SeeHow: Workflow Extraction from Programming Screencasts through Action-Aware Video Analytics

DEHAI ZHAO, Austrlian National University, Australia
ZHENCHANG XING, Austrlian National University, Australia
XIN XIA, Monash University, Australia
DEHENG YE, Tencent AI Lab
XIWEI XU, CSIRO's Data61, Australia
LIMING ZHU, CSIRO's Data61, School of CSE, UNSW, Australia

Programming screencasts on the web are one of the most preferred media type for developers to expand their programming knowledge and learn new concepts as they describe the workflow of completing a task. Nonetheless, the streaming characteristic of programming screencasts limits the searchability of specific content via text based retrieval techniques, resulting in the gap to access useful information and interact with it. In this work, we leverage Computer Vision (CV) techniques to build a programming screencast analysis tool which can automatically extract workflow from videos. We recognize programming actions and code snippets from frame sequences which are used to split videos into fragments. Tutorial watchers can interact with the extracted workflow which presents video summary, actions, code, time stamps and concept explanations. In addition, our tool can analyze programming screencasts straightly without extra software instrumentation or re-recording the videos. The proposed method is evaluated on 40 hours of tutorial videos and live coding screencasts with diverse programming environments. The results demonstrate our tool can extract fine-grained workflow accurately.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**.

Additional Key Words and Phrases: Programming screencast, Action Recognition, Workflow Extraction

ACM Reference Format:
Dehai Zhao, Zhenchang Xing, Xin Xia, Deheng Ye, Xiwei Xu, and Liming Zhu. 2020. SeeHow: Workflow Extraction from Programming Screencasts through Action-Aware Video Analytics. *ACM Trans. Softw. Eng. Methodol.* 37, 4, Article 111 (August 2020), 25 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Computer programming involves two types of knowledge: *knowing-what* (a.k.a declarative knowledge) and *knowing-how* (a.k.a procedural knowledge). Fig. 1 illustrates an example. Knowing-what involves facts or propositions of specific programming concepts or entities. For example, Activity in Fig. 1(a) is an Android class which takes care of creating a window in which developers can place app UI, and Bundle in Fig. 1(b) is a mapping from string keys to various parcelable values.

Authors' addresses: Dehai Zhao, dehai.zhao@anu.edu.au, Austrlian National University, 108 North Rd, Acton, ACT, Australia, 2601; Zhenchang Xing, zhenchang.xing@anu.edu.au, Austrlian National University, 108 North Rd, Acton, ACT, Australia, 2601; Xin Xia, xin.xia@monash.edu, Monash University, Wellington Rd, Melbourne, Victoria, Australia; Deheng Ye, dericye@tencent.com, Tencent AI Lab; Xiwei Xu, xiwei.xu@data61.csiro.au, CSIRO's Data61, Garden St, Sydney, NSW, Australia; Liming Zhu, liming.zhu@data61.csiro.au, CSIRO's Data61, School of CSE, UNSW, Garden St, Sydney, NSW, Australia.

1049-331X/2020/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

Fig. 1. An Example of Programming Workflow

Knowing-how involves the workflow to complete a programming task step by step. For example, the developer first creates a class Splash extending Activity class (Fig. 1(a)), and then overrides the method OnCreate() and calls various APIs (e.g., setContentView(), R.layout) in the overriding method (Fig. 1(b)). Next, she creates another activity class StartingPoint and declares several fields (Fig. 1(c)). Finally she sets the bundle mappings in the XML file (Fig. 1(d)).

Programming screencasts are screen recordings of the developers performing programming tasks. Typical examples include programming videos on Youtube and live coding streams on Twitch. The programming screencast for the programming workflow example in Fig. 1 is available at http://seecollections.com/seehow/example. Programming screencasts are an important medium for observing and learning programming workflow (i.e., developers' coding steps and outcomes) in detail, for example, how to create a new activity class (either manually or though IDE support), how to enter each API call incrementally, how to interweave the development of activity classes and how to bundle resources. Such workflow dynamics is much more informative than using static text and screenshots to describe the programming steps.

However, the utilization of the workflow information in programming screencasts has been limited by the image nature of the screencasts. That is, *the workflow is implicitly embedded in the sequence of screenshots*. This makes it hard for developers to get a quick overview of the workflow or to navigate and search the workflow steps [5, 7]. This also creates a significant barrier for developing easy-to-deploy tools for monitoring, analyzing and recommending the workflow steps, for example, finding that the developer does not set activity bundle after creating a new activity class, which cause the failure of starting the activity.

There are two types of methods for making the workflow in programming screencasts explicit. First, *record workflow steps together with programming screencasts* [5]. Workflow recording can be achieved by instrumenting software tools or underlying operating systems [7] to capture the developers' interactions with software tools, from which workflow steps can be derived. Although the instrumentation-based method can capture fine-grained workflow steps, its deployability is

constrained by the availability of accessibility APIs [4]. Second, *extract workflow from programming screencasts by comparing content changes between consecutive screenshots* [15, 25]. Workflow extraction is easy-to-deploy as it does not demand any special API support. But content-based analysis can extract only very coarse grained programming activities. For example, content-change analysis can segment the switching from code editing to bundle editing in Fig. 1. However, it cannot distinguish the editing of two different activity classes and the specific coding steps (e.g., method overriding, API calls). In a comparative study of 135 developers, Bao et al. [5] show that fine-grained coding steps are much more effective for learning knowing-how knowledge than coarse-grained programming activities. Readers are referred to Section 2.4.1 for further discussion.

In this work, we propose a novel workflow extraction approach that integrates the strengths of the existing two types of methods but overcomes their weaknesses. As a workflow extraction method, our approach is easy-to-deploy. Unlike content-based workflow extraction methods, our approach is action-aware, which extracts fine-grained coding steps from programming screencasts at the line granularity that the workflow-recording methods support. In this work, we extract four types of coding steps: *enter text*, *delete text*, *edit text*, and *select text*. Text can be source code or other textual content (e.g., command-line commands, xml configurations, and text field input). We develop deep-learning based computer vision methods to recognize and aggregate primitive coding actions and corresponding text edits.

To evaluate the effectiveness of our method, we build a dataset of programming screencasts, including 260 programming videos from YouTube and 10 live coding streams from Twitch (see Table 1). These screencasts involve multiple programming languages (e.g., Python and Java) and many different programming tasks (e.g., programming basics, Android app, web app, and game). They were created by 8 different developers (6 for YouTube and 2 for Twitch), using very different development tools (e.g., Eclipse, Pycharm, Sublime). The total duration of these screencasts is 41 hours. Two authors invest significant manual efforts to label the coding steps in these screencasts, including the starting and ending frame of a coding step and the text edited by the step[1]. We obtain 5,466 coding steps with variant durations (7.62 ± 10.82 seconds).

On this dataset, our workflow extraction method achieves 0.501 in F1-score at IoU = 1. IoU measures the intersection over union between the video fragments of the identified steps and the ground-truth steps. IoU = 1 means the perfect match of the two fragments. For the not-perfectly-aligned identified steps, 94.2% have only 0 or 1 frame offset. Our evaluation also confirms the stability of our method for different programming languages, developers, and development tools. We invite three developers to evaluate the quality of 2,605 not-perfectly-aligned coding steps. They rate 83% of these coding steps as correct, and their ratings have substantial agreements with Fleiss' kappa of 0.88. This suggests that the small misalignments of most of the not-perfectly-aligned steps do not affect the correct understanding of the identified steps.

Our work makes the following contributions:

- We develop the first computer-vision based action-aware method for extracting coding steps at the line granularity from programming screencasts.
- We contribute a large dataset of programming screencasts with 5,466 manually labeled coding steps for evaluating computer-vision based workflow extraction methods.
- We conduct extensive experiments to evaluate the accuracy and generality of our method and the quality of the extracted coding steps for human understanding.

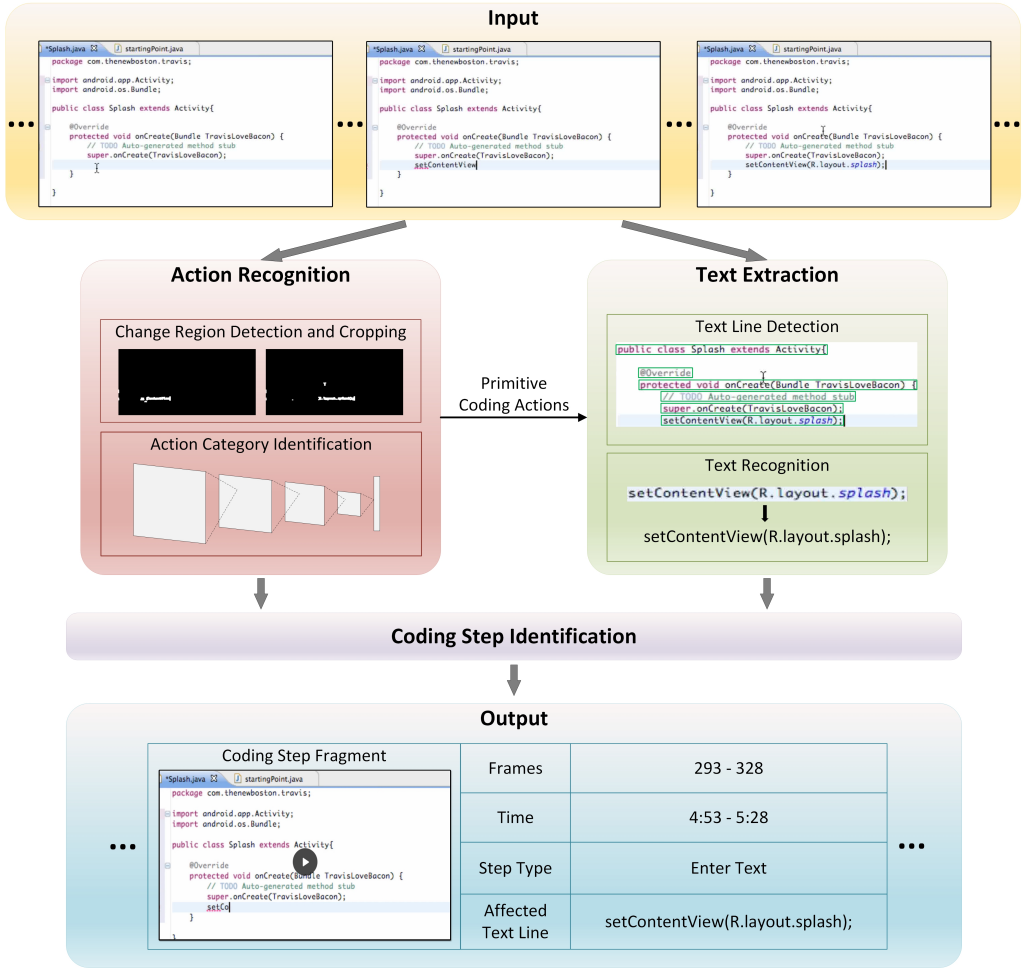


Fig. 2. Main Steps of Our Approach

2 APPROACH

Fig 2 presents the main steps of our approach. Our approach takes as input a programming screencast (i.e., a sequence of screenshots), and outputs a sequence of coding steps (Section 2.1). We develop computer-vision based techniques for action recognition and text extraction. By analyzing the consecutive frames, action recognition extracts screenshot regions affected by primitive Human-Computer Interaction (HCI) actions and determines action categories (Section 2.2), and text extraction extracts and aligns text lines affected by coding actions (Section 2.3). The action and text information are then aggregated to determine coding steps and the corresponding video fragments (Section 2.4).

[1]The dataset can be downloaded at this Github repository https://github.com/zzzzdh/SeeHow.

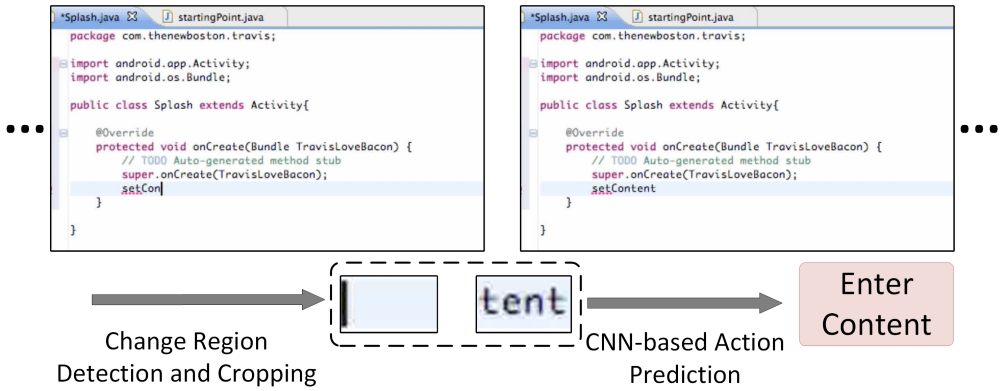


Fig. 3. Illustration of how ActionNet works

2.1 Input and Output

As illustrated in Fig. 2, an input programming screencast is a sequence of screenshots taken at a specific time interval (e.g., 1 second). Each screenshot is a frame in the screencast. Recording a screencast can be achieved by using operating-system level APIs, without the need for application-specific support. Taking advantage of this generality, our approach does not constrain the types of programming tasks, nor the development tools and programming languages used in the programming tasks. It also does not make any assumption about computer settings (e.g., screen resolution, window size and arrangement) and screenshot-taking interval. Furthermore, our approach does not rely on any workflow information recorded by instrumentation-based methods (e.g., ActivitySpace [7]).

As illustrated in Fig. 2, an output coding step consists of five pieces of information: the start and end frame, the start and end time, the corresponding video fragment, the type of coding step, and the text line affected by the code step. A sequence of identified coding steps constitutes a trailer of the input programming screencast. In this work, we consider four types of coding steps: enter text, delete text, edit text and select text. Although being referred to as coding steps, the text being entered, deleted, edited or selected includes not only source code but also other software text, such as command-line commands, console output, text field input like file name or search keywords, XML file content, and web page content. The granularity of text being manipulated is one line of text. This line granularity represents basic and coherent steps in programming tasks, which is a common granularity for version control. If a block (several lines) of text is manipulated as an atomic unit, we consider this block of text as a special line of text. Typical examples include a block of text being cut, pasted or selected as a whole, or a code block being added by code auto-completion or quick-fix assistants.

2.2 Action Recognition

When identifying coding steps, our approach is aware of primitive HCI actions that constitute the coding steps. This action awareness differentiates our approach from existing action-agnostic, content-only workflow extraction methods [6, 15, 26]. It allows our approach to filter out irrelevant content changes on the computer screen, resulting from non-coding actions such as switch windows, trigger or leave pop-ups, which always confuse action-agnostic methods. It also allows our approach to aggregate continual HCI actions on the same text line into coding steps in a coherent way (see Section 2.4), which has never been achieved by content-only analysis.

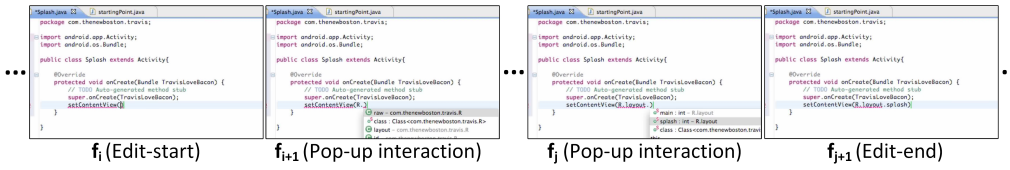


Fig. 4. An example of processing pop-up assisted code editing

In this work, we adopt ActionNet [39] to recognize primitive HCI actions in programming screencasts. ActionNet recognizes three categories, nine types of frequent HCI actions in programming work: *move cursor/mouse* (move cursor, move mouse over editable text, and move mouse over non-editable text), *edit content* (enter content and delete content), *interact with app* (select content, scroll content, trigger or leave pop-ups, and switch windows). All other HCI actions (e.g., move/resize windows, zoom-in/out text, and click UI elements like menu or list items) are recognized as an other-action type.

Fig. 3 illustrates an example of how ActionNet recognizes enter-content. The input to ActionNet are two consecutive frames in a programming screencast. First, ActionNet detects different pixels between the two frames based on the structure similarity index [33] between the same-position pixels of the two frames. If the two frames are identical, ActionNet reports no-action between the two frames. Otherwise, it computes the bounding box (i.e., the smallest rectangle region) that contains different pixels, and crops the corresponding image regions in the two input frames as the change regions. Then, the change regions are fed into a CNN-based image classifier, which predicts the HCI action that most likely results in the change regions on the two frames. Predicting HCI actions based on the change regions rather than the whole frames is a unique design of ActionNet. This design takes into account the characteristics of HCI actions - the instant screen changes by HCI actions and the (usually small) scale of screen changes, which are different from physical action recognition [30, 32, 36]

The consecutive frames capturing no actions or move-mouse/cursor actions are discarded for the coding step analysis, because they have no content changes. For coding step analysis, enter/delete/select-content are considered as coding actions. Select-content is considered as a coding action because the selected content is often edited afterwards. Scroll-content, trigger/leave pop-ups, switch-windows and other-action are considered as non-coding actions.

Although trigger/leave pop-ups is not a coding action, pop-ups are often used to display code completion or fix suggestions in the IDEs, which assist code editing (see Fig. 4). The interaction with pop-ups results in complex screen changes. For example, the pop-up window may block code and its content is surrounded by code. As the developer is typing the code, the pop-up window may move and its content keeps being updated. This makes it very difficult for content-only analysis [6, 15, 26] to infer pop-up assisted coding actions.

Our approach addresses this difficulty by identifying the frames before and after pop-up interactions. As shown in Fig. 4, when a trigger-popup is recognized between the two frames f_i and f_{i+1}, our approach first records f_i as a potential edit-start frame. Then, when a leave-popup is recognized between the two frames f_j and f_{j+1}, it records f_{j+1} as a potential edit-end frame. Finally, our approach feeds f_i and f_{j+1} into ActionNet to infer coding actions. If a coding action is recognized, f_i and f_{j+1} are used in the coding step analysis, while pop-up interactions in between f_i and f_{j+1} are ignored. If a coding action is not recognized (e.g., the developer cancels pop-ups), the original frame and action sequence will be used.

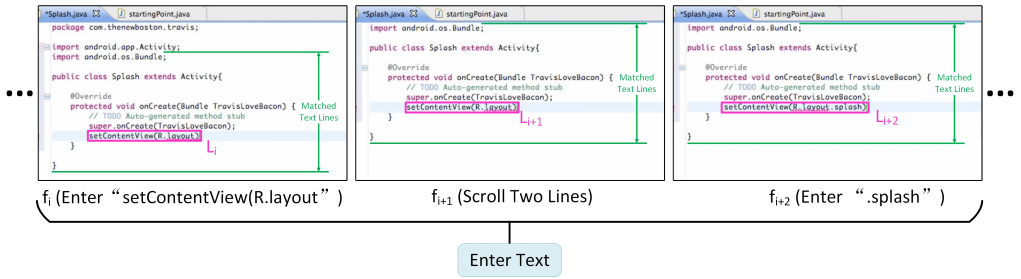


Fig. 5. An example of scroll-content in between coding actions

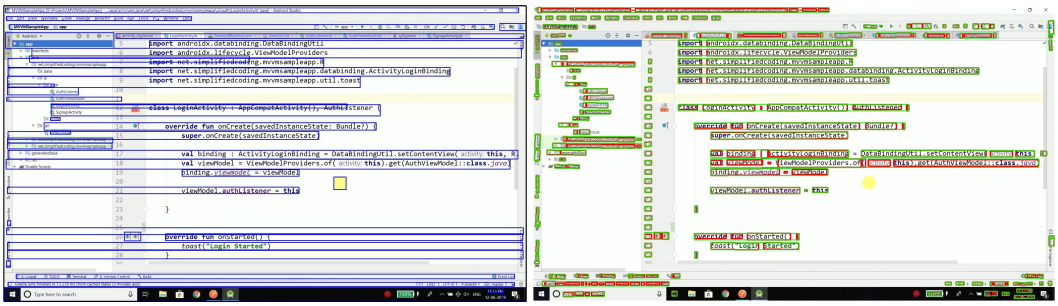
2.3 Text Extraction

If the two frames capture a coding action, our approach extracts and aligns text lines in the two frames. We also extract and align text lines in the two frames capturing a scroll-content action. Text-line alignment helps to aggregate continual coding actions on the same text line, even they may be separated by small content scrolling in between (see Fig. 5). Non-coding actions may change the displayed content (usually significantly). But such screen changes are due to the appearance of new content or the rearrangement of the displayed content, rather than the actual changes of the current content. Our action-aware approach ignores such meaningless content changes resulting from non-coding actions.

Extracting text from UI screenshots has two sub-tasks: *text line detection* - locate the bounding box of text elements (e.g., words) in the image, and *text recognition* - convert a text image into text characters. As shown in Fig. 6(a), UIs are usually composed of multiple windows. Different windows use different text styles (e.g., font, size, alignment, color, contrast). Even the content in the same window may use different text styles, for example, programming language keywords, constants or comments in code editor. UIs also contain visual components such as menu, tool bar, tab which also display text. All these UI text characteristics affect the choice of appropriate text extraction techniques.

Optical Character Recognition (OCR) tools (e.g., Tesseract [1]) are commonly used for extracting text from UI screenshots [6, 15, 26]. OCR tools are ideal for processing document images, but their performance degrades for UI screenshots [6] due to complex view layouts and text styles. Fig. 6(a) shows some inaccurate text lines on an IDE screenshot detected by Tesseract. Furthermore, the UI screenshot resolution (e.g., 96 dpi) is usually lower than the recommended resolution (300 dpi) for the OCR processing. A recent large-scale empirical study on GUI widget detection [11] shows that UI text should be treated as scene text (usually in low resolution and with messy text layouts and styles) rather than document text. Therefore, we adopt the state-of-the-art scene text detection tool EAST [42] to detect text lines in UI screenshots.

EAST detects text at the word level, for example, the red word boxes in Fig. 6(b). Word-level detection helps to distinguish component labels (e.g., menu text) from regular text (e.g., code). Our approach scans the detected word text boxes from left to right and top to bottom, and merges the horizontally adjacent boxes iteratively. Let $B_1(x_1, y_1, x_2, y_2)$ and $B_2(x_1, y_1, x_2, y_2)$ be the two adjacent text boxes, where (x_1, y_1) is the top-left coordinate and (x_2, y_2) is the bottom-right coordinate, and B_2 is right to B_1 (i.e., $B_2.x_1 > B_1.x_2$). B_1 and B_2 will be merged into one text box if they satisfy the two conditions: 1) The vertical coordinate of the horizontal middle line of B_2 (i.e., $(B_2.y_1 + B_2.y_2)/2$) is within the top and bottom line of B_1 (i.e., $[B_1.y_1, B_1.y_2]$; 2) The minimum horizontal distance between B_1 and B_2 is less than the average width of the characters in the two boxes. The average



(a) Text lines detected by Tesseract **(b)** Text lines merged from word boxes by EAST

Fig. 6. Example of text line detection

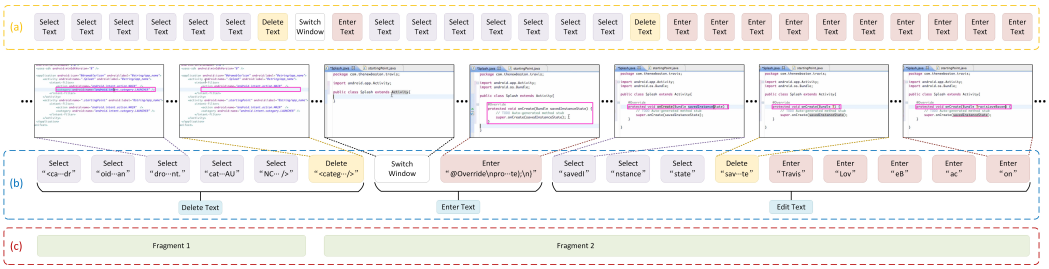


Fig. 7. Illustration of three coding step identification methods. (a) Output of ActionNet. (b) Output of our method. (c) Output of CodeTube

character width is computed as $((B_2.x_2 - B_2.x_1) + (B_1.x_2 - B_1.x_1))/num_c$ where num_c is the total number of characters in B_1 and B_2 recognized by CRNN [29]. If B_2 cannot be merged with B_1, B_2 is used as the new starting box to merge the word boxes to the right of B_2. This merging process continues until no more word boxes are left to merge. It outputs text lines such as those in green boxes in Fig. 6(b).

Compared with the text lines by Tesseract, text lines obtained by our approach corresponds intuitively to how human view text on the screen. Given a detected text line, our approach crops the image region by the text-line bounding box and uses the CRNN tool [29] to convert the cropped text image into text characters. CRNN stands for Convolutional Recurrent Neural Network, which is the state-of-the-art model for text recognition. CRNN uses convolutional neural network to extract image features, which makes it robust to text images with various fonts, colors and backgrounds.

After extracting text lines from the two frames, our approach aligns the extracted text lines to facilitate the identification of coding steps. It first traverse the text-line boxes in a frame from left to right and top to bottom, and produces a sequence of text-line boxes. It then computes the longest common subsequence of text-line boxes between the two frames. Each text-line box is treated as an atomic text element for string matching by longest common substring. We set the string matching threshold at 0.95. This high threshold allows only a small number of character differences (e.g., the OCR errors or character edits) between the two matched text lines.

2.4 Coding Step Identification

2.4.1 Comparison with ActionNet and CodeTube. Before introducing our coding step identification method, we first use an illustrative example (see Fig. 7) to compare the key differences between

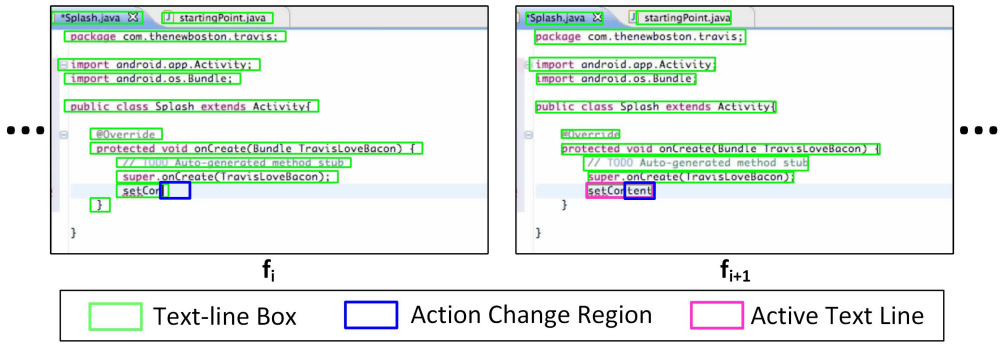


Fig. 8. An example of locating active text line(s)

our SeeHow method and two previous work (i.e. ActionNet [39] and CodeTube [25]), in terms of *information granularity* and *workflow extraction method*. Readers are referred to Section 4.6 for the empirical comparison among the three methods.

ActionNet detects actions between only two consecutive frames. As shown in Fig. 7(a), this finest-grained analysis recognizes only a sequence of primitive HCI actions, such as mouse movement, character-level text changes, and window switching or popup. Furthermore, ActionNet does not extract corresponding text content changed by the primitive actions. Unless such primitive, content-agnostic actions are aggregated appropriately by an approach like our SeeHow, they do not intuitively correspond to any meaningful coding steps, and thus cannot directly support any developers' information needs, such as programming video search.

In contrast to the primitive HCI actions by ActionNet, CodeTube outputs coarse-grained video fragments based on the analysis of only frame content similarity. As shown in Fig. 7(c), CodeTube detects only two long video fragments, one for editing configuration file, and the other for editing Splash activity class. Because CodeTube is action-agnostic, a coarse-grained video fragment will involve many fine-grained coding steps, for examples, how to override the onCreate() method step by step. The study by Bao et al. [5] shows that such fine-grained coding steps are crucial for searching, exploring and learning programming video tutorials.

As shown in Fig. 7(b), our SeeHow method aims to extract coding steps at the line granularity in between the primitive HCI actions by ActionNet and the coarse-grained video fragments by CodeTube. CodeMotion also attempts to recover code-line changes by comparing frame content changes. However, because CodeMotion is action-agnostic, it suffers greatly from irrelevant frame content changes, such as window switching and popup. Although instrumentation-based methods [5, 7] can extract coding steps at the line granularity, they are limited by the underlying operating system's accessibility APIs. In contrast, our SeeHow is a generic, vision-based method.

2.4.2 Identifying Coding Steps by SeeHow. A coding step at line granularity may consist of some primitive HCI actions. For example, entering a line of code may consist of a sequence of enter-content actions, some delete-content actions to correct typing errors, and also non-coding actions, such as move cursor, scroll content, pop-up interactions. Based on the recognized actions and the extracted text lines, our approach aggregates continual coding actions into coding steps at line granularity.

Fig. 7 (b) illustrates the examples of coding step identification by our method. Our approach scans the sequence of N screenshots from beginning to end. Recall that the frames capturing no actions or move-mouse/cursor actions have already been discarded in the step of action recognition. Let f_i

be the current frame pointer ($1 \leq i \leq N-1$). Let act_i be the ith action in the sequence, captured in f_i and f_{i+1}. If act_i is a non-coding action (e.g, the 3rd frame in Fig. 7), the approach increments the frame pointer by 1. If act_i is a coding action (e.g., the 5th frame in Fig. 7), it records f_i as the start frame of the coding step to be identified, and analyzes the action and text-line information of f_i and the subsequent frames to identify the coding step. After outputting a coding step, the approach continues to process the next frame.

As the coding step to be identified is enter-text, delete-text, edit text or select-text, our approach first locates the text line(s) L_i in f_i and L_{i+1} in f_{i+1} affected by the coding action act_i. This is determined by overlapping the change regions of act_i with the text-line boxes in f_i (or f_{i+1}). As shown in Fig. 8, the area of an action change region and the affected text line(s) may differ greatly, for example, only a few characters are added in a long code line. However, the action change region and the affected text line(s) should vertically overlap, no matter how different their areas are. Therefore, our approach identifies a set of vertically continuous text lines (denoted as L) with the maximum vertical overlap with the action change region (denoted as R). Multiple text lines is possible as some coding actions may affect a block of text as a whole. The vertical overlap vo of L and R is $[R.y_1, R.y_2] \cap [L.y_1, L.y_2] / [R.y_1, R.y_2] \cup [L.y_1, L.y_2]$, where y_1 and y_2 are the vertical coordinates of the top and bottom line of L or R. If $vo > 0.75$, L is identified as active text line(s) for act_i. The threshold 0.75 is determined by examining the correct and wrong overlaps in our dataset.

If L_{i+1} for enter/select-content includes multiple text lines (e.g., the fourth frame in Fig. 7), our approach outputs f_i and f_{i+1} as a coding step. The step type is the corresponding action type. If L_{i+1} for enter/select-content is empty, act_i is regarded as an invalid action (usually an erroneously recognized coding action). That is, active text-line analysis helps to filter out erroneous coding actions by ActionNet. If act_i is delete-content and L_i includes multiple text lines, our approach outputs f_i and f_{i+1} as a delete-text step.

If L_{i+1} for a coding action includes only one text line, our approach attempts to aggregate continual coding actions on the same text line (e.g., the 1st and 3rd step in Fig. 7). If the action act_{i+1} of f_{i+1} and f_{i+2} is a coding action, the approach identifies active text line(s) L_{i+1} for act_{i+1}. If L_{i+1} contains one text line, the approach determines if L_i and L_{i+1} is a pair of matched text lines between f_{i+1} and f_{i+2}. If that is the case, act_{i+1} and act_i are continual coding actions on the same text line. Otherwise, the approach computes the vertical overlap of L_i and L_{i+1}. If the vertical overlap is greater than 0.75, L_i and L_{i+1} are also considered as matched text lines. This helps to deal with large text edits on the same text line, resulting from text paste or the use of code completion assistant.

The action aggregation stops when a non-coding action (except scroll-content) is encountered or L_i and L_{i+1} cannot be matched in neither way. If a scroll-content action is encountered, our approach examines if L_i has a matched text line L_{i+1} on f_{i+1}. If so, the algorithm continues with L_{i+1} as the active text line for aggregating subsequent coding actions as illustrated in Fig. 5. When the aggregation stops, the approach records the latter frame of the last aggregated coding action as the end frame of the coding step. If active text line in the start frame is empty, the code step is marked as enter-text (the 2nd step in Fig. 7), otherwise as edit text (the 3rd step in Fig. 7). If L_{i+1} is empty (i.e., the entire line is deleted like the 2nd frame in Fig. 7), the coding step is marked as delete-text. The algorithm uses the active text line of the last coding action as the text line affected by the coding step.

3 EXPERIMENT SETUP

This section describes our experimental dataset and evaluation metrics for evaluating our workflow-extraction approach.

Table 1. Details of our programming screencast dataset

Programming videos from Youtube					
Language	**Idx**	**Videos**	**Dur.(h)**	**Tool**	**Task**
Python	P1	52	4.98	Terminal & Notepad	Programming basics
	P2	43	3.95	Pycharm	Programming basics
	P3	17	3.22	Terminal & Notepad	Programming basics
Java	P4	99	10.08	Eclipse	Android development
	P5	11	2.99	Android studio	Android development
	P6	38	5.83	Eclipse	Web app development
Live coding streams from Twitch					
Multiple	L1	5	5.00	Online IDE	Web app development
	L2	5	5.00	Sublime & Shell	Game development
Total	**8**	**270**	**41.05**	-	-

3.1 Coding-Step Dataset

We manually label 41 hours of programming screencasts created by eight different developers, and identify 5,466 line-granularity coding steps with variant durations. To the best of our knowledge, our dataset is the first of its kind for evaluating the extraction of line-granularity coding steps.

3.1.1 Data Collection. We collect programming screencasts from two sources: Youtube and Twitch. We search Youtube for the programming tutorials on Python and Java (the two most popular programming languages). We select the returned playlists that demonstrate programming tasks using software development tools, rather than those explaining programming concepts and knowledge on black board or slides. To test the capability boundary of our approach, we select the playlists using different software development tools (e.g., Eclipse, PyCharm, Notepad, Terminal), recorded with different computer settings (e.g., screen resolution, window theme, text style), and demonstrating diverse development tasks (e.g., programming basics, mobile, web or game development). Finally, we select six playlists (three for Python and three for Java) by six developers, which contain 260 videos with over 31 hours total duration (see Table 1). The duration of the videos ranges from 4 to 15 minutes (median 8 minutes).

We collect 10 live coding sessions on Twitch by the two developers. Each session is 1 hour. The two developers use very different computer settings and tools. Different from Youtube tutorials which demonstrate a specific task in a video, live coding sessions record the actual, continual development work by developers. In our collected streams, the two developers use Python, Java and/or Javascript, and occasionally use some shell scripts (e.g., PowerShell). Live coding developers may switch between programming languages and development tools for different development tasks during a living coding session.

3.1.2 Manual Labeling. We decode the collected 270 programming screencasts into 270 sequences of screenshots at the rate of 1 frame per second (fps) by the ffmpeg tool [2]. We manually label the resulting 41,591 non-identical frames into coding steps, in terms of the start and end frame and the type of each coding step. The labeling uses the same coding-step identification criteria as described in Section 2.4. That is, we use human annotators as the most accurate "computer-vision" technique to identify ground-truth coding steps for evaluating our computer-vision based approach. The first and second authors work independently to label the coding steps. We use the Fleiss' kappa measure

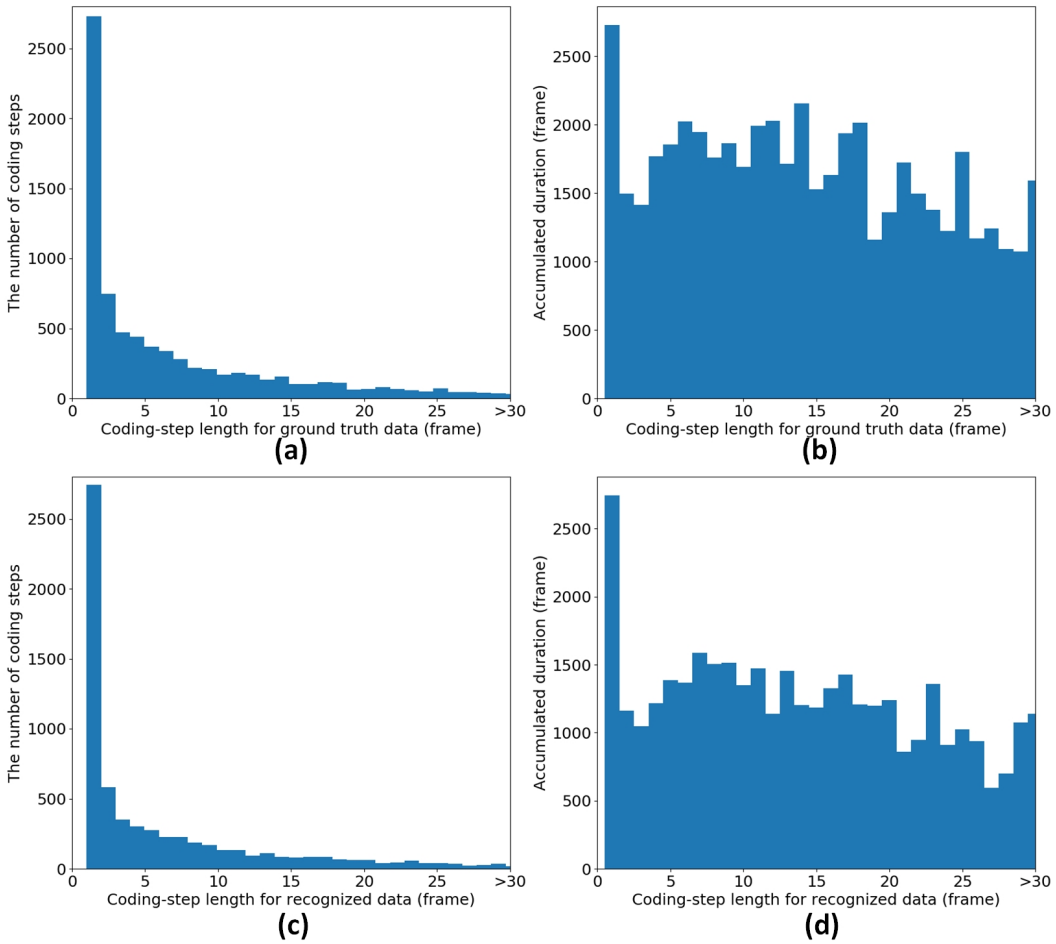


Fig. 9. Distribution of coding-step length and total duration

to examine the agreement between them. The overall kappa value is 0.93, which indicates almost perfect agreement between the labelers. After completing the labeling process, the two labelers discuss the disagreements to obtain the final labels.

We finally obtain 5,466 coding steps, with the duration of 7.62 ± 10.82 frames (i.e., seconds at 1 fps). There are 5,457 non-coding video fragments, with the duration of 16.17 ± 17.88 frames. The ratio of coding and non-coding fragments is about 1:1 because the two types of fragments usually appear alternately in the programming screencasts. Among the 5,466 coding steps, enter-text, delete-text, edit-text and select-text account for 36.5%, 0.5%, 20.9%, and 42.1%, respectively. Delete-text is much less than the other three types because developers either create new programs or modify existing code. Select-text is relatively more because developers often select code to explain in Youtube video tutorials.

Fig. 9(a) shows the distribution of coding-step length. The shortest coding step contains two frames. Coding steps with 5 or less frames account for about 61.7% of the 5,466 ground-truth coding steps. There are many short coding steps because developers often make small modifications to

existing lines of text, or make atomic text-block modifications. However, as shown in Fig. 9(b), the accumulated duration of different lengths of coding steps does not differ as much as the frequency of different code-step lengths. The accumulated duration of coding steps with 5 or less frames account for only 18.6% of the total duration of the 5,466 coding steps.

3.2 Evaluation Metrics

We compute Precision, Recall and F1-score of the identified coding steps in a programming screencast against the ground-truth coding steps. Precision is the portion of coding steps that are correctly recognized among all the identified coding steps. Recall is the portion of the ground-truth coding steps that that are correctly recognized. F1-score conveys the balance between Precision and Recall which is computed by $2 \times (Precision \times Recall)/(Precision + Recall)$.

To determine the correctness of an identified coding step, we need to first determine the accuracy of the boundary (i.e., start and end frame) of the identified coding step against the ground-truth coding step. To that end, we compute Intersection over Union (IoU) of the two video fragments. Let r be the coding-step fragment identified by our method, and gt be the ground-truth fragment. IoU is computed by $r \cap gt/r \cup gt$, i.e., the number of common frames between the two fragments over the total number of frames in the set of two fragments. If r and gt have no overlapping, their IoU is 0.

Given the sequence of identified coding-step fragments $R = \{r_1, r_2, \cdots, r_m\}$ and the sequence of ground-truth fragments $G = \{gt_1, gt_2, \cdots, gt_n\}$ of a programming screencast, we compute the longest common subsequence between R and G, with the goal of maximizing the IoU of the matched r_i and gt_j. When a r_i overlaps multiple ground-truth fragments, we select the gt_j with the highest IoU with r_i. When a r_i is matched with a gt_j, the matching continues from the frame next to the end frame of r_i. We allow multiple r_i to be matched to different fragments of one gt_j. This could happen when a long ground-truth fragment has been recognized as several consecutive short fragments.

Given a pair of matched coding-step fragments r_i and gt_j, if their IoU is above a threshold, we consider r_i as a correctly identified coding-step fragment. The higher the IoU threshold is, the more accurate the boundary of the identified coding-step fragment is, but the less number of fragments can be qualified as correct. If the step type of a correctly identified r_i matches the step type of gt_j, the coding step of r_i is regarded as a correctly identified coding step.

IoU has been widely used in object detection filed [27, 28] but it is not always the perfect evaluation metric. As described in [17, 37], even the best matching anchor box of a small object has a low IoU value typically. Similarly, IoU is sensitive to short fragment length. For example, assume gt has 4 frames and r has 3 frames. The highest IoU is only 0.75, when all three frames in r overlap gt. When IoU threshold is above 0.75, the identified fragment r, albeit high quality, will be regarded as incorrect. Therefore, we compute time offset as a complementary metric to understand the mismatch between the two overlapping fragments. When IoU > 0, the time offset is computed by $min(|F_r.start - F_{gt}.start|, |F_r.end - F_{gt}.start|, |F_r.start - F_{gt}.end|, |F_r.end - F_{gt}.end|)$. Time offset represents the least effort required to navigate from the start or end frame of the identified coding step to the start or end frame of the overlapping ground-truth step. As shown in Fig. 9, the majority of coding steps are short. Therefore, it would not be too difficult to identify the complete coding step after locating its start or end frame. Given a pair of matched r_i and gt_j, if their time offset is below a threshold, we consider r_i as a correctly identified coding-step fragment.

4 EXPERIMENT RESULTS AND FINDINGS

We conduct extensive experiments on our coding-step dataset to investigate the following four research questions:

- **RQ1.** How do the coding-step trailers extracted by our approach compare with the ground truth trailers?
- **RQ2.** How well does our approach perform under different IoU and time-offset threshold settings?
- **RQ3.** How well does our approach perform for different developers, development environments, programming languages, and programming tasks?
- **RQ4.** How well do developers rate the quality of the identified coding steps?
- **RQ5.** How dose SeeHow differ from ActionNet and CodeTube?

4.1 Comparison of Extracted and Ground-Truth Trailers (RQ1)

4.1.1 Motivation. A coding-step trailer provides a concise overview of coding steps in a programming screencast. Each screencast has an extracted trailer and a ground-truth trailer. We want to investigate how the identified coding steps compare with the ground-truth coding steps at the trailer level.

4.1.2 Method. We make three types of comparisons between the extracted trailers T_i and the ground-truth trailers T_{gt}. First, we compare the distribution of coding-step length and total duration in T_i versus T_{gt}. Second, we compare the frame coverage of a set of trailers (T_i or T_{gt}) over the original programming screencasts. Third, we compute three percentages: the overall IoU of T_i and T_{gt}, the percentage of the frames only in T_i (false positive), and the percentage of the frames only in T_{gt} (false negative).

4.1.3 Results. Our approach identifies 4,648 coding steps in the 270 programming screencasts. Fig 9(c) and Fig 9(d) show the frequency and total duration of different lengths of the identified coding steps, respectively. Compared with those of the ground-truth coding steps in Fig 9(a) and Fig 9(b), we observe similar overall distribution. About 66.0% of the identified coding steps have 5 or less frames. This percentage is relatively higher than the percentage of the ground-truth counter part (61.7%). Furthermore, the ground-truth trailers have relatively more long coding steps than the extracted trailers, and relatively longer total duration. This is because our approach may identify several shorter coding steps for a long coding step. However, the overall frame coverage does not differ significantly, with 26.9% for the extracted trailers and 28.1% for the ground-truth trailers. The overall IoU of the extract trailers and the ground-truth trailers is 0.661, only 22.4% of the frames in the extracted trailers are not in the ground-truth trailers, and only 21.3% of the frames in the ground-truth trailers are not covered in the extracted trailers.

The coding steps identified by our approach have similar length and duration distributions to the ground-truth steps. They also have similar overall coverage of the programming screencasts and low false-positive and false-negative frames.

4.2 Performance Under Different Matching Thresholds (RQ2)

4.2.1 Motivation. The correctness of coding steps depends on the IoU (or time-offset) threshold. The strictness of the threshold affects the boundary accuracy and the number of coding-step fragments that can be treated as correct, which in turn affects the performance of coding step extraction. In this RQ, we investigate the impact of the IoU and time-offset threshold on the extraction of coding steps.

4.2.2 Method. We experiment six IoU threshold $\{0, 0.3, 0.5, 0.7, 0.9, 1.0\}$. IoU > 0 means the identified coding step and the ground-truth step have at least one overlapping frame, and IoU = 1.0 means the identified step and the ground-truth step have identical frames from start to end. We experiment

Table 2. Performance at different IoU thresholds

IoU	Precision	Recall	F1-score
>0	0.891	0.942	0.909
>0.3	0.772	0.738	0.744
>0.5	0.703	0.671	0.677
>0.7	0.612	0.586	0.590
>0.9	0.548	0.525	0.529
=1.0	0.520	0.498	0.501

Table 3. Performance at different time-offset thresholds

Time-offset	Precision	Recall	F1-score
=0	0.838	0.824	0.820
≤1	0.869	0.864	0.856
≤3	0.882	0.891	0.876
≤5	0.890	0.911	0.891
≤7	0.892	0.926	0.900
≤9	0.894	0.937	0.907

six time-offset threshold {0, 1, 3, 5, 7, 9} (frame). time-offset = 0 means the identified coding step and the ground-truth step have at least one matched boundary, and time-offset ≤ 9 means the closest boundaries of the identified step and the ground-truth step have at most 9 frames gap. At each threshold, we obtain a set of correctly identified coding steps. We compute precision, recall and F1-score of these identified steps.

4.2.3 Results. Table 2 shows the overall evaluation metrics of all 4,648 identified coding steps at different IoU thresholds. At the strictest threshold IoU = 1.0, our approach still achieves 0.5 F1-score, which means about half of the identified coding steps match perfectly with the corresponding ground-truth steps. As the IoU threshold decreases, the boundary accuracy criteria is loosen, and more and more identified coding steps are qualified as correct. Consequently, the F1 increases gradually from 0.5 for IoU = 1.0 to 0.74 for IoU > 0.3. At the lowest threshold IoU > 0, the F1-score reaches the upper bound 0.9. That is, about 90% of the identified coding steps have at least one overlapping frame with the ground-truth steps.

Fig 10(a) is the distribution of different lengths of identified coding steps at different IoU threshold ranges. About 71.0% of coding steps with 5 or less frames (red bar) have IoU$\in (0.7, 1.0]$ with the ground-truth steps. Only a small number of coding steps with 5 or less frames have IoU$\in (0.7, 0.9]$, but about 13.2% of coding steps with 5 or less frames have IoU$\in (0, 0.3]$. This is because it is hard for short coding steps to achieve high IoU when they do not completely match the ground-truth steps. In contrast, about 60.4% of coding steps with more than 5 frames have IoU$\in (0.7, 1.0]$, and the rest of such long coding steps are roughly evenly distributed in different IoU ranges. This is because IoU metric is more robust when the length of coding steps increases.

We further analyze time-offset based performance metrics, as shown in Table 3. We can see that the upper bound of time-offset based metrics at time-offset ≤ 9 is almost the same as those at IoU > 0. However, when the time-offset threshold becomes stricter, the metrics decreases much slower, compared with the metric decrease as IoU increases. At the strictest time-offset = 0, the F1 is 0.82,

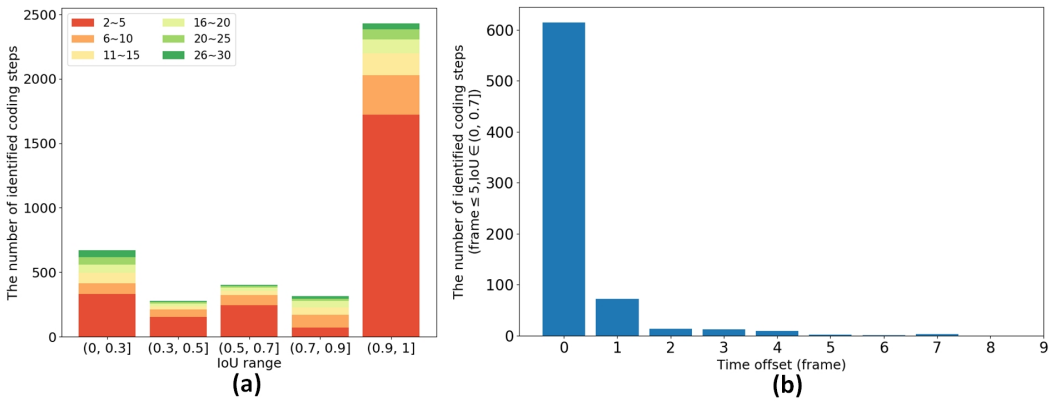


Fig. 10. Coding-step distribution at different IoU and time-offset thresholds

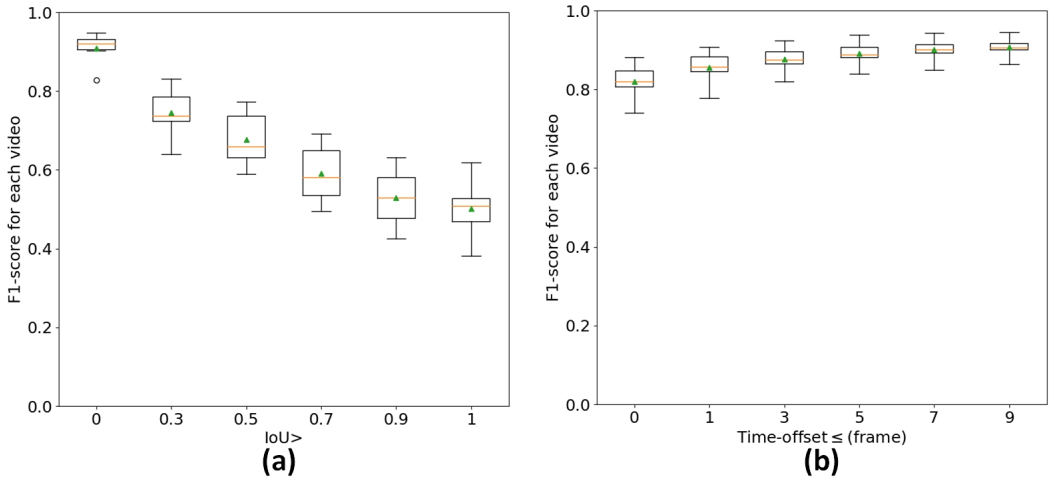


Fig. 11. Per-video F1 distribution at different IoU and time-offset thresholds

which means about 82% of the identified coding steps have at least one boundary matching the boundary of the corresponding ground-truth steps.

Furthermore, time-offset threshold at 5 or above improves the performance very marginally. This suggests that the boundary of correctly identified coding steps is usually very close (1-4 frame gap) to that of the ground-truth steps. Fig. 10(b) shows the time-offset distribution of the identified coding steps with 5 or less frames and having IoU$\in (0, 0.7]$ with the ground-truth steps. 94.2% of these coding steps have time-offset = 0 or 1. In fact, this is the main reason why time-offset based performance is much better than IoU based.

Our approach can accurately extract coding steps, even at the strictest boundary accuracy criteria. A large portion of short coding steps have low IoU with the ground-truth steps, but they have only 0-1 frame time offset.

Table 4. Performance on different playlists

Language	Idx	IoU > 0.7 Precision	IoU > 0.7 Recall	IoU > 0.7 F1-score	Time-offset ≤ 3 Precision	Time-offset ≤ 3 Recall	Time-offset ≤ 3 F1-score	#Id steps
Programming videos from Youtube								
Python	P1	0.703	0.696	0.691	0.936	0.932	0.923	498
	P2	0.645	0.553	0.585	0.917	0.838	0.863	661
	P3	0.705	0.644	0.670	0.905	0.902	0.901	649
	Ave.	0.684	0.631	0.648	0.919	0.891	0.896	-
Java	P4	0.535	0.526	0.522	0.883	0.905	0.880	877
	P5	0.505	0.512	0.495	0.811	0.859	0.820	489
	P6	0.578	0.588	0.579	0.842	0.910	0.871	577
	Ave.	0.539	0.542	0.532	0.882	0.891	0.876	-
Live coding streams from Twitch								
Multiple	L1	0.601	0.649	0.615	0.938	0.833	0.882	384
	L2	0.625	0.526	0.571	0.923	0.800	0.857	513
	Ave.	0.613	0.587	0.593	0.930	0.816	0.869	-

#Id steps means identified steps in each playlist

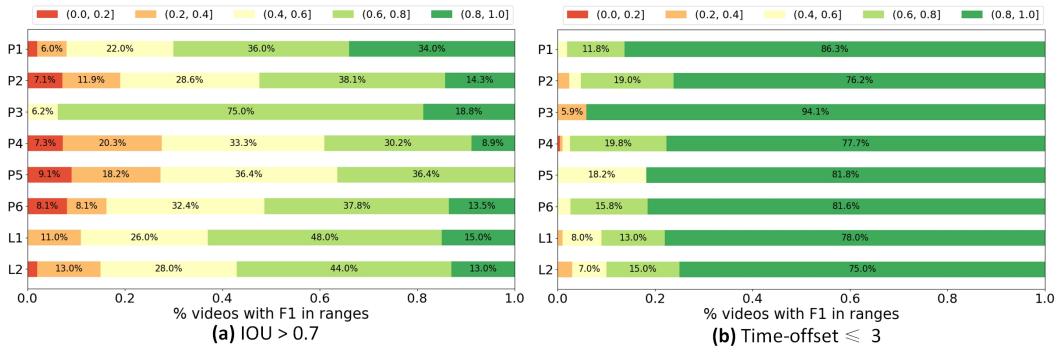


Fig. 12. Coding-step distribution at different IoU and time-offset thresholds

4.3 Performance On Diverse Programming Screencasts (RQ3)

4.3.1 Motivation. Developers use different programming languages and tools in their work, and their computer settings (e.g., screen resolution, window theme, text style) also vary. There are also many different types of programming tasks. This RQ aims to investigate the performance stability and variation of our approach in such diverse settings.

4.3.2 Method. Our coding-step dataset consists of 8 sets of programming screencasts, created by 8 different developers with diverse computer, tool, programming language and task settings (see Table 1). We first analyze the performance variations across 270 programming screencasts at different IoU and time-offset thresholds. Then, we zoom into the performance of each set of programming screencasts at the IoU > 0.7 or time-offset ≤ 3.

4.3.3 Results. Fig 11(a) and Fig 11(b) show the distribution of the F1-score of 270 programming screencasts at different IoU and time-offset thresholds, respectively. The interquartile range of the

box plots is mostly 0.1 between the 25th percentile and the 75th percentile for IoU-based evaluation, and less than 0.05 for time-offset based evaluation. Some screencasts are more challenging than others, which results in about 0.2 difference between the best and the worst F1-score for IoU, and about 0.1 F1-score difference for time-offset. Time-offset based evaluation is more stable than IoU based evaluation, due to the time-offset's tolerance of small mismatches.

Table 4 shows the metrics on the eight sets of programming screencasts. The performance rankings of these eight sets are consistent across IoU>0.7 and time-offset≤3 threshold. This indicates the consistency between the two thresholds, even though they represent different criteria for determining the correctness of the identified coding steps. P1 has the best performance, and P5 has the worst performance. Compared with the overall average F1-score (0.59 for IoU>0.7 and 0.876 for time-offset≤3), only P1 and P5 have relatively large variation (±0.1 in F1-score for IoU>0.7 and ±0.05 in F1-score for time-offset≤3). The 3 Python playlists have relatively better performance, the 3 Java playlists have relatively worse performance, and the 2 live coding sets are in between.

Fig 12 shows the percentage of videos in each set of programming screencasts whose F1-scores fall into different ranges. We show percentage because different sets have different numbers of screencasts. For 6 sets of programming screencasts, more than half of the videos have F1-score > 0.6 (light green and green bars) for IoU-based evaluation. For all 8 sets of programming screencasts, more than 81.3% of the videos have F1-score > 0.8 (green bars) for time-offset based evaluation. The videos with low F1-scores ((0.0,0.2] in red or (0.2,0.4] in orange) usually account for a small portion of a set of programming screencasts. This is especially the case for time-offset based performance.

We examine the videos with low F1-scores to understand what causes the low accuracy. We find that the challenging screencasts usually involve complex human computer interactions. For example, P6 demonstrates Web app development, in which the developer frequently switch between web design tool, command-line terminal and web browser to edit, deploy and view the web page back and forth. Sometimes, different windows may even overlap. As another example, Java IDEs (e.g., Eclipse) often provide sophisticated pop-ups. These pop-ups blur the coding steps with very dynamically changed pop-up contents. In contrast, Python developers often use text editors (e.g., Notepad) where coding steps are clear to identify. Complex human computer interaction poses a significant challenge for ActionNet to accurately recognize primitive HCI actions of interests. Some meaningless HCI actions may also confuse our approach. For example, some developers like to randomly select some code fragments, switch windows or scroll content while thinking about code features or errors, which result in many meaningless actions. The erroneously recognized or meaningless HCI actions negatively affect the downstream inference of coding steps.

Our approach performs stably across diverse programming screencasts. It can accurately identify coding steps in over 81% of the programming screencasts in our dataset by time-offset based evaluation. Improving the ActionNet's recognition capability for complex human computer interaction could further improve the performance.

4.4 Coding Step Quality by Human Evaluation (RQ4)

4.4.1 Motivation. The identified coding steps may not perfectly match the ground-truth coding steps at both boundaries. The evaluation against these ground-truth coding steps, no matter IoU or time-offset based, mechanically judge the correctness of the identified coding steps by an overlapping threshold. This RQ aims to investigate how developers judge the correctness of not-perfectly-aligned coding steps.

4.4.2 Method. We exclude the identified coding steps with IoU = 1.0 as they perfectly match the ground-truth coding steps labeled by the two authors. For the 270 programming screencasts in

our dataset, we keep the 2,605 not-perfectly-aligned coding steps (i.e., IoU$\in (0, 1.0)$). We recruit 3 developers who have at least 3 years programming experience on Python and Java. We ask them to watch a programming screencast in our dataset and then review the non-perfectly-aligned coding steps for this screencast. The developers can contrast these coding steps against the original programming screencast and/or the ground-truth coding steps. They give a binary rating for each non-perfectly-aligned coding steps: correct or incorrect. Correct rating means the developers believe the inaccurate boundaries of the identified coding steps do not affect their correct understanding of the identified steps. The 3 developers examine all 270 programming screencasts in our dataset, and perform the correctness annotation independently. We compute Fleiss's Kappa [13] to evaluate the inter-rater agreement, and use the majority of three ratings for a step as the final rating.

4.5 Results

The 3 developers have almost perfect agreement (Fleiss' kappa = 0.88) for marking the correctness of the not-perfectly-aligned coding steps. Fig 13(a) shows the distribution of the percentage of correct-rated coding steps per video in the 8 sets of programming screencasts. The overall average correctness percentage is 83.6%. This correctness percentage is for not-perfectly-aligned coding steps, but it is close to the overall F1-score 79.3% at Io>0.2 and the overall F1 85.6% at time-offset$\leq$1. The correctness percentage has small variations within and across the 8 sets of programming screencasts. The correctness percentage of Python programming tutorials and live coding streams are slightly higher than that of Java programming tutorials. These correctness rating results are consistent with those by IoU and time-offset based evaluation.

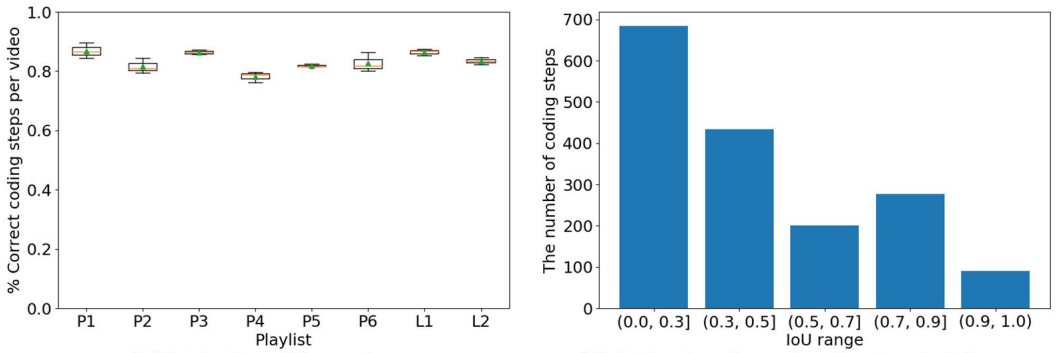


Fig. 13. Correctness ratings by human evaluation

Fig. 13(b) shows the distribution of the correct-rated coding steps in different IoU ranges with the ground-truth coding steps. We see that correct-rated coding steps are scattered in all IoU ranges. About 700 correct-rated coding steps have $\leq$ 0.3 IoU with the ground-truth coding steps. Those coding steps are regarded as incorrect steps when the IoU threshold is above 0.3. But they are rated as correct when reviewed by the 3 developers. As discussed in Section 4.2, a large portion of identified coding steps with low IoU have a small boundary gap (0-3 frames) from the ground-truth steps. When watching a coding step in the programming screencast, such a small gap do not affect the understanding of the identified coding steps.

We also find that the correct-rated coding steps may have low F1-score with the ground-truth steps due to the over fragmentation. That is, one ground-truth step may be identified as several smaller steps. This usually happens when the coding step interweaves with the interaction with

Table 5. Performance of three methods

Method	ActionNet	SeeHow	CodeTube
Output Amount	33727	5466	578
Mean fragment length	2	7.62 ± 10.82	130.50 ± 114.43

pop-ups. Although we develop special processing to mitigate the impact of pop-up interactions (see Section 2.2), the complexity of pop-up interactions may still result in one coding step being recognized as several smaller fragments. Those smaller fragments have low IoU with the ground-truth coding steps. But it is not difficult to associate them with the surrounding related steps when watching the video fragments of these steps. Therefore, the developers often still rate these low IoU steps as correct.

The developers' correctness ratings of the identified coding steps are similar to time-offset base evaluation. They can understand the identified steps in face of their small boundary inaccuracy or over-segmentation.

4.6 Comparison of the Outputs by SeeHow, ActionNet and CodeTube (RQ5)

4.6.1 Motivation. As mentioned in Section 2.4, the proposed SeeHow is methodologically different from previous work such as ActionNet [39] and CodeTube [25]. The most straightforward way to reflect the difference is to analyze the outputs of the three methods.

4.6.2 Method. We use ActionNet's source code and pretrained model from the author's Github repository https://github.com/DehaiZhao/ActionNet_0 with default settings. As CodeTube dose not release source code, we implement it carefully with the top ranked parameters in the paper: $\alpha = 5\%$, $\beta = 15\%$ and $\gamma = 50s$, where α is minimum percentage of Longest Common Substring (LCS) overlap between two frames to consider them as containing the same code fragment, β is minimum textual similarity between two fragments to merge them in a single fragment and γ is minimum video fragment length. We run the two methods on all 270 programming screencasts in our dataset and analyze the amount of output and the mean fragment length output by the three methods.

4.6.3 Results. Table 5 shows the results. ActionNet has the highest amount of output (33,727) because it recognizes programming actions for adjacent two frames as long as there is a (samll) change region between them, which is also the reason why its mean fragment length is two. CodeTube finally extracts 578 video fragments from 270 screencasts, which is about 10 times less than the number of coding steps identified by SeeHow. These video fragments have mean fragment length of 130.50 ± 114.43 frames. The high standard deviation is caused by the diversity of the tutorial authors' programming habits. For example, some authors keep coding in one window even though the contents are not so relevant, while others switch window frequently in one video. Our method output 5466 coding-step fragments with mean fragment length of 7.62 ± 10.82 frames, which is between ActionNet and CodeTube.

SeeHow is essentially different from ActionNet and CodeTube in terms of the information granularity, and its output fills the gap between the primitive HCI actions by ActionNet and the coarse-grained video fragments by CodeTube.

5 RELATED WORK

Programming screencasts record UI screenshots while developers perform coding steps. The image nature of UI screenshots limits the ways developers can interact with the recorded coding steps. VT-Revolution [5] develops a novel way for making interactive programming screencasts. During screen recording, it relies on software instrumentation to record corresponding coding steps. The recorded coding steps allows developers to scan, search and navigate programming screencast just like textual tutorials. Although the enhanced interaction with programing screencasts is intriguing, its reliance on software instrumentation is a major barrier for the wide adoption of VT-Revolution. The authors of VT-Revolution envision to make regular programming screencasts interactive through computer-vision based workflow extraction.

CodeTube [25, 26] and CodeMotion [15] share this vision. CodTube extracts code-like content from screenshots and splits a long video into shorter fragments based on the content similarity of adjacent screenshots. The fragments (usually in minutes) correspond to coarse-grained programming activities (e.g., all changes in a file). In contrast, the coding steps our approach extracts are at the line granularity and have second-level durations, close to what VT-Revolution records by software instrumentation. CodeMotion extracts code content in a similar way to CodeTube, but it attempts to infer more fine-grained coding actions from content changes. However, as CodeMotion is action agnostic, its action inference is unreliable in face of complex window interactions.

Our method is related to action detection in natural scene [9, 14, 18, 36, 41], which aims to detect the start and end time of action instances in untrimmed videos. R-C3D [36] applies 3D convolutional neural network [32] to extract spatial-temporal features from video frames for the prediction of action intervals. Yue et.al [41] proposes to model the temporal structure of each action instance by a structured temporal pyramid network. We adopt the IoU and time-offset metrics from these works for evaluating action detection in untrimmed videos. Our approach extends ActionNet [39], a specially designed neural model for recognizing HCI actions in programming screencasts. However, ActionNet only makes predictions for two consecutive frames, and it recognizes only primitive actions. Our approach enhances the ActionNet's output and exploits both primitive HCI actions and coding-action-relevant text content to infer coding step fragments and types.

An alternative way to obtain coding steps is software instrumentation [7]. Being able to obtain fine-grained coding steps is an advantage of software instrumentation over video analytics like CodeTube [25] and CodeMotion [15]. Our work closes the gap between video analytics and software instrumentation, and get rid of the hard-to-deploy issue of software instrumentation. We see fast growth in applying computer-vision techniques to GUI data, such as GUI widget detection [11], GUI testing [8, 12, 34, 38], GUI code generation [10], GUI visual defect detection [19, 20, 35, 40], GUI evolution analysis [21], GUI content prediction [23, 24]. Our work adds to this picture a new way to "see and understand" dynamic programming behaviors. All these techniques together gradually move us towards to the envisioned human-AI pair programming [31].

6 THREATS TO VALIDITY

Threats to internal validity refer to factors internal to our studies that could have influenced our experimental results:

(1) *The evaluation of SeeHow:* The coding fragments and non-coding fragments are labeled manually for evaluating our method, which means error labels are inevitable. In order to minimize the human label errors, three annotators are recruited and label independently, and their labels achieve almost perfect agreement. The high agreement benefits from the explicit boundaries of coding steps at the line granularity. Some disagreements come from identifying

whether the video fragments (e.g., typing a class name in configuration window) are relevant to coding. But the annotators were able to finally reach an agreement after discussion.

(2) *The knowledge background of annotators:* In RQ4, three developers were recruited to mark the correctness of the non-perfectly-aligned coding steps. A correct mark will be given if the developers believe they can understand the identified steps, even though these coding steps have inaccurate boundaries. The three developers suggested that it is not difficult for them to understand the concepts in tutorial videos, especially those videos for beginners, by only watching a short fragment. A bias could be introduced because they have certain level of programming knowledge and experience.

Threats to external validity refer to the generalizability of the results in this study:

(1) *Scale of the dataset:* We construct the first dataset of the line-granularity coding steps, which covers two programming languages, 8 playlists, 270 videos and 41 hours duration. The dataset, far from covering all knowledge about programming, is a typical example of programming tutorial video, as those programming tutorials are not much different in their structure and content than others. In the future, we plan to collect more tutorial videos with different programming languages, tools and configurations to further evaluate SeeHow. In addition, labeling 41 hours video costs about 40 hours for every annotator because of the required code-step granularity. For example, the annotator cannot exactly determine it is the end of a coding step when the author types the last character of a code line in frame f_i. Because more editing actions could be done in this code line. The annotator has to keep watching until the author starts editing a new line in frame f_j. Then he needs to revert back to f_i and mark it as the end of a coding step. Such a large number of reverting operations makes the dataset very time consuming to create.

(2) *Noise in text recognition results:* We use computer vision and deep learning methods to recognize code from frames, which is inevitable to have error results. In [6], the authors try to denoise code extracting results from programming screencasts by the statistical language model of a large corpus. However, we have to update the corpus when adding more programming screencasts and create a corpus for each programming language, which is an enormous work. We find that noises have no impact on our experimental results. Therefore, we ignore the denoise process and focus on the code recognition quality such as applying the deep learning based text detection [42] and text recognition [29] methods in this work. The noisy results may affect the user experience of our SeeHow's web application when they read the extracted coding steps. We plan to integrate the denoise process such as psc2code [6] into our tool in the future.

(3) *Selection of OCR engine:* We apply EAST [42] and CRNN [29] to as OCR engines in this work. There are many other alternative methods such as google vision OCR [3], which has been shown to perform extremely well on programming screencasts and has been successfully used in previous work [6, 16, 22]. We also test google vision OCR and achieve similar performance with EAST and CRNN. EAST and CRNN run locally on GPU, and google vision OCR is cloud service. We finally choose EAST and CRNN mainly because of the following reasons: First, despite the network latency, the network security cannot be guaranteed, which is also a problem in software engineering field. Second, some devices with high security level are not allowed to connect to public network and it is infeasible to call google cloud service. Running source code locally is reliable and safe. Third, EAST and CRNN is free, but Google vision service is not.

7 CONCLUSION AND FUTURE WORK

This paper presents the first approach for extracting the line-granularity coding steps from programming screencasts, without the need of software instrumentation. The innovation lies in the action-aware text context extraction and coding step identification. Furthermore, our approach incorporates the recent advances in HCI action recognition and GUI text detection, rather than relying on the long-used OCR-based video content analysis. The evaluation on 41 hours diverse programming screencasts shows that half of the coding steps identified by our approach match perfectly with the ground-truth steps, and most of the not-perfectly-matched steps have small (0-3) frame offset. The three developers judge 83% of the identified coding steps as correct, with substantial inter-rater agreement. Our study shows that the performance of our method is dependent on the quality of the programing screencasts. It would still be desirable for it to work well on the low quality videos. In the future, we will investigate two interesting topics through computer-vision techniques: making regular programming screencasts interactive, and tracking and analyzing programming behaviors.

REFERENCES

[1] August 28, 2020. https://github.com/tesseract-ocr.
[2] August 28, 2020. https://ffmpeg.org/.
[3] August 28, 2020. https://cloud.google.com/vision.
[4] Lingfeng Bao, Jing Li, Zhenchang Xing, Xinyu Wang, and Bo Zhou. 2015. scvRipper: video scraping tool for modeling developers' behavior using interaction data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 673–676.
[5] Lingfeng Bao, Zhenchang Xing, Xin Xia, and David Lo. 2018. Vt-revolution: Interactive programming video tutorial authoring and watching system. *IEEE Transactions on Software Engineering* 45, 8 (2018), 823–838.
[6] Lingfeng Bao, Zhenchang Xing, Xin Xia, David Lo, Minghui Wu, and Xiaohu Yang. 2020. psc2code: Denoising Code Extraction from Programming Screencasts. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 3 (2020), 1–38.
[7] Lingfeng Bao, Deheng Ye, Zhenchang Xing, Xin Xia, and Xinyu Wang. 2015. Activityspace: a remembrance framework to support interapplication information needs. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 864–869.
[8] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. 2020. Translating Video Recordings of Mobile App Usages into Replayable Scenarios. *arXiv preprint arXiv:2005.09057* (2020).
[9] Fabian Caba Heilbron, Victor Escorcia, Bernard Ghanem, and Juan Carlos Niebles. 2015. Activitynet: A large-scale video benchmark for human activity understanding. In *Proceedings of the ieee conference on computer vision and pattern recognition*. 961–970.
[10] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*. 665–676.
[11] Jieshan Chen, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, and Liming Zhu. 2020. Object Detection for Graphical User Interface: Old Fashioned or Deep Learning or a Combination? *arXiv preprint arXiv:2008.05132* (2020).
[12] Christian Degott, Nataniel P Borges Jr, and Andreas Zeller. 2019. Learning user interface element interactions. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 296–306.
[13] Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin* 76, 5 (1971), 378.
[14] Rui Hou, Chen Chen, and Mubarak Shah. 2017. Tube convolutional neural network (T-CNN) for action detection in videos. In *Proceedings of the IEEE international conference on computer vision*. 5822–5831.
[15] Kandarp Khandwala and Philip J Guo. 2018. Codemotion: expanding the design space of learner interactions with computer programming tutorial videos. In *Proceedings of the Fifth Annual ACM Conference on Learning at Scale*. 1–10.
[16] Abdulkarim Khormi, Mohammad Alahmadi, and Sonia Haiduc. 2020. A study on the accuracy of ocr engines for source code transcription from programming screencasts. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 65–75.
[17] Mate Kisantal, Zbigniew Wojna, Jakub Murawski, Jacek Naruniec, and Kyunghyun Cho. 2019. Augmentation for small object detection. *arXiv preprint arXiv:1902.07296* (2019).

[18] Tianwei Lin, Xu Zhao, and Zheng Shou. 2017. Single shot temporal action detection. In *Proceedings of the 25th ACM international conference on Multimedia*. 988–996.

[19] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2020. Owl Eyes: Spotting UI Display Issues via Visual Understanding. In *Proceedings of the 35th International Conference on Automated Software Engineering*.

[20] Kevin Moran, Boyang Li, Carlos Bernal-Cárdenas, Dan Jelf, and Denys Poshyvanyk. 2018. Automated reporting of GUI design violations for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering*. 165–175.

[21] Kevin Moran, Cody Watson, John Hoskins, George Purnell, and Denys Poshyvanyk. 2018. Detecting and summarizing gui changes in evolving mobile apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 543–553.

[22] Parisa Moslehi, Bram Adams, and Juergen Rilling. 2018. Feature location using crowd-based screencasts. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 192–202.

[23] Jordan Ott, Abigail Atchison, Paul Harnack, Adrienne Bergh, and Erik Linstead. 2018. A deep learning approach to identifying source code in images and video. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 376–386.

[24] Jordan Ott, Abigail Atchison, Paul Harnack, Natalie Best, Haley Anderson, Cristiano Firmani, and Erik Linstead. 2018. Learning lexical features of programming languages from imagery using convolutional neural networks. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 336–3363.

[25] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Mir Hasan, Barbara Russo, Sonia Haiduc, and Michele Lanza. 2016. Too long; didn't watch! extracting relevant fragments from software development video tutorials. In *Proceedings of the 38th International Conference on Software Engineering*. 261–272.

[26] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Barbara Russo, Sonia Haiduc, and Michele Lanza. 2016. CodeTube: extracting relevant fragments from software development video tutorials. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 645–648.

[27] Joseph Redmon and Ali Farhadi. 2018. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[28] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*. 91–99.

[29] Baoguang Shi, Xiang Bai, and Cong Yao. 2016. An end-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition. *IEEE transactions on pattern analysis and machine intelligence* 39, 11 (2016), 2298–2304.

[30] Karen Simonyan and Andrew Zisserman. 2014. Two-stream convolutional networks for action recognition in videos. In *Advances in neural information processing systems*. 568–576.

[31] Xin PENG Zhenchang XING Jun SUN. 2019. AI-boosted software automation: learning from human pair programmers. *Science China (Information Sciences)* 10 (2019), 6.

[32] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. 2015. Learning spatiotemporal features with 3d convolutional networks. In *Proceedings of the IEEE international conference on computer vision*. 4489–4497.

[33] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing* 13, 4 (2004), 600–612.

[34] Thomas D White, Gordon Fraser, and Guy J Brown. 2019. Improving random GUI testing with image-based widget detection. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 307–317.

[35] Ziming Wu, Yulun Jiang, Yiding Liu, and Xiaojuan Ma. 2020. Predicting and Diagnosing User Engagement with Mobile UI Animation via a Data-Driven Approach. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–13.

[36] Huijuan Xu, Abir Das, and Kate Saenko. 2017. R-c3d: Region convolutional 3d network for temporal activity detection. In *Proceedings of the IEEE international conference on computer vision*. 5783–5792.

[37] Jiangqiao Yan, Hongqi Wang, Menglong Yan, Wenhui Diao, Xian Sun, and Hao Li. 2019. IoU-adaptive deformable R-CNN: Make full use of IoU for multi-class object detection in remote sensing imagery. *Remote Sensing* 11, 3 (2019), 286.

[38] Rahulkrishna Yandrapally, Andrea Stocco, and Ali Mesbah. 2020. Near-duplicate detection in web app model inference. In *ACM*. 12.

[39] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xin Xia, and Guoqiang Li. 2019. ActionNet: vision-based workflow action recognition from programming screencasts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 350–361.

[40] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Seenomaly: Vision-Based Linting of GUI Animation Effects Against Design-Don't Guidelines. In *42nd International Conference on Software Engineering (ICSE'20). ACM, New York, NY*.

[41] Yue Zhao, Yuanjun Xiong, Limin Wang, Zhirong Wu, Xiaoou Tang, and Dahua Lin. 2017. Temporal action detection with structured segment networks. In *Proceedings of the IEEE International Conference on Computer Vision*. 2914–2923.

[42] Xinyu Zhou, Cong Yao, He Wen, Yuzhi Wang, Shuchang Zhou, Weiran He, and Jiajun Liang. 2017. East: an efficient and accurate scene text detector. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 5551–5560.