Non-Linear Editing of Text-Based Screencasts

Jungkook Park[1]*, **Yeong Hoon Park**[2]*, **Alice Oh**[1]
[1]School of Computing, KAIST, Republic of Korea
[2]Department of Computer Science and Engineering, University of Minnesota, USA
pjknkda@kaist.ac.kr, park1799@umn.edu, alice.oh@kaist.edu

ABSTRACT
Screencasts, where recordings of a computer screen are broadcast to a large audience on the web, are becoming popular as an online educational tool. To provide rich interactions with the text within screencasts, there are emerging platforms that support text-based screencasts by recording every character insertion and deletion from the creator and reconstructing its playback on the viewer's screen. However, these platforms lack support for non-linear editing of screencasts, which involves manipulating a sequence of text editing operations. Since text editing operations are tightly coupled in sequence, modifying an arbitrary part of the sequence often creates ambiguity that yields multiple possible results that require user's choice for resolution. We present an editing tool with a non-linear editing algorithm for text-based screencasts. The tool allows users to edit any arbitrary part of a text-based screencast while preserving the overall consistency of the screencast. In an exploratory user study, all subjects successfully carried out a variety of screencast editing tasks using our prototype screencast editor.

Author Keywords
Screencast; Screencast editing; Non-linear editing; Operational transformation

INTRODUCTION
Instructional screencasts are increasingly widespread as an online educational tool for a variety of topics. Most screencasts are recorded in a video format because they are generally created using a screen recording software. However, a video is mainly a graphical, view-only medium, so it is difficult to provide rich interactions with the in-video contents such as dragging and copying text shown in the video. To provide text-based interactions with the text within a screencast, there are emerging online platforms that support text-based screencast for demonstrating terminal sessions [6] or programming tutorials [8, 7]. Instead of recording the screen as a video, these platforms capture text insertions and deletions at the character level, as well as cursor and selection changes and other relevant events from the creator's editing activities. This allows viewers to interact with the text at any moment in the screencast. For example, when viewing a programming tutorial screencast, a viewer may pause, edit and run the code to better understand the content, and then resume watching the screencast.

Despite the advantages of text-based screencasts, most of the currently available systems lack adequate support for creating such content compared to other media such as videos. In particular, little effort has been devoted to supporting *non-linear editing* of text-based screencasts. Thus it is very difficult to edit an arbitrary part of a text-based screencast, in contrast to videos whose non-linear editing can be easily done using any modern video-editing software.

Implementing a non-linear editing system for text-based screencasts is technically challenging. Unlike a video whose frames are independent of each other, each text editing operation of a text-based screencast is dependent on all of its prior operations. Hence, editing a part of the text-based screencast involves adjusting the offsets (numeric values indicating in which position the text is edited) of all subsequent text editing operations. Moreover, rewriting the text editing operations can introduce ambiguity in reconstructing the latter part of the screencast. While several studies previously introduced non-linear editing systems for document change history [2] or code change history [4, 1, 5], most are based on snapshots or line-based diff system, which cannot give users a character-level control over the text editing operations for the purpose of editing a screencast.

In this paper, we describe a non-linear editing algorithm for text-based screencasts. We illustrate how the ambiguity of text editing operations can occur during the non-linear editing process and how such ambiguity can be detected and resolved with user's resolution. To evaluate the efficacy of our proposed algorithm, we introduce a prototype of a text-based screencast editor that provides non-linear editing functionality using the proposed algorithm. Finally, we report the results of an exploratory user study with six subjects. Findings from the study include: (i) our screencast editor is flexible enough to support a diverse range of editing tasks, yet (ii) it is difficult to learn and get used to screencast editing due to the complexity of ambiguity resolution, and (iii) we identified potential opportu-

*Both authors contributed equally to this work.

UIST '18, October 14–17, 2018, Berlin, Germany
 ISBN 978-1-4503-5948-1/18/10…$15.00
DOI: https://doi.org/10.1145/3242587.3242654

nities for improving the design based on the findings from the exploratory user study.

The contributions of this paper are developing and presenting the following:

- A non-linear editing algorithm for text-based screencasts.
- A prototype editor that implements non-linear editing functionality for text-based screencasts.
- An exploratory study demonstrating that users can successfully edit a text-based screencast using our editor in various editing scenarios.

RELATED WORK

In this section, we situate our work in two related research fields, operational transformations and non-linear editing of text editing history.

Operational Transformation

The operational transformation (OT) [10] is an effective approach to record document changes at a fine granularity. OT was originally developed to support real-time collaborative text editing, but it can also be used to track document changes in virtue of its simple structure. While there are several different specifications of OT [13], we decided to adhere to the most simple specification available to make our non-linear editing algorithm simple and to work adaptively even when it is applied with other extended OT specifications.

Most previous OT research addresses the problem of maintaining the *syntactic consistency* of a shared document. In case of a conflict, any of the possible syntactically consistent resolutions is considered valid, without regard to *semantic correctness* or *user's intention*. Since an OT algorithm would not automatically decide which outcome 'makes sense' the most, the resulting text may not be semantically correct, or violate the user's intention [9, 11]. This problem of preserving semantic consistency is generally addressed in an application-specific manner to determine what makes sense for the particular context [12]. To the best of our knowledge, our approach is the first to show users all possible resolutions and explicitly let them make the decision for resolving OT conflicts in text-based screencasts.

Non-Linear Editing of Text Editing History

Timewarp [2] is a tool to support collaborative editing of a document. Timewarp investigates the idea of allowing users to interact with a document at any previous version of its history. A similar idea is adopted among the software developers using Git VCS, where a user can rewrite history through 'rebase' mechanism. It is a common practice among Git users to use rebase to change the order of commits, squash or flatten commits in order to better organize commit history [1, 3]. Furthermore, a system for demonstrating multi-stage code examples has been proposed by [4] using Git commit history. The system also allows users to edit intermediate stage of the code by rebase-like rewriting mechanism. However, since commit history based on Git stores text editing history in a line-level granularity, reconstructing a screencast from such history data has a limitation in the level of the expressiveness compared to a video. Also, these editing systems have a constraint on the part of history that can be edited to be merged without introducing content ambiguity. As far as we know, our proposed non-linear editing algorithm is the first algorithm to provide editing functionality of an arbitrary part of a text-based screencast recorded at the character-level.

NON-LINEAR EDITING ALGORITHM

In this section, we propose an algorithm for non-linear editing of text-based screencast. First, we introduce the symbolic notations we use in describing the screencast editing algorithm. Second, we define the *Ambiguous Positioning Problem* where a non-linear edit can result in some of the subsequent OTs with multiple possibilities for their positions. Third, we describe the mechanism for detecting and resolving the positional ambiguities during the editing process. Lastly, we explain the proposed non-linear editing algorithm and describe how the ambiguity can be detected and resolved by the user within the ambiguity resolution process. Our implementation is publicly accessible at https://github.com/elicast-research/non-linear-edit.

Notations

A text operational transformation (OT) is denoted as $\delta(s,e,t)$ and is interpreted as "overwrite current text from position s to position e with new text t". For example, OT $\delta(0,1,\text{“}h\text{”})$ changes text "cat" to "hat". This notation can represent text inserting or deleting operation if $s = e$ or $t =$ "" respectively. Also, when the current text (in the screencast) T is given, we can calculate the inverse of OT as $\delta^{-1}(s,e,t) = \delta(s, s + t.length, T.substring(s,e))$.

A text-based screencast is represented as a list of text OTs. For example, a playback of a screencast $[\delta(0,0,\text{“}a\text{”}), \delta(1,1,\text{“}bc\text{”}), \delta(0,2,\text{“}x\text{”})]$ begins with an empty string, then have 3 subsequent frames: "a", "abc", and "xc".

In practice, each OT also includes a timestamp information and other metadata, which we will omit in this paper to simplify the algorithm description.

Ambiguous Positioning Problem

Unlike non-linear editing of video, editing a part of the screencast without properly handling all of its subsequent OTs will result in unexpected screencast content. This is because each text editing operation of a text-based screencast is dependent on all of its prior operations. For instance, suppose we have a screencast with a list of OTs $[\delta(0,0,\text{“}Hi\text{”}), \delta(2,2,\text{“}World\text{”})]$ whose playback begins with an empty string, then the next frame shows "Hi" and the final frame shows "HiWorld". Let us assume we are trying to edit this screencast as if we wrote the word "Hello" instead of "Hi" in the beginning. We might want to replace the first OT to $\delta(0,0,\text{“}Hello\text{”})$. However, without transforming the subsequent OT $\delta(2,2,\text{“}World\text{”})$, the final content of the playback will become an unexpected text "HeWorldllo". To produce the intended text "HelloWorld", we would need to apply some transformation function to the subsequent OTs so that the original OT $\delta(2,2,\text{“}World\text{”})$ can be transformed into $\delta(5,5,\text{“}World\text{”})$.

However, calculating the amount of position shift of the subsequent OTs is often ambiguous; there can be multiple possible outcomes that the ambiguity cannot be resolved unless the

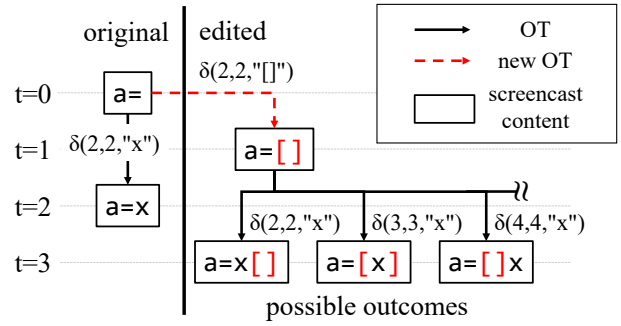

Figure 1. Transformation of the subsequent OT can vary depending on the inserted screencast content or the intention of screencast creator. When a OT $\delta(2,2,$"[]"$)$ is newly inserted before the existing OT $\delta(2,2,$"x"$)$, the existing OT can be transformed to 6 possible positions by Algorithm 2, resulting different content at the end of the edited screencast. This figure shows three examples among them.

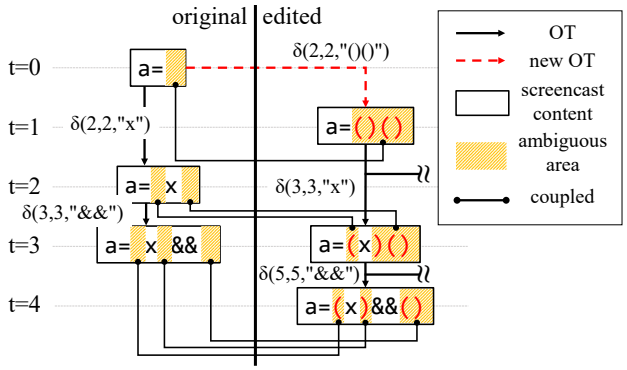

Figure 2. An example of coupled ambiguous area. The area is first created by the inserted OT $\delta(2,2,$"()()"$)$ and then updated by the subsequent OTs. The orange-color-filled areas represent ambiguous areas and the links between two ambiguous areas represent one to one mappings. In this figure, we only display one example among several possible transformations.

user's choice is given. Thus, unless the intention of screencast creator is known, transformed position of the subsequent OTs cannot be determined. Figure 1 shows an example where a subsequent OT can be transformed into one of the multiple possible outcomes when a new OT is inserted in a prior time frame. In the figure, the original screencast is "a=" (t=0) → "a=x" (t=2) and the user inserts new OT $\delta(2,2,$"[]"$)$ between the two operations. In this case, the newly inserted OT introduces an area $[2,2]$ which any OT overlaps with the area becomes ambiguous. There can be 6 possible transformations for this ambiguity (by Algorithm 2), and the figure shows three examples among them.

Ambiguity Detection & Resolution

We first define *ambiguous area*, a text range where, if an OT overlaps with the area, the transformation of the subsequent OTs result in ambiguous positions. In the example shown in Figure 1, any subsequent OT $\delta(a,b,t)$ whose $[a,b]$ includes 2 (the last position of text "a=") results in an ambiguous positioning problem. Thus, the ambiguous area of the example becomes $[2,2]$. More concretely, an ambiguous area $[a,b]$ is created when a new OT $\delta(a,b,t)$ is inserted. Extending the same rule, we can calculate the ambiguous areas given

Algorithm 1 An algorithm to get a coupled ambiguous area

```
1:  function GetCoupledAmbiguousArea(L_N)
      // Get a coupled ambiguous area introduced by OTs L_N
2:    A_before ← GetAmbiguousAreas(L_N)
3:    L_N^inv ← L_N.map(x → x^{-1}).reverse()
4:    A_after ← GetAmbiguousAreas(L_N^inv)
5:    C ← []
6:    while A_before ≠ ∅ do  // always |A_before| = |A_after|
7:      Pop leftmost interval a_before from A_before
8:      Pop leftmost interval a_after from A_after
9:      C.append((a_before → a_after))
10:   return C
11: function GetAmbiguousAreas(L_N)
      // Get ambiguous areas introduced by OTs L_N
12:   A ← {}
13:   for i ∈ {0,1,...,N−1} do
14:     a ← {x ∈ ℝ | L[i].s ≤ x ≤ L[i].e}
15:     for j ∈ {i−1, i−2, ..., 0} do
16:       a ← Γ(a, L[j].s, −L[j].t.length)
17:       a ← Γ(a, L[j].s, L[j].e − L[j].s)
18:     A ← A ∪ a
19:   return A
20: function Γ(a, p, d)
      // Transform area a at the position p with the amount d
21:   return {x ∈ ℝ | (x ≤ p ∧ x ∈ a) ∨ (p + d^+ ≤ x ∧ x − d ∈ a)}
```

a list of OTs by iteratively combining the ambiguous area created from each OT (**GetAmbiguousAreas** function in in Algorithm 1) However, when the current text T and a list OTs $L_N := [x_1, x_2, ..., x_N]$ is given, the ambiguous area of x_i $(1 < i \leq N)$ is not $[x_i.s, x_i.e]$ because the start/end positions of x_i are based on the intermediate text $T + \sum_{j=1}^{i-1} x_j$, not on the original text T. Therefore, the algorithm adjusts the range $[x_i.s, x_i.e]$ to be based on T by removing the range affected by x_j $(1 \leq j < i)$ iteratively in reverse order. For example, the ambiguous area of the second OT in $[\delta(0,1,$"xy"$), \delta(1,5,$"z"$)]$ is calculated by $\Gamma(\Gamma([1,5],0,-2),0,1) = \Gamma([0,3],0,1) = \{[0,0],[1,4]\}$.

If we detect any ambiguity from subsequent OTs using ambiguous areas, we then identify possible positions of the transformed OTs by using two *coupled ambiguous areas*; one that corresponds to the original playback of the screencast, the other that corresponds to the new version of the playback that is being rewritten. Each of the areas on both sides has one to one mapping. For example, in the case of Figure 2, the coupled ambiguous area at the point of resolving the first subsequent OT is $\{[2,2] \rightarrow [2,6]\}$ because any OT located at the position $[2,2]$ can be transformed anywhere in the position $[2,6]$. After the transformation of the first subsequent OT is done, the coupled ambiguous areas are updated as follows: original OT $\delta(2,2,$"x"$)$ divides original side of ambiguous area into $\{[2,2],[3,3]\}$ and transformed OT $\delta(3,3,$"x"$)$ divides rewritten side of ambiguous area into $\{[2,3],[4,7]\}$. At the same time, the mapping between the two sides is updated as $\{[2,2] \rightarrow [2,3],[3,3] \rightarrow [4,7]\}$. Because all characters affected by OTs in both sides are in the ambiguous areas, in

Algorithm 2 An algorithm to get possible transformations of OT given coupled ambiguous area

1: **function** GETPOSSIBLETRANSFORM(C, x)
   // Get all possible transformations of OT x given by the coupled ambiguous area C
2: $p_a \leftarrow a_{after}$ s.t. $(a_{before} \rightarrow a_{after}) \in C \wedge x.s \in a_{before}$
3: $p_b \leftarrow a_{after}$ s.t. $(a_{before} \rightarrow a_{after}) \in C \wedge x.e \in a_{before}$
4: if $p_a = \emptyset$, then $p_a \leftarrow [x.s + \Delta(C, x.s), x.s + \Delta(C, x.s)]$
5: if $p_b = \emptyset$, then $p_b \leftarrow [x.e + \Delta(C, x.e), x.e + \Delta(C, x.e)]$
6: $P_x \leftarrow []$
7: **for** $a', b' \in p_a \times p_b$ **do**
8: **if** $a' \leq b'$ **then**
9: $P_x.append(\delta(a', b', x.t))$
10: **return** P_x
11: **function** $\Delta(C, p)$
   // Get the amount of position shift at p by the coupled ambiguous area C
12: $d \leftarrow \sum_{\substack{(a_{before} \rightarrow a_{after}) \in C \\ \wedge a_{before} \leq p}} \left(a_{after}.length - a_{before}.length\right)$
13: **return** d

every transformation step, the characters that are not included in any ambiguous areas are identical for both sides.

The calculation of coupled ambiguous area is based on the symmetric property of OT; if ambiguous areas corresponds to the original screencast is calculated by L_N, then coupled ambiguous areas corresponds to the rewritten screencast is calculated by the symmetric inverse of L_N (**GetCoupledAmbiguousArea** function in Algorithm 1). After we get coupled ambiguous area C from a newly added screencast L_N, we can get possible transformations of an arbitrary OT when the OT is transformed by L_N as follows (**GetPossibleTransform** function in Algorithm 2):

- If an OT is located in an ambiguous area, then a transformed OT can be located in any position within the corresponding coupled ambiguous area (Line 2-3).
- If an OT is located in a non-ambiguous area, then a transformed OT has no ambiguity and preserve same relative position to the surrounding characters (Line 4-5).

Screencast Editing Algorithm

Once we calculate coupled ambiguous areas for a screencast, we can define a function that inserts a new screencast content into the middle of the original screencast. The insertion procedure can be outlined as follows (**InsertScreencast** function in Algorithm 3):

For the purpose of the algorithm description, let us consider a non-linear editing scenario where a user makes a new screencast recording L_N to be inserted at the k-th frame of the original screencast L_O.

1. The algorithm first makes a new screencast L by concatenating $L_O[0:k]$ and L_N. Since the new screencast L_N is recorded based on the text at k-th frame built by $L_O[0:k]$, concatenation of $L_O[0:k]$ and L_N results in a valid screencast. At this point, L does not yet contain the latter part of the original screencast $L_O[k:]$.
2. Calculate the coupled ambiguous area C for the new screencast L_N. At this point, $a_{before} \in C$ and $a_{after} \in C$ represent the ambiguous areas based on $L_O[0:k]$ and $L[0:k + L_N.length]$ respectively.
3. For each OT in the latter part of screencast, iteratively apply ambiguity detection and resolution:
   (a) Let i be the iteration index.
   (b) Get P – all possible transformations of OT $L_O[k+i]$ according to C.
   (c) If there is only one possible transformation $p := P[0]$, it means there is no ambiguity. However, if there are more than one possible transformations, the algorithm cannot determine the intended transformation of the screencast creator. Thus, the algorithm asks user to select one of the transformation among P. After user's selection, append the chosen transformation $p \in P$ to the incomplete screencast L.
   (d) Update C according to the resolution p. After the update, a_{before} and a_{after} represent ambiguous areas of L_N based on the text produced by $L_O[0:k+i+1]$ and $L[0:k+L_N.length+i+1]$ respectively.

Non-linear editing of screencast can be done by using the algorithm that inserts a screencast into the middle of another screencast (**ReplaceScreencast** function in Algorithm 3). With the insertion function, removing of arbitrary range of screencast

Algorithm 3 An algorithm to replace a part of screencast

1: **function** REPLACESCREENCAST(L_O, L_N, s, e)
   // Replace a part of screencast $L_O[s:e]$ with OTs L_N
2: $L_T^{inv} \leftarrow L_O[s:e].map(x \rightarrow x^{-1}).reverse()$
3: $L'_N \leftarrow L_T^{inv} \cdot L_N$
4: $L \leftarrow$ InsertScreencast(L_O, L'_N, e)
5: $L' \leftarrow L[0:s] \cdot L[2e-s:]$
6: **return** L'
7: **function** INSERTSCREENCAST(L_O, L_N, k)
   // Insert a new screencast L_N between two OTs $L_O[k-1]$ and $L_O[k]$
8: $L \leftarrow L_O[0:k] \cdot L_N$
9: $C \leftarrow$ GetCoupledAmbiguousArea(L_N)
10: **for** $i \in \{0, 1, ..., |L_O| - k - 1\}$ **do**
11: $x := L_O[k+i]$
12: $Y \leftarrow$ GetPossibleTransform(C, x)
13: **if** $|Y| > 1$ **then**
14: Ask user to choose one y from Y
15: **else**
16: $y \leftarrow Y[0]$ // No ambiguity
17: $L.append(y)$
18: **for** $(a_{before} \rightarrow a_{after}) \in C$ **do**
19: $a_{before} \leftarrow \Gamma(\Gamma(a_{before}, x.s, x.s - x.e), x.s, x.t.length)$
20: $a_{after} \leftarrow \Gamma(\Gamma(a_{after}, y.s, y.s - y.e), y.s, y.t.length)$
21: **return** L

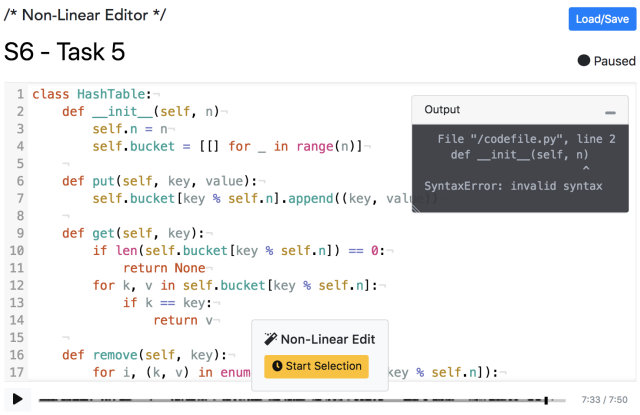

Figure 3. The interface of web-based screencast editor with non-linear editing functionality.

$(L_O[s:e])$ can be done by inserting the inverse of the OTs of the removing part in reversed order (L_T^{inv}) and then cut the OTs of $L_0[s:e] \cdot L_T^{inv}$ out after ambiguity resolution. Since L_T^{inv} is the inverse of $L_0[s:e]$, the coupled ambiguity area from $L_0[s:e] \cdot L_T^{inv}$ has no ambiguous area, thus the cutting operation can be done safely without introducing any ambiguity on subsequent OTs. Then, replacing function can also be defined as described in by combining removing existing part of screencast and inserting a newly added screencast.

INTERFACE

We built a web-based screencast editor (Figure 3) as a prototype tool where a user can record a text-based screencast and perform non-linear edits to the recorded screencast. This section describes the user interface of the prototype highlighting its key components that enable non-linear editing of text-based screencasts.

Interface Overview

We designed the editor to mainly support the recording of instructional programming demonstrations, so the editor is equipped with a code editor with Python 3 syntax highlighting. The code editor is used for both recording and editing a text-based screencast. Users can play, pause, and seek the screencast using the play/pause button and the timeline slider. The record button is only visible when a user reaches the end of the screencast; user can press the record button to append a new recording at the end of the screencast. While recording, all text editing operations in the editor are recorded along with a voice narration ingested from a microphone device. The "Start Selection" button located at the lower center starts a non-linear editing of the screencast.

Edit Range Selection

User starts editing of the screencast by clicking the "Start Selection" button. Then, the editor changes to the *selection* mode as shown in Figure 4. In the selection mode, user is asked to select a part of the screencast to be re-recorded. The start position of the selection is automatically set to the current thumb position when the user clicks the "Start Selection" button.

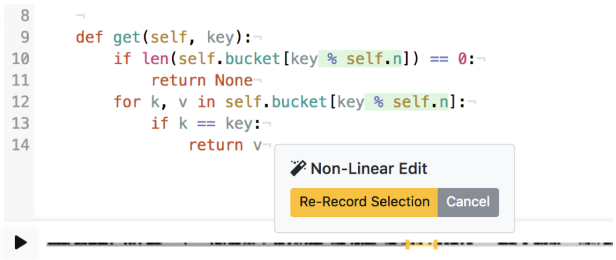

Figure 4. The interface for selecting a part of the screencast to be re-recorded. The selected range is marked in yellow on the timeline slider. The characters added in the selected part of the screencast are highlighted in green in the code editor.

User can set the end of the selection range by seeking the timeline slider. She can press the play/pause button and seek the timeline slider as usual, except she cannot move the thumb position left to the beginning of the selection. The selected range is displayed in yellow on the timeline slider and the characters added in the selected part of the screencast are highlighted in green in the code editor.

Re-Record Screencast

After selecting the part of the screencast to be re-recorded, user can start recording a new screencast by clicking the "Re-Record Selection" button. Then, recording starts immediately for a new screencast. Since the new screencast will overwrite the selected part of the screencast, the editor makes an initial state of the new screencast identical to the beginning state of the selected part of the screencast. Thus the user will receive the same experience when she appends a new screencast at the end of the existing screencast.

After recording, user clicks the pause button to stop the new recording, then the editor automatically replaces the selected part of the screencast with the new recording using the non-linear editing algorithm. If no ambiguity occurs, the interface returns to the default mode with the edited screencast; otherwise, it turns into the *ambiguity resolution* mode.

Ambiguity Resolution

If ambiguity is detected from an edit, the editor turns into the *ambiguity resolution* mode and the "Ambiguity Resolver" modal appears. The modal shows 4 code editors: the two editors at the top show the effect of the OT in the original screencast, and the two editors at the bottom show the expected effect of the OT in the currently editing screencast. To give a quick preview of the effect when the OT is applied, the removed text by the OT is highlighted in red and inserted text by the OT is highlighted in green. The user can explore the possible transformations of the OT by dragging 'from' and 'to' handle in the left bottom editor. Each of the handles can only be moved within the range calculated by the editing algorithm. After the user found the intended transformation, the user can click the "Resolve" button to resolve the ambiguity. The modal shows each of the ambiguities until all ambiguities are resolved, and then the interface returns to the default mode with the edited screencast.

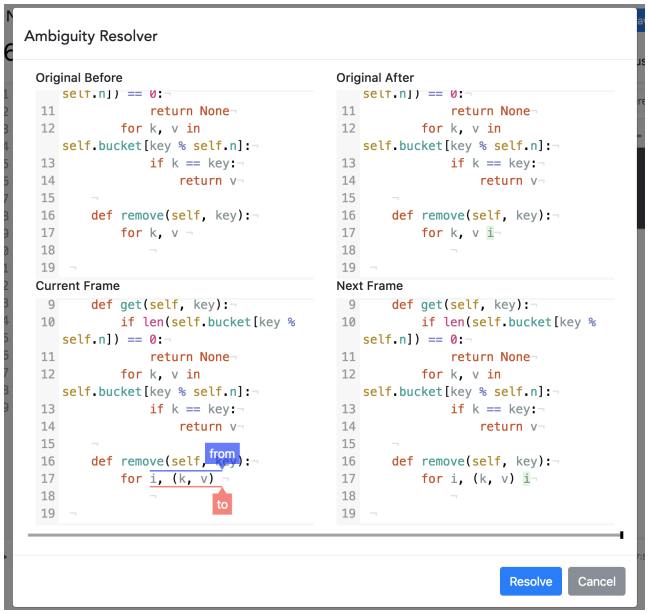

Figure 5. The interface of ambiguity resolver. For an OT that causes ambiguity, the resolver shows 4 code editors where each show the content of (top) the original screencast (bottom) the edited screencast (left) before the OT is applied (right) after the OT is applied. Users can choose one of the possible transformations of OT by dragging 'from' and 'to' handles within the editor in the lower left.

EVALUATION

We performed an exploratory study with six participants where each participant recorded a 10-minute screencast and edited the screencast following a given set of general editing tasks designed to be as close as possible to real-world use cases. To investigate the efficacy of our screencast editing algorithm and users' screencast editing patterns and behaviors, we designed the experiment to answer the following questions: (i) Can users successfully edit a screencast and get desired results in diverse usage scenarios? (ii) What kinds of different editing patterns emerge when carrying out different editing tasks? (iii) How difficult is it for a user to learn and get used to screencast editing features (i.e. timeline selection and ambiguity resolution)?

Procedure

We recruited 6 participants (4 graduate students and 2 recent graduates) who have served as teaching assistants for introductory computer science courses. At the beginning of each 1.5-hour session, we gave a 10-minute tutorial on how to record and edit a screencast using our tool. As part of the tutorial, we prepared 5 representative cases of editing the past recording of a screencast involving different usages of timeline selection and conflict resolution, which we demonstrated first then let the participants repeat as exercises.

Then, we asked the participants to complete 5 tasks in order, first of which is to create the initial recording of a screencast and the remaining 4 tasks to edit the recorded screencast in various ways. We encouraged the participants to think aloud when editing the screencast. The tasks are designed to resemble possible real world scenarios so that a range of natural screencast edit behaviors would be observed. The tasks are described as follows:

Task	# Edits		Median Time Spent for Ambiguity Resolution
	All	w/ Ambiguity (%)	
2	91	37 (40.7%)	18.56s
3	26	5 (19.2%)	47.79s
4	36	27 (75.0%)	21.04s
5	34	23 (67.6%)	14.79s

Table 1. The statistics of the participants' screencast edits for each task. Task 1 is excluded since it does not involve non-linear edits.

1. Record a 10-minute long screencast on how to implement a basic hash table in Python 3. Implement a hash table class with $put(key, value)$ and $get(key)$ methods without docstring, then write test cases.
2. Choose and edit 3 parts where your mistakes are recorded or any short span of the screencast that is not smooth or you want to re-record.
3. Edit the screencast such that it shows you implementing $remove(key)$ method after implementing *put* and *get* methods but before writing test cases.
4. Choose a variable or method argument that is referenced at least 3 times, and edit the screencast such that the variable (argument) has been defined and referenced in a different name from the beginning of its existence.
5. Edit the screencast such that it shows you writing docstring for each *put* and *get* method when you first begin implementing the methods in the screencast.

We recorded each participant's screen and voice as well as logging all screencast editing events with timestamps throughout the experiment. Upon completing all tasks, participants were asked to answer 4 post-study survey questionnaires about their perceived learnability and efficiency of the timeline selection and ambiguity resolution features, and we conducted a semi-structured interview for 15 minutes.

Results

All participants (S1-S6) completed all 5 tasks after spending 56 minutes on average; they could successfully record a screencast and edit different parts of the screencast. For Task 2 through Task 5, each participant made 31 screencast edits on average, and 49.7% (94/189) of the edits introduced an ambiguity that needed to be resolved explicitly by the participant. In such ambiguous cases, they had to make 1.46 decisions on average for choosing one of the possible transformations, and they spent median time of 19.1 seconds to complete the ambiguity resolutions for the edit (7.1 seconds for choosing each single transformation).

In order to compare the screencast editing patterns between different tasks, we analyzed the number of total screencast edits and the number of edits that caused ambiguity for each task, and the median time spent for doing ambiguity resolutions when an edit introduced ambiguity, as shown in Table 1.

We found that certain types of edits can have a significantly lower chance of introducing ambiguity than other types of edits. In particular, the screencast edits for Task 3, mostly performed with zero-length timeline selection and a few big chunk of insertions (*remove* method definition) surrounded by whitespaces, are not further edited in the subsequent part of the screencast. In contrast, the edits for Task 4 had the highest chance of creating ambiguity since variable declarations are almost always followed by nearby text changes.

Although all participants successfully completed the given editing tasks, they felt it is not easy to fully understand the process of ambiguity resolution. In the post-study survey, participants responded to the 5-point Likert survey questionnaires (1: Strongly Disagree, 5: Strongly Agree) that while it was easy to learn ($\mu = 4.50$, $\sigma = 0.84$) and efficiently use once they understood timeline selection feature ($\mu = 4.67$, $\sigma = 0.52$), ambiguity resolution was not easy to learn ($\mu = 2.17$, $\sigma = 1.17$) and they were not able to resolve ambiguities efficiently even after learning the feature ($\mu = 2.83$, $\sigma = 0.98$). Here we describe the key findings from their experience of using ambiguity resolution based on the analysis of the post-study interview.

Rich Context is Needed for Ambiguity Resolution

5 out of 6 participants mentioned that, before they pressed 'finish recording' button, they could not predict whether an ambiguity would occur, or at which point in the screencast an ambiguity would occur, because they could not recall every detail of the recorded screencast in high precision. S4 said *"I cannot be conscious of the whole changes from the beginning to end when I'm editing"*. Moreover, there were instances where participants completely forgot some of the changes they made while recording the screencast. S4 said: *"I think I forget about the word entirely when I delete it"*. S3 mentioned that he did not know the fact that he had a habit of *"unconsciously inserting characters and removing them while I'm thinking"*, so he could not recall that those changes were recorded at all.

Participants argued they needed to be given more context in order for them to recall what has happened earlier and resolve ambiguity more easily. In particular, 3 participants felt that viewing only the *Next Frame* is not enough for them to remember what other changes were made right after, as S2 mentioned: *"I don't know what comes next.. with this one character diff."*. Participants suggested that tool should be able to somehow visualize several contiguous changes that comes after, or all relevant changes in the subsequent part of screencast.

Different Strategies of Dealing with Ambiguities

Some participants described their thoughts on the complexity of ambiguity resolution. S6 said: *"I think the concept itself is confusing in the first place because the process is naturally unintuitive for human to understand, like RSA algorithm. I just keep forgetting how it works because it's so unnatural."* Also, to cope with the complex nature of ambiguity resolution, they developed different strategies of their own to simplify the work.

Some participants tried to avoid causing lots of cumbersome ambiguities as possible by *selecting a coarse-grained time range* when multiple text changes are made contiguously within single nearby location and re-record the whole part. For example, when S1 recorded several mistakes while writing a docstring for Task 5, instead of editing the compact time range of the recording where he makes the mistakes, he cut most part of the recording and replaced with a new lengthy recording, so that he is left with a few trivial ambiguities to resolve.

We also observed that some participants were editing *backwards in time*, so they do not have to deal with ambiguities they would have had otherwise. For example, when S6 was trying to eliminate a mistake in Task 2 where he defined an unused variable which he later deleted, he first eliminated the latter part where he deletes the variable, then eliminated the part he defines the variable. These edits did not cause any ambiguity, whereas if he eliminated the earlier part first an ambiguity would have occurred due to the changes in the latter part.

Some other participants said they would prefer to edit screencast with zero-length time range selection whenever possible, since it has lower chance of causing ambiguity because there are no previous recording that are replaced; it has a zero-length ambiguous area.

Users Resolve Ambiguities with Little Cognitive Effort

Lastly, we found that the participants tended not to give much cognitive effort when choosing the desired transformation for an ambiguity, even if they fully understood what the from/to handle represents and how it affects the subsequent part of the screencast. Most of the time, however, they were able to choose the desired transformation simply by focusing on how the *Next Frame* changes as they moved each handle's position, until the *Next Frame* looked as they desired. Among three participants who mentioned this in the interview, S6 said: *"I didn't give too much thought into 'why' or 'how' before every move because it's complicated. Besides, it wasn't really necessary to think about it to make the change as I wanted."* and S2 mentioned that he developed some level of *"intuition"* about how dragging each from/to handle would change the next frame.

Discussion and Future Work

One of the goals of our exploratory user study was to serve as a proof-of-concept to identify the potential opportunities for further improvements. The study revealed both the potential and limitations of the non-linear editing in its current form. On a positive note, we observed that the screencast editing feature is flexible enough to support a diverse range of editing scenarios. Since the alternative to the non-linear editing approach is users having to re-record the entire latter part of the screencast from the point of edit, non-linear editing would be preferable in most editing scenarios, especially if users need to correct a mistake they have made early on. However, the study also revealed that the additional amount of effort for ambiguity resolution is often a significant burden for users. Thus, further research effort should be focused on reducing user's burden of performing ambiguity resolution. We suggest two prominent directions for usability improvement based on the findings from the study.

As the users stated during the interviews, users often have difficulty understanding how each choice of ambiguity resolution would impact the latter part of the screencast, largely because they often do not recall every fine-grained text edits in the screencast they are recording. Thus, we believe further research is needed in exploring the design space of conflict resolution interface to give users a more comprehensive view of what changes they are making during ambiguity resolution. In particular, the interface should be able to visualize a more complete picture of each resolution outcome than just showing only the "Next Frame", and make it easier to navigate the screencast during ambiguity resolution to help users get more context by looking at the previous text edits they may have forgotten.

In addition, reflecting on the fact that most participants developed their own unique strategies with the goal of *avoiding* complex ambiguity resolutions, another possible direction for future work would be to simplify the ambiguity resolution process itself, such that it can be perceived not as a complicated process that often yields unpredictable results, but as a series of simple tasks that require a small amount of cognitive effort. One idea is to provide a "recommended" transformation as default, by automatically inferring the intention of the edit using a domain-specific algorithm (e.g. in the case of programming screencasts, a programming language parser) or machine learning model. For example, suppose a user made a non-linear edit and needs to choose a transformation between (b) and (c) in Figure 1, an algorithm could analyze the content and infer that (b) $a = [x]$ is a more "preferable" choice for the transformation than the other. If such "preferable" transformation could be accurately inferred each time and shown to the user as a "recommended" choice of transformation, users would be able to resolve the ambiguity with much less cognitive effort.

CONCLUSION

Text-based screencasts are widely used, but they are quite cumbersome to edit, unlike videos with the modern editing tools that allow flexible and easy editing. In this paper, we introduced a non-linear editing algorithm for text-based screencasts and a prototype screencast editor that provides non-linear editing functionality using the proposed algorithm. We conducted an exploratory study with six subjects demonstrating that users were successful in using our prototype screencast editor to create and edit a screencast in various editing scenarios. While the participants found the ambiguity resolution process difficult to learn and use due to its complex nature, the study revealed some possible directions for improvements, such as the editor providing more context, and the editor letting users to be fully aware of the changes made from ambiguity resolution.

ACKNOWLEDGEMENTS

This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2017M3C4A7065962).

REFERENCES

1. Christian Bird, Peter C Rigby, Earl T Barr, David J Hamilton, Daniel M German, and Prem Devanbu. 2009. The Promises and Perils of Mining Git. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*. IEEE, 1–10.
2. W Keith Edwards and Elizabeth D Mynatt. 1997. Timewarp: Techniques for Autonomous Collaboration. In *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*. ACM, 218–225.
3. Daniel M German, Bram Adams, and Ahmed E Hassan. 2016. Continuously Mining Distributed Version Control Systems: an Empirical Study of How Linux Uses Git. *Empirical Software Engineering* 21, 1 (2016), 260–299.
4. Shiry Ginosar, De Pombo, Luis Fernando, Maneesh Agrawala, and Bjorn Hartmann. 2013. Authoring multi-stage code examples with editable code histories. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. ACM, 485–494.
5. Shinpei Hayashi, Takayuki Omori, Teruyoshi Zenmyo, Katsuhisa Maruyama, and Motoshi Saeki. 2012. Refactoring Edit History of Source Code. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 617–620.
6. Marcin Kulik and contributors. 2012. asciinema. `https://asciinema.org`. (2012).
7. Qualified. 2018. Qualified: Codecast. `https://www.qualified.io/codecast`. (2018).
8. Scrimba. 2017. Scrimba. `https://scrimba.com`. (2017).
9. Chengzheng Sun. 2010. OT FAQ: Can OT solve semantic consistency problems? `http://www3.ntu.edu.sg/home/czsun/projects/otfaq/#_Toc321146139`. (2010).
10. Chengzheng Sun and Clarence Ellis. 1998. Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*. ACM, 59–68.
11. Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. 1998. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)* 5, 1 (1998), 63–108.
12. David Sun, Chengzheng Sun, Steven Xia, and Haifeng Shen. 2012. Creative conflict resolution in realtime collaborative editing systems. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*. ACM, 1411–1420.
13. David Sun, Steven Xia, Chengzheng Sun, and David Chen. 2004. Operational Transformation for Collaborative Word Processing. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*. ACM, 437–446.