VizProg: Identifying Misunderstandings By Visualizing Students' Coding Progress

Ashley Zhang
University of Michigan
Ann Arbor, Michigan, USA
gezh@umich.edu

Yan Chen
Virginia Tech
Blacksburg, Virginia, USA
ych@vt.edu

Steve Oney
University of Michigan
Ann Arbor, Michigan, USA
soney@umich.edu

ABSTRACT

Programming instructors often conduct in-class exercises to help them identify students that are falling behind and surface students' misconceptions. However, as we found in interviews with programming instructors, monitoring students' progress during exercises is difficult, particularly for large classes. We present VizProg, a system that allows instructors to monitor and inspect students' coding progress in real-time during in-class exercises. VizProg represents students' statuses as a 2D Euclidean spatial map that encodes the students' problem-solving approaches and progress in real-time. VizProg allows instructors to navigate the temporal and structural evolution of students' code, understand relationships between code, and determine when to provide feedback. A comparison experiment showed that VizProg helped to identify more students' problems than a baseline system. VizProg also provides richer and more comprehensive information for identifying important student behavior. By managing students' activities at scale, this work presents a new paradigm for improving the quality of live learning.

KEYWORDS

programming education at scale, code visualization

ACM Reference Format:
Ashley Zhang, Yan Chen, and Steve Oney. 2023. VizProg: Identifying Misunderstandings By Visualizing Students' Coding Progress. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems (CHI '23), April 23–28, 2023, Hamburg, Germany.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3544548.3581516

1 INTRODUCTION

Programming instructors often conduct in-class coding exercises—short programming activities that students perform independently—to give students hands-on practice, assess students' progress, and identify students that are falling behind. By identifying and working with struggling students, instructors can strengthen students' understanding of the material and give them a better intuition for important concepts. However, if left unaddressed, small misunderstandings can escalate to become long-term learning barriers for students. Therefore, instructors should be able to identify struggling students and their misunderstandings during in-class exercises promptly and reliably. However, identifying problems in real time is difficult for several reasons. First, misunderstandings tend to be implicit, abstract, and not readily apparent without carefully reading students' code. However, it is often not possible to read students' code at scale in large classes or for shorter exercises. Second, there are many aspects of students' code (including aspects that the instructor might not anticipate) that instructors need to consider to gain insight into potential learning barriers. This suggests that there needs to be a better way to monitor students' code at scale.

Past research has explored ways to address these challenges. For example, Codeopticon [16] allows instructors to monitor students' code in real-time. However, Codeopticon requires that instructors read students' code individually, making it difficult to assess students' performance as a whole, particularly when needing to scale to large classes. Overcode [14] addresses the scalability issue by clustering and visualizing student code submissions [14]. However, it was designed for post-hoc analyses rather than providing real-time feedback and does not consider the need to monitor students over time. We also found in our interviews with programming instructors that time sensitivity and large class sizes make it difficult for instructors to identify learning challenges during in-lecture exercises. Ideally, instructors should be able to easily identify problems among many students' coding activities in real-time.

In this paper, we propose new techniques to address these problems and allow instructors to visualize and understand students' status in real-time for in-class programming exercises. Our design takes inspiration from maps of physical spaces. On a map, if we know a person's starting point, destination, and location, we can easily determine how close they are to their destination. With real-time updates, we could also determine if they are progressing to their destination or if they might be lost. What if checking where a student is on a programming exercise could be as easy as seeing *where* they are on a map? Although prior work has represented code in 2-D spaces [14, 19, 36], our approach is the first to do so in a way that explicitly encodes human-understandable meaning to the space (their problem-solving approach and their progress) and that can work in real-time (as students are typing). This work represents an initial step to show the feasibility and benefits of this approach.

We introduce VizProg, a tool that allows instructors to monitor and inspect large numbers of students' coding submissions over time by presenting students on a 2D map. In VizProg, students' status is represented as a position that encodes 1) similarities in students' code (as 2D Euclidean distances); 2) how students *approach* the exercise (using vertical space); 3) students' progress—how close

CHI '23, April 23–28, 2023, Hamburg, Germany

ACM ISBN 978-1-4503-9421-5/23/04…$15.00
https://doi.org/10.1145/3544548.3581516

they are to a correct solution (using horizontal space); and 4) how students' status changes over time. This is done by computing the semantic similarity and edit distance between students' code and solution code. Additionally, VizProg allows instructors to navigate the temporal and structural evolution of students' code at different levels of granularity, understand relationships between code, determine when to provide feedback, and assess who might need feedback the most.

We conducted a within-subject experiment to evaluate the effectiveness of VizProg. In a simulated live coding exercise setting, we found that compared to a baseline system, VizProg can help participants to 1) discover more than twice as many student misunderstandings, and 2) find the misunderstandings with less than half of the time and fewer interactions. Furthermore, participants reported that VizProg provides richer and more comprehensive information for identifying important student behaviors. This work can help instructors improve the live learning experience by better understanding students' mental models and providing tailored feedback at scale. This work makes the following contributions:

- A better understanding of the needs and challenges that instructors have when monitoring students' in-class coding exercise, based on interviews with programming instructors.
- A novel algorithm for representing students' progress in coding exercises as a 2D Euclidean spatial map that encodes their approach and progress towards a solution.
- VizProg, a system that builds on this algorithm to facilitate monitoring students' progress in real-time.
- Evidence showing that VizProg can help identify more misconceptions and important student behaviors in coding exercises than a baseline system.

2 RELATED WORK

VizProg is inspired by and primarily contributes to two research fields: programming education at scale, and source code visualization.

2.1 Programming Education at Scale

2.1.1 Understanding Students' Progress. Prior research has recognized the importance and difficulty of instructors understanding students' progress in programming exercises. For instance, Markel and Guo examined the step-by-step dynamic of one-on-one tutoring by undergraduate teaching assistants (TA) in a laboratory study [23]. Their research suggests that TAs' greatest difficulty is understanding students' mental models of course content. Further, early-stage students often have difficulty phrasing their questions clearly and make wrong assumptions about their problems, making it challenging for instructors to understand what they struggle with [23]. Wang et al. also conducted interviews with instructors and identified challenges they face when coordinating in-class programming exercises [38]. They describe how time and physical constraints make it difficult to observe students' progress while conducting in-class programming exercises. Due to the lack of understanding of students' backgrounds, it is also difficult to pair students for discussion by matching those with similar backgrounds. Our interview studies corroborate these findings.

Prior work has also proposed ways to help instructors better understand students' progress and thought processes. Kim et al. introduce RIMES [21], which supports authoring, recording, and reviewing interactive exercises in video lectures to give insights into students' thought process. RIMES was found to be useful in identifying and helping struggling students, as well as providing qualitative feedback to students [21]. Guo developed Codeopticon, an interface that enables instructors to get a real time view of students' actions by monitoring and chatting with dozens of students [16]. However, these tools are limited to small-scale sessions where instructors have the bandwidth to provide one-on-one feedback. VizProg instead proposes using a visualization approach to understand students' progress at scale. Its visualization is complementary to prior approaches and could be used in combination with them.

2.1.2 Providing Feedback At Scale. In order to generate feedback that scales to large introductory programming courses while still ensuring feedback is personalized enough to be helpful, instructors need to understand the variation among student solutions and what they struggle with. Markel and Guo discussed the difference between teaching generalizable knowledge and fixing bugs in introductory programming courses [23]. Teaching generalizable knowledge requires instructors to understand what knowledge each student comprehends and struggles with.

Researchers developed systems to help instructors understand students' solutions and provide feedback at scale. Nbgrader helps instructors generate feedback at scale by automatically generating a student version of Jupyter Notebook without solutions and grading assignments using notebooks executing results [1]. Overcode and Foobaz use the same clustering pipeline to generate feedback for correct student code solutions at scale [13, 14]. Autostyle uses clustering tools to broadcast actionable hints asynchronously regarding code style as well as the correctness and completeness of code solutions [25]. Singh et al. present a feature grammar to capture semantic relationships within programs and a supervised model to grade programming exercises in an independent manner [32]. Singh et al. also introduce a system using reference code and potential corrections to errors to automatically provide feedback for introductory programming problems [33]. Head et al. proposed MistakeBrowser and FixPropagator to generate feedback for incorrect solutions by clustering the transformation of fixing buggy programs [18]. Other research uses crowdsourcing to generate timely, customized feedback at scale. TutorASSIST provides on-demand assistance to students by crowdsourcing from teachers outside the classroom [27]. AXIS provides learners with crowd-sourced explanations on how to solve a problem from MTurk and allows learners to revise and evaluate them [40].

Most of the works mentioned above are designed to give asynchronous feedback [1, 13, 14, 18, 25, 33], but have not been applied to real-time feedback generation. PuzzleMe makes it easier for instructors to provide feedback at scale by using peer assessment, where students test and review peer's solutions [38]. Codeopticon helps instructors give students support in real time by watching students editing and debugging and chatting with them [16]. However, these tools are not meant to help instructors understand students' solutions at scale in large classrooms, as VizProg is designed

for. Codeopticon shows a list of tiles with every student's coding process and a chat box, which can be very messy at scale. The process of checking on each student's status to give feedback is time-consuming for instructors. Moreover, Codeopticon does not support instructors in understanding students' progress, since instructors focus on directly solving students' problems. It is challenging for instructors to get a general sense of the whole classroom using these tools. To overcome these problems, we designed VizProg, which visualized students' progress in a large classroom to help instructors understand issues and provide feedback in real-time.

2.1.3 Artificial Intelligence in Education. Artificial Intelligence (AI) has an increasingly important role in education [5]. Most of these systems aim to complement instructors by helping them scale their capabilities—for example, by producing immediate helpful responses to frequently asked questions [15], picking practice problems that are appropriate for a given student [7], and allowing instructors to create course-specific intelligent tutoring systems that give students hands-on problem solving guidance [39]. VizProg and our algorithms for representing code in 2D maps fit within the larger research area of AI in education. VizProg leverages AI to help instructors make more informed decisions while teaching. By better understanding which students are struggling, how many students are struggling, the problem solving approaches that students take, and the speed of progress, instructors can adapt their in-class exercises to be more responsive to students. For example, they might use this information to decide when to help individual students, to address common issues with the whole class, whether to extend the time given for an exercise, or how to group students into mixed teams for group exercises.

2.2 Code Visualization

2.2.1 Two-Dimensional Visualizations of Code. Prior work has explored ways to visualize code in two-dimensional space. Taniguchi et al. built a system that visualizes mutual edit distances between large groups of code [36]. They use these distances to compute high-dimensional vectors for every code sample in a larger set and use T-SNE [37] to reduce to two dimensions. There are two key limitations to this approach that VizProg aims to address. First, although there is a clear meaning to the *relative* positions of two points (closer means smaller edit distance), there is no clear human-understandable meaning for the *absolute* position of code locations. Thus, it can be difficult to tell if students are making progress. Second, there is no clear way to represent different *approaches* or the differences between approaches, as there is no semantic information included in the visualization.

Similarly, Huang et al. [19] mapped out semantic similarity between students' submissions in a Massive Open Online Course (MOOC). They used syntactic and functional similarity metrics to create their 2-D maps. However, again, this produces a visualization where there is meaning in the relative positions of code locations but no clear meaning in the absolute positions of code embeddings. Researchers have also used clustering methods to create visualizations of code without 2-D position meaning [14]. For instance, OverCode [14] visualizes a list of code clusters from correct student solutions, ordering them by cluster sizes. However, visualization without position meaning is insufficient for instructors to track and understand the students' progress in real-time, such as how students come up with a solution from scratch.

2.2.2 Clustering Submissions. The high variances in students' code and its high dimensionality make it difficult to interpret students' behaviors in a scalable manner. However, clustering students' code in real time may help reduce the number of submissions that instructors need to manually check. Researchers have explored approaches that combine visualization and clustering techniques to reduce the instructor's workload and the number of variations they have to handle. The ability to identify and cluster semantically similar submissions in a robust, general manner presents both an opportunity and a challenge. Earlier work clusters code submissions with Abstract Syntax Tree (AST) edit distance in order to evaluate syntax similarity and functional similarity [19]. Kaleeswaran et al. analyze data submissions on DP programming exercises by solution strategy, checking how students manipulate arrays in their solution [20]. The Codewebs project created a method for quickly determining semantically equivalent code snippets and allowing efficient indexing of all submissions within MOOC programming assignments [26]. Overcode uses both static and dynamic analysis to cluster similar, correct code submissions that perform the same computation, and provides a visualization to help instructors understand code solution variation [14]. Building on Overcode, Head et al. propose to cluster incorrect code solutions by transformation rather than clustering only correct solutions [18]. This helps instructors better understand students' bugs and create reusable feedback that scales to a large class. Piech et al. introduced a method to encode student programs as embeddings in neural networks and propose feedback generation at scale based on the clusters learned on the embedding space [28].

In addition to clustering tools, there is a series of tools that support comparison between programs. File comparison tools like Microsoft Win Diff highlights text that is different between files. Schleimer et al. proposes MOSS for finding similarities among student programs to detect plagiarism [31]. Taherkhani et al. use machine learning methods to identify sorting algorithm implementation [34, 35]. With the ability of clustering techniques to support generating feedback at scale, we provide process information that had been overlooked by previous clustering tools to make feedback tailored for students' problems while simultaneously supporting introductory programming courses at scale.

2.2.3 Real time code sharing. To support instructors observing students' progress in programming exercises, one challenge is to maximize the use of information on students' progress in real time code sharing. Real time code sharing between instructors and students offers many benefits to introductory programming courses. Prior research has shown that real-time code sharing could minimize context switching, facilitates knowledge sharing, lowers both student's cognitive load and instructor's teaching load, and improves student engagement in classes [3, 4, 17, 38]. Instructors share code in real-time in settings including MOOCs, lecture videos, online livestreams, and real classrooms [6]. Real time code sharing facilitates communication between students and instructors. Instructors broadcast programming activities to students, and students share their progress with instructors. Researchers have developed a series of tools to support real time code sharing in educational settings.

For instance, Chen and Guo developed Improv, which synchronizes code and output blocks with slides, therefore minimizing context switching and lowering cognitive load [4]. The Codestrates platform integrates code sharing into literate computing for collaboration on computational notebooks [29]. Borowski et al. use Codestrates to support real time code sharing among students in computational notebooks [2]. Codechella combines automated visualization of running states with real-time code sharing in online educational settings to enable learners to remember, comprehend and apply knowledge [17]. PuzzleMe combines peer assessment with real time code sharing where students share test cases and provide timely feedback to their peers, thus helping instructors create engaging introductory programming courses [38]. Byun et al. proposed CoCode, a visual program that shows students' code editors and output in real time to improve student's social presence for online courses [3]. While promising, this work is limited to small scale code sharing. VizProg shares all students' code at a keystroke level and visualizing them at scale in an easily interpretable way for instructors to analyze students' behaviors.

3 NEEDS AND CHALLENGES IN IN-CLASS CODING EXERCISES

We conducted interviews to better understand how instructors conduct and monitor in-class coding exercises. Our interviews allowed us to better understand how well existing practices and tools work. We recruited six participants (three self-identified as women, three as men) who had taught introductory programming classes in which they conducted in-class coding exercises. The classes the participants taught had more than 150 students. We found our participants through local mailing lists and personal connections. Participants had no prior knowledge of the purpose of the interviews. We asked participants about how they currently conduct in-class coding exercises, how they monitor and understand students' progress, how they provide feedback, when they move on, and how their future teaching strategies can be influenced by their students' performance. We summarize our key findings from these interviews as one need and three challenges below.

3.1 Need 1 (N1): Need to see students' coding progress in real-time

Four out of six participants (P1, P2, P4, P5) mentioned the importance of monitoring students' coding progress during in-class exercises. According to these participants, knowing the progress of the exercise can allow them to gain a more detailed understanding of their students' knowledge in the specific topics, provide more tailored feedback, and make better decisions regarding the exercise progression (e.g., how much more time to give students to complete their work). "So cannot give like infinite time for them to finish the exam. So if we get, we get a point that even though like no one solved that problem. They're still thinking, we'll just stop, try to solve it" (P5). Furthermore, understanding students' coding progress can help participants get feedback on their own teaching performance, and make plans for the remainder of the class. "it's important for like time allocation for the rest of the class" (P2). This illustrates how effectively understanding students' coding progress benefits both instructors and students.

3.2 Challenge 1 (C1): Understanding students' progress at different granularity

Participants reported their strategies for tracking students' progress both online and in person. For online settings, two participants used 'breakout rooms' (smaller virtual meetings that split students into groups) to group students for coding exercises (P1, P2). While conducting the exercises, teaching assistants will monitor the progress of students by jumping between rooms and observing or conversing with them. The instructors will then gather information regarding the performance of the students from these teaching assistants. For in-person settings, two participants stated that they would walk around and monitor individual students' computers or group discussions (P3, P4). Sometimes they ask students directly about their understanding or check to see if they have any questions. However, many students worry about what their classmates will think if they ask questions or otherwise reveal that they do not understand the material. Thus, our participants found that asking students directly might not be helpful, as students "sometimes pretend to understand to avoid looking 'stupid' in front of their peers" (P4). This indicates a need for instructors to monitor students' progress at various levels (e.g., the individual level, the group level) during in-class activities in an accurate manner.

3.3 Challenge 2 (C2): Inability to validate students' progress at scale

Our participants' opinions were split when asked how accurate they believe they are at understanding their students' progress. One third of the participants felt they had a good enough understanding of their students' progress, even if it was not necessarily very accurate (P1). Other participants are reluctant to claim a good understanding of students' progress. For example, P2 expressed that they understood "barely anything, I can only tell whether they're finished or not. [...] I cannot observe where they were stuck at." In light of this, instructors need a means of validating their understandings of student progress at a class scale.

3.4 Challenge 3 (C3): Scaling tailored feedback on progress is difficult

More than half of the respondents (4/6) said they sometimes did not have enough time to provide feedback after seeing issues during in-class exercises. "We actually don't have enough time to make sure everybody completes it" (P1). Typically, participants only had time to provide feedback to a small number of students, which is not scalable. Combining this finding with C2, our participants might also be spending their time with the students who need feedback the most. This indicates that instructors need a quick and efficient means of providing feedback to students as they progress through exercises. Further, it is important that they know who would benefit from feedback the most.

4 VIZPROG

4.1 System Design Goals

Led by prior work and our interviews with instructors, we developed three design goals (DG1-DG3) to guide the design of VizProg

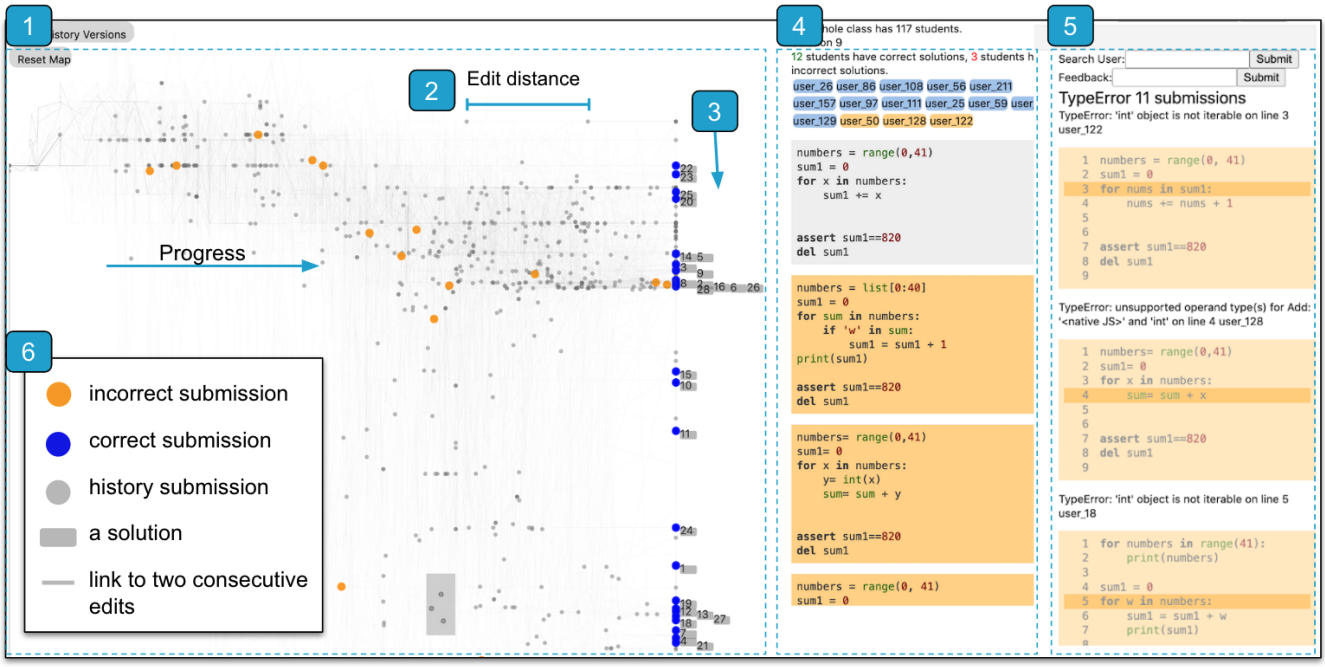


Figure 1: VizProg's User Interface. There are three main view panels: the overall class progress *2D map* view (1), a *solution-centered* view (4), and a *progress detailed* view (5). On the 2D map view, each dot represents a student's submission, each line between two dots indicates the edit movement. The x-axis encodes the size of a code edit to be proportional to the distance (2), and the y-axis represents different kinds of solutions for this exercise (3).

to help instructors monitor students' in-class exercise progress in real-time.

- **DG1: Easily view students' progress in real-time:** The system should provide students' progress in real-time so that instructors could observe students' current progress. In the context of in-class programming exercises, the closest we can get to real-time feedback is by providing feedback *as students are typing*.
- **DG2: Easily compare the difference between code.** Although participants in our interviews did not raise this issue specifically, we believe instructors would benefit from understanding students' *approaches* for solving the problem. This would allow instructors to see which solutions are commonly used, observe if any students solved the problem in an unusual way, and if students are solving the problem using the concepts they learned in class.
- **DG3: Ability to inspect and navigate students' progress at different granularity:** Instructors should be able to inspect students' progress at the level of individuals and as collective groups.

With these design goals in mind, we designed VizProg, a visualization system that allows instructors to navigate the temporal and structural evolution of students' code, understand relationships between code, and determine when to provide feedback to students. In the following sections, we describe VizProg's user interface and the algorithms used to realize its features in detail.

4.2 VizProg's User Interface

Figure 1 shows VizProg's user interface which consists of three main panels: the overall class progress ***2D map*** view (1), a ***solution-centered*** view (4), and a ***progress detailed*** view (5). As soon as a user starts VizProg, the system continuously monitors each student's code editor at a keystroke level. On the 2D map view, it uses a color-coded dot to indicate a student's code status (correctness) and a gray line to show how their status changes over time (Figure 1.6). As they progress through the coding exercise, the 2D map updates in real-time to always reflect their current status. Instructors can interact with VizProg during the exercise (Figure 1.4, 5) to track class-wide performance or individual progress. Additionally, VizProg provides a lightweight feedback feature that enables users to send text messages to individual students or to a group of students (Fig. 2.d). Below, we describe the user interface design for VizProg.

4.2.1 2D Map View: Overall Class Progress. To clearly convey when *progress* is being made—when a student's position changed (DG1), VizProg depicts progress by left-to-right motion (Fig. 1.1), since rightward movement is a common representation of progress (and there might be a strong psychological basis for this in other domains [10]). A gray line was used to connect two consecutive edits. Only when a student *submits* their code will a dot appear on the 2D map. Small gray dots represent historical code versions (meaning a student submitted that code but has since moved on). Larger dots

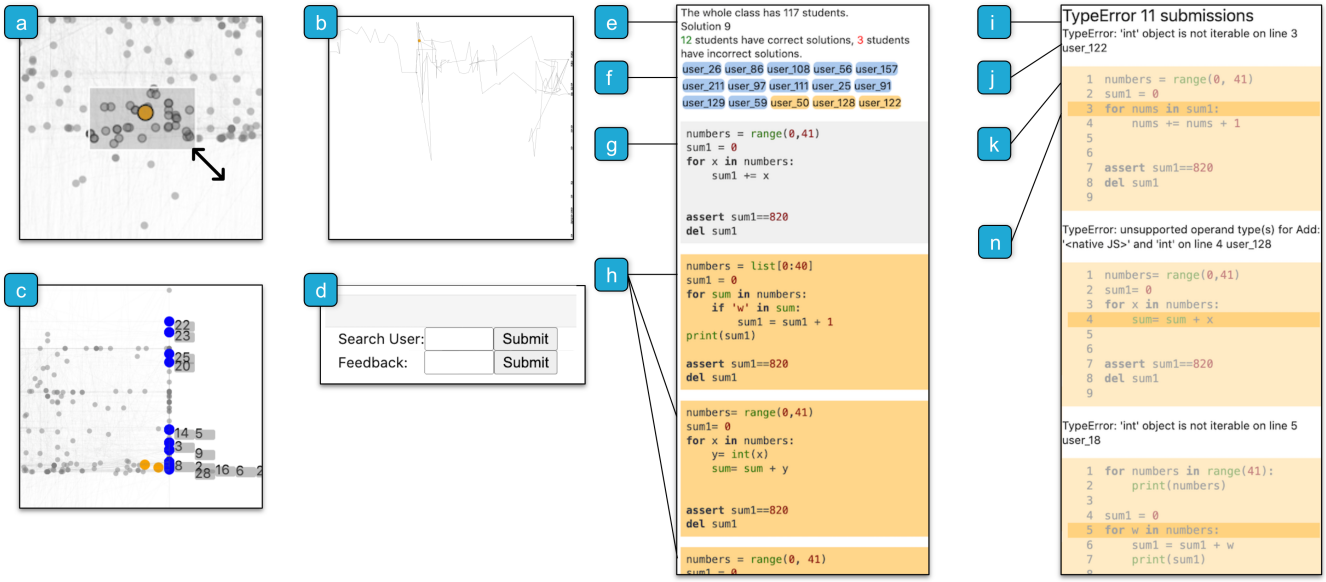


Figure 2: A detailed view of VizProg's user interface. Instructors can crop a region on the 2-D map to view solutions in that area (a). Cropped regions appear as gray rectangles surrounded by dots (a), and the progress detailed view shows statistics for the region (i, j, k, n). Correct solutions are displayed on the right of the map as blue dots with gray labels (c). When the instructor clicks a student name (f), the 2-D map will highlight this student's trajectory (b). The solution-centered view consists of statistics related to the solution (e), a list of students' IDs (f), and correct (g) and incorrect submissions (h) made by these students. Instructors can search for a specific student (d). They can also send feedback to either an individual student or a group of students (d).

represent students' *current* 'location'. Orange dots represent students who have not yet found a correct solution (as determined by the instructor's unit tests). Blue dots represent students who have found a correct solution (Fig. 1.6). We also used a gray rectangle to indicate one type of solution (Fig. 1.3).

To ensure these updates are done in real-time, VizProg computes locations over intermediate code edits rather than submissions. These edits often include syntax errors, which makes many prior techniques that rely on building Abstract Syntax Trees (ASTs) [14, 26] not feasible to apply. VizProg instead uses transformer-based code vectorization [11], which can encode code semantics even when there are syntax errors (as we will describe later). VizProg is designed to make the generated space to *continuous* and *proportional* to the size of code edits—the size of a code edit should be proportional to the distance moved in 2-D space (Fig. 1.2). That is, if the Euclidean distance between dot_a and dot_b is shorter than that between dot_b and dot_c, then student_a and student_b should have a more similar solution than that of student_b and student_c. In addition, instructors should expect a similar coding pattern when inspecting submissions that are close to each other. In this manner, instructors can identify submissions that are far from being correct.

To help instructors understand the variety of students' solutions—to identify which solutions might be common and which might be abnormal (DG2), VizProg uses vertical space to represent different kinds of solutions (Fig. 1.3). We applied the same clustering algorithm that we used for individual submissions to display similar solutions vertically closer to each other.

4.2.2 Solution-Centered View: Students End With A Solution. To inspect the submissions from all the students who arrived at the same solution (DG3), instructors can click a solution (Fig. 1.3) and see the solution-centered view (Fig. 1.4). The list contains three main sections: statistics related to the solution (Fig. 2.e), a list of IDs of students who are close to (or found) this solution (Fig. 2.f), and a list of submissions made by these students (Fig. 2.h). This view also summarizes the number of correct and incorrect submissions that are approaching this solution. The list of student IDs is color coded to represent correctness. The instructor can click each student ID to see the trajectory of the selected student's submissions on the 2D map view. The progress detailed view also displays all the submission made by this student throughout the history of the exercise (Fig. 1.5).

4.2.3 Progress Detailed View: A Selected Submission(s) View. To allow users to navigate students' progress at different granularities, VizProg lets users examine the code progress of both groups and individuals at the code level (DG3). For group progress, instructors can crop a region on the 2D map to see the submissions only within that region (Fig. 2.a). When a region is selected, the area on the map will be a gray rectangle surrounding multiple dots. As long as the selected region has at least one submission, the 2D map view will hide all the dots outside of the region, and the progress detailed

view will also display only the selected submission code (Fig. 1.5). To assist users in identifying common misconceptions, VizProg lists these submissions by error type frequency in descending order (Fig. 2.i)[1]. Furthermore, VizProg color codes each submission by its correctness, where orange indicates an incorrect submission and gray indicates a correct submission (Fig. 2.n). VizProg also displays the error message (Fig. 2.j) and highlights the line of code that caused the error when there is an error in the submission (Fig. 2.k). By resizing the overlay or dragging the overlay on the 2D map, the user can view real-time updates on the progress detailed view of the selected region (Fig. 2.a). For individual progress, users can either search by student ID (Fig. 1.5), or click a student ID on the solution-centered view (Fig. 1.4). This student's progress will also be represented by a trajectory line on the 2D map (Fig. 2.b). After cropping a region or selecting submissions of a student, instructor can use the lightweight feedback feature (Fig. 2.d) to send feedback to the students that are selected.

4.2.4 VizProg visualization compared to alternatives. We compare the visualization of VizProg to alternative tools in Table 1. We choose OverCode [14] and Codeopticon [16] as alternatives, which provide the state-of-the-art support for instructors to view students' solutions in programming courses. The comparison is based on the four aspects listed in Table 1. First, VizProg and Codeopticon provide dynamic visualizations that update in real-time for instructors to monitor students' progress, while OverCode analyzes students' final solutions that are correct without regard to how they come up with the solutions. Second, the visualization of VizProg and OverCode is more concise than Codeopticon's. VizProg encodes students' progress into a 2-D map, where instructors can view hundreds of students' progress in a single page without scrolling. By displaying clusters of student solutions, OverCode saves space by eliminating solutions with the same computation but different variable names. In Codeopticon, each learner's progress is summarized in a tile, and the instructors interact with a dashboard consisting of a list of tiles. As a result, Codeopticon is not suitable for large programming courses. Third, VizProg and Codeopticon visualize solutions from all students, while OverCode visualize only solutions that can be executed without syntax errors. Fourth, VizProg summarizes editing history using 2-D trajectories, whereas OverCode and Codeopticon do not. OverCode displays only final submissions. Codeopticon shows code edits as diffs, but does not summarize editing history.

4.3 VizProg's Algorithm

4.3.1 Naïve Approaches. The easiest approach would be to compute a code vector (using CodeBERT [11] or similar tools) and perform dimensionality reduction (using T-SNE [37] or similar algorithms) to reduce each code sample to two dimensions that can be displayed to instructors. However, this approach has two important downsides. First, we found that small edits can result in disproportionately large "jumps" in 2-D space by using vector embeddings alone. Second, this approach does not represent data points in a way that are necessarily intuitive; it is difficult to infer whether a student is close to a solution from their position alone. Finally, depending on the approach for dimensionality reduction, the lower-dimensional embeddings might need to be re-computed frequently, which is too computationally expensive for real-time updates.

4.3.2 Normalizing Code. We refer to a given piece of code as c, a string of characters. $c_{s,t}$ refers to the code of student s at time t. We will use is_correct(c) $\in \{true, false\}$ to represent if a solution c is correct (is_correct(c) = $true$) or incorrect (is_correct(c) = $false$), as determined by unit tests. We will use only_correct(C), where C is a set of code samples, to represent the subset of C where is_correct = $true$. This means that: only_correct(C) $\subseteq C$.

We rely on two separate similarity metrics to determine how to represent a student with code c in VizProg: edit distance and vector similarity (both described below). However, neither of these similarity metrics account for differences that have no functional meaning to the Python interpreter, such as differences in variable names, comments, and spacing. For example, both similarity metrics would determine that the following pieces of code are different, even though they are functionally nearly identical (the only difference being that the code sample on the right prints `'Done!'`):

```
  my_variable = 10
  my_dictionary = {}


  for key, value in my_dictionary.items():
      other_value = value + 1
      print(key, other_value)
```

```
v = 10
d = {}

# loop over all of the items in d
for k, v in d.items():
    w = v + 1   # add 1 to the value
    print(k, w)
print('Done!')
```

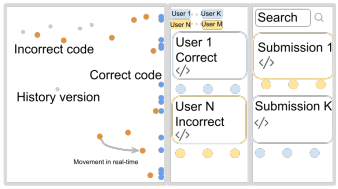
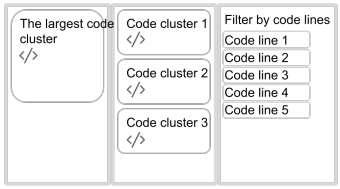
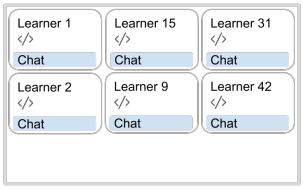
Prior work has accounted for this by computing the Abstract Syntax Trees (ASTs) of both samples and modifying variable names between code samples to match [14]. However, this approach relies on building an AST, which is typically not possible in the presence of syntax errors. As we discuss above, we designed VizProg to work with code that has syntax errors. VizProg instead relies on text-based normalization, which attempts to normalize code by performing string-level operations, using regular expressions. Our normalization performs the following:

- Removes extra spacing (newline characters, `\n`) in the code
- Removes code comments
- Detects variable names, as defined through assignment (e.g., `varname = . . .`) or implicit declaration (e.g., `for varname in . . . :`).
- Removes calls to the `print()` function, as these are often used by students to debug their code and are not typically part of the problem definition

For example, VizProg's normalization on the above code samples would produce:

[1]Failing the unit tests for the problem also produces a runtime error.

Table 1: We compared the visualization of VizProg to OverCode [14] and Codeopticon [16] based on four features. A "Dynamic" visualization is one that updates in real-time. "Concise" means that the information can be read in a single page without having to scroll. The term "Represent All" indicates whether it displays all the students in the class. "Summary History" refers to whether history editing is summarized.

Features	VizProg (this work)	OverCode [14]	Codeopticon [16]
			
Dynamic	✓	×	✓
Concise	✓	✓	×
Represent All	✓	×	✓
Summarize History	✓	×	×

```
v0 = 10
v1 = {}

for v2, v3 in v1.items():
    v4 = v3 + 1
```

We refer to the normalized version of code c as norm(c). Our normalization method has several drawbacks. First, it could result in small changes producing large semantic changes. For example, if a student s as code at time t $c_{s,t}$ and at time $t+1$, they add a '#' to comment out some portion of code, the distance between $norm(c_{s,t})$ and $norm(c_{s,t+1})$ could be large. Second, there are still several non-functional changes that it does not account for. For example, changing the order of declaration of `v0` and `v1` in the above code makes no functional difference to the code execution but is not accounted for in our normalization technique. Still, we have found that these issues have a small impact on our underlying algorithm. One of the reasons we used short variable names like `v0` is that there is a relatively small cost for naming mistakes; for example, the edit distance between `'v0'` and `'v5'` is small. However, future work could further improve our normalization method to account for these challenges.

We divide our discussion into our techniques for determining students' *approach* and their *progress*.

4.3.3 Representing Students' Problem-Solving Approaches in VizProg. We represent students' approach on the y-axis and we use the *vector* similarity between a students' solution and existing solutions to determine which approach they are using.

Vector Similarity The first distance metric that VizProg uses is *vector similarity*. VizProg leverages CodeBERT [11], a pre-trained transformer model capable of representing code, to convert code into a vector (with 768 dimensions by default). vec(c) $\in \mathbb{R}^{768}$ represents the vectorized version of code c, as computed by CodeBERT. We can compute the vector similarity of two different code samples c_1 and c_2 using the cosine similarity, after normalizing the code samples (using the normalization technique described above):

$$\text{vec\_sim}(c_1, c_2) := \frac{\text{vec}(\text{norm}(c_1)) \cdot \text{vec}(\text{norm}(c_2))}{|\text{vec}(\text{norm}(c_1))||\text{vec}(\text{norm}(c_2))|}$$

This produces a single number in the range $[-1, 1]$ where higher numbers represent higher similarity. In practice, this vector similarity tends to be very close to 1 when comparing code samples for the same exercise, even when comparing different approaches to the same problem (empirically, in the range $[0.96, 1.0]$).

Building a Solution Space In order to build a Euclidean space for code solutions to a given problem, VizProg first needs a pre-existing set of prior solutions. In practice, these prior solutions might come from previous class sessions, previous semesters, instructor-written solutions, or could be collected after some subset of students has completed the exercise. The source of prior solutions may affect the solution space. Ideally, solution sets should be seeded from a source that contains a diverse and comprehensive set of approaches to solving the problem. We will discuss the problem of seeding VizProg in more detail in section 4.3.6. We denote the set of prior solutions as PAST_CODE $= \{p_1, p_2, \cdots, p_{n_{past}}\}$, where there are n_{past} prior code examples. Ideally, PAST_CODE should contain several examples of correct solutions (is_correct(p) $=$ *true* for some $p \in$ PAST_CODE) but typically should contain a mixture of correct and incorrect solutions.

We first build a matrix P containing the vector representation of every item p_n in PAST_CODE (after normalizing the code):

$$P = \begin{bmatrix} | & | & & | \\ \text{vec}(\text{norm}(p_1)) & \text{vec}(\text{norm}(p_2)) & \cdots & \text{vec}(\text{norm}(p_{n_{past}})) \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{n_{past} \times 768}$$

We then reduce P from 768 rows to 1 row, first using Principal Component Analysis (PCA) (to reduce from $(n_{past} \times 768)$ to $(n_{past} \times 40)$) and then T-SNE [37] (to reduce from $(n_{past} \times 40)$ to $(n_{past} \times 1)$). This reduces P to a single vector, which we call $\vec{y}$ =T-SNE(PCA(P, 40), 1) $\in \mathbb{R}^{n_{past}}$, because we will use it to compute the vertical (y)

position of students' code. $\vec{y_p} \in \mathbb{R}$ denotes the position of prior code sample p. We go through this process in order to distinguish between solutions p_i and p_j that are very similar ($\vec{y_{p_i}} \approx \vec{y_{p_j}}$) or different ($\vec{y_{p_i}} \not\approx \vec{y_{p_j}}$).

In addition, we use OverCode [14] to cluster similar correct solutions from PAST_CODE more robustly. A cluster in OverCode [14] is a set of correct solutions that perform the same computation. For a given problem, we get distinct solution clusters, which we use to label correct solutions along the y-axis in VizProg (Fig. 1.3).

Encoding Approach To determine which approach students are attempting to use, we use the vector similarity between students' solutions and prior solutions (all after normalizing the code). For a student's code c we first select the n_{vec_sim} prior solutions in PAST_CODE that are correct and most similar to c and store the result in NEAR_APPROACH. Formally, this is:

$$\text{NEAR\_APPROACH}(c) = \underset{PC \subseteq \text{only\_correct}(\text{PAST\_CODE}), |PC| = n_{vec\_sim}}{\arg\max} \left(\sum_{p \in PC} \text{vec\_sim}(c, p) \right)$$

In Python code, this could be computed as (assuming `c` is defined as the current code sample):

```
NEAR_APPROACH = sorted(filter(is_correct, PAST_CODE),
    key=lambda p: vec_sim(c, p))[n_vec_sim:]
```

This produces the subset of PAST_CODE with most semantically similar correct solutions. Smaller values of n_{vec_sim} produce movement that better reflects the solution that a given code sample is closest to but it can result in frequent vertical jumps as the closest solution changes. Larger values of n_{vec_sim} produce movement over time that is smoother but can be less accurate. VizProg uses $n_{vec\_sim} = 10$.

We then compute the y position of code c as the weighted average of these similar solutions:

$$\text{y\_position}(c) := \sum_{n \in \text{NEAR\_APPROACH}(c)} \vec{y_n} \cdot \text{softmax}(\text{vec\_sim}(c, p_n)^3)$$

Where $\vec{y_n} \in \mathbb{R}$ represents the y position of code n (as computed above). We cube the vector similarity to better differentiate between several similarities that are close to 1, while preserving the sign of the vector similarity.

4.3.4 Representing Students' Progress in VizProg. The second distance metric that VizProg uses is *edit distance*. We represent students' progress on the x-axis and we use the edit distance to determine how far they are from a correct solution.

Computing Edit Distance We use the normalized Levenshtein edit distance [22] ('levenshtein(a, b)' denotes the distance between a and b) to determine the edit distance between code samples:

$$\text{edit\_distance}(c_1, c_2) := \frac{\text{levenshtein}(\text{norm}(c_1), \text{norm}(c_2))}{\max(\text{len}(\text{norm}(c_1)), \text{len}(\text{norm}(c_2)))}$$

where $\text{len}(c)$ represents the number of characters in c (a positive integer $\in \mathbb{N}$) and $\max(a, b)$ represents a if $a \geq b$ and b otherwise. We normalize (divide by the maximum length code sequence) in order to avoid disproportionately long or short solutions or submission from overly influencing the edit distance. Thus, $\text{edit\_distance}(c_1, c_2)$ always returns a positive number between $[0, 1]$ where 0 would mean c_1 and c_2 are functionally identical (small edit distance).

Encoding Progress To determine how close students are to a correct solution, we use the edit distance between students' solutions and prior solutions (all after normalizing the code). We first find the n_{edit_sim} closest solutions by edit distance. VizProg uses $n_{edit\_sim} = 10$. Formally:

$$\text{NEAR\_EDIT}(c) = \underset{PC \subseteq \text{only\_correct}(\text{PAST\_CODE}), |PC| = n_{edit\_sim}}{\arg\min} \left(\sum_{p \in PC} \text{edit\_distance}(c, p) \right)$$

In Python code, this could be computed as (assuming `c` is defined as the current code sample):

```
NEAR_EDIT = sorted(filter(is_correct, PAST_CODE),
    key=lambda p: edit_distance(c, p))[:n_edit_sim]
```

If solution c is correct (passes the instructor's unit tests) then we assign its x position to 0. If it is not correct, we compute the x position as the average edit distance for items in NEAR_EDIT:

$$\text{x\_position}(c) := \begin{cases} 0, & \text{if } correct(c) = true \\ \underset{p \in \text{NEAR\_EDIT}(c)}{\text{average}} \; (-1 \cdot \text{edit\_distance}(c, p)), & \text{otherwise} \end{cases}$$

Where 'average' represents the arithmetic mean. Note that we negate the edit distances, meaning $\text{x\_position}(c)$ is always ≤ 0 and larger numbers signify that c is *more similar* to existing solutions.

4.3.5 Computational Efficiency and Displaying Progress in Real-Time. The process of building a solution space is computationally expensive but only needs to be done once before the instructor begins an exercise. After $\vec{y}$ has been computed, we can use it to quickly compute $\text{x\_position}(c)$ and $\text{y\_position}(c)$ for any student code c at little computational cost. Computing the position of $c_{s,t}$ requires doing a forward pass of $c_{s,t}$ through the CodeBERT transformer, building $\text{NEAR\_APPROACH}(c_{s,t})$, taking the weighted sum to compute y_position, computing $\text{NEAR\_EDIT}(c_{s,t})$, and taking the weighted sum to compute x_position. Of these operations, the most computationally expensive is the forward pass through CodeBERT, which executes almost instantly on any modern GPU.

4.3.6 Seeding VizProg with good sets of solutions. Instructor-written solutions may only represent a subset of possible student solutions. Using solution sets that lack diversity in code could make the 2-D solution space less meaningful. Mapping instructor-written solutions directly to vertical positions on the map could exclude the diverse approaches that students take. It is likely the calculated vertical positions of some student solutions are outside the solution space. When visualizing these solutions on the 2-D map, the corresponding dots will be displayed at the edges of the map. Therefore, to build a solution space that would have meaningful results, users should collect a solution set that has diverse code in it, such as students' solutions from previous semesters.

4.3.7 Limitations. There are several limitations of our approach. First, it requires an existing repository of attempted solutions to build a 2-D solution space. However, in future work, this solution

space could instead be built as users complete their code without requiring any prior submissions. The primary challenge of doing this is that it would require re-building the P matrix and re-performing the steps for dimensionality reduction as PAST_CODE grows. This could be optimized by using incremental dimensionality reduction [12] rather than re-doing the process from scratch. This way, most of the computational cost would determined by the amount of new data, rather than the total size of PAST_CODE. We could also perform these steps asynchronously to ensure that there is no delay when rendering the visualization. We could also animate changes to the 2D solution space to make it easier for instructors to follow the visualization as it evolves. Second, VizProg relies on code to assess students' progress but progress is not necessarily visible from the code alone. For example, students might write commented pseudo-code to describe the algorithm they plan to use for a problem before they start writing code. VizProg does not recognize or represent this kind of progress in its visualization. If a student is falling behind in VizProg, this is a sign that instructors should check in on the student—not definitive evidence that they are struggling. This means that VizProg can help instructors determine who to help but still requires instructors to use their judgement.

5 USER STUDY

We conducted a within-subject study to evaluate the effectiveness of VizProg for identifying students' problems. In the study, we provided participants with a replay of pre-recorded authentic examples of students solving a programming problem, and asked them to answer quiz questions on students' errors and progress. As our 'baseline' system, we used OverCode [14], with two augmentations described in Section 5.1.4. We chose OverCode as a baseline because it is a state-of-the-art tool, it is open-source, and it shared a similar design goal with ours.

5.1 Method

5.1.1 Recruitment. Because the target users of VizProg are instructors, we primarily recruited participants with experience teaching programming courses. We reached out senior students from the *[redacted for anonymity]* and *[redacted for anonymity]* programs at *[redacted for anonymity]*. In the screening session, participants indicated their prior experience teaching Python. Qualified participants were experienced Python programmers, including teaching assistants, tutors, and senior students who have taken advanced Python programming courses before. We recruited 16 participants (10 self-identified as male, 6 as female) from a local participant pool. All 16 participants participated in the first session, and 15 participants participated in the second session. As Table 2 shows, participants have experience in Python programming varies from 1 to more than 6 years. Among these participants, 14 had experience teaching programming courses.

5.1.2 Live Simulation. In order to ensure the data we used in the study were authentic, we used data collected from a large introductory programming course at *[redacted for anonymity]*. The course size ranges from 130–190 students. Our data were collected from an exercise within an interactive Python textbook used by the course. The data represented students' attempts at solving exercises on their own time (rather than during time-limited class exercises) but they contained genuine examples of misunderstandings and challenges that students faced when attempting the exercises. We first collected students' submissions for 100 programming exercises from the course. We filtered the dataset by the number of students who submitted solutions to the exercise, and the number of submissions made per students. We ended up getting 69 programming exercises which have more than 100 students' submissions and each student have more than 2 submissions in average. We chose two programming exercises from the filtered dataset, one for each session, that were roughly equivalent in terms of complexity:

Exercise 1 (E1): Provided is a string saved to the variable `s1`. Create a dictionary named `counts` that contains each letter in `s1` and the number of times it occurs.

Exercise 2 (E2): Create a list of numbers `0` through `40` and assign this list to the variable `numbers`. Then, accumulate the total of the list's values and assign that sum to the variable `sum1`.

E1 had 627 Python code snippets from 109 students. E2 had 823 Python code snippets from 117 students. The solutions varied from 2 lines to 20 lines of code. The submission time ranges from a few minutes to several days. We trim the submission by setting a time threshold and them normalized the time to a 15 minute time window. We also checked each submission to ensure that it did not contain any identifying information or present any privacy concerns and anonymized appropriately[2].

The data captured contained a snapshot of every *submission* that students made (every time they ran the code). However, we want our evaluation to work with keystroke-level data. To maintain the setting realism and ensure participants' experience quality, we generated synthetic keystroke-level data from the submissions to simulate students' typing activities. For each submission, we

Table 2: For the user study, we recruited 16 teaching assistants, tutors, instructors, and senior students who are experienced in Python programming.

PID	Gender	Teaching Exp.	Python Prog. Exp.
P1	Female	Tutor	2 Years
P2	Male	Teaching Assistant	3 Years
P3	Male	Teaching Assistant	1 Year
P4	Male	Teaching Assistant	6+ Years
P5	Female	Teaching Assistant	3 Years
P6	Female	Teaching Assistant	3 Years
P7	Male	Teaching Assistant	2 Years
P8	Male	Teaching Assistant	1 Year
P9	Male	Teaching Assistant	5 Years
P10	Female	Teaching Assistant	4 Years
P11	Male	Instructor	1 Year
P12	Male	Teaching Assistant	2 Years
P13	Female	Teaching Assistant	6 Years
P14	Male	None	4 Years
P15	Female	Teaching Assistant	3 Years
P16	Male	None	2 Years

[2]In our examples, there was no identifying information contained in code. In other examples, students might use their given name as a variable name or output their name in their code.

compared it with the most recent previous submission, calculated the string difference between them. For each addition and deletion in the difference, we split it into character level editing activities. With the keystroke-level data, participants observed students consistently changing from one submission to the next submission character by character, rather than sudden jumps in the solution space.

Finally, we computed our visualization in a way that the visualization of $c_{s,t}$ could never depend on student s's code after time t (no forward dependencies). This means that the trajectory for each student is what would be generated if that student's solutions were embedded in a space generated from the rest of the solutions, a setup conceptually similar to cross-validation.

5.1.3 Study Setup. Our evaluation was within-subjects, where participants joined two sessions—one with the baseline system and one with VizProg. We counterbalanced the order of the systems and tasks. We provided 15 minutes of training on how to use each system. To ensure participants get enough practice of using the system, we provided an example using scenario for users to explore the user interface. In the practice example, participants watched a replay of 20 students solving a programming problem and we asked them to perform some exercises using the system.

After training, participants began the study. Participants watched a replay of students solving a programming exercise for 15 minutes (109 students for E1 and 117 students for E2). During the replay, participants were asked to use the system to answer quiz questions around students' errors and progress. After the replay, participants had 20 minutes to finish the quiz questions based on the final results of the replay. After each session, participants were asked to complete a survey regarding their experience of using the system. After the second session (with the same procedure but a different system), we conducted a reflective interview for comparing the systems.

We conducted this study remotely using Zoom, and each session lasted about 60 minutes. We recorded the screencast of them performing the task, their answers to the survey, and the audio of their think aloud process and their answers to our follow up interview. We compensated each participant with a $25 USD Amazon Gift Card for each session.

5.1.4 Baseline. We chose OverCode [14] as the baseline, which clusters correct code submissions by computational results. OverCode is an effective tool for understanding submissions at scale. However, OverCode [14] is not designed for live settings. To make the comparison fairer (not biased in favor of VizProg), We developed a Jupyter Lab extension that updated OverCode results in real-time. Both the baseline system and VizProg are implemented as Jupyter Lab extensions. During the study, participants used Jupyter Lab to watch the replay. For participants who used OverCode [14], we also provided the original OverCode user interface after the replay finished.

5.1.5 Data Collection. In the screening session, we collected data on participants' teaching experience and Python programming experience. For each session, one member of the research team was present. We created a list of code scheme of behaviors observed from the study. For students' answers to the quiz questions, one member of the research team graded the correctness of the answers. At the end of each session, we asked participants to fill out a survey and compared the two session's results. After the second session, we conducted interviews with participants, where we asked participants to compare the baseline system and VizProg. We worded our questions in a way that tried to elicit more honest feedback by not revealing which system was the 'control'.

5.2 Results

The quiz was designed as multiple-choice questions and open-ended questions. We graded participants' answers to each multiple-choice question. One member of the research team created a list of correct answers to the quiz questions based on the replay. To grade participants' answers, we calculated

$$\frac{\text{the number of matched answers}}{max(\text{total number of correct answers, total number of selected answers})}$$

to work out their grades for the quiz. We also coded the screen recordings to analyse the time spent on each quiz question. We used a two-tailed Welch's t-test to determine significance for our statistical analysis. For the open-ended questions in the quiz, we analysed the screen recordings to understand how participants interact with the tool to perform the tasks.

5.2.1 Participants understand students' problems more accurately using VizProg than the baseline. In the first session, we designed the quiz questions as multiple-choice questions, which were more direct and required users to find the information about specific students. In the second session, we designed the quiz questions to be more open-ended, which encouraged participants to explore the system and interact with the whole dataset. For the multiple-choice questions, We found that participants' accuracy with VizProg ($\mu = 79.6\%$, $\sigma = 0.1$) is significantly higher than participants' grade with the baseline system ($\mu = 51.4\%$, $\sigma = 0.2$, $p < 0.0001$).

In the open-ended questions, we asked participants to use the system to find common misunderstandings of the whole class. We coded the misunderstandings participants found during the study. Comparing to the list of existing misunderstandings generated by the researcher, we calculated the number of valid misunderstandings participants mentioned. As shown in Table 3, the valid misunderstandings participants found using VizProg ($\mu = 4.5$, $\sigma = 1.5$) is significantly more than what they found using the baseline system ($\mu = 2.4$, $\sigma = 0.5$, $p < 0.01$). We listed the misunderstandings participants found in two conditions in Table 3. In the control condition, 4 out of 7 participants described misunderstandings in a general way, using terms including "Name Error", "Type Error" and "Syntax Error". In VizProg, 6 out of 8 participants described misunderstandings more specifically by pointing out the parts that made the solution incorrect.

5.2.2 VizProg helps participants understand issues faster in live settings than the baseline. To investigate how the two systems help participants understand students' problems in live settings, we calculated 1) when participants started finding errors, and 2) how much time they spent to find students' errors. We found participants using the baseline system started identifying errors significantly later than participants using VizProg ($p < 0.05$). In the baseline system, participants started finding errors 1069.9 seconds after

Table 3: Common misunderstandings participants found in the second session. The third column lists all the misunderstandings per participant, and the fourth column calculates the total number of the misunderstandings the participant identified in the study.

PID	Condition	Misunderstandings	Count
P3	Baseline	Use wrong variable in summation, Range excludes parameter "end"	2
P4	Baseline	Name Error, Syntax Error, Use wrong variable in summation	3
P5	Baseline	Use variable not defined, Do not know how to create a new list	2
P6	Baseline	Type Error, Name Error, Use wrong variable in summation	3
P11	Baseline	Type Error, Name Error	2
P14	Baseline	Name Error, Do not know how to create a new list, Range excludes parameter "end"	3
P15	Baseline	Use variable not defined, Use wrong variable in summation	2
P1	VizProg	Type Error, Syntax Error, Use variable not defined, Iterate on int object, Use wrong variable in summation	5
P2	VizProg	VizProg Iterate on int object, Use wrong variable in summation, General usage of range, Return value of range	4
P7	VizProg	Use wrong variable in summation, Use append method in an incorrect way, Initialize range	3
P9	VizProg	Type Error, Name Error, Key Error	3
P10	VizProg	Use variable not defined, For loop on item not iterable, Use append method in an incorrect way, Add up array and int, Use methods that do not exist, General usage of range	6
P12	VizProg	Use variable not defined, Use wrong variable in summation, Hard code, Incomplete expression, Did not add numbers in loop	5
P13	VizProg	Use variable not defined, Iterate on int object, Use wrong variable in summation, Use append method in an incorrect way, Add up array and int, Index on variable that does not support indexing, Miss right bracket on print	7
P16	VizProg	Use variable not defined, Do not know how to create a new list, Range excludes parameter 'end'	3

the replay starts in average (σ = 360.2). In VizProg, participants started finding errors 500.1 seconds after the replay starts in average (σ = 468.2). As the replay lasted 15 minutes, this indicated that with VizProg, people are able to understand students' problems synchronously with students working on the problem, while in the baseline system, people tended to wait until students finished the exercise. We did not find significant difference of how much time they spent on finding errors between two conditions (VizProg μ = 261.5 seconds, baseline μ = 214.9 seconds, $p > 0.05$)

5.3 System Usability and Study Insights

To better understand VizProg's usability benefits or issues, we ran a thematic analysis on the interview transcripts with our own observations of participants' behavior patterns from the video.

5.3.1 VizProg helps participants understand in live settings with less context switching. As we showed in Section 5.2, participants can understand students' problems faster and more accurately using VizProg than the baseline system. Based on our observation, we found that using VizProg takes participants less context switch to understand errors in live settings. In VizProg, instructors can quickly understand errors of the whole classroom by watching the 2-D map instead of checking every students' editing history. In both sessions, we pointed participants to the most popular solution, and asked participants to find out errors for that solution. In the control condition, 8 participants needed to look at individual students' submissions to understand what were the misunderstandings they had. 3 participants (P1, P6, P10) went through every student in that group and spent more than 8 minutes looking at their submissions. Given the large number of students in the class, 12 out of 15 participants randomly selected a few students from the group and checked their history versions.

While in the treatment condition (with VizProg), 15 out of 16 participants used the 2D Euclidean map to interact with the whole class's submission. They brushed and selected an area on the map and checked in on different error types on the right side of the tool. Participants found VizProg helpful because similar submissions are grouped in a small area (P2, P9, P10, P14, P16) and incorrect submissions are grouped by error types(P1-3, P5, P7, P9-12, P14-16). VizProg also enables participants to understand students' problems synchronously as students are working on the problem, while in the control condition, participants need to wait until they finish(P3-6, P14-15).

5.3.2 Visualizing progress on a 2D map takes participants less effort to validate students' progress. In the second session, we asked participants to find students who did not finish the programming problem. Among these students, participants were asked to decide

who were close to a correct solution and who needed more help. In the control condition, participants looked at the history versions of all students that did not have a correct solution, and then decided whether they need further help. In the treatment condition, participants checked the trajectory on the map for each student that had an incorrect submission at the end. Participants found that some students "were very close to the correct solutions on the map, I think they only need to replace the variable name (P9-10, P15)," while some other students "they were very far away from the map, I think they don't understand the concepts and would talk to them (P9)." Participants also noticed that "they actually already reach the correct solution but they went back changing to a different approach, and never get it correct later, I don't understand what's going on. (P12)"

"... The whole movement from left to right is basically telling me the progress the student is making towards the right answer, whereas in OverCode it was more like just set of blocks, and there was no indication whether the final one is the right or not. Um! It was just showing me the dotted saying that this is the current portion of it. But as this was more easier, even just from the graph aspect as well... (P1)"

5.3.3 VizProg enables users to quickly form a strategy to decide who to give feedback to. In the second session, we asked participants to first find students that need help, and then give feedback to these students. We observed different behavior between two conditions for giving feedback to the students.

In the control condition, participants looked at all the final submissions and found there were 18 students who did not have a correct solution when the simulation stopped. They decided that all the 18 students need help. 2 out 7 participants went over every student's final submission and generated feedback based on the errors in each submission. 5 participants randomly looked at some of the students' final submission and gave feedback to them, skipped the other students.

In the treatment condition, participants first searched for dots that were out of place in the 2D map. Participants then brushed and selected areas on the map, and found that dots that are close to each other had similar misconceptions, so they decided to give the same feedback to them. Participants also found that dots on the far left side of the map had submissions that were very far away from a correct solution, while dots on the right side of the map had submissions only needed a few edits to become a correct solution. 5 out of 8 participants decided to give more detailed feedback for the dots on the far left side and talk to the students, and guide the dots on the right side to a correct solution that is close on the map.

This indicated that in the control condition, participants lacked a strategy in giving feedback and made decisions based on randomly looking at students' code. In the treatment condition, participants quickly formed a strategy of using the map to generate feedback at different granularity. Using VizProg, participants can use the visual guidance on the map to give feedback at various levels (i.e., the individual level, the group level) shortly.

5.3.4 Visualizing progress at scale can still be overwhelming. Despite of the benefits mentioned above, 4 participants (P4, P6, P12-13) found the 2D map in VizProg overwhelming when all the dots started to move.

"...I had to select a particular set of students. I had to drag and drop on the graph, and I wasn't entirely sure how that was going, plus the the whole thing about the student moving from one solution to another... (P4)"

Even in the control condition where we use conventional code editors instead of a 2D map, participants still found it overwhelming when many students started working on their code (P2, P9-10, P12, P15).

Although we followed the rationale that students moving from left to the right means that they are moving from incorrect solution to a correct solution, the map does not have explicit semantic meaning for the position. P1 mentioned that "I have to brush and select an area to look at the code." P1 and P10 wished they "can see more information from the map without extra interaction". However, encoding more information on the map could lead to more cognitive load for users to validate progress at scale. P6 and P13 said they prefer the baseline system because they can directly see what's going on in each student's editor instead of remembering what each dot means on the map.

6 DISCUSSION

6.1 VizProg's Visualization is Intuitive for Participants

As our findings show, VizProg helps participants to identify more issues while spending less time analyzing students' submissions. These findings suggest that VizProg's visualization and interactions are intuitive compared to an enhanced version of OverCode, and can help them identify students' problems at scale. One participant commented "...I would say initially, the whole movement from left to right is basically telling me the progress the student is making towards the right answer, whereas in Overcode it was more like just a set of blocks, and there was no indication whether the final one is the right or not..." (P1). This makes sense because the 2D map can off load users' effort of tracking students' history activities to visual information such as the color and the position of the dots. With the encoded information, users can more quickly decide on which students to focus on, shaping their strategies on analyzing students' behaviors. Additionally, VizProg's features—most notably, the ability to brush to select a group of students and examine progress at different levels (i.e., group level, individual level)—allowed participants to analyze student behaviors at scale with fewer context switches.

6.2 Trajectories in VizProg ease the reasoning progress

VizProg offers an innovative way to visualize students' coding progress, which not only reduces instructor's memory load, but also provides a clear visual guide to reasoning about students' behaviors. For instance, when students are working toward a final solution, instructors can clearly see how a dot moves along a trajectory and easily recall history versions by looking at trajectory's position. In a conventional timeline view, every time students make a new submission, users need to look at previous versions to recall what this student submitted before. Additionally, VizProg also helps participants identify abnormal behaviors. For instance, participants

found that some students' trajectories first reached the rightmost side and then wander back to the left side of the map, which means the students had correct submissions and then changed it to incorrect solutions. They reasoned that these students might be exploring other approaches and did not need help. Participants also found that students were wandering in the middle of the map and never reached the right side of it during the whole exercise. Participants then decided to talk to the student and give tailored feedback.

6.3 The Student Experience with VizProg

Beyond helping instructors, VizProg might also benefit to students in several ways. Most directly, by helping instructors identify struggling students and class-wide patterns, VizProg allows instructors to adapt their instruction to students' needs. For example, instructors might discuss mistakes that they observed across many students, give students tailored feedback, or create impromptu in-class exercises in response to what observe in VizProg. Future versions of VizProg could incorporate additional student information[3] that might help instructors better understand if there might be class-wide equity issues (e.g., if there are hidden barriers that prevent a group of people from being able to meaningfully participate in class exercises). Future versions of VizProg could also help to increase student engagement and improve the effectiveness of peer learning. Prior work found that in peer learning, students are grouped without regard to their diverse backgrounds, solution approaches, and levels of knowledge, which could lead to less meaningful and less fruitful group discussions [38]. By encoding students' progress into a 2D map, VizProg can help instructors connect students with each other strategically—for example, to form groups of students who took different approaches or to pair students who are struggling with peers that can help them. Third, Denny et al. [8] found that students benefit from exposure to a wider diversity of solutions by reviewing others' code. With the assistance of VizProg, instructors could easily guide students to diverse solutions from other students without revealing their identities, thus giving students a deeper understanding of how they can apply the concepts they learn in class.

6.4 Ethical Implications and Privacy in VizProg

VizProg provides a code-centered view, where instructors can focus on the code itself without sensitive information such as students' names, genders, grades, or race. Given sensitive information such as identities, instructor may stereotype some groups of students. Inequities embedded in and around computing courses can be barriers to participation and promote bias in class [24, 30]. Therefore, the code-centered view in VizProg could potentially reduce bias and help create a fairer environment for students. Future deployments of VizProg should also give students the option to opt out of sharing their data. The current VizProg interface allows instructors to monitor students' progress in real-time without regard to students' consent to submit. This could harm students' privacy and make students less motivated to engage in class due to social pressure. We can extend VizProg at the student side to give students the option to turn monitoring mode on and off. In addition, students should be aware of being monitored when working on programming exercises in class. VizProg can be extended with an in-editor notification to inform them that they are being passively monitored by the instructor.

[3]The design and presentation of this information would need to be considered carefully, as including demographic information might have important drawbacks, as we discuss in section 6.4.

7 LIMITATIONS

Our user study has three primary limitations. First, although we used authentic student data, we used a simulated setting rather than a real classroom. As a result, we removed distractions and psychological intensity for participants is reduced as compared to a live classroom setting. Second, the feedback generated by instructors was not forwarded to actual students, which might make participants less inclined to provide timely and nuanced feedback during the study. Third, we evaluated on relatively short snippets of code. More advanced programming courses might have exercises that require writing dozens of lines of code across multiple files, which could be more difficult to map and visualize or might require visualizing individual components separately. Additionally, there are limitations to the system. Several participants noted that both the baseline map as well as the 2D Euclidean map in VizProg can be overwhelming when many students are editing the code simultaneously. Although our results showed that participants were still able to find the information they needed quickly in VizProg, conveying high volume of information is still challenging using a 2D visualization.

8 FUTURE WORK

Beyond what we created and demonstrated in this work, VizProg has the potential to better capture and represent the process of solving exercises at scale and for instructors to precisely analyze behavior with lower cognitive effort. In this work, our target audiences are instructors of large introductory level programming courses, who need more efficient and intuitive support to understand students' progress and misconceptions. But we believe that our approach can be generalized to many contexts that need to visualize progress changes at scale in real time. For example, the high level idea of visualizing incremental changes on a 2D map can be applied to monitoring students writing short-answer questions, identifying the spread of viruses, or analyzing spatio-temporal traffic flows [9].

In this work, VizProg updates the visualization upon each student keystroke, where the dots move in real-time as students type. We chose keystroke-level updates because it reveals more information than the granularity of every time students execute their code. In programming exercises, students have different coding habits and execute code at varying frequencies. Some students run code very often, while others run code only when they are ready to submit it. Instructors may not be able to observe the full process of how students develop a solution from scratch at the granularity of each execution of code. Nevertheless, as participants reported in the user studies, keystroke-level updates can be overwhelming and distracting especially when a bunch of dots move at the same time. Future work can explore different granularities of updates, such as every time students run the code or every few minutes. Additionally, we can explore various encoding models to reduce

large spatial "jumps" between updates. We can also explore visualizing trajectories with color cues to help instructor better identify behavior patterns, such as jumping between different approaches and being stuck at a certain area.

9 CONCLUSION

In this work, we explored a design that allows instructors to visualize and understand students' status in real-time for in-class programming exercises. We introduce VizProg, a tool that allows instructors to monitor and inspect large numbers of students' coding submissions over time by presenting students on a 2D map. In VizProg, students' status is represented as a position that encodes similarities in students' code, how students approach the exercise, students' progress—how close they are to a correct solution, and how students' status changes over time. Our comparison study showed that VizProg can help participants to discover more than twice as many student problems, and find these problems with less than half of the time and fewer interactions. Furthermore, participants reported that VizProg provides richer and more comprehensive information for identifying important student behavior. This work illustrates how we can further improve teaching by better understanding students' mental models and providing tailored feedback at scale.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under DUE 1915515.

REFERENCES

[1] Douglas S Blank, David Bourgin, Alexander Brown, Matthias Bussonnier, Jonathan Frederic, Brian Granger, Thomas L Griffiths, Jessica Hamrick, Kyle Kelley, M Pacer, et al. 2019. nbgrader: A tool for creating and grading assignments in the Jupyter Notebook. *The Journal of Open Source Education* 2, 11 (2019).

[2] Marcel Borowski, Johannes Zagermann, Clemens N Klokmose, Harald Reiterer, and Roman Rädle. 2020. Exploring the Benefits and Barriers of Using Computational Notebooks for Collaborative Programming Assignments. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 468–474.

[3] Jeongmin Byun, Jungkook Park, and Alice Oh. 2021. Cocode: Providing Social Presence with Co-learner Screen Sharing in Online Programming Classes. *Proceedings of the ACM on Human-Computer Interaction* 5, CSCW2 (2021), 1–28.

[4] Charles H Chen and Philip J Guo. 2019. Improv: Teaching programming at scale via live coding. In *Proceedings of the Sixth (2019) ACM Conference on Learning@ Scale*. 1–10.

[5] Lijia Chen, Pingping Chen, and Zhijian Lin. 2020. Artificial intelligence in education: A review. *Ieee Access* 8 (2020), 75264–75278.

[6] Yan Chen, Walter S Lasecki, and Tao Dong. 2021. Towards supporting programming education at scale via live streaming. *Proceedings of the ACM on Human-Computer Interaction* 4, CSCW3 (2021), 1–19.

[7] Albert T Corbett and Akshat Bhatnagar. 1997. Student modeling in the ACT programming tutor: Adjusting a procedural learning model with declarative knowledge. In *User modeling*. Springer, 243–254.

[8] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Codewrite: supporting student-driven practice of java. In *Proceedings of the 42nd ACM technical symposium on Computer science education*. 471–476.

[9] Somayeh Dodge and Evgeny Noi. 2021. Mapping trajectories and flows: facilitating a human-centered approach to movement data analytics. *Cartography and Geographic Information Science* 48, 4 (2021), 353–375.

[10] Matthew L Egizii, James Denny, Kimberly A Neuendorf, Paul D Skalski, and Rachel Campbell. 2012. Which way did he go? Directionality of film character and camera movement and subsequent spectator interpretation. In *International Communication Association conference, Phoenix, AZ.*

[11] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[12] Takanori Fujiwara, Jia-Kai Chou, Shilpika Shilpika, Panpan Xu, Liu Ren, and Kwan-Liu Ma. 2019. An incremental dimensionality reduction method for visualizing streaming multidimensional data. *IEEE transactions on visualization and computer graphics* 26, 1 (2019), 418–428.

[13] Elena L Glassman, Lyla Fischer, Jeremy Scott, and Robert C Miller. 2015. Foobaz: Variable name feedback for student code at scale. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 609–617.

[14] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. 2015. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 1–35.

[15] Ashok K Goel and Lalith Polepeddi. 2018. Jill Watson: A virtual teaching assistant for online education. In *Learning engineering for online education*. Routledge, 120–143.

[16] Philip J Guo. 2015. Codeopticon: Real-time, one-to-many human tutoring for computer programming. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 599–608.

[17] Philip J Guo, Jeffery White, and Renan Zanelatto. 2015. Codechella: Multi-user program visualizations for real-time tutoring and collaborative learning. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 79–87.

[18] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing reusable code feedback at scale with mixed-initiative program synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale*. 89–98.

[19] Jonathan Huang, Chris Piech, Andy Nguyen, and Leonidas Guibas. 2013. Syntactic and functional variability of a million code submissions in a machine learning mooc. In *AIED 2013 Workshops Proceedings Volume*, Vol. 25. Citeseer.

[20] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. 2016. Semi-supervised verified feedback generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 739–750.

[21] Juho Kim, Elena L Glassman, Andrés Monroy-Hernández, and Meredith Ringel Morris. 2015. RIMES: Embedding interactive multimedia exercises in lecture videos. In *Proceedings of the 33rd annual ACM conference on human factors in computing systems*. 1535–1544.

[22] Vladimir I Levenshtein et al. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. Soviet Union, 707–710.

[23] Julia M Markel and Philip J Guo. 2021. Inside the Mind of a CS Undergraduate TA: A Firsthand Account of Undergraduate Peer Tutoring in Computer Labs. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. 502–508.

[24] Paola Medel and Vahab Pournaghshband. 2017. Eliminating gender bias in computer science education materials. In *Proceedings of the 2017 ACM SIGCSE technical symposium on computer science education*. 411–416.

[25] Joseph Bahman Moghadam, Rohan Roy Choudhury, HeZheng Yin, and Armando Fox. 2015. AutoStyle: Toward coding style feedback at scale. In *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*. 261–266.

[26] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. 2014. Codewebs: scalable homework search for massive open online programming courses. In *Proceedings of the 23rd international conference on World wide web*. 491–502.

[27] Thanaporn Patikorn and Neil T Heffernan. 2020. Effectiveness of crowd-sourcing on-demand assistance from teachers in online learning platforms. In *Proceedings of the Seventh ACM Conference on Learning@ Scale*. 115–124.

[28] Chris Piech, Jonathan Huang, Andy Nguyen, Mike Phulsuksombati, Mehran Sahami, and Leonidas Guibas. 2015. Learning program embeddings to propagate feedback on student code. In *International conference on machine Learning*. PMLR, 1093–1102.

[29] Roman Rädle, Midas Nouwens, Kristian Antonsen, James R Eagan, and Clemens N Klokmose. 2017. Codestrates: Literate computing with webstrates. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 715–725.

[30] Kevin Robinson, Keyarash Jahanian, and Justin Reich. 2018. Using online practice spaces to investigate challenges in enacting principles of equitable computer science teaching. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 882–887.

[31] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. 2003. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 76–85.

[32] Gursimran Singh, Shashank Srikant, and Varun Aggarwal. 2016. Question independent grading using machine learning: The case of computer program grading. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 263–272.

[33] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 15–26.

[34] Ahmad Taherkhani, Ari Korhonen, and Lauri Malmi. 2012. Automatic recognition of students' sorting algorithm implementations in a data structures and algorithms course. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*. 83–92.

[35] Ahmad Taherkhani and Lauri Malmi. 2013. Beacon-and Schema-Based Method for Recognizing Algorithms from Students' Source Code. *Journal of Educational Data Mining* 5, 2 (2013), 69–101.

[36] Yuta Taniguchi, Tsubasa Minematsu, Fumiya Okubo, and Atsushi Shimada. 2022. Visualizing Source-Code Evolution for Understanding Class-Wide Programming Processes. *Sustainability* 14, 13 (2022), 8084.

[37] Laurens Van der Maaten and Geoffrey Hinton. 2008. Visualizing data using t-SNE. *Journal of machine learning research* 9, 11 (2008).

[38] April Yi Wang, Yan Chen, John Joon Young Chung, Christopher Brooks, and Steve Oney. 2021. PuzzleMe: Leveraging Peer Assessment for In-Class Programming Exercises. *Proceedings of the ACM on Human-Computer Interaction* 5, CSCW2 (2021), 1–24.

[39] Daniel Weitekamp, Erik Harpstead, and Ken R Koedinger. 2020. An interaction design for machine teaching to develop AI tutors. In *Proceedings of the 2020 CHI conference on human factors in computing systems*. 1–11.

[40] Joseph Jay Williams, Juho Kim, Anna Rafferty, Samuel Maldonado, Krzysztof Z Gajos, Walter S Lasecki, and Neil Heffernan. 2016. Axis: Generating explanations at scale with learnersourcing and machine learning. In *Proceedings of the Third (2016) ACM Conference on Learning@ Scale*. 379–388.

SemanticOn: Specifying Content-Based Semantic Conditions for Web Automation Programs

Kevin Pu
jpu@dgp.toronto.edu
University of Toronto

Rainey Fu
rainey.fu@mail.utoronto.ca
University of Toronto

Rui Dong
ruidong@umich.edu
University of Michigan

Xinyu Wang
xwangsd@umich.edu
University of Michigan

Yan Chen
yanchen@dgp.toronto.edu
University of Toronto

Tovi Grossman
tovi@dgp.toronto.edu
University of Toronto

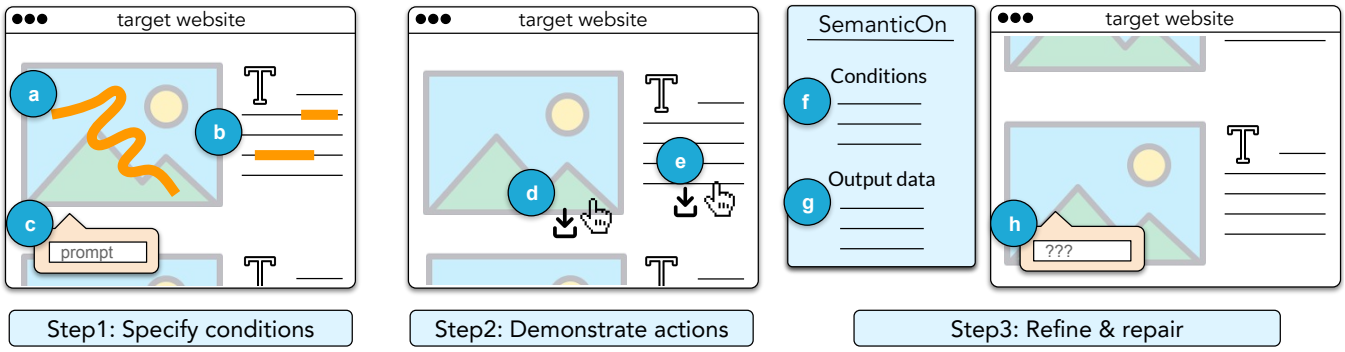


Figure 1: The workflow of SemanticOn. There are three steps to creating a web automation program with semantic conditions using SemanticOn. (Step 1) To specify semantic conditions, users can either describe their intent in text (*User Enters*, ⓒ) or indicate the section of interest by brushing through an image ⓐ or highlighting parts of a text ⓑ (*System Suggests*). SemanticOn then encodes these specifications with computer vision and natural language processing techniques into web program conditions. (Step 2) To create the intended web automation program, users demonstrate the actions on the website using WebRobot, including image downloading ⓓ and text scraping ⓔ. (Step 3) Once the program is executed, users can also easily coordinate with SemanticOn to refine the semantic conditions (ⓕ, ⓗ) or take back control to add or remove data manually ⓖ.

ABSTRACT

Data scientists, researchers, and clerks often create web automation programs to perform repetitive yet essential tasks, such as data scraping and data entry. However, existing web automation systems lack mechanisms for defining conditional behaviors where the system can intelligently filter candidate content based on semantic filters (e.g., extract texts based on key ideas or images based on entity relationships). We introduce SemanticOn, a system that enables users to *specify*, *refine*, and *incorporate* visual and textual semantic conditions in web automation programs via two methods: natural language description via prompts or information highlighting. Users can coordinate with SemanticOn to refine the conditions as the program continuously executes or reclaim manual control to repair errors. In a user study, participants completed a series of conditional web automation tasks. They reported that SemanticOn helped them effectively express and refine their semantic intent by utilizing visual and textual conditions.

KEYWORDS

Web automation, PBD, user intent, semantics

ACM Reference Format:
Kevin Pu, Rainey Fu, Rui Dong, Xinyu Wang, Yan Chen, and Tovi Grossman. 2022. SemanticOn: Specifying Content-Based Semantic Conditions for Web Automation Programs. In *The 35th Annual ACM Symposium on User Interface Software and Technology (UIST '22), October 29-November 2, 2022, Bend, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3526113.3545691

UIST '22, October 29-November 2, 2022, Bend, OR, USA

ACM ISBN 978-1-4503-9320-1/22/10…$15.00
https://doi.org/10.1145/3526113.3545691

1 INTRODUCTION

Enterprises, governments, and schools often use web-based applications to manage their businesses and services. Other than information consumption, users such as clerks, data scientists, and researchers often employ these web platforms to conduct tasks that are repetitive yet essential, such as data scraping and data entry. Performing these tasks manually can often lead to human errors (e.g., data duplicates, missed entries), which can cause inefficiencies.

Web automation offers a solution that leverages bots to mimic human interactions on web applications. It assists users with tedious and recurring tasks and has proven to be faster and more accurate for various task types compared to manual effort [42].

Past research has developed techniques to help users of all expertise levels to quickly and accurately create their intended web automation programs [25–27, 44, 68]. However, these techniques are limited to creating programs with requirements at the website syntax or structural level (e.g., scraping the first two items in each row of a table). Tools capable of creating logic based on the meaning of the content (semantics) remain unexplored. For instance, commercial tools such as iMacros [2] and UiPath [8] enable users to perform record-and-replay interactions for web automation and testing. Research tools such as Helena [15] further this technique by lowering the learning curve, allowing users with little programming experience to create complex programs that can handle hierarchical data (e.g., tree-structured data) and distributed data spread across multiple websites.

We identified a need for web automation with semantic conditions through prior user studies [24] and analysis of real user requests in online forms [3, 7]. This includes vision-related semantic conditions, such as scraping images that meet specific criteria (e.g., a photography student wants to study group interaction portraits on a gallery website with thousands of photos) or text-related semantic conditions, such as scraping text only when it expresses particular sentiments (e.g., a film critic wants to evaluate positive reviews of a movie star's acting from dozens of news articles in a journal). With current techniques, users cannot specify these semantic intents in web automation programs. As noted above, semantic information often varies by content type, which makes it hard to design a universal interaction that is both easy to use and sufficiently expressive. Additionally, unlike other AI systems that provide results immediately after the provision of user inputs (e.g., chatbots), once executed, a web automation program will continuously output results as it iterates over web contents. This makes monitoring and error handling difficult, as the program may encounter unforeseen and problematic cases.

This paper explores interactive techniques to enable content-based semantic condition specification for web automation programs. We introduce SemanticOn,[1] a system that allows users to *specify*, *refine*, and *incorporate* visual and textual semantic information as conditions in web automation programs via two methods: natural language description via prompts or detailed information highlighting with system support. We define them as *User Enters* and *System Suggests*, respectively. SemanticOn combines the relative strengths of neural models (Transformer) for unstructured information and program synthesis techniques for web automation. By doing so, we introduce a new interaction paradigm for users to continuously add/refine semantic conditions in a programming-by-demonstration system. Specifically, SemanticOn builds upon WebRobot [24], a *program synthesis* system that enables users to create web automation programs by demonstrating actions on the target websites. WebRobot employs a no-code development approach that requires only web interactions in place of programming knowledge from users, which is consistent with our design goal.

[1]SemanticOn is an acronym for **semantic con**dition

Figure 1 depicts the three steps of using SemanticOn. (Step 1) To specify semantic conditions, users can either describe their intent in a sentence (Fig. 1.c), indicate their area of interest by brushing through an image (Fig. 1.a), or highlight parts of a text (Fig. 1.b). SemanticOn uses similarity-based computer vision and natural language processing techniques to encode these specifications into web program conditions. (Step 2) To create the intended web automation program, users will demonstrate actions on the website using WebRobot, including image downloading (Fig. 1.d) and text scraping (Fig. 1.e). (Step 3) Once the program is executed, users can also easily coordinate with SemanticOn to refine the semantic conditions based on the automatically detected information (Fig. 1.f, Fig. 1.h) or reclaim control to manually add or remove data if the program has misjudged (Fig. 1.g). To our knowledge, SemanticOn is the first system to explore content-based semantic specification interactions for web automation programs.

We conducted a user study with 10 participants to evaluate SemanticOn's overall usability and efficiency and to compare the semantic condition specification of each method (*User Enters* and *System Suggests*). We found that participants using SemanticOn successfully extracted 80.8% of the required data with an average time of 06:10 minute:second per task. The participants found that SemanticOn helped them effectively express their semantic intent by prompting them to consider their visual and textual perceptions of the tasks. We found a sense of control vs. effort trade-off, where participants enjoyed composing their conditions in *User Enters* but had to spend more time and mental effort devising a description to encapsulate the semantic condition. On the other hand, while participants could specify and refine conditions more easily via highlighting content details and selecting generated conditions in *System Suggests*, they had less freedom to express their intent when system suggestions were inaccurate.

In the final section of this work, we analyze the human-AI collaboration workflow in SemanticOn, discuss the implications of adding similarity-based models in a symbolic PBD system, and explore future work that can adapt our approach to other types of interactive AI systems that require semantic conditions. This work is an essential step towards the vision of natural, intent-unambiguous end-user programming with a focus on web automation creation. This paper makes the following contributions:

- The *User Enters*, *System Suggests* interaction designs, implementations, and evaluations that allow users to specify and demonstrate their intent during web automation creation,
- The refinement and error-handling techniques to clarify and improve semantic filters in a continuous human-AI collaboration process,
- SemanticOn, along with a user study showing its usability and effectiveness in helping users specify semantic conditions for web automation programs.

2 RELATED WORK

SemanticOn builds on decades of web automation systems and innovations. In this section, we draw our design goals and guidance from three areas of work: web automation, programming-by-demonstration, and user intent specification and refinement.

2.1 Web Automation

Web automation is a software technique that leverages bots to perform tedious and recurring web tasks by mimicking human interactions, such as data entry and data extraction. Data scientists, UI testers, and clerks all use web automation to help complete their domain-specific tasks [40, 43, 49, 69]. Social scientists, for example, might want to develop web data scraping programs to collect necessary web datasets. UI testers might want to create an automated browser testing program to help developers find front-end defects. Data workers might envision a data entry program for routine tasks like entering large amounts of data into a digital system (e.g., booking flights for all employees).

Creating web automation programs is a non-trivial and complex task. Many web automation tools require users to have domain knowledge (e.g., understand the Document Object Model (DOM) structure) and programming experience. Commonly used tools like Puppeteer [4], Selenium [6], Scrapy [5], and Beautiful Soup [1] require users to learn code syntax, understand the task content architecture (e.g., DOM tree hierarchy), and have software testing experience. Prior work has shown that even for professional developers, creating automation programs is time-consuming. Krosnick and Oney studied the challenges of writing web macros using common web automation frameworks for experienced programmers [35]. They found that a primary challenge for participants was the labor of checking syntactical element selectors to create their programs, which was inefficient and prone to mistakes. In addition, the program might not generalize to cross-webpage selections where the elements don't have syntactic similarity. Our work enables users to specify the semantic meaning of their target content, bypassing the issues caused by implementation.

Researchers have developed many helpful tools to reduce the effort of program creation. For desktop application automation, systems like Sikuli [76] allow users to identify a GUI element (e.g., an icon or a toolbar button) by taking its screenshot. Using computer vision techniques, it analyzes patterns in the screenshots to locate the appropriate elements when automating GUI interactions. Although this approach is promising, it requires programming knowledge and cannot disambiguate similar elements or text information. For UI testing, researchers have proposed and studied crowdsourcing and automated testing strategies to help increase the testing coverage and reduce the effort of creating programs [21, 23]. While helpful, outputs produced with these tools are hard to generalize to new UIs or contexts.

2.2 Programming-by-Demonstration

To further reduce the expertise required, many tools have used a programming-by-demonstration (PBD) approach where users only have to interact with the target applications rather than writing code [12, 36, 38]. These span a variety of application domains including text manipulation [13, 39, 54, 60, 75], image or video editing [37, 47, 51], and GUI synthesis [55, 57, 61, 71]. In the context of web applications, PBD delivers on this first design requirement, offering web automation without requiring users to understand browser internals or manually reverse-engineer target pages. The PBD approach has produced great successes in the web automation domain, most notably CoScripter [42], Vegemite [48], Rousillon [15], and iMacros [2].

Some of these systems require users to edit their traces to add parametrization. For instance, CoScripter and iMacros offer record-and-replay functionality; users record themselves interacting with the browser—clicking, entering text, and navigating between pages—and the tool writes a loop-free script that replays the recorded interaction. Because they lack support for control constructs and function composition, these systems require users to have logic skills. Other systems support iteration using program synthesis, automatically discovering loops given a demonstration of one or a few iterations. While less domain knowledge is needed, the synthesizer can make mistakes in which the user must provide more demonstrations or edit the DSL to correct it (e.g., Helena), which can be frustrating. SemanticOn instead allows users to effectively coordinate with PBD systems by smoothly switching agency and editing constraints at any time during program execution.

2.3 User Intent Specification and Refinement

User intent specification is an important and challenging component of human-AI collaboration. Ideally, users should be able to easily and naturally specify their intent to a system while understanding its states. However, given the limited capabilities of AI understanding techniques, high-level user intent can be difficult for systems to comprehend. Many systems have proposed bridging the gap between user intent and system understanding. For instance, PLOW [9] and PUMICE [46] allow users to express concepts (e.g., hot weather) in natural language and then learn the concepts to generalize the automation. Systems like Scout [70], Designscape [64], and Iconate [80] allow users to iteratively refine their intent by directly manipulating the AI-generated artifacts. Other studies have shown that this refinement interaction can even be delegated to crowd workers [18]. Another work, APPINITE [45], also encapsulates user's intent in natural language instructions and clarifies the intention in a back-and-forth conversation with the AI. While these approaches are promising, user intents can involve visual and cognitive details such as identifying visual relationships in images or parsing texts to match a high-level idea. The user's semantic level intents are often not fully or accurately expressed through natural language or limited examples only, leading to information loss during communication and rendering the communication ineffective [19].

Similar to PBD systems, programming-by-example (PBE) is another approach to facilitate program creation for various tasks such as data wrangling [29, 30, 34] and data visualization [52, 72]. Many PBE and PBD systems require users to provide additional examples to disambiguate user intent. Falx allows users to specify visualization examples using a small amount of data and then infers and transforms the data to match the design [73]. Sporq allows users to more accurately and quickly search code patterns in large codebases by prompting them to refine their intent by annotating a batch of negative examples and adding specific constraints [58]. Other works enable users to directly annotate their input examples (augmented examples) to disambiguate user intent [66, 78]. Or they employ data visualization techniques to showcase the generated

programs, allowing users to tweak the path of program generation in a tree view [77]. While promising, providing additional examples increases users' cognitive demand. In this work, we focus on addressing the ambiguous semantic conditions and designing human-AI collaboration interaction solutions to help refine the constraints based on the content.

Using machine learning (ML) models to refine intent has been a recent focus in the field of interactive ML. One common interactive ML approach allows users to offer feedback during the model training process for more effective ML model creation [16]. Work by Cai et al. allows users to adjust the search algorithm iteratively with different types of similarities at different moments [14]. Projects by Austin et al. and Jiang et al. allow users to interact with large language models to help refine their intent when writing code snippets [11, 33]. Work by Amershi et al. allows users to identify new friend groups on social media by analyzing the examples presented [10]. Software developed by Fogarty et al. helps users to create their own rules to improve the search results [28]. This research inspires our work, but instead, we focus on helping users refine their intent while interacting with continuous AI systems—web automation programs that require constant monitoring and that effectively coordinate the turn-taking.

3 BACKGROUND AND DESIGN GOALS

Our work is built upon an existing web automation system, WebRobot [24], that only uses web interactions and requires no programming knowledge from its users. This is consistent with our design goal. In addition to prior work, we derive our design goals from WebRobot's user study. In this section, we provide necessary background information on the WebRobot system and then discuss the design goals for our system SemanticOn.

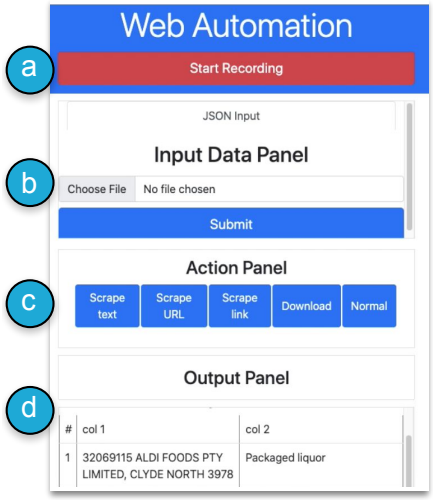


Figure 2: A screenshot of the WebRobot system UI.

3.1 The WebRobot System Workflow

WebRobot is designed to facilitate web automation program creation. Figure 2 shows the WebRobot user interface. To create a web automation program for a data entry or scraping task, a user first starts recording their actions (Fig. 2.a). Then they can either upload a `JSON` file (Fig. 2.b) if the task involves data entry, or they can choose an appropriate action (e.g., Scrape Text) in the action panel (Fig. 2.c) and perform the required actions (e.g., clicking the desired text data on the website). After each scraping action, they will see the data appended to the output panel (Fig. 2.d). Behind the scenes, WebRobot records every user action on the website and its associated action type (e.g., Scrape Text). After a few demonstrations, WebRobot synthesizes a program P from the trace A of demonstrated actions. In particular, WebRobot guarantees that P not only *reproduces* the demonstrated actions from A but also generalizes beyond A. This typically implies P would contain loops that can be used to automate the user-intended task. Finally, WebRobot executes P to automate the rest of the actions in the task.

procedure SYNTHESIZE (A)
input: $A = [a_1, \cdot\cdot, a_m]$ is a trace of user-demonstrated actions.
output: a program P that generalizes A.
1: $P_0 := a_1; \cdot\cdot; a_m$;
2: $W := \{P_0\}$; $\widetilde{P} := \emptyset$;
3: **while** $W \neq \emptyset$
4: $P := W$.remove();
5: **if** P *generalizes* A **then** $\widetilde{P}$.add(P);
6: $W' :=$ REWRITE(P);
7: $W := W \cup W'$;
8: **return** RANK($\widetilde{P}$);

Algorithm 1: Rewrite-based program synthesis algorithm.

3.2 WebRobot's Synthesis Algorithm

In a nutshell, WebRobot's synthesis algorithm (Algorithm 1) generalizes an input action trace A into a program P (with loops) by iteratively rewriting A to loops in P *from the inside out*. Initially, it creates a program P_0 with exactly those actions in A (line 1): while P_0 reproduces A, it does not generalize A (i.e., it does not produce new actions after A). Therefore, the algorithm performs iterative rewriting to gradually "compress" P_0 into more compact and general programs using a worklist algorithm (lines 2-8). The worklist W is initialized to have only P_0, and we use $\widetilde{P}$ to keep track of all programs that generalize A (line 2). Whenever W is not empty (line 3), the algorithm would remove a program P from W (line 4). It then checks to see whether P generalizes A; if so, P is added to $\widetilde{P}$ (line 5). After this, in line 6, the algorithm tries to rewrite P into more general programs, which are stored in W'. The key idea underlying our REWRITE procedure is to perform *semantic rewriting* using a methodology called *speculate-and-rewrite*. Intuitively, it inspects P, identifies repetitive patterns in P, hypothesizes potential loops that correspond to P, and finally synthesizes programs with one more level of loop. How WebRobot's speculative writing process works is beyond the scope of this work; we refer interested readers to the original WebRobot paper [24] for details. Once P is rewritten to a new set of programs W' (line 6), the algorithm simply merges W' into W (line 7). The worklist loop terminates when no programs

can be rewritten and it finally returns the smallest program in $\widetilde{P}$ using a ranking function (line 8).

3.3 User Feedback

In WebRobot's user study, participants reported that while WebRobot can help lower barriers of entry for the creation of web automation programs and handling a more comprehensive range of tasks, they wished that they could express conditions to filter the content. For instance, one participant said, *"maybe some conditional scraping [can be included], not based on whether the element exists in the webpage, but based on some other conditions."* Consistently, we found posts on forums such as iMacros [3] and Stack Overflow [7] that request the creation of web automation programs with content-based conditions. Participants also wished to refine their intent when interacting with the WebRobot system. For instance, participants reported that they wanted to *"undo my wrong manipulations"* or *"edit my history."* However, as noted in the related work, WebRobot and other systems do not effectively support actions such as undo or history manipulation.

3.4 Design Goals

Based on this prior work, we devised the following three design goals to help users easily create web automation programs with semantic conditions.

- **DG1: Ability to express content-based semantic conditions:** Users can specify semantic conditions when creating web automation programs.
- **DG2: Accessible user intent refinement:** Users can iteratively refine their semantic conditions at any time of the program creation process.
- **DG3: Responsive error handling for mistakes made by users and the system:** Users need to modify inaccurate conditions and edit scraped data easily.

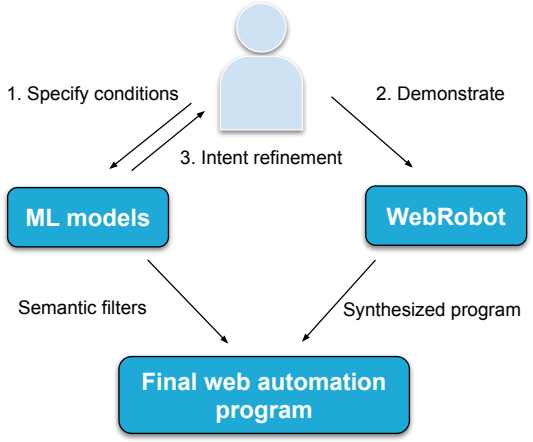


Figure 3: SemanticOn's system architecture.

4 SEMANTICON

With the three design goals above, we created SemanticOn to help users specify, refine, and incorporate semantic conditions in automated web data scraping. Figure 3 shows SemanticOn's system architecture and main user interactions at a high level. Instead of writing web macros from scratch for each website and filtering the scraped content in post-processing, users can interact with SemanticOn to compose semantic conditions on the content they want to scrape (Step 1 Fig. 3), then demonstrate actions on the target website to synthesize automation programs for different websites without writing a single line of code (Step 2 Fig. 3). Throughout this process, users can communicate with SemanticOn, which is capable of parsing text and image content through machine learning models. The users and SemanticOn work together to refine the condition set and repair errors in result selection, continuously improving both the system's and user's understanding of the filter criteria (Step 3 in Fig. 3). In this section, we first illustrate SemanticOn's user experience with a sample scenario that embodies common semantic conditions. We then detail the design and implementation of SemanticOn.

4.1 The SemanticOn User Experience

Mia, an outdoor enthusiast, wants to extract online information about travel destinations where outdoor activities are available. To help her make an informed decision, Mia wants to scrape the text description and the image for each location from an article to build potential itineraries. One option is to read through every paragraph, look at each picture, and manually copy and paste the relevant information, but that process would be tedious and repetitive. On the other hand, Mia could write a web scraping script using Python. She has some coding experience, but writing a script and filtering the results based on her preference would also be time-consuming and laborious. Instead, Mia uses SemanticOn to efficiently demonstrate her conditions and web actions and synthesize a web automation program that completes the task for her.

To begin, Mia sets the semantic conditions for the intended content (Step 1 Fig.3). She first clicks "Text Condition". She then selects *User Enters* (Fig. 4.d) to specify the semantic condition in her own words. Mia represents her high-level requirement in the system prompt by typing, *"This is a great location for outdoor activities"* (Fig. 6.c). She believes this sentence is likely semantically similar to the relevant content in this article. After clicking "Add", the condition is appended to the text condition table (Fig. 4.h) in the condition panel. Furthermore, Mia decides to add a condition to the corresponding destination image. She wants to travel to a place where hiking and water activities are accessible. To that end, she uses *System Suggests*, clicks on an ideal image, and highlights the mountain and lake in the picture (Fig. 6.a). The system detects several objects and summarizes the image content into a sentence. Mia also adds the relevant objects and caption to the corresponding tables (Fig. 4.f,g) in the condition panel.

After specifying two initial conditions, Mia decides to start the demonstration process (Step 2, Fig.3). She clicks the "Start Recording" button (Fig. 4.a) to start the web macro recording for program synthesis. Then, Mia specifies the task name as "Travel Destination Search" and sets the column number to 2, one for text descriptions and one for the associated images.

Next, Mia selects "Download Image" and hovers the mouse to highlight the desired image element (Fig. 4.j). As she clicks on the element, the image is downloaded and put into the first column

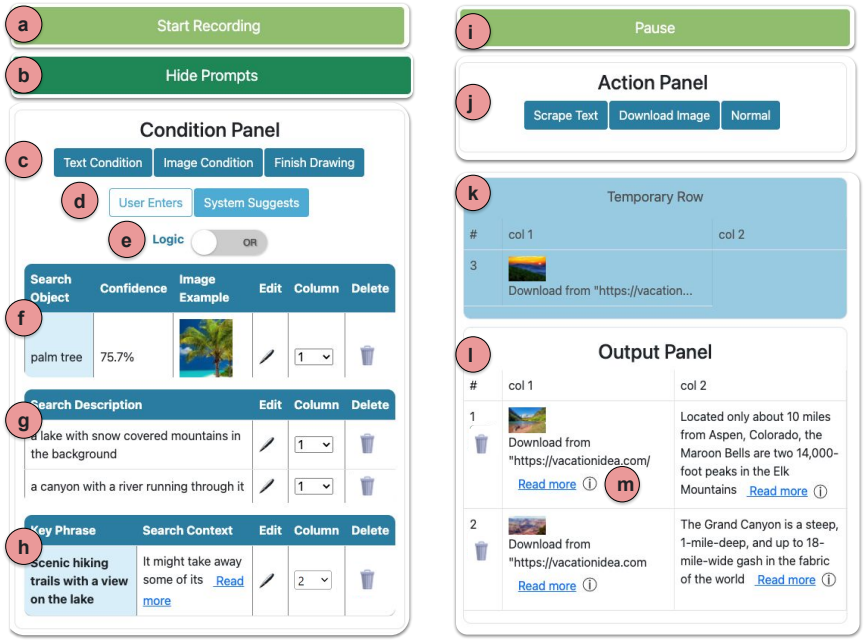


Figure 4: Overview of SemanticOn's user interface. The user begins program synthesis by clicking Start Recording ⓐ. The user can disable or enable system suggested prompts in automation ⓑ. The user can specify conditions for images or text ⓒ and choose between manually input or select system-suggested conditions ⓓ. The user can also toggle between OR and AND logic for the condition set ⓔ. A unique table displays image object conditions ⓕ, image description conditions ⓖ, and text description conditions ⓗ. The user may pause ⓘ at any time to make changes that will not affect automation. To interact with the web page and demonstrate actions for program synthesis, the user will select their action type by choosing an option in the action panel ⓙ. The system will predict the next action, shown in the temporary row ⓚ. If the semantic conditions are met, the temporary row will be appended to the output panel ⓛ. To learn more about how the content match with the conditions, the user may click on the information icon ⓜ.

of the Temporary Row (Fig. 4.k). SemanticOn evaluates conditions associated with each column before the row is added to the output panel or skipped. However, at the demonstration stage, the condition will not be triggered, as we assume users will only demonstrate macro actions on the content they want to include. Mia then selects the "Scrape Text" action and repeats the steps for the destination description. As both columns in the temporary row are filled, the entire row is added to the output panel (Fig. 4.l).

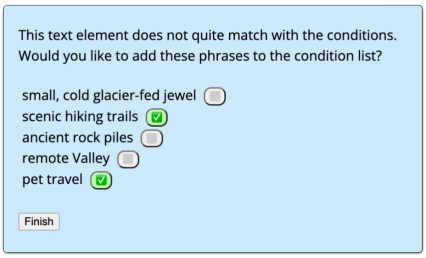


Figure 5: Example of system prompt when new content does not match with current condition.

Mia repeats her demonstration for a second destination that fits her requirement. After two rows of user demonstration, WebRobot can synthesize the web automation program and predict the user's next step, so the system now enters a guided semi-automation mode. SemanticOn scrolls into the next row of the destination and highlights the next element to be extracted. For the image, a prompt states that the generated caption matched with the initial image condition *"a lake beneath the mountain range"*. Mia confirms this selection and lets SemanticOn continue. However, Mia finds that her initial text condition is too general, as the system informs her the text condition does not match the current text and presents key phrases from this element to be potentially used as refinement (Fig.5). She realizes that *"scenic hiking trails"* would be a good textual condition to include. She also discovers *"pet travel"* as a key phrase. Mia would love to take her dog Sushi on the trip, so she also selects that key phrase. After she clicks "Finish," those two phrases are added to the text condition table, and the highlighted text is considered accepted.

After inspecting several elements in semi-automation mode and refining the condition set with system prompts, Mia feels satisfied with the system's interpretation of her requirements. She clicks on the "Hide Prompt" button (Fig. 4.b) to allow the system to automatically predict and filter the rest of the web page content without stopping for user confirmation. Mia sits back and waits for the program to finish. However, she notices that for one destination, *"Grand Canyon, USA,"* the system filters the image as it does not match *"a lake beneath the mountain range"*. However, she actually wants to include this destination in her list, as she can still engage in outdoor activities such as hiking and kayaking. To repair this error,

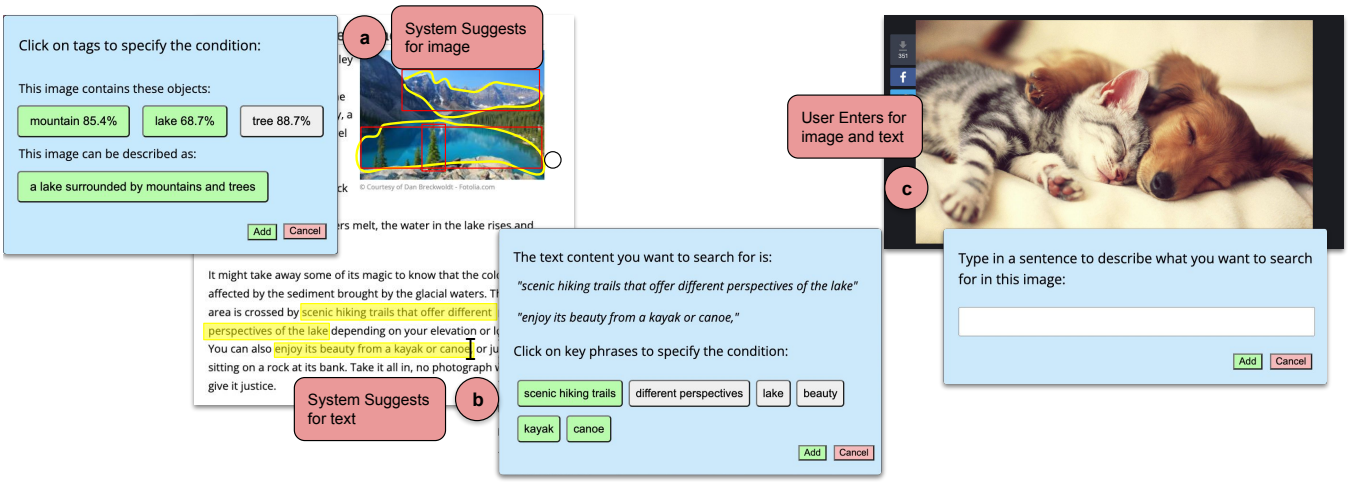


Figure 6: Two condition specification methods in SemanticOn. After entering the system suggests mode, the user may draw on images. Then, a prompt ⓐ containing detected objects within that region, and a general description of the image will be shown. Similarly, the user may highlight sections of text, and the system will prompt ⓑ important key phrases within the highlighted portion. Users may manually enter conditions by clicking on an image or section of text and entering their conditions ⓒ.

Mia pauses the automation process (Fig. 4.i, 7.b). She then manually scrapes the text description and downloads the image using the action panel (Fig. 4.j). It's worth noting that during the pause, Mia's web macros are not used for program synthesis and therefore do not affect the current automation program. Mia also adds another image condition using *User Enters*, specifying *"a canyon valley with river"* as another instance of her ideal destination. While the program is paused, Mia scrolls through, evaluates the output panel, and can delete output rows that were added by system mistake or modify/delete conditions during the pause (Fig. 7.d). After she is satisfied with the refined condition set and the current results, Mia clicks "Resume" to continue the automation. The program iterates through the rest of the web page and collects filtered data based on Mia's semantic conditions. After the program ends, Mia stops the web macro recording, and the collected results are exported as a `JSON` file. Mia uses SemanticOn to create a tailored web automation program without writing any code. She also obtains a high-accuracy result set of her ideal travel destinations by continuously working with SemanticOn to clarify and refine conditions and repair errors.

4.2 Design and Implementation

We implemented SemanticOn as a Chrome browser extension, incorporating the core program synthesis engine from an existing system, WebRobot. Primarily, it uses plain JavaScript for the front-end interactions. For semantic condition comprehension, we adopted two off-the-shelf Transformer models.

4.2.1 Step 1: Specify Semantic Conditions. To enable content-based semantic conditions specification (DG1), SemanticOn allows users to add condition criteria to specific content on the target website. Currently, the system supports conditions on text and image content (Fig. 4.c). The user can choose one of the two methods of specification: 1) *User Enters*, where the user composes the condition using natural language, or 2) *System Suggests*, a novel specification technique where the user highlights relevant content for SemanticOn to analyze and provide suggested conditions (Fig. 4.d). Upon selecting the type of condition and the specification method, the user chooses an element on the web page as the basis of the semantic condition.

Inspired by recent prompt-based interactions [33, 50, 79], a prompt displays next to the selected text or image element for *User Enters*, encouraging the user to enter a semantic description of their search criteria (Fig. 6.c). This description, along with the context of the text or image element, is added to the conditional panel on SemanticOn (Fig. 4.f,g,h). In comparison, for *System Suggests*, the user needs to highlight the crucial part of the content to set the condition on.

For images, the user brushes over an area of interest with their mouse (Fig. 6.a) to illustrate. The image is fed through an off-the-shelf pre-trained multi-layer Transformer model that learns to align image-level tags with their corresponding image region features [32]. The model detects objects for the illustrated section and generates a caption for the entire image. For texts, the user highlights relevant phrases or sentences with their mouse (Fig. 6.b). Similarly, the text is processed by an off-the-shelf unsupervised language model where the noun phrases in the input text are first detected and then ranked based on frequency and co-occurrence. The model generates key phrases in the passage highlighted by the user. The user can then pick any object, caption, or phrase tags to add to the condition table (Fig. 4.f,g,h). We used these models through the Microsoft Azure Cognitive Services and Cloud platform.[2] Guided by the design of other systems [17, 31], we also implemented edit, delete, and logical operations for multiple condition specifications, allowing the user to modify, remove, or set the `AND/OR` logic switch on the conditions (Fig. 4.e). They can also apply the condition to one or all of the output columns.

4.2.2 Step 2: Demonstrate Actions and Automate. Once the user sets initial conditions, they can start the demonstration process by clicking "Start Recording" (Fig. 4.a). This initializes the WebRobot

[2]Microsoft Azure, https://tinyurl.com/55puwcf2

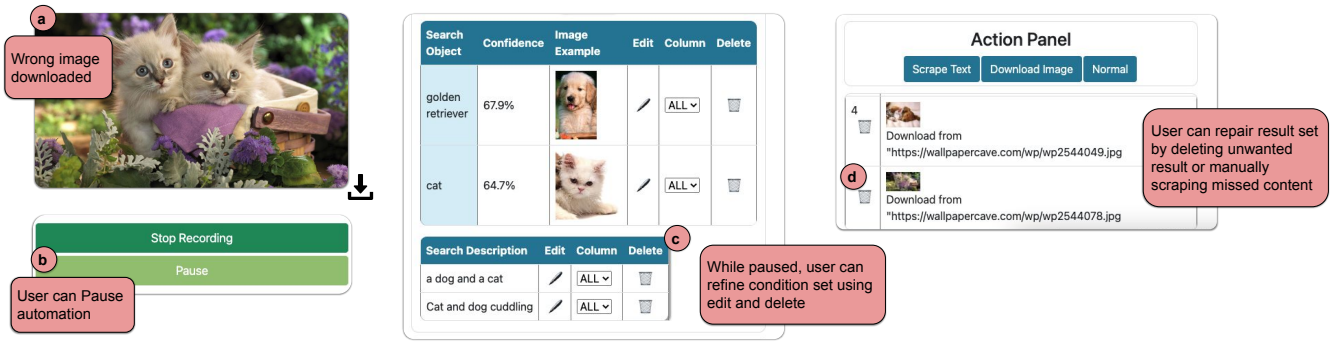


Figure 7: SemanticOn Error Repair Workflow. A user wants to download cute pictures of a cat and a dog cuddling, but they notice a photo of two cats is being scraped by mistake ⓐ. The user pauses the system ⓑ. Then, the user can edit the condition description, delete conditions ⓒ, and delete the incorrectly scraped result ⓓ.

system on the current web page. The user can then select one of the three actions under the action panel: Scrape Text, Download Image, and Normal (Fig. 4.j). The Scrape Text and Download Image actions allow users to click on the desired text or image element and extract the information into the table in output panel. In contrast, the Normal action allows the user to navigate and paginate the web page. For example, the user can use Normal to click into a web page, use Scrape Text to extract a passage, and switch back to Normal to return to the previous page. WebRobot registers this entire sequence of web macros and generates an automation program to repeat it.

After the user demonstrates their intended pattern of actions twice, WebRobot generates an automation program and predicts the next action. SemanticOn also enters a guided *semi-automation* mode. That is, for every data extraction action predicted, the system parses the content through machine learning models. A new image is considered matching with the current conditions if 1) an object specified in the image condition is detected (e.g. user specified a dog, and the new image consists of a human petting a dog), or 2) the new caption yields a similarity score above a certain threshold when compared with captions specified in the image condition. The image caption and text content similarity thresholds were set to 0.5 and 0.4, respectively, based on our benchmark testing. A new text is considered matching if it yields a similarity score above a threshold when compared with specified text conditions. We used a Sentence-Transformer to calculate the semantic similarity between captions and texts [67]. The model utilizes Sentence-BERT (SBERT), a pre-trained network that derives semantically meaningful sentence embeddings that can be compared using cosine-similarity. To avoid the low-score comparison between an entire paragraph and a key phrase, we chunk the text content and conditions into sentences and see if any pair results in a high similarity score.

4.2.3 Step 3: Refine, Repair, and Coordinate. To help users easily refine their intent (DG2) and repair mistakes (DG3), we adopted the *human-in-the-loop* approach [24, 59, 63] and introduced a *semi-automation mode* to ease the transition between manual demonstration and full system automation. First, to repair unexpected results or conditions, SemanticOn offers users a program pause function, which is important and unique in continuous AI systems. In the semi-automation mode, if new text or image content matches with the condition set, the user is notified by a prompt informing them which part of the content was matched. In contrast, if the new content is ambiguous or does not match with the condition set, the user is prompted with a set of suggested conditions (objects and captions for images, key phrases for text) generated by ML model processing (Fig. 5). The user can clarify their intent by selecting suggested conditions to append to the condition set. The new content will be accepted and added to the output results. Alternatively, the user can reject the system's prediction by clicking "Finish", in which case the condition set remains unchanged, and the new content is discarded. The semi-automation mode guides the user through condition specification and refinement. The user might begin with a high-level idea of their search query but can refine and adjust it upon seeing SemanticOn's interpretation of the result. The user can add new conditions to cover unforeseen cases or to modify/delete vague or over-specified conditions that filter results in an unintended way.

After going through predicted content, administering decisions, and refining conditions, the user might feel satisfied with the set of conditions. In that case, the user can choose to hide system prompts (Fig. 4.b) for rejected content and enter full automation. In this mode, SemanticOn continuously executes the predicted web macros and assesses each new piece of content, outputting the information accordingly. This is done by adding a condition evaluation step after executing every macro before extracting the data. However, when the user detects a filtering error, they can use the "Pause Automation" button to reclaim manual control (Fig. 7.b). When the system is paused, the user is free to modify the output result table. For each assessed piece of output, an information icon is displayed at the end of the table cell (Fig. 4.m). Once clicked, it will expand and display the ML processing results for that content. The user can repair system-made errors by deleting rows of ambiguous results accepted by the system but misaligned with the user's mental model (Fig. 7.d). They can also manually use the action panel to add content rejected by the system but matches the user's needs. In addition, the user can learn from the filtering results to add new conditions they omitted or under-specified. Likewise, condition editing and deletion are also available during the pause (Fig. 7.c). This way, the user can feel confident about the automatic selection of results, as they always have the power to reclaim manual control, repair

system mistakes, and clarify their intents. Note that during pause, none of these steps are recorded for program synthesis and do not affect the automation program.

5 SYSTEM EVALUATION

We conducted an in-person user study to evaluate SemanticOn's usability and compare the two methods for condition specifications. This evaluation was guided by the usage evaluation in the HCI toolkit evaluation strategy classification [41].

5.1 Participants

We recruited 10 people (5F5M, mean age 24.3, mean coding experience 4.6 years) at a large public university. Participants are denoted as P1-P10 in subsequent sections. Seven of the participants were graduate students, and three were undergraduates. Five participants have written web automation programs or used commercial tools to some extent. The participants were contacted by email to participate in a study where they would interact with computer software to create web automation programs.

5.2 Study Design

After signing our consent form, each participant first watched a tutorial video of SemanticOn's interface and features. Then participants performed five tasks using SemanticOn. For each task, they were given a task sheet with a semantic conditional intent. Similar to the posted tasks in online forums, the descriptions were intentionally vague, so the user could not copy the instructions verbatim and would need to formalize their own idea of how to specify and refine the conditions. The participants could request the experimenter's assistance at any time during the session. After the participants completed the tasks, we conducted a short interview with them regarding their experience. Additionally, they filled out a short survey with Likert scale and short-answer questions when they exited. The participants were compensated $25 for their time. Each session took 60-75 minutes and was conducted in person on our machine. All sessions were screen- and audio-recorded. Our study is approved by the IRB at our institution.

5.3 Tasks

Through an analysis of online web scraping request posts from iMacros Forum [3], and Stack Overflow [7], we designed five tasks that represent the challenge of creating web automation programs with semantic conditions. The Appendix (Fig 8) shows the details of the user study tasks. To evaluate the usability and utility of the system and compare the effectiveness of two intent specification methods, we designed the tasks with the following goals in mind: 1) the web content to be scraped should be realistic and easily understood by participants, 2) the tasks should extract information based on text and image content, 3) the condition criteria and the corresponding results should contain some ambiguity to allow room for refinement, and 4) the tasks should help SemanticOn to demonstrate the system's full capabilities. To achieve this goal, we adopted and piloted five web scraping requests by real users on online forums, two based on image-heavy websites (pet wallpapers[3], street photography[4]), two with text-heavy content (movie star biographies[5], list of novels[6]), and one task with a variety of both image and text content (beautiful places in the world[7]). Participants needed to extract only images or only text content for the first four tasks. In addition, they were randomly assigned to use *User Enters* and *System Suggests* as their specification method for the first four tasks. We counterbalanced the ordering of image versus text tasks and the assigned specification method to eliminate sequence and learning effects. To standardize task difficulty, participants needed to extract the top 15 text passages and/or images from the website based on a semantic condition. For the final task, participants needed to scrape both images and text in two separate columns and were free to use both condition specification methods. Task 5 served as an exploration task for participants to evaluate and compare the two specification methods and was not constrained by a result set or time. Participants engaged in this task until the end of the study. Therefore, Task 5 was not included in the quantitative analysis for accuracy and duration in the later sections.

We designed the conditions based on the content itself and established a ground truth result set that was compared with the participant's result to measure accuracy. The conditions were designed so that in the ground truth set, the number of results passing and failing the condition across all tasks was approximately the same (31 passes, 29 fails).

5.4 Results

5.4.1 Time and accuracy. The user study recorded 40 scenarios (10 participants x 4 tasks); one instance was discarded from analysis due to a recording issue, resulting in 39 total task completions. Table 1 lists the average time (in minute:second) each participant spent and the accuracy (percentage of correctly included and excluded results) on each task. The overall average duration is 06:10 (image tasks *mean*=05:55, text tasks *mean*=06:25), and the overall average task accuracy is 80.7% (image tasks $mean = 83.0\%$, text tasks $mean = 78.5\%$). We could not identify statistical significance in the difference in accuracy and duration across image and text tasks.

We also analyzed the data comparing usages of *User Enters* versus *System Suggests*. The average duration was 06:22 for *User Enters* tasks and 05:57 for *System Suggests* tasks. In terms of mean accuracy, participants achieved 83.0% for *User Enters* tasks and 78.5% for *System Suggests* tasks. Again, the differences are unable to be identified as statistically significant.

5.4.2 Overall effectiveness in intent specification. In the exit survey, participants rated the ease of use of SemanticOn overall, the ease of use of each specification method, their mental effort, and their trust in the system. Table 2 displays the average score for Likert scale questions on SemanticOn's usability. On a scale of 1 (strongly disagree, very negative) to 7 (strongly agree, very positive), participants believed their specified conditions were displayed in an easily understandable way ($mean = 6.0, SD = 1.1$) and that the

[3]Pet Wallpapers, https://tinyurl.com/47kr3mz6
[4]Street Photographers, https://tinyurl.com/bdppfa48
[5]Movie Stars, https://tinyurl.com/zsnkne5r
[6]Best Romance Novels, https://tinyurl.com/mryjs47h
[7]Vacation Destinations, https://tinyurl.com/4peucx3p

Task index	Completion time (mean, SD in mm:ss)	Accuracy
1	05:06 (01:43)	85.3%
2	06:58 (02:28)	68.9%
3	06:44 (01:50)	81.3%
4	05:56 (01:50)	86.0%

Table 1: Average time spent on each task. Tasks 1 and 3 are image tasks; tasks 2 and 4 are text tasks.

system-generated prompts in semi-automation mode helped them refine their intent ($mean = 5.6, SD = 0.97$). However, participants had polarized opinions on the statement *"it was easy to coordinate (claim controls, pause/resume, show/hide prompts) with SemanticOn to repair under- or over-specifications,"* with 5 participants agreeing and 3 participants disagreeing ($mean = 4.7, SD = 1.7$). Overall, the users widely accepted the semi-automation workflow towards automation and the opportunity to adjust conditions later on by pausing. 9 out of 10 participants added refinement conditions for Tasks 1-4, and all participants utilized the system prompts in semi-automation mode to refine their conditions in Task 5.

5.4.3 Error-handling Analysis. As per DG3, SemanticOn offers error-handling techniques for users during full automation. We measured participants' usage of three error repair methods: condition deletion, result deletion, and manual result addition. Most users opted to use the pause and repair functionalities, as 7 participants deleted their specified conditions due to inaccuracies, 6 participants removed undesired output to repair system mistakes, and 2 participants manually added desired contents missed by the filter. The utilization of the error-handling features varied based on the task type. For condition deletion, we found 16 instances across image tasks and 0 instances for text tasks. Similarly, there were 11 instances of result deletion for image tasks and 2 for text tasks. As for manual result addition, we found 3 instances for image tasks and none for text tasks. This difference can be related to the content processing speeds for visual versus textual materials. The condition specification method also plays a role in the usage of error-handling features. We observed 4 instances of condition deletions for *User Enters* and 12 for *System Suggests*. In addition, we found 4 result deletion instances for *User Enters* and 9 for *System Suggests*. This can be attributed to the difference in the required effort for each workflow. We expand on these varieties in our Discussion.

5.4.4 Comparison between User Enters and System Suggests. Regarding the difference between the two specification methods, participants found both experiences comparable but thought *System Suggest* was easier to use. P10 said, *"[My preference] really depends on the task, but I really liked the System Suggests mode, especially for words it was really easy to use and accurate."* When assessing the overall experience to specify conditions, participants rated *User Enters* slightly higher ($mean = 5.3, SD = 1.1$) than *System Suggests* ($mean = 5.1, SD = 1.2$). But when evaluating the ease of use for each interaction, users rated entering their own natural language descriptions lower ($mean = 5.1, SD = 1.1$) than brushing, highlighting, or choosing system-suggested conditions ($mean = 5.7, SD = 0.82$). We were unable to identify statistical significance in these differences. However, when measuring the average number of initial conditions set before the demonstration, participants specified an average of 2.6 conditions for *User Enters* and 4.1 conditions for *System Suggests* ($p < 0.05$). This points to a difference in ease-of-use and condition-generation capability between the two methods, which is further explored in Discussion.

Question	Likert Scale (Mean, SD)
User Enters Experience	5.3 (1.0)
System Suggests Experience	5.1 (1.2)
User Enters Ease of Use	5.1 (1.0)
System Suggests Ease of Use	5.7 (0.8)
Coordination to Refine Specifications	4.7 (1.6)
Usefulness of Generated Prompts	5.6 (0.9)
Conditions Displayed Clearly	6.0 (1.0)
Trust in Full Automation	4.5 (1.5)
Success in Completing Task	5.0 (0.9)
Mental Demand	4.6 (1.7)
Effort to Achieve Results	4.9 (1.3)
Feelings of Insecurity and Stress	5.2 (1.7)
Feelings of Being Hurried or Rushed	4.9 (1.6)

Table 2: Survey Responses. For section one (top), 1 is very negative, and 7 is very positive. For section two (bottom), 1 is very high mental demand, effort, insecurity and stress, and feelings of being hurried and rushed.

5.4.5 Mental effort and AI system trust. Participants also reported on their mental effort in completing the tasks and their trust in the AI system in the survey. Participants generally reported medium to high mental effort using SemanticOn, especially at the start of the study when they had to familiarize themselves with the interface and remember the workflow (P2, P5, P6, P7). On a scale of 1 (very demanding, very hard, not successful) to 7 (not demanding, not hard at all, very successful), participants experienced medium mental demand ($mean = 4.6, SD = 1.8$) and medium effort ($mean = 4.9, SD = 1.4$). They believed they were relatively successful in accomplishing the tasks ($mean = 5.0, SD = 0.94$).

We also found that the participants exhibited a medium level of trust ($mean = 4.5, SD = 1.6$) towards the AI system's prediction in full automation. Six users were comfortable entering full automation mode for at least one task. However, P3 and P7 expressed very low trust in the automated prediction (both rated 2 out of 7 in survey). During the interview, some participants (P5, P9) also commented that their trust in the system depended on the content type and the number of refinements required for each task. For example, P5 mentioned they *"trust this picture task a bit more...[because] text has more group[ing]s and potential variations"* when they used 11 text refinements and only 1 image refinement. We analyze the effect of content type on users' perception of the task and our system in the Discussion.

6 DISCUSSION

Based on the evaluation, we present analysis of the human-AI collaboration techniques and the role of each agent in SemanticOn. We also discuss the implications of adding similarity-based machine learning models in symbolic PBD systems. In addition, we report findings on users' trade-offs in using *User Enters* and *System Suggests*, as well as the intent specification effectiveness and system limitations to provide insights on future designs.

6.1 Human-AI Collaboration

The user workflow of SemanticOn consists of both a human and an artificial intelligence agent; they exchange control in different parts of the interaction and collaborate to build a complete and accurate web automation program. In SemanticOn, we utilize machine learning models' classification efficiency to rapidly process texts and images, summarize the content, and provide suggested semantic filters to the user. This saves the user's mental effort significantly, as it is no longer the human's role to parse content and identify an inclusion or exclusion decision. However, it also introduces a new source of error to web automation, as similarity-based models can mislabel inputs and result in false positive and false negative data. This is why human is always involved in the continuous human-AI collaboration in SemanticOn. After a couple of demonstrations, the user is prompted with potential semantic conditions for each processed content in the semi-automation mode. Additionally, even when the user feels satisfied with the set of conditions, the program can still be paused at any time if a mistake is spotted or the user wants to edit the conditions.

From the evaluation of this interaction paradigm, most participants (7/10) reported that the workflow's path from demonstration to guided semi-automation to full automation was intuitive and useful. The semi-automation phase allowed users to calibrate conditions with SemanticOn and better understand the AI system's interpretation of the conditions. This is important as users might not have had a well-defined semantic condition at the program's start. P4 believed that the *"human brain still needs to think about [how to describe the condition]...and link to all the keywords. These words might not specifically come to mind to the human."* In this case, the initial conditions may not include all of the user's needs, resulting in low filter accuracy. The semi-automation mode provides aid for this. For example, P3 mentioned that even if the results were not accurate at the start, *"I can interactively try to make [automated prediction] better as it goes."* P4 also mentioned that *"the transitional period to trust the process is good, saves a lot of fuss."*

However, the steps to refine conditions and results through system prompts and automation pauses could require too much user effort and become time-consuming. P5 commented that the entire user experience was *"a little slow...I think the fine-tuning part is slow. Once you start [full automation], then it's fine."* P2 reported that *"it took me multiple times to figure out the sequence of the buttons."* When they spotted system mistakes, P1 and P4 decided not to pause and amend the errors because it took too much effort. They believed the misclassified content was not detrimental to the automation task, as there was no requirement for accuracy. While supplying users with multiple refinement tools enhances robustness, the system must be cautious not to overload users with interaction techniques and interruptions.

6.2 Similarity-based Model in PBD Systems

A main contribution of SemanticOn is introducing a new interaction paradigm for users to continuously add/refine semantic conditions in programming-by-demonstration (PBD) systems. In particular, it enables users to express intent via similarity-based machine learning models. This specification is at a higher abstraction level than pure symbolic systems based on DOM structure. Here, we discuss the advantages and disadvantages of expressing intent using similarity-based statistical models versus pure symbolic systems, the implications of adding statistical models in PBD systems, and the generalizability and viability of SemanticOn against errors.

Traditional symbolic automation systems like WebRobot are robust at understanding specific user actions and generalizing them into repeatable steps in a synthesized program. This is achieved by iterating over the elements in the DOM structure of the websites and generalizing the pattern. Meanwhile, statistical machine learning models are useful at parsing high-level user intent and matching unstructured, but semantically similar concepts, for example extracting key entities from a natural language input. However, both systems alone have their limitations. Similarity-based models lack reasoning capability. The machine learning models we employ classify the inclusion and exclusion of content based on semantic similarity, but they cannot derive further actions based on specific criteria. For example, the models alone could not construct a program that repeatedly takes in an image URL, searches it, and downloads the image based on semantic conditions. It is efficient in the last filtering step but ineffective when structural elements need to be generalized. In contrast, symbolic models that reason about a set of user instructions over a certain website structure might fail to generalize if the query is unstructured or outside the scope of the task domain, while similarity models can cover a wider range of input content that are not constrained by the structure. SemanticOn combines these two techniques in an effective way that seals the gap between users' semantic intent and web automation.

From a practical standpoint, training an intelligent statistical model for the tasks we cover in SemanticOn's user evaluation would be effortful. Systems like Calendar.help [22] require many expert heuristics and even human workers to automate a very specific task in event scheduling across different parties. In SemanticOn, users compromise some effort by doing two rounds of demonstrations of their desired web actions. Still, the automation tasks can be generalized to a diverse set of data on many different websites. Introducing similarity-based models in symbolic systems does present a new source of error as the models can filter content incorrectly, in addition to program synthesis errors in pure-symbolic systems. To amend this, SemanticOn provides continuous condition refinement and output edit/delete options so that users can repair classification errors easily and yield more accurate results.

Our work does not rely on a specific program synthesizer to generate a web automation program or a specific machine learning model to filter content semantically. The novelty of SemanticOn is the set of condition specification, refinement, and error-correction interactions, which can be generalized to other PBD systems.

6.3 User Enters and System Suggests

One emphasis of this project is to compare the usability of the two semantic condition specification methods. To summarize the findings, participants shifted their preference between *System Suggests* and *User Enters* based on two factors: perceived effort and the type of content presented. When participants expressed their preferences and the rationale behind using each specification method, we also discovered a trade-off between their sense of control and the perceived effort.

6.3.1 Perceived effort based on content types. Perceived effort plays an important role in user preference. When it was easy to summarize the target content on which users needed to set conditions, participants preferred to use *User Enters* to express their intent instead of selecting *System Suggests*, which could be too broad or too specific. P3 remarked that *"[User Enters makes] you feel like [you have] more control because you are looking for a specific thing."*

On the contrary, when the target content was information-heavy and hard to encapsulate in a short natural language description, participants preferred to use *System Suggests* to highlight the details and let SemanticOn present potential conditions. P6 commented that *"I do like the possibility of being able to say this is what I see in the image [based on System Suggests]... and [highlight] a part of the image that I think is important."*

When participants were free to use both *User Enters* and *System Suggests* (i.e. Task5 in Appendix Fig 8), we found that text content inherently took more effort to process, and 7 out of 10 participants opted to use *System Suggests* to avoid reading a large block of text. In contrast, despite containing a variety of objects and entity relationships, the corresponding image content could be encoded and summarized more quickly, and 9 out of 10 participants chose to specify their intents with *User Enters*.

This preference based on content type persisted during condition refinement in semi-automation and full automation modes. Participants expressed difficulty in parsing textual information rapidly, as they could not distinguish whether the system made the correct selection. P3 said that *"text task in general is much more difficult,"* and P5 commented, *"when it comes to text task, I'm a little lazy to read through all this... so system can just get me relevant keywords."* Our observations support this as users were far more likely to repair errors for image tasks than text tasks by deleting inaccurate conditions and incorrect results. Participants can quickly identify whether an image fits the condition set, but they can not parse through a block of text as the automation quickly processes each row of content. Based on this finding, future designs should account for the processing effort for text content and provide users more time and aid in understanding and summarizing the information.

6.3.2 Sense of Control and mental effort trade-off. Another key factor for specification preference was the sense of control. Three participants (P3, P7, P10) listed a better sense of control as to why they preferred *User Enters*, where they could tailor the condition using their terms. In addition, in cases where *System Suggests* suggestions were inaccurate (e.g., failing to detect relevant text entities or generate image captions at the appropriate granularity), some participants (P2, P4, P6, P10) switched to *User Enters*.

However, there was a trade-off between the sense of control and the required effort. Participants rated a lower ease-of-use score on average for *User Enters* (5.1 versus 5.7 for *System Suggests)*, despite it having fewer interaction steps. Three participants (P4, P5, P10) pointed out that they preferred *System Suggests* due to the convenience and ease of mental effort. *System Suggests* is also advantageous in reporting details that escaped the user's attention. For example, P4 pointed out that they preferred *System Suggests* mode because it sometimes captured things that the user failed to recognize or think of. Additionally, P7 mentioned that they *"quite like System Suggests more, [because it is] more systematic and highlights specific things."* Participants could avoid cognitive overload by handing the processing labor to SemanticOn through *System Suggests*, but they were limited to the generated set of conditions for each item. This trade-off also affects participants' usage of error handling features during automation. With the increased cognitive load in *User Enters*, participants are much less likely to pause and repair their mistakes than when using *System Suggests*. However, this could be explained by participants adopting less accurate prompts from *System Suggests*, which leads to more errors overall for users to fix. Future work could potentially create a combined approach where users can edit system-suggested conditions, yielding a sense of control and avoiding heavy user efforts.

6.4 System Effectiveness

Most participants (8/10) agreed that SemanticOn is an effective tool for web automation and that the condition specification aspect is useful. Compared with manual effort and traditional data scraping methods such as writing automation scripts, participants found SemanticOn's no-code solution novel and easier to use. P5 mentioned that in manual scraping scenarios, *"Control-F would only get you so far"* and that the system is *"much easier than writing your own script...[and] more functionalit[ies]"* In addition, P7 thought this tool *"would be super helpful for someone not comfortable with code. [It provides] low effort but maximum reward."*

More than half of the participants (6/10) believed that SemanticOn could be applied to realistic tasks. However, some focused on the conditional selection aspect, and others emphasized on the complexity of the task. For example, P2 commented that the system is *"very good for selective tasks [when you] only want a few images from a lot of them."* Similarly, P5 suggested that SemanticOn could be used for websites without robust filtering, search engine, or categories because *"it could easily make any website sort-able."* On the other hand, P1 enjoyed using this tool on Task 5 with both image and text conditions combined and proposed that *"this type of tool would benefit... power users [for tasks] like creating a dataset...[and] dealing with complex things."* These reports provide evidence for the need of no-code solutions in conditional web automation tasks.

6.5 System Limitations

There are several limitations to our system. First is the efficiency of coordination. As mentioned, some participants found the refinement process complicated and slow and were reluctant to utilize the functionality to increase system accuracy. In future designs,

the researchers could simplify the user interface and the interaction steps and condense the system prompt information to reduce cognitive load.

Another limitation is the relatively low accuracy of the machine learning models it relies on to classify text and image content. Partially, this relates to a design decision we made when building SemanticOn. We first used two state-of-the-art models—OFA [74], which uses a sequence-to-sequence learning framework, and ImageAI, which uses a convolutional neural network [56], for image captioning and object detection, and we were able to achieve high accuracy. However, the image processing was computation-heavy and took more than 30 seconds per image on an off-the-shelf laptop. To reduce the gulf of evaluation [62], we switched to the Microsoft Azure computer vision model [32] for near-instant cloud processing but with lower accuracy and detail level for some content domains. Similarly, for key phrase extraction, we employed the Microsoft Azure language model, sending text contents via REST API calls for near-instant processing. But the extracted key phrases often have a low level of abstraction (i.e. extracting entities and nouns in a sentence without indication of interaction) and could not provide high-level descriptions like those used for image captions.

Additionally, SemanticOn builds upon an existing symbolic program synthesis system, WebRobot, which has its own constraints and requirements. And the introduction of similarity-based machine learning models also presents ambiguity to the source of error between incorrect program synthesis steps and result misclassifications. The user might be confused when encountering unexpected behavior, as it could be a product of a user demonstration error or the system's misunderstanding of user intents, which requires a different workflow to repair. However, in our user study, we emphasize repairing errors from machine learning model mislabelling as that is the novel part of SemanticOn.

6.6 Future Work

We summarized three points of feedback that are relevant for future works. First, we found that participants spent a longer time encoding and processing texts than images. This is coherent with previous studies on human capability in consuming different types of information [53]. But, this finding is significant in full automation, when the system rapidly loops through web page content, giving users little time to identify potential selection errors. Future works could enable multi-modal interaction (e.g., voice) to reduce the effort of information processing [20]. Additionally, one can provide more indicators for users to quickly recognize the information represented in text elements (e.g. highlighting matching key phrases) and decide whether the content should be included based on the semantic condition.

Second, participants separately favored one of the two condition specification methods, *User Enters* and *System Suggests*, depending on the content type and the amount of information they needed to process. Similar to the neurosymbolic program synthesis approach [16, 65], future works could create a unified technique where users could benefit from the instant and comprehensive results from machine learning models while preserving users' power to specify conditions on their own terms.

Lastly, future designs could further improve the system usability by reducing the mental effort. The dual process of collaborating with an AI on both demonstrating web macros to synthesize an automation program and specifying semantic conditions to filter web content requires a decent amount of mental effort, especially for users who are less familiar with programming or web scraping. Therefore, future designs should alleviate users' cognitive load with a more minimalist UI and more user guidance.

7 CONCLUSION

In this work, we designed and developed SemanticOn, a collaborative system that allows users to specify and refine visual and textual conditions through user-entered descriptions and system-suggested prompts in a web automation program. In a system evaluation, we found that participants can effectively use SemanticOn to create conditional filters and refine them via continuous human-AI collaboration, collecting selective web content with high accuracy. Participants' feedback also suggested that a guided semi-automation mode, where users authorize system predictions, helped clarify user intents. We also found a trade-off between *User Enters* and *System Suggests* regarding user effort and the sense of control. Our work can point directions to future system and interaction designs for user-intent specification and refinement in a continuous human-AI collaboration setting.

ACKNOWLEDGMENTS

We thank all our participants and reviewers. This research was supported in part by the National Sciences and Engineering Research Council of Canada (NSERC) under Grant IRCPJ 545100 - 18.

REFERENCES

[1] 2022. Beautiful Soup. http://www.crummy.com/software/BeautifulSoup/ Accessed: April, 2022.
[2] 2022. iMacro. https://www.progress.com/imacros Accessed: April, 2022.
[3] 2022. iMacro Forum. https://forum.imacros.net/ Accessed: April, 2022.
[4] 2022. Puppeteer. https://pptr.dev/ Accessed: April, 2022.
[5] 2022. Scrapy. https://scrapy.org/ Accessed: April, 2022.
[6] 2022. Selenium. https://www.selenium.dev/ Accessed: April, 2022.
[7] 2022. Stack Overflow. https://stackoverflow.com/search?q=%5Bselenium%5D+semantic Accessed: April, 2022.
[8] 2022. UiPath. https://www.uipath.com/ Accessed: April, 2022.
[9] James Allen, Nathanael Chambers, George Ferguson, Lucian Galescu, Hyuckchul Jung, Mary Swift, and William Taysom. 2007. Plow: A collaborative task learning agent. In *AAAI*, Vol. 7. 1514–1519.
[10] Saleema Amershi, James Fogarty, and Daniel Weld. 2012. Regroup: Interactive machine learning for on-demand group creation in social networks. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 21–30.
[11] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).
[12] A Blackwell. 2000. Your Wish is My Command: Giving Users the Power to Instruct their Software, chapter SWYN: a visual representation for regular expressions. *M. Kaufmann* (2000), 245–270.
[13] Alan F Blackwell. 2001. SWYN: A visual representation for regular expressions. In *Your wish is my command*. Elsevier, 245–XIII.
[14] Carrie J Cai, Emily Reif, Narayan Hegde, Jason Hipp, Been Kim, Daniel Smilkov, Martin Wattenberg, Fernanda Viegas, Greg S Corrado, Martin C Stumpe, et al. 2019. Human-centered tools for coping with imperfect algorithms during medical decision-making. In *Proceedings of the 2019 chi conference on human factors in computing systems*. 1–14.
[15] Sarah E Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 963–975.
[16] Qiaochu Chen, Aaron Lamoreaux, Xinyu Wang, Greg Durrett, Osbert Bastani, and Isil Dillig. 2021. Web question answering with neurosymbolic program

synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 328–343.

[17] Yan Chen and Tovi Grossman. 2021. Umitation: Retargeting UI Behavior Examples for Website Design. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 922–935.

[18] Yan Chen, Jaylin Herskovitz, Walter S Lasecki, and Steve Oney. 2020. Bashon: A Hybrid Crowd-Machine Workflow for Shell Command Synthesis. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–8.

[19] Yan Chen, Sang Won Lee, and Steve Oney. 2021. CoCapture: Effectively Communicating UI Behaviors on Existing Websites by Demonstrating and Remixing. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*.

[20] Yan Chen, Sang Won Lee, Yin Xie, YiWei Yang, Walter S Lasecki, and Steve Oney. 2017. Codeon: On-demand software development assistance. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. 6220–6231.

[21] Yan Chen, Maulishree Pandey, Jean Y Song, Walter S Lasecki, and Steve Oney. 2020. Improving crowd-supported gui testing with structural guidance. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–13.

[22] Justin Cranshaw, Emad Elwany, Todd Newman, Rafal Kocielnik, Bowen Yu, Sandeep Soni, Jaime Teevan, and Andrés Monroy-Hernández. 2017. Calendar.help. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM. https://doi.org/10.1145/3025453.3025780

[23] Biplab Deka, Zifeng Huang, Chad Franzen, Jeffrey Nichols, Yang Li, and Ranjitha Kumar. 2017. Zipt: Zero-integration performance testing of mobile app designs. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 727–736.

[24] Rui Dong, Zhicheng Huang, Ian Iong Lam, Yan Chen, and Xinyu Wang. 2022. WebRobot: Web Robotic Process Automation using Interactive Programming-by-Demonstration. *arXiv preprint arXiv:2203.09993* (2022).

[25] Ian Drosos, Titus Barik, Philip J Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *Proceedings of the 2020 CHI conference on human factors in computing systems*. 1–12.

[26] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-step live programming by example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 614–626.

[27] Michael H Fischer, Giovanni Campagna, Euirim Choi, and Monica S Lam. 2021. DIY assistant: a multi-modal end-user programmable virtual assistant. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 312–327.

[28] James Fogarty, Desney Tan, Ashish Kapoor, and Simon Winder. 2008. CueFlik: interactive concept learning in image search. In *Proceedings of the sigchi conference on human factors in computing systems*. 29–38.

[29] Philip J Guo, Sean Kandel, Joseph M Hellerstein, and Jeffrey Heer. 2011. Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. 65–74.

[30] William R Harris and Sumit Gulwani. 2011. Spreadsheet table transformations from examples. *ACM SIGPLAN Notices* 46, 6 (2011), 317–328.

[31] Björn Hartmann, Leslie Wu, Kevin Collins, and Scott R Klemmer. 2007. Programming by a sample: rapidly creating web applications with d. mix. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*. 241–250.

[32] Xiaowei Hu, Xi Yin, Kevin Lin, Lijuan Wang, Lei Zhang, Jianfeng Gao, and Zicheng Liu. 2020. VIVO: Visual Vocabulary Pre-Training for Novel Object Captioning. *arXiv preprint arXiv:2009.13682* (2020).

[33] Ellen Jiang, Edwin Toh, Alejandra Molina, Aaron Donsbach, Carrie J Cai, and Michael Terry. 2021. Genline and genform: Two tools for interacting with generative language models in a code editor. In *The Adjunct Publication of the 34th Annual ACM Symposium on User Interface Software and Technology*. 145–147.

[34] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the sigchi conference on human factors in computing systems*. 3363–3372.

[35] Rebecca Krosnick and Steve Oney. 2021. Understanding the Challenges and Needs of Programmers Writing Web Automation Scripts. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–9.

[36] David Kurlander, Allen Cypher, and Daniel Conrad Halbert. 1993. *Watch what I do: programming by demonstration*. MIT press.

[37] David Kurlander and Steven Feiner. 1992. A history-based macro by example system. In *Proceedings of the 5th annual ACM symposium on User interface software and technology*. 99–106.

[38] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. 2003. Programming by demonstration using version space algebra. *Machine Learning* 53, 1 (2003), 111–156.

[39] Tessa A Lau, Pedro M Domingos, and Daniel S Weld. 2000. Version Space Algebra and its Application to Programming by Demonstration.. In *ICML*. Citeseer, 527–534.

[40] Vu Le and Sumit Gulwani. 2014. Flashextract: A framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 542–553.

[41] David Ledo, Steven Houben, Jo Vermeulen, Nicolai Marquardt, Lora Oehlberg, and Saul Greenberg. 2018. Evaluation strategies for HCI toolkit research. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–17.

[42] Gilly Leshed, Eben M Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: automating & sharing how-to knowledge in the enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 1719–1728.

[43] Ian Li, Jeffrey Nichols, Tessa Lau, Clemens Drews, and Allen Cypher. 2010. Here's what I did: Sharing and reusing web activity with ActionShot. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 723–732.

[44] Toby Jia-Jun Li, Amos Azaria, and Brad A Myers. 2017. SUGILITE: creating multimodal smartphone automation by demonstration. In *Proceedings of the 2017 CHI conference on human factors in computing systems*. 6038–6049.

[45] Toby Jia-Jun Li, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenze Shi, Wanling Ding, Tom M. Mitchell, and Brad A. Myers. 2018. APPINITE: A Multi-Modal Interface for Specifying Data Descriptions in Programming by Demonstration Using Natural Language Instructions. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 105–114. https://doi.org/10.1109/VLHCC.2018.8506506

[46] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M Mitchell, and Brad A Myers. 2019. Pumice: A multi-modal agent that learns concepts and conditionals from natural language and demonstrations. In *Proceedings of the 32nd annual ACM symposium on user interface software and technology*. 577–589.

[47] Henry Lieberman. 1994. A user interface for knowledge acquisition from video. In *AAAI*. Citeseer, 527–534.

[48] James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A Lau. 2009. End-user programming of mashups with vegemite. In *Proceedings of the 14th international conference on Intelligent user interfaces*. 97–106.

[49] Greg Little, Tessa A Lau, Allen Cypher, James Lin, Eben M Haber, and Eser Kandogan. 2007. Koala: capture, share, automate, personalize business processes on the web. In *Proceedings of the SIGCHI conference on Human factors in computing systems*. 943–946.

[50] Vivian Liu and Lydia Chilton. 2022. Design Guidelines for Prompt Engineering Text-to-Image Generative Models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*.

[51] David L Maulsby, Ian H Witten, and Kenneth A Kittlitz. 1989. Metamouse: Specifying graphical procedures by example. *ACM SIGGRAPH Computer Graphics* 23, 3 (1989), 127–136.

[52] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User interaction models for disambiguation in programming by example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 291–301.

[53] Richard Mayer and Joan Gallini. 1990. When Is an Illustration Worth Ten Thousand Words? *Journal of Educational Psychology* 82 (12 1990), 715–726. https://doi.org/10.1037/0022-0663.82.4.715

[54] Dan H Mo and Ian H Witten. 1992. Learning text editing tasks from examples: a procedural approach. *Behaviour & Information Technology* 11, 1 (1992), 32–45.

[55] Francesmary Modugno and Brad A Myers. 1994. *Pursuit: Visual programming in a visual domain*. Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE.

[56] Moses and John Olafenwa. 2018–. ImageAI, an open source python library built to empower developers to build applications and systems with self-contained Computer Vision capabilities. https://github.com/OlafenwaMoses/ImageAI

[57] Brad A Myers, Dario A Giuse, Roger B Dannenberg, Brad Vander Zanden, David S Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. 1995. GARNET comprehensive support for graphical, highly interactive user interfaces. In *Readings in Human–Computer Interaction*. Elsevier, 357–371.

[58] Aaditya Naik, Jonathan Mendelson, Nathaniel Sands, Yuepeng Wang, Mayur Naik, and Mukund Raghothaman. 2021. Sporq: An Interactive Environment for Exploring Code using Query-by-Example. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 84–99.

[59] Julie L Newcomb and Rastislav Bodik. 2019. Using human-in-the-loop synthesis to author functional reactive programs. *arXiv preprint arXiv:1909.11206* (2019).

[60] Wode Ni, Joshua Sunshine, Vu Le, Sumit Gulwani, and Titus Barik. 2021. reCode: A Lightweight Find-and-Replace Interaction in the IDE for Transforming Code by Example. In *The 34th Annual ACM Symposium on User Interface Software and Technology*. 258–269.

[61] Jeffrey Nichols and Tessa Lau. 2008. Mobilization by demonstration: using traces to re-author existing web sites. In *Proceedings of the 13th international conference on Intelligent user interfaces*. 149–158.

[62] Don Norman. 2013. *The design of everyday things: Revised and expanded edition*. Basic books.

[63] Besmira Nushi, Ece Kamar, Eric Horvitz, and Donald Kossmann. 2017. On human intellect and machine failures: Troubleshooting integrative machine learning systems. In *Thirty-First AAAI Conference on Artificial Intelligence*.

[64] Peter O'Donovan, Aseem Agarwala, and Aaron Hertzmann. 2015. Designscape: Design with interactive layout suggestions. In *Proceedings of the 33rd annual ACM conference on human factors in computing systems*. 1221–1224.

[65] Emilio Parisotto, Abdel-rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. 2016. Neuro-symbolic program synthesis. *arXiv preprint arXiv:1611.01855* (2016).

[66] Hila Peleg and Nadia Polikarpova. 2020. Perfect is the enemy of good: Best-effort program synthesis. *Leibniz international proceedings in informatics* 166 (2020).

[67] Nils Reimers and Iryna Gurevych. 2019. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (2019).

[68] Alborz Rezazadeh Sereshkeh, Gary Leung, Krish Perumal, Caleb Phillips, Minfan Zhang, Afsaneh Fazly, and Iqbal Mohomed. 2020. VASTA: a vision and language-assisted smartphone task automation system. In *Proceedings of the 25th international conference on intelligent user interfaces*. 22–32.

[69] Atsushi Sugiura and Yoshiyuki Koseki. 1998. Internet scrapbook: automating web browsing tasks by demonstration. In *Proceedings of the 11th annual ACM symposium on User interface software and technology*. 9–18.

[70] Amanda Swearngin, Chenglong Wang, Alannah Oleson, James Fogarty, and Amy J Ko. 2020. Scout: Rapid Exploration of Interface Layout Alternatives through High-Level Design Constraints. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–13.

[71] Priyan Vaithilingam and Philip J Guo. 2019. Bespoke: Interactively synthesizing custom GUIs from command-line applications by demonstration. In *Proceedings of the 32nd annual ACM symposium on user interface software and technology*. 563–576.

[72] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Interactive query synthesis from input-output examples. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1631–1634.

[73] Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J Ko. 2021. Falx: Synthesis-powered visualization authoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–15.

[74] Peng Wang, An Yang, Rui Men, Junyang Lin, Shuai Bai, Zhikang Li, Jianxin Ma, Chang Zhou, Jingren Zhou, and Hongxia Yang. 2022. Unifying Architectures, Tasks, and Modalities Through a Simple Sequence-to-Sequence Learning Framework. *CoRR* abs/2202.03052 (2022). arXiv:2202.03052 https://arxiv.org/abs/2202.03052

[75] Andrew J Werth and Brad A Myers. 1993. Tourmaline (abstract) macrostyles by example. In *Proceedings of the INTERACT'93 and CHI'93 Conference on Human Factors in Computing Systems*. 532.

[76] Tom Yeh, Tsung-Hsiang Chang, and Robert C Miller. 2009. Sikuli: using GUI screenshots for search and automation. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology*. 183–192.

[77] Tianyi Zhang, Zhiyang Chen, Yuanli Zhu, Priyan Vaithilingam, Xinyu Wang, and Elena L. Glassman. 2021. Interpretable Program Synthesis. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 105, 16 pages. https://doi.org/10.1145/3411764.3445646

[78] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L Glassman. 2020. Interactive Program Synthesis by Augmented Examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. 627–648.

[79] Zheng Zhang, Zheng Xu, Yanhao Wang, Bingsheng Yao, Daniel Ritchie, Tongshuang Wu, Mo Yu, Dakuo Wang, and Toby Jia-Jun Li. 2022. StoryBuddy: A Human-AI Collaborative Agent for Parent-Child Interactive Storytelling with Flexible Parent Involvement. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*.

[80] Nanxuan Zhao, Nam Wook Kim, Laura Mariah Herman, Hanspeter Pfister, Rynson WH Lau, Jose Echevarria, and Zoya Bylinskii. 2020. Iconate: Automatic compound icon generation and ideation. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–13.

A APPENDIX

#	Task Description	Condition Type	Inclusion Example	Exclusion Example
1	Download pictures with a dog and a cat interacting	Image only	 Dog and cat interacting	 Cat only, no interactions
2	Scrape book descriptions that highlight a female protagonist's story	Text only	 Heroine perspective	 No female perspective
3	Download photos if it contains multiple people interacting	Image only	 Two people interacting	 No interactions
4	Scrape movie star biographies that writes about their acting	Text only	 Biography includes acting	 No acting in biography
5	Scrape the description and download the image if: 1) Text mentions outdoor activities and 2) Image contains mountain and water	Text and image	 Image has mountain and water and description mentions outdoor activities	 No outdoor activity description, no mountains

Figure 8: Task descriptions with inclusion and exclusion examples