeok Yoon
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA, USA
youngseok@cs.cmu.edu

Brad A. Myers
Human-Computer Interaction Institute
Carnegie Mellon University
Pittsburgh, PA, USA
bam@cs.cmu.edu

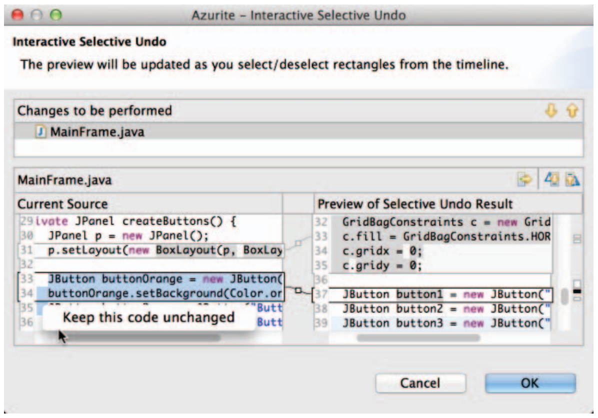
***Abstract*—Programmers often need to revert some code to an earlier state, or restore a block of code that was deleted a while ago. However, support for this *backtracking* in modern programming environments is limited. Many of the backtracking tasks can be accomplished by having a *selective undo* feature in code editors, but this has major challenges: there can be conflicts among edit operations, and it is difficult to provide usable interfaces for selective undo. In this paper, we present AZURITE, an Eclipse plug-in that allows programmers to selectively undo fine-grained code changes made in the code editor. With AZURITE, programmers can easily perform backtracking tasks, even when the desired code is not in the undo stack or a version control system. AZURITE also provides novel user interfaces specifically designed for selective undo, which were iteratively improved through user feedback gathered from actual users in a preliminary field trial. A formal lab study showed that programmers can successfully use AZURITE, and were twice as fast as when limited to conventional features.**

***Index Terms*—Selective undo, backtracking**

## I. Introduction

Since programmers are human, they often make mistakes while writing program code. In other cases, programmers may intentionally make temporary changes to the code, either as an experiment or to help with debugging. As a consequence, programmers often need to *backtrack* while coding. We define backtracking as "going back at least partially to an earlier state either by removing inserted code or by restoring removed code" [1]. Previously, we conducted a lab study, a survey of backtracking practices of programmers, and a longitudinal study of 1,460 hours of actual code editor use from 21 programmers. The results confirmed that backtracking is in fact prevalent and programmers often report having problems when they want to backtrack [1]. The programmers in the longitudinal study backtracked 10.3 times per hour on average, and 34% of all the detected backtracking instances were performed manually without using the undo command or any other tool support [2].

Others have also shown that programmers frequently backtrack while coding. From a high level view, programmers often do *exploratory programming*: they build quick prototypes, see how they work, and then backtrack and refine the program and the requirements [3, 4]. Also, backtracking becomes important in a situation where alternative solutions need to be managed for a given task. Several variation management tools have been developed [5, 6], but these are limited in that users cannot easily backtrack and add a new alternative from there if they did not plan ahead where they would need new alternatives.


Azurite – Interactive Selective Undo
Interactive Selective Undo
The preview will be updated as you select/deselect rectangles from the timeline.
Changes to be performed
MainFrame.java
MainFrame.java
Current Source
Preview of Selective Undo Result
Keep this code unchanged
Cancel
OK

```
ivate JPanel createButtons() {
  JPanel p = new JPanel();
  p.setLayout(new BoxLayout(p, BoxLay

  JButton buttonOrange = new JButton(
  buttonOrange.setBackground(Color.or
                                  "Butt
                                  Butt
```

```
  GridBagConstraints c = new Grid
  c.fill = GridBagConstraints.HOR
  c.gridx = 0;
  c.gridy = 0;

  JButton button1 = new JButton("
  JButton button2 = new JButton("
  JButton button3 = new JButton("
```

Fig. 1. The interactive selective undo dialog of AZURITE. The user can mark some code in the left panel, and ask to "Keep this code unchanged". This can be repeated until the preview in the right panel matches what is desired.

However, backtracking support is limited in modern interactive development environments (IDEs). The restricted linear undo model [7], which is used in most text and code editors, is not suitable for all situations. The most significant problem is that users can only undo the most recent edits. This can be very inconvenient when users realize that they made a mistake after making some other changes that they want to retain. In addition, programmers may intentionally make changes to the code that they later want to remove. Another option is to use a version control system (VCS) such as Subversion or Git to revert some code to a previous version. However, this approach relies on the assumption that the desired code is already committed to the VCS, which may not always be the case.

These problems can be resolved by having a *selective undo* feature in code editors. Users could select specific edit operations performed in the past and invoke the selective undo command to revert only the code affected by the those operations. Our study showed that 9.5% of all the backtrackings performed by the participants were selective, meaning that they could not have been handled by the conventional undo command [2]. Selective undo has been well researched in the area of graphical editors [7, 8, 9]. However, this technique has not been used with text or code editors due to the many text-specific challenges which we address here.

First, as Berlage pointed out, existing selective undo mechanisms are designed to work best when the system has identifiable objects that are affected by operations, but text does not have the notion of objects but rather has a stream of characters [7]. Second, there can be many regional conflicts among the edit operations. A *regional conflict* can occur when the region of a later edit overlaps the region of the earlier edit which the user wants to selectively undo. When there is a regional conflict among the edit operations, the result of a selective undo may not be well defined. To illustrate this point, consider the following example. An edit operation $e_1$ changes the code from "`myFontSize = 12;`" to "`myRectangleSize = 12;`" and sometime later, another operation $e_2$ changes it to "`myRegionArea = 12;`". This is an example of regional conflict because the affected ranges of the two operations are overlapping and the "`Rectangle`" text inserted by $e_1$ is only partially available in the current code. In this case, it is not clear what the result of selectively undoing operation $e_1$ alone should be. The system should be able to detect such cases and provide an appropriate approach to resolving them.

A final challenge of providing selective undo for code is that it is difficult to provide intuitive user interfaces for the user to *find* what to selective undo. Many existing selective undo user interfaces for graphics present a list of edit operations performed in the past along with human-readable descriptions of individual operations [7, 8, 9]. However, text editing operations are much more fine-grained than graphical editing, so it is hard for the users to interpret the high level edit intent just by looking at the individual text edits. In addition, graphical applications can use a thumbnail to represent a snapshot of the graphics at a certain point of time, which makes it easier to present the edit history to the user [6, 10, 11, 12, 13]. In contrast, a thumbnail of a piece of a large text file does not give much information to the users.

To resolve these problems and complement the existing tools, we devised a novel selective undo mechanism and applied it to the code editor. The system takes fine-grained code edits as input and maintains the mapping between the different segments of the current source file and the edit operations that introduced those segments. The system also keeps track of regional conflict relationships among edit operations (Section III). The system then uses this information to provide selective undo in code editors (Section IV).

Our tool is called AZURITE[1], and is an Eclipse plug-in that provides a new algorithm and new user interfaces for selective undo. We previously published an overview of the initial user interfaces of AZURITE [14], but the current paper is the first to present the selective undo mechanism capable of handling regional conflicts, and many new user interface features for selective undo based on user feedback from a preliminary field trial with the version described in the previous paper (Section V). We also present here the performance feasibility analysis (Section VI) and our new formal user study which shows that AZURITE is effective and usable (Section VII). We discuss the design space of selective undo tools and their trade-offs, and future work directions (Section VIII). Then we discuss the related work (Section IX) and conclude the paper (Section X).

[1] AZURITE is a blue mineral, and here stands for **A**dding **Z**est to **U**ndoing and **R**estoring **I**mproves **T**extual **E**xploration. The plug-in is available open-source for general use at: `http://www.cs.cmu.edu/~azurite/`.

## II. MOTIVATING EXAMPLE

Imagine a scenario where a programmer working on a graphical user interface (GUI) in Java Swing wants to implement a simple panel with three vertically arranged buttons, as shown in Fig. 2. There should be a fixed amount of padding inside the entire panel and between the buttons. First, she starts out with having a stub method that returns an empty panel. She then makes the following changes in order.

Button1
Button2
Button3

Fig. 2. A sketch of the desired UI

1. She creates three button objects and adds them to the panel (see Fig. 3a).
2. Running the application shows horizontally laid out buttons, so she looks for some layout manager to use. She tries `GridBagLayout` (Fig. 3b).
3. The intermediate code seems too complicated for just a simple vertical layout. She looks for a simpler layout manager, and discovers `BoxLayout`. She uses undo command multiple times to get rid of all the `GridBagLayout` code (*backtracking* to Fig. 3a).
4. She writes some code with `BoxLayout`, resulting in much simpler code and vertically laid out buttons (Fig. 3c).
5. She changes some properties of the buttons, such as the background color and button text (Fig. 3d).

She now tries to polish the layout and add some spacing between the buttons before moving further. However, she realizes that `BoxLayout` does not support spacing while `GridBagLayout` does. Therefore, she wants to restore the `GridBagLayout` code she wrote in step 2, while keeping the changes from step 5.

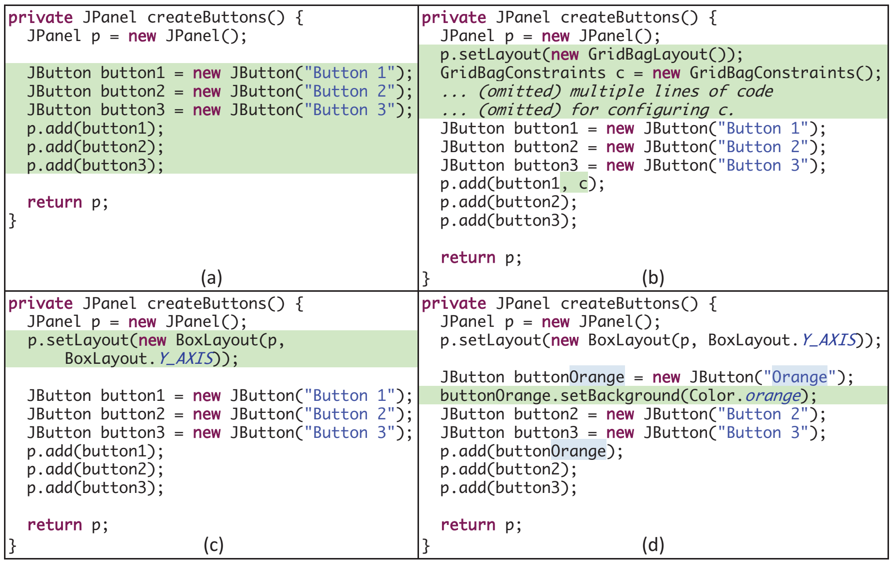
At this point, the undo command cannot be used because she had previously used the undo command to remove the `GridBagLayout` code and then she made some *new* changes from there, so the needed operations have been eliminated from the undo stack. Even if she had not used the undo command in step 3, the undo command would still be inappropriate for this situation, because it will necessarily revert the changes made in step 5, which is not desired. Moreover, it would be very unlikely that the `GridBagLayout` code had been committed to a version control system, because the code was still an intermediate state. The only option she now has is to reproduce the `GridBagLayout` code from scratch, which is inefficient. It would be much more convenient for her if there was at least a semi-automatic way of restoring the desired code from Fig. 3b while keeping the subsequent desired edits from Fig. 3d.

To address these problems, we developed a tool called AZURITE, which allows programmers to *selectively undo* fine-grained code changes made in the code editor. In this example, the programmer with AZURITE can restore the deleted `GridBagLayout` code without losing the changes related to `buttonOrange`, with the following steps:

1. Find the point in time in the past where the text "`GridBagLayout`" existed in the `createButtons` method using AZURITE's history search.

2. Select all the edit operations within the `createButtons` method performed since the point found in step 1.
3. Launch the interactive selective undo dialog (Fig. 1). Then, from the left panel, indicate the parts of the current code that should be kept unchanged.
4. After checking the preview of the selective undo result shown in the right panel, press the OK button to actually perform the selective undo.

AZURITE provides a rich set of user interfaces designed to help users complete various backtracking tasks. The list of steps described above is just one example, and there are several different ways to achieve the same result using AZURITE, as will be described below. Users can use AZURITE in the way that they feel the most comfortable.



(a)
```
private JPanel createButtons() {
  JPanel p = new JPanel();

  JButton button1 = new JButton("Button 1");
  JButton button2 = new JButton("Button 2");
  JButton button3 = new JButton("Button 3");
  p.add(button1);
  p.add(button2);
  p.add(button3);

  return p;
}
```

(b)
```
private JPanel createButtons() {
  JPanel p = new JPanel();
  p.setLayout(new GridBagLayout());
  GridBagConstraints c = new GridBagConstraints();
  ... (omitted) multiple lines of code
  ... (omitted) for configuring c.
  JButton button1 = new JButton("Button 1");
  JButton button2 = new JButton("Button 2");
  JButton button3 = new JButton("Button 3");
  p.add(button1, c);
  p.add(button2);
  p.add(button3);

  return p;
}
```

(c)
```
private JPanel createButtons() {
  JPanel p = new JPanel();
  p.setLayout(new BoxLayout(p,
      BoxLayout.Y_AXIS));

  JButton button1 = new JButton("Button 1");
  JButton button2 = new JButton("Button 2");
  JButton button3 = new JButton("Button 3");
  p.add(button1);
  p.add(button2);
  p.add(button3);

  return p;
}
```

(d)
```
private JPanel createButtons() {
  JPanel p = new JPanel();
  p.setLayout(new BoxLayout(p, BoxLayout.Y_AXIS));

  JButton buttonOrange = new JButton("Orange");
  buttonOrange.setBackground(Color.orange);
  JButton button2 = new JButton("Button 2");
  JButton button3 = new JButton("Button 3");
  p.add(buttonOrange);
  p.add(button2);
  p.add(button3);

  return p;
}
```

Fig. 3. The code changes for the motivating example. The green highlight shows newly inserted lines, and the blue highlight shows updates to the existing code. The omitted code is partially shown in Fig. 1.

## III. EDIT HISTORY MANAGEMENT

In this section, we explain the technical details of the edit history management algorithm used for selective undo.

### *A. Definitions*

In text editing, there exist only a few types of primitive edit operations. Here, the following three operations are considered as primitive: *insert*, *delete*, and *replace*. By primitive operation, we mean that these operations are the basic undoable units in the system. A replace operation is a primitive operation even though it can be considered as a delete followed by an insert with the same offset, because a replace operation should be undoable as an atomic unit. All of the higher-level editing commands that change the text, such as find-and-replace and cut-and-paste, result in one or more of these primitive operations. Note that the history records *every* operation that happens, and new operations are always added to the end of the history. In particular, regular undo commands and our selective undo commands are also included in the history using these primitives. For example, the undo of a delete operation is added to the end of the history as an insert operation.

However, it is not enough to simply keep the edit history to provide the selective undo feature. When performing undo on some past operations, the system must be able to determine which locations *in the current state* correspond to the locations where the operations were originally performed. This is not trivial because the offsets change whenever some text is inserted or removed above the location in the file of the past edit. This problem has been identified by others (e.g., [15]) and arises from the requirement that code be stored as plaintext without embedded meta-information like bookmarks. Thus, before a selective undo can be performed, offsets of previous operations in the file may need to be adjusted dynamically. We will refer to these adjusted offsets as *dynamic offsets*.

The idea of dynamic offsets here is similar to the notion of *dynamic pointers* introduced by Abowd and Dix in the collaborative editing context [16]. Our approach differs in that we keep track of dynamic *segments* instead of pointers. We introduced our dynamic segment management technique at an abstract level in our previous paper [14], but here we provide full technical details.

A dynamic segment is defined as a 4-tuple $(k, o, \tau, r)$, where the value of $k$ is either `ins` or `del`, $o$ is the dynamic offset of this segment, $\tau$ is the deleted or inserted text, and $r$ is a relative offset value (can be `nil`) which will be explained in Section III.D. There are two types of dynamic segments: *insert segments* ($k =$ `ins`) and *delete segments* ($k =$ `del`) – a replace will use both.

Let $\Omega$ be the set of all possible primitive edit operations. An edit operation $e \in \Omega$ can then be defined as a 3-tuple $(t, d, I)$, where $t$ is the time the operation was performed, $d$ is the delete segment associated with the edit operation (can be `nil`), and $I$ is the set of insert segments (can be `nil`). Each dynamic segment is always associated with an edit operation. The segment information is updated whenever a new operation is added to the history. $I$ is a set rather than a single insert segment, because an insert segment can be split by some later performed operation, as will be explained below in Section III.C.

Finally, the edit history is then defined as a chronological history list of edit operations, $H \coloneqq e_1 e_2 \dots e_N$, where $N$ is the number of all edit operations performed so far, and $e_N$ is the last (newest) edit operation.

### *B. Regional Conflicts of Edit Operations*

A selective undo operation may not be well defined when there are *regional conflicts* among the edit operations. Recall the example from Section I, where a replace operation $e_1$ changes the code from "`myFontSize = 12;`" to "`myRectangleSize = 12;`" and at some time later, another operation $e_2$ changes it to "`myRegionArea = 12;`". What should be the re-

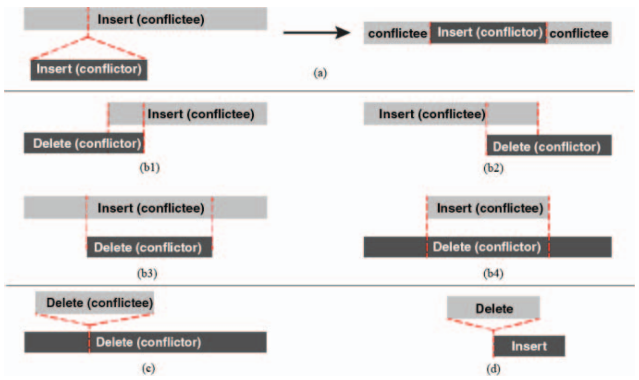


Fig. 4. Types of regional conflicts illustrated

sult of selectively undoing operation $e_1$ alone? There are multiple options:

**A1:** Selectively undo $e_1$ while leaving in the parts changed by $e_2$, resulting in "`myFontgionArea = 12;`"

**A2:** Also undo the conflicting operation $e_2$ and revert the code to the way it was before operation $e_1$ was performed, resulting in "`myFontSize = 12;`"

**A3:** Tell the user that there is a regional conflict and do not allow the selective undo, so the code stays as "`myRegionArea = 12;`"

Unfortunately, since it is not entirely clear what the user wants, it would not be appropriate for the system to choose arbitrarily without user intervention. Throughout the paper, we will use the notation $e_i \rightarrow e_j$ to represent a regional conflict where the edit region of $e_j = (t_j, d_j, I_j)$ overlaps with the region of $e_i = (t_i, d_i, I_i)$ at least partially, and $t_i < t_j$. For example, $e_1 \rightarrow e_2$ holds in the example above. Note that these two operations need not necessarily be consecutive in time, and there may be arbitrarily many edit operations in between.

As the arrow between the two operations implies, a regional conflict is always one-directional: the arrow starts from some earlier performed operation and points to a later performed operation. For convenience, when there is $e_i \rightarrow e_j$ conflict, we will call $e_i$ the *conflictee*, and $e_j$ the *conflictor*.

## C. Types of Regional Conflicts

Regional conflict detection happens at the segment level. Since there are only two types of dynamic segments, insert and delete, we get a total of four combinations of possible regional conflicts (Fig. 4a-d).

*1) $Insert \rightarrow Insert$ Conflict:* This occurs when the second insertion is performed somewhere in the middle of an existing insert segment (Fig. 4a). When the user tries to selectively undo the first insert operation, it is not clear whether the user really wants to remove only the text inserted by the first operation, or to remove them both together. When an $Insert \rightarrow Insert$ conflict is created (that is, when the user performs the conflictor edit), our algorithm splits the conflictee's insert segment into two pieces.

*2) $Insert \rightarrow Delete$ Conflict:* An $Insert \rightarrow Delete$ conflict occurs when the deletion range overlaps at least partially with an existing insert segment, which can happen in four different ways (Fig. 4b). Fortunately, there is no ambiguity when the user wants to selectively undo the conflictee; all the remaining text that came from that insert operation should be removed.

*3) $Delete \rightarrow Delete$ Conflict:* A $Delete \rightarrow Delete$ conflict occurs when the second deletion range encloses the first deletion range (Fig. 4c). Similar to the $Insert \rightarrow Insert$ case, when the user tries to selectively undo the first delete operation, it is not clear if the user really wants to restore only the text deleted by the first deletion, or restore the whole text deleted by both operations.

*4) $Delete \rightarrow Insert$ Conflict:* As shown in Fig. 4d, there can never be a $Delete \rightarrow Insert$ conflict because the range of deletion becomes a single point, and insert operations also only happen at a single point, and two points cannot conflict.

## D. Segment Closing / Reopening

When managing the dynamic segment information, a naïve implementation may lose some of the required information for selective undo. Suppose that there was originally "`xyz`" in the code at offset 10, and two delete operations $e_a$ and $e_b$ are performed, whose delete segments are $d_a = (\text{del}, o_a, \tau_a, r_a)$ and $d_b = (\text{del}, o_b, \tau_b, r_b)$, respectively. First, $e_a$ deletes "`y`" in the middle, resulting in $o_a = 11$. Next, $e_b$ deletes "`xz`", which results in $o_b = 10$ and a $Delete \rightarrow Delete$ conflict. In addition, $o_a$ is adjusted to 10, so that selectively undoing just $e_a$ would put "`y`" at the correct offset. What would happen if we undo both $e_a$ and $e_b$ at this point?

Since undo has to be performed backwards (to avoid messing up dynamic segments while performing the selective undo), $e_b$ would be undone first, and "`xz`" would be put back at offset 10. Next, $e_a$ would be undone and it would put back "`y`" at offset 10, which is the current value of $o_a$. The resulting code would look like "`yxz`" instead of "`xyz`", which is clearly not what the user wanted.

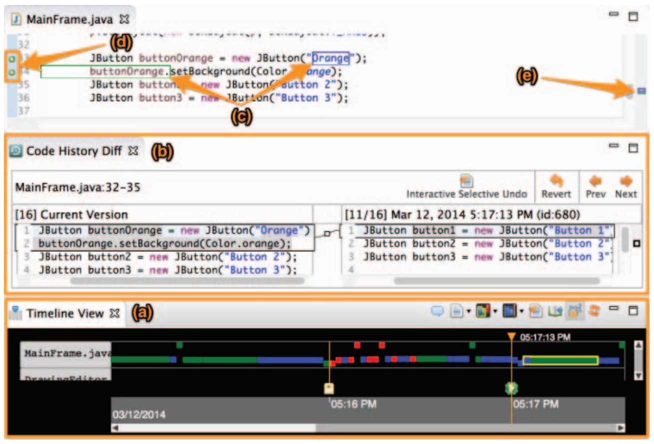
This happens because the dynamic segment for the conflictee ($e_a$) loses the offset information relative to the conflictor ($e_a$)'s offset when a $Delete \rightarrow Delete$ conflict occurs. To solve this problem, we need to (1) store this relative offset and (2) restore the offset when the conflictor is undone. We call these two processes *segment closing* and *segment reopening*, which work as follows:

**close**: $r_a \coloneqq o_a - o_b$ **reopen**: $o_a \coloneqq o_b + r_a$
$r_a \coloneqq nil$

In the previous example, closing $d_a$ will store $o_a - o_b = 11 - 10 = 1$ into $r_a$, right before $o_a$ is adjusted to 10. When undoing $d_b$, $d_a$ will be reopened, which will restore the offset $o_a$ to $o_b + r_a = 10 + 1 = 11$ and make $r_a$ be `nil`. Once a segment is closed ($r \neq nil$), then its $r$ value is never updated unless the segment is reopened. The same applies to $Insert \rightarrow Delete$ conflicts.

## E. Time Complexity of Edit History Management

Each new operation can add at most 4 dynamic segments: 1 delete segment, 1 insert segment, and up to 2 more segments (Fig. 4b3) if it happens to split an existing insert segment. Since the system iterates through all the segments in the history and



```
JButton buttonOrange = new JButton("Orange");
buttonOrange.setBackground(Color.orange);
JButton button2 = new JButton("Button 2");
JButton button3 = new JButton("Button 3");
```

```
JButton buttonOrange = new JButton("Orange")
buttonOrange.setBackground(Color.orange);
JButton button2 = new JButton("Button 2");
JButton button3 = new JButton("Button 3");
```

```
JButton button1 = new JButton("Button 1"
JButton button2 = new JButton("Button 2"
JButton button3 = new JButton("Button 3"
```

Fig. 5. The updated timeline visualization (a) and the code history diff view (b) of AZURITE. The code corresponding to the selected rectangles (yellow-outlined) in the timeline are highlighted in the editor by the boxes (c), the small icons on the left ruler (d), and the markers on the scrollbar on the right side (e). The colors of the boxes match the rectangle colors in the timeline.

performs constant time update work for each segment whenever a new edit is performed, the worst-case time complexity of updating the entire history is $O(N)$, where $N$ is the total number of edit operations in the current history $H$ (this potential performance issue is discussed in Section VI.B).

## IV. Selective Undo Mechanism

Provided that the system has all the dynamic segments information for the entire history, it can perform any selective undo. The user selects some edit operations (see Section V), and invokes the selective undo command. Our selective undo is performed in two phases: *determining code chunks*, and *performing selective undo for each chunk*.

### A. Phase #1: Determining Code Chunks

Unlike the conventional undo, our selective undo allows the user to select *multiple* operations to be undone together. Since there is no guarantee that all the selected operations were performed at the same place, the selective undo mechanism must first find the different *code chunks* affected by the selected operations. A code chunk consists of one or more operations and their dynamic segments, which constitute a continuous area in code. We use three rules to determine the code chunks: (1) all the segments associated with a same operation must be added to the same chunk, (2) all the segments located between two segments in a same chunk must also be added to that chunk, (3) all abutting chunks are merged into a single chunk.

### B. Phase #2: Selective Undo for Each Chunk

Once the chunks are determined, selective undo is performed for each chunk independently. For convenience, this is done from the bottommost chunk within the file so as not to affect the dynamic offsets of the segments of the other chunks while performing selective undo. Selective undo is performed in two different ways, depending on the existence of regional conflicts outside of the chunk, which is defined as following:

$$rcExists = \exists e_i, e_j \in \Omega \text{ such that } e_i \in \Sigma \wedge e_j \notin \Sigma \ \wedge e_i \rightarrow e_j,$$

where $\Omega$ is the set of all edit operations, and $\Sigma \subseteq \Omega$ is the set of all the selected operations in the chunk. The point is that regional conflicts can be automatically resolved if the conflictor is also selected ($e_j \in \Sigma$), while user intervention is required when the conflictor is *not* selected ($e_j \notin \Sigma$).

*1) $If\, rcExists = false$:* In this case, selective undo can always be performed without user intervention by applying the inverse changes of all the operations involved in this chunk, working backwards from the most recent operation to the oldest one. The inverse changes should be applied to each dynamic segment when there are multiple segments associated with an operation. When applying the inverse change of a delete segment, all the segments closed by this segment should be reopened (see Section III.D).

*2) $If\, rcExists = true$:* When there are conflicts, the users should be provided with the different alternatives of possible resulting code so that they can choose one of them or cancel the selective undo. Using the font size example from Section III.B, all three alternatives, A1, A2, and A3, should be provided to the user. A3 is simply the same code as it is, so the system only needs to calculate A1 and A2.

A1 is obtained by selectively undoing the conflictees while leaving in the parts changed by the conflictors. It turns out that A1 can be obtained by applying the same algorithm for conflict-free chunks used when rcExists=false without extra work.

In order to get A2, the chunk needs to be *expanded* to include its conflicting operations. However, there might be other operations conflicting with the ones just included in the chunk. For instance, in our example, there could be another later operation $e_3$ which changes the code from "`myRegionArea = 12;`" to "`myRegionBreadth = 12;`", which conflicts with operation $e_2$ but not with $e_1$. Currently, AZURITE includes all of these transitive conflictors to get the expanded chunk[2]. Once the expanded chunk is obtained, A2 is calculated by applying the selective undo algorithm as for non-conflicting chunks.

## V. User Interface Design

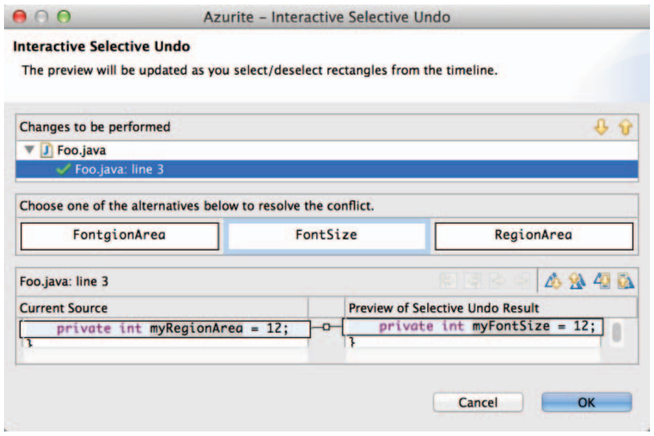
The selective undo algorithm described in Sections III & IV requires that the user select the correct set of edit operations to be undone. However, from the user's point of view, this can be a difficult task when the history gets bigger. There needs to be intuitive ways for users to express what they want to revert and/or what they want to get as the result of the selective undo.

### A. Initial Design

We first summarize the initial user interfaces of AZURITE designed to work with our selective undo mechanism presented earlier [14], to provide enough context to understand our new user interfaces.

The timeline visualization lets users see and interact with the edit history (Fig. 5a shows the updated version of the timeline). The horizontal axis represents time, and the edit history of each file is shown in a corresponding row. Each edit opera-

[2] In edge cases such as if the user replaces an entire file with an older version, AZURITE extracts the diffs between the two versions and logs the diffs as separate operations to minimize the potential future regional conflicts.


Azurite – Interactive Selective Undo

Interactive Selective Undo

The preview will be updated as you select/deselect rectangles from the timeline.

Changes to be performed

- Foo.java
  - Foo.java: line 3

Choose one of the alternatives below to resolve the conflict.

FontgionArea | FontSize | RegionArea

Foo.java: line 3

| Current Source | Preview of Selective Undo Result |
|---|---|
| `private int myRegionArea = 12;` | `private int myFontSize = 12;` |
| `}` | `}` |

Fig. 6. The interactive selective undo dialog when there is a chunk with regional conflicts. The user can choose one of the provided options to resolve regional conflicts. Here, the second option (FontSize) is chosen by the user.

tion is represented as a rectangle, which are color-coded according to the type of edit: inserts are green, deletes are red, and replacements are blue. The user can click or drag to select one or more rectangles. Once some of the operations are selected, the user can invoke a popup context menu containing various commands including selective undo.

The code history "diff" view (Fig. 5b) is a code-compare view with two panels. Users can select an arbitrary region of code from the regular code editor, and launch this view to see how the selected region has evolved over time, or to revert the code to an earlier version. Users can drag the orange vertical marker in the timeline back and forth to move between different versions of the code snippet while this view is open.

To leverage the observation that programmers remember some characteristics of the code that they want to backtrack [1], AZURITE provides a *history search* feature where users can search backwards through time (*not* through the current code), meaning that even deleted code can be found. All of these search results are also highlighted in the timeline, and the user can further investigate the selection or invoke selective undo.

## B. Field Trial with the Initial Design

After implementing the initial user interfaces (Section V.A), we conducted a field trial by deploying AZURITE to the students in Masters of Software Engineering program at Carnegie Mellon. We asked them to try using AZURITE while working on their studio projects in summer 2013. We then interviewed two of the active users to get more detailed feedback.

They both described that it was difficult to determine what the resulting code would look like, just by looking at the selected rectangles. For this reason, one of them used the code history diff view instead, because it worked as a "preview" of the selective undo result for him. He mentioned that he used the code history diff view instead of using the regular undo command, because it was convenient to see a preview before undoing. He also mentioned that he often wanted to keep some comments and selectively undo only the nearby code because for him the comments were usually the high-level description of a certain algorithm and only the code may be wrong. Both users wanted a feature to "tag" a point in time in the timeline.

The following subsections present the new user interfaces which address these concerns.

## C. Regional Undo Shortcut

We have found that the most popular form of selective undo is reverting a specific region of code to an old version, which has been referred to as "regional undo" [17]. In the current version of AZURITE, users can select some region in the regular code editor and use a keyboard shortcut (Ctrl+R by default, ⌘R on a Mac) one or more times to perform selective undo on that region directly within the code editor.

## D. Improvements to the Timeline View

We made some significant improvements to the timeline visualization (see Fig. 5a). One of the common ways of backtracking is to go back to a certain point in the past when a specific event happened. As Beck says in his book, "it would be great if the programming environment helped me with this, working as a checkpoint for the code every time all of the tests run" [18]. To support this, AZURITE detects significant coding events and displays them in the timeline view. Currently, the displayed events include running JUnit tests that pass () or fail (), running the application under development (), saving files (), and version-control system related commands such as commit (). An event is displayed on the timeline as a vertical line with an icon representing that event at the bottom (Fig. 5a). Further event types can be trivially added in the future. Users can also "tag" the current or any previous point in time, which was one of the most requested features from the field trial. A tag also shows an icon ( , Fig. 5 at 5:16), which can be named (shown on mouse hover), or stay anonymous. Users can left-click on any icon to move the orange marker to that point to see how the code looked then. Right-clicking any of the icons shows a context menu providing useful commands such as "Undo All Files to This Point," which can be viewed as a lightweight, automatic versioning feature.

Another requested addition is that when one or more rectangles are selected in the timeline, the corresponding areas of code are highlighted in the code editor (Fig. 5c-e). This mitigates the users' reported problem that it can be difficult to mentally associate the rectangles with the code.

## E. Interactive Selective Undo

We created another new user interface called the *interactive selective undo* dialog (Fig. 1) in response to feedback from the field trial. The design was inspired by Eclipse's refactoring wizard, which shows a preview of all changes to be made before actually changing the code. Similar to a typical refactoring wizard, our interactive selective undo dialog shows a side-by-side "diff" view where the left panel shows the current code and the right panel shows the preview of the selective undo based on the currently selected rectangles in the timeline. On the top panel is the list of all the affected files.

The interactive selective undo dialog is modeless, and the rectangles can be added to and/or removed from the selection while the dialog is open, which immediately updates the preview result shown in the dialog. By allowing this, users need not worry about selecting the exact set of rectangles on their first attempt. Users can manipulate the selection until the preview shows the desired result. This dialog is also capable of dealing with regional conflicts. When there is a chunk which contains regional conflicts, the dialog shows a red X icon (⊗) beside the chunk, and the OK button is disabled temporarily. Once the user selects the chunk from the top panel, the dialog shows the three alternative options described in Section III.B so that users can choose which one they want. Once an option is chosen, the chunk is marked as resolved with a green check icon (✓, Fig. 6), and the OK button becomes enabled when the user resolves all existing regional conflicts. Note that in most use cases of AZURITE, regional conflicts will not come up since rectangles will be selected with the aid of AZURITE features such as history search and the code history diff view. The conflict-resolution interface is provided for the sake of soundness.

Additionally, users can select an arbitrary region of the code in the left panel, right-click to bring up the context menu, and select "Keep this code unchanged" (Fig. 1). Doing this searches for all operations affecting that selected region of code and excludes those operations from what will be undone. This feature provides a significant usability improvement compared to requiring users to select the exact set of operations, because users can easily get the desired results by roughly over-specifying the selected operations, and then marking all the code fragments desired to be in the resulting code.

## VI. Performance Feasibility

### A. Size of the Logs

AZURITE uses the fine-grained code change history generated by FLUORITE [19], which is a recording plug-in that captures all of the low-level editor commands and edit operations. FLUORITE keeps the history as log files, and AZURITE uses the log files when the user wants to bring an old history back into the timeline. The total size of FLUORITE log data from 21 programmers, containing 1,460 hours of active coding activities, collected for our previous study [2] was 377MB, which gives a log size growth rate of about 264.5KB/hr during active editing. Given the spacious hard drives used nowadays, we believe this will not be a critical issue for most users. Moreover, the same study [2] discovered that 99% of the backtracking instances are performed within 3 editing sessions, implying that purging old editing histories would be safe enough in most cases.

### B. Edit History Management Performance

As described in Section III.E, the edit history management algorithm gets slower as the history gets bigger, because its time complexity is $O(N)$, where $N$ is the number of all operations in the current history. We measured[3] the actual time it takes to add a new operation (which would usually contain one or more tokens, not just a single character) to the history under varying sizes of $N$, and it took 3.93ms when $N$ = 10,000, and 39.58ms when $N$ = 100,000. According to the log data we collected, 10,000 operations is approximate one week of coding work (avg. # of edits: 251.9/hour). This shows that the edit history management algorithm, even unoptimized, will work in practice without causing significant delay. Future work includes optimizing the edit history's scalability, for example by using a tree-based data structure with relative offsets.

[3] Measured on a PC running Windows 8 with a 2.60GHz CPU.

## VII. User Study

We conducted a small controlled study to evaluate the usability and usefulness of AZURITE, after implementing all the new user interfaces described in Sections V.C-E. We used a between-subjects design with two groups: AZURITE and Eclipse only (as the control). All the tasks were performed using the Eclipse 4.4 IDE on a 15" Macbook Pro machine running OS X 10.9. The study took about 1.5 to 2 hours for each participant.

### A. Participants

We recruited 12 programmers at Carnegie Mellon, 8 males and 4 females (median age=27), who were randomly assigned to either group. All participants reported that they had at least 3 years of programming experience in general (median = 5 yrs), and at least 2 years of Java experience (median = 4 yrs). No one had previously seen or used AZURITE before the study.

### B. Tasks

There were a total of 6 Java programming tasks, performed in the same order for all participants (Table I). A separate code base was provided for each task. For tasks 1 through 5, each task was composed of a series of steps: a number of normal (non-backtracking) programming steps described in Table I, followed by a backtracking step (which is always the last step) to revert the changes made in the *underlined step* while keeping the changes from the other steps. This was because we wanted to have the participants create the edit history themselves, and wanted to use backtracking situations that are not trivial to perform using regular undo. For example, for T1, participants were asked to revert the factorial method back to the `for loop` version after finishing step (3). For task 6, the participants were provided with an existing code edit history and then performed backtracking from there. This allowed us to test if users can effectively perform selective backtracking when the code changes are scattered in multiple files and locations, without needing to require the users to spend time creating a long and complex edit history.

Because it would be impossible to gather backtracking timing data if the participant becomes stuck in one of the non-backtracking steps, we set up a 4-minute limit for each step, and the experimenter helped the participant after the time limit. No help was provided for the backtracking steps from which the reported results are measured.

### C. Study Procedure

After obtaining the informed consent and demographic information, for both groups we alternated between one tutorial session and two programming tasks, resulting in a total of 3 tutorial sessions and 6 tasks. The tasks were designed in a way that they can be effectively performed by using the feature they learned from the previous tutorial session, but the participants were told that they were free to use any strategies.

In the tutorial sessions, the AZURITE group learned how to use the regional undo shortcut (Section V.C), the "Undo All

Files to This Point" feature from the timeline view (Section V.D), and the interactive selective undo dialog (Section V.E). The control group learned about the local history feature of Eclipse, which is a built-in feature that keeps a per-file local history of every saved version of the source code [20], and is the closest existing feature that can help with the tasks. In the three tutorial sessions, this group learned how to manually perform selective undo using the local history, how to replace the entire source code with one of the saved revisions, and how to perform selective undo in multiple files. All participants in both groups were familiar with and were able to use the regular linear undo as well. After each tutorial, the participants were given a written document explaining the feature they learned during the tutorial with screenshots, so that they could refer to it later.

### *D. Results*

All the participants in both groups successfully completed all the backtracking steps, so we compared the completion time of the backtracking step of each task. An independent-samples t-test was conducted to compare the backtracking completion time between the two groups. Over all tasks, the AZURITE group took significantly less time (mean=386.3s) to perform all the backtracking steps than the control group (mean=768.8s, $p < 0.01$), which is roughly twice as fast. Fig. 7 shows the average backtracking completion time for the individual tasks for the 6 participants in each group. For tasks 2, 4, and 6, the AZURITE group was significantly faster compared to the control group ($p < 0.05$). For T1, the AZURITE group was faster but this was not statistically significant (p=0.25), mainly because one participant in the AZURITE group mistyped the shortcut key twice, so he started over and took much longer (173s).

T3 & T4 were non-selective backtracking tasks, meaning that they could have been performed with multiple invocations of the regular undo command. One participant in the control group in fact used the regular undo command instead of using the local history feature. Still, the AZURITE group was generally faster because they could quickly skim through the timeline to find the last successful unit test run and then undo all files to that point with a single command. There was one participant in the AZURITE group who did not use AZURITE for T3 and took 167s reproducing the code manually, which heavily affected the mean value and made the average time difference statistically non-significant (p=0.16) for T3.

The AZURITE group did not perform better for T5 (p=0.65), and the completion time varied much more (sd=108.4) than the control group (sd=37.11). This task was meant to be completed using the interactive selective undo dialog, but the participants had to also manually edit the resulting code after using the dialog, due to the difference between the two layout managers they were using. Two participants quickly realized this and roughly performed selective undo with the dialog and manually edited the resulting code. One participant used the regional undo instead. The other 3 participants tried to complete the backtracking using only the interactive selective undo dialog, which resulted in much longer time because the current interactive selective undo dialog does not support manual editing within the dialog itself. On the other hand, the "compare view" of the Eclipse local history supports manual editing, and resulting code can be manually edited within the compare view. So this is a feature we will add to AZURITE in the future. T6 was also meant to be completed with the interactive selective undo dialog, but in this case everything could be done solely with the dialog, and thus the AZURITE group dramatically outperformed the control group (mean=128.5s v. 345.0s, $p < 0.02$).

After finishing all the tasks, we asked whether the tool they used was useful for them with a 5-point Likert scale. A Wilcoxon rank sum test showed that AZURITE was more useful (median=5) than the Eclipse local history (median=3.5, $p<0.05$).

### *E. Limitations*

In order to focus on whether the tool helps in various backtracking situations, we provided specific steps of tasks without the surrounding exploratory programming context. To prevent the tasks from being too artificial, we designed the tasks based on actual backtracking situations observed from previous studies [1, 2]. In addition, at the end of the study, we gave a 5-point Likert question asking if the given scenarios were plausible in that similar situations come up when they are programming. 10 out of 12 participants agreed or strongly agreed, but the other two participants did not (median=4). Another limitation is that the control group was not trained to use any VCS such as Git. This was primarily because the step-by-step nature of the task

TABLE I. SUMMARY OF TASKS

| Task | Individual Steps | SEL | MF |
|---|---|---|---|
| T1 | (1) Implement factorial method with a loop<br>*(2) Modify factorial method to use recursion*<br>(3) Make a few more independent changes | √ | |
| T2 | *(1) Delete some existing sorting code*<br>(2) Make a few more independent changes | √ | |
| T3 | (1) Implement a simple Number class (unit tests given)<br>*(2) Modify Number to be an immutable class* | | √ |
| T4 | (1) Implement a Stack with inheritance (unit tests given)<br>*(2) Modify Stack to use composition instead* | | |
| T5 | (1) Layout GUI controls using GridLayout<br>*(2) Change the layout manager to GridBagLayout*<br>(3) Add another GUI control | √ | |
| T6 | Remove all the debugging specific code (e.g., println in multiple places) while keeping the actual bug fix code. An edit history is provided to begin with. | √ | √ |

SEL: selective backtracking (cannot be done with regular undo)
MF: multiple files are involved
Task T5 is similar to the motivating example of Section II.

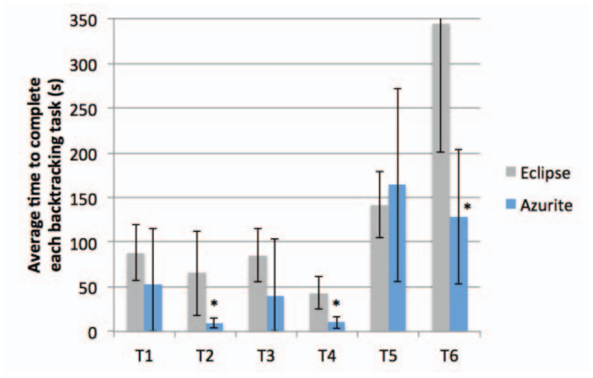


Fig. 7. The average backtracking completion time for each task. The error bars indicate the standard deviations. *differences are statistically significant.

instructions could bias the behavior of the participants towards commiting a version after each step, even when they do not usually make such small, frequent commits in their real environments. Carefully comparing AZURITE and VCSs on the backtracking tasks remains as future work.

The participants were all recruited from the local university, not from industry. However, given that our previous study found that programmers backtrack frequently [2], and 10 out of 12 participants had previous industry experience, we believe that AZURITE would be useful in real environments as well.

## VIII. DISCUSSION AND FUTURE WORK

### *A. Linear vs. Tree/Graph Based History*

We intentionally chose to use a linear history model instead of a tree or graph model as used in US&R [21] or Git. In our history model, all the code changes (including the undo commands themselves) are added to the end of the timeline for two reasons. First, although several text editors and plug-ins have provided tree-structured visualizations that allow users to move among different nodes [22, 23], it is difficult to understand the tree as the history gets bigger. This is because the nodes do not provide useful information for the user to navigate the tree, which is why we believe these have not caught on in popularity. In contrast, the linear history model and timeline visualization of AZURITE would match programmers' episodic memory [24] and have been shown to be understandable in our studies.

Second, it is not trivial to represent a selective undo operation in a tree-structured history. Unlike the regular undo, selectively undoing some changes does not result in one of the previously visited nodes in the history tree. Rather, it creates a *new* node that has never existed before. Also, a graph-based history as provided in Git would be inappropriate because a selective undo operation is not used for merging.

### *B. Granularity of Edit Operations*

Having different approaches for merging / dividing edit operations might affect the usability of a selective undo tool. On the one hand, if each character-level edit is logged individually, the tool could be less usable because all of the individual operations would have to be tediously identified and selected character by character. To mitigate this problem, AZURITE combines consecutive typing performed within 2 seconds (configurable), similar to the way typical text/code editors do. On the other hand, if the operations are too coarse-grained, then there would be many more regional conflicts among the operations. This would also give less control to the users, which was a real problem observed during the pilot for the user study. Based on our observation that users wanted to control the undo range at least at the line level, we decided to prevent a single operation from spanning across multiple lines. For example, when the user types multiple lines or pastes a large block, AZURITE divides the operation into multiple insert operations, each having one line of code. This approach worked very well during the actual user study.

### *C. Using Textual vs. Structural Changes as Input*

While there are development tools which use the abstract syntax tree (AST) level changes of the code as input [25, 26, 27], AZURITE uses the textual code changes instead. There are trade-offs between these two choices. On the one hand, by using textual changes as input, the mechanism becomes language agnostic, as is the conventional undo command and most commercially available merge tools [28]. Besides, there are certain types of edits that cannot be captured at the AST-level, such as reformatting code or changing a comment section. By using the textual input, our system can capture and undo these types of changes as well.

On the other hand, by using AST-level changes, the selective undo mechanism could use the additional information and better handle *semantic conflicts*. A semantic conflict can occur when a set of edit operations are semantically related to each other in the code. For example, when a method is renamed, its definition and all the call-sites must change together. Selectively undoing only one of these rename operations would not cause any *regional* conflicts because the changes were all made in totally different locations. However, this would result in inconsistent code containing compile errors and thus could be considered to result in *semantic* conflicts.

Since these edits are likely to have been performed close together in time, users may be able to easily select them together in the timeline. However, if not, we believe the compile errors would catch these kinds of problems and users would be able to perform the remaining backtracking steps to complete what they wanted to achieve. We observed this situation occurring during the user study, especially in the backtracking step of T5. In order to keep the added GUI control correctly, the users had to keep both the member field declaration and the code for creating the object and adding it to the panel. Some participants only kept the creation code and forgot to keep the declaration, resulting in a compilation error. All of them immediately realized their mistake by reading the error message and went to the field declaration location and performed selective undo there to restore the code. Even though we found these kinds of conflicts to be easy to detect and fix, we plan to investigate adding features to AZURITE to handle them directly by augmenting the structural information of the changes. In fact, as we showed in our previous study of backtracking [2], the textual changes can easily be transformed into AST-level changes with a parser, which means that we could use both the textual and the structural changes as input.

### *D. Scalability of AZURITE*

As it stands now, AZURITE has only been tested with visualizing less than a week of coding histories. We believe this is an appropriate level of scalability for its purpose, given that we found 97% of the programmers' backtracking is performed within the same editing session [2]. However, because there was a general consensus from the users that the rectangles displayed in the timeline are too fine-grained, we will enhance the timeline by applying semantic zooming and coalescing the edits for the same task or those related to a working set.

## IX. Related Work

### A. Selective Undo in Other Contexts

Selective undo has been extensively studied for graphical, interactive editors. Berlage introduced the selective undo model implemented in GINA, which adds the reverse operation of the selected command to the current context [7]. The Amulet [8] and Topaz [9] systems had a similar selective undo feature, but these allowed repeating a selected command even on a new object. Selective undo was applied in spreadsheets [29] by allowing users to select a region in the spreadsheet and perform regional undo. Dwell-and-spring [30] is a recent selective undo mechanism for direct manipulation. It provides an interface for undoing any press-drag-release interaction. All of these approaches assume that there is an object on which the operations can be performed: primitive graphical objects such as shapes in graphical editors, and individual cells in spreadsheets. In contrast, there is no clear notion of objects in text editors, since edit operations typically affect ranges of text, and the text itself moves around and is changed.

There are other undo models that support selective undo by providing additional commands beyond undo and redo. The US&R model [21] allows users to *skip* redoing an operation, using a complicated tree-based data structure. Users can selectively undo an isolated operation, by undoing multiple steps until the target operation gets undone, skipping the redo command once, and then redoing the rest of the operations. The triadic model [31] uses a simpler structure composed of a linear history list, and a circular redo list which can be *rotated* by users. Undoing an operation puts the operation at the beginning of the redo list, and rotating the redo list takes one operation at the beginning of the list and puts it at the end. Since the rotate command can be used to skip a redo command, users can selectively undo a certain operation in a similar way. However, both models require deep understanding of the underlying history structure to correctly perform selective undo. In addition, selective undo cannot be done in one step, which can be cumbersome for users.

### B. Regional Undo in Text Editors

Some text editors such as Emacs and DistEdit [32] support regional undo, where the user undoes the most recent operation that affected a specific selected region of text. Regional undo is useful and also relatively easy to implement compared to the generic selective undo, because it always undoes the most recent operation performed in the selected region, which guarantees that there are no regional conflicts with the target operation (see Section III.B). Regional undo is directly supported in AZURITE using the keyboard shortcut, by searching for all edits for the region of code and invoking selective undo on the last one, or by using code history diff view and using the revert button. In regional undo, however, there can be an ambiguity if the user selects a region which partially overlaps with an operation's effective region. Li and Li refer to this problem as region overlapping, and introduce the idea of *partial undo* as a solution, which undoes only overlapped part of the operation when an operation partly falls in the given undo region [17]. In this situation, AZURITE would do the same when using the code history diff view or the regional undo shortcut to revert a certain region of code to one of the previous versions.

### C. Variation Management Tools

Version control systems (VCSs) can be seen as temporal variation management systems. However, there are many cases where a VCS cannot directly help with backtracking, as discussed in Section I. Git provides features related to selective undo, such as reverting a whole commit, and selectively committing local code changes. However, selectively reverting changes from an existing commit is a very involved process. Moreover, these features cannot be used for restoring deleted code that is neither in the commit history nor in the local code. Most IDEs support automatic local history keeping features [33, 34, 35, 36], but they are limited in that (1) the history is shown in a linear list without any human-readable descriptions or cues, (2) history search is not supported, and (3) selective undo is not directly supported, so users must compare the local and the desired older versions to merge the wanted changes manually. Similar to IDEs, cloud-based text editors such as Google Docs [37] support linear revision history with the same limitations.

There are a few other systems such as Juxtapose [5] and Parallel Pies [6], which facilitate design exploration by providing ways of adding alternatives at any time and moving among the alternatives. However, users must know in advance when they want to add variations in Juxtapose, and Parallel Pies works only in the graphical editing context.

Other work has studied ways to manage source code variations. Choice calculus provides a generalized representation for software variations at the source code level and provides theoretical foundations of variation management [38]. However, the Choice calculus cannot handle regional conflicts. Barista [39] had an alternative expressions tool which allows selecting an alternative by clicking on one of the listed choices, but it was restricted to the expression level.

## X. Conclusion

Although selective undo can be a powerful way to backtrack, this idea has not previously been implemented in any popular code editors, due to the challenges of selective undo for text editing. AZURITE overcomes these challenges and provides a practical and usable system, which our formal user study suggests can make programmers more effective and efficient by allowing them to backtrack more easily. We expect that the tool would give programmers more confidence while exploring, because they can back out of incorrect edits at any time. AZURITE is available for general use as an Eclipse plugin for Java, and we invite your feedback: www.cs.cmu.edu/~azurite/.

## Acknowledgments

We thank Sebon Koo for helping us implement the early version of the timeline, and the participants of our studies. Funding for this research comes in part from the Korea Foundation for Advanced Studies (KFAS) and in part from NSF grants IIS-1116724 and IIS-1314356. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of KFAS or the National Science Foundation.

## REFERENCES

[1] Y. Yoon and B. A. Myers, "An Exploratory Study of Backtracking Strategies Used by Developers," *Proc. International Workshop on Cooperative and Human Aspects of Software Engineering* (CHASE 2012), 2012, pp. 138-144.

[2] Y. Yoon and B. A. Myers, "A Longitudinal Study of Programmers' Backtracking," *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing* (VL/HCC 2014), 2014, pp. 101-108.

[3] D. W. Sandberg, "Smalltalk and Exploratory Programming," *SIGPLAN Notices,* vol. 23, 1988, pp. 85-92.

[4] J. Sametinger and A. Stritzinger, "Exploratory Software Development with Class Libraries," *Proc. Joint Conference of the Austrian Computer Society*, 1992.

[5] B. Hartmann, L. Yu, A. Allison, Y. Yang, and S. R. Klemmer, "Design as Exploration: Creating Interface Alternatives through Parallel Authoring and Runtime Tuning," *Proc. ACM Symposium on User Interface Software and Technology* (UIST 2008), 2008, pp. 91-100.

[6] M. Terry, E. D. Mynatt, K. Nakakoji, and Y. Yamamoto, "Variation in Element and Action: Supporting Simultaneous Development of Alternative Solutions," *Proc. SIGCHI Conference on Human Factors in Computing Systems* (CHI 2004), 2004, pp. 711-718.

[7] T. Berlage, "A Selective Undo Mechanism for Graphical User Interfaces Based on Command Objects," *ACM Transactions on Computer-Human Interaction,* vol. 1, 1994, pp. 269-294.

[8] B. A. Myers and D. S. Kosbie, "Reusable Hierarchical Command Objects," *Proc. SIGCHI Conference on Human Factors in Computing Systems* (CHI 1996), 1996, pp. 260-267.

[9] B. A. Myers, "Scripting Graphical Applications by Demonstration," *Proc. SIGCHI Conference on Human Factors in Computing Systems* (CHI 1998), 1998, pp. 534-541.

[10] D. Kurlander and S. Feiner, "Editable Graphical Histories," *Proc. 1988 IEEE Workshop on Visual Languages*, 1988, pp. 127-134.

[11] S. R. Klemmer, M. Thomsen, E. Phelps-Goodman, R. Lee, and J. A. Landay, "Where Do Web Sites Come From?: Capturing and Interacting with Design History," *Proc. Conference on Human Factors in Computing Systems* (CHI 2002), 2002, pp. 1-8.

[12] D. Kurlander and S. Feiner, "A Visual Language for Browsing, Undoing, and Redoing Graphical Interface Commands," *Visual languages and visual programming,* 1990, pp. 257-275.

[13] M. Chii, M. Yasue, A. Imamiya, and M. Xiaoyang, "Visualizing Histories for Selective Undo and Redo," *Proc. 3rd Asia Pacific Computer Human Interaction*, 1998, pp. 459-464.

[14] Y. Yoon, B. A. Myers, and S. Koo, "Visualization of Fine-Grained Code Change History," *Proc. IEEE Symposium on Visual Languages and Human-Centric Computing* (VL/HCC 2013), 2013, pp. 119-126.

[15] S. P. Reiss, "Tracking Source Locations," *Proc. International Conf. on Software Engineering* (ICSE 2008), 2008, pp. 11-20.

[16] G. D. Abowd and A. J. Dix, "Giving Undo Attention," *Interacting with Computers,* vol. 4, 1992, pp. 317-342.

[17] R. Li and D. Li, "A Regional Undo Mechanism for Text Editing," *Proc. International Workshop on Collaborative Editing Systems* (IWCES 2003), 2003.

[18] K. Beck, *Test-Driven Development: By Example*, Addison-Wesley Professional, 2002.

[19] Y. Yoon and B. A. Myers, "Capturing and Analyzing Low-Level Events from the Code Editor," *Proc. ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools* (PLATEAU 2011), 2011, pp. 25-30.

[20] "Eclipse Local History," http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2FgettingStarted%2Fqs-55.htm.

[21] J. S. Vitter, "US&R: A New Framework for Redoing (Extended Abstract)," *SIGSOFT Software Engineering Notes,* vol. 9, 1984, pp. 168-176.

[22] S. Losh, "Gundo - Visualize your Vim Undo Tree," 2012; http://sjl.bitbucket.org/gundo.vim/.

[23] T. Cubitt, "undo-tree.el version 0.3.1 --- Treat undo history as a tree," 2010; http://www.dr-qubit.org/emacs.php.

[24] C. Parnin and S. Rugaber, "Programmer Information Needs after Memory Failure," *Proc. IEEE International Conference on Program Comprehension* (ICPC 2012), 2012, pp. 123-132.

[25] R. Robbes and M. Lanza, "A Change-based Approach to Software Evolution," *Electronic Notes in Theoretical Computer Science,* vol. 166, 2007, pp. 93-109.

[26] T. Omori and K. Maruyama, "A change-aware development environment by recording editing operations of source code," *Proc. international working conference on Mining software repositories* (MSR 2008), 2008, pp. 31-34.

[27] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining fine-grained code changes to detect unknown change patterns," *Proc. International Conference on Software Engineering* (ICSE 2014), 2014, pp. 803-813.

[28] T. Mens, "A State-of-the-Art Survey on Software Merging," *IEEE Transactions on Software Engineering,* vol. 28, 2002, pp. 449-462.

[29] Y. Kawasaki and T. Igarashi, "Regional Undo for Spreadsheets," *Proc. ACM Symposium on User Interface Software and Technology* (UIST 2004), 2004.

[30] C. Appert, O. Chapuis, and E. Pietriga, "Dwell-and-Spring: Undo for Direct Manipulation," *Proc. SIGCHI Conference on Human Factors in Computing Systems* (CHI 2012), 2012, pp. 1957-1966.

[31] Y. Yang, "Undo Support Models," *International Journal of Man-Machine Studies,* vol. 28, 1988, pp. 457-481.

[32] A. Prakash and M. J. Knister, "A Framework for Undoing Actions in Collaborative Systems," *ACM Transactions on Computer-Human Interaction,* vol. 1, 1994, pp. 295-330.

[33] Eclipse Foundation, "Eclipse - The Eclipse Foundation open source community website.," http://www.eclipse.org/.

[34] Oracle Corporation, "NetBeans IDE," http://netbeans.org/.

[35] M. Wilson-Thomas, "Auto History Extension in Visual Studio 2013," 2014; http://blogs.msdn.com/b/visualstudio/archive/2014/01/23/auto-history-extension-in-visual-studio-2013.aspx.

[36] Apple Inc., "Comparing Versions of a File with the Timeline," 2013; https://developer.apple.com/library/ios/recipes/xcode_help-versioneditor/VersionTimeline.html.

[37] Google Inc., "Google Docs," https://docs.google.com/.

[38] M. Erwig and E. Walkingshaw, "The Choice Calculus: A Representation for Software Variation," *ACM Transactions on Software Engineering and Methodology,* vol. 21, 2011.

[39] A. J. Ko and B. A. Myers, "Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors," *Proc. CHI 2006*, pp. 387-396.