CrossCode: Multi-level Visualization of Program Execution

Devamardeep Hayatpur
University of California, San Diego
La Jolla, California, USA
dshayatpur@ucsd.edu

Daniel Wigdor
University of Toronto
Toronto, Canada
daniel@dgp.toronto.edu

Haijun Xia
University of California, San Diego
La Jolla, California, USA
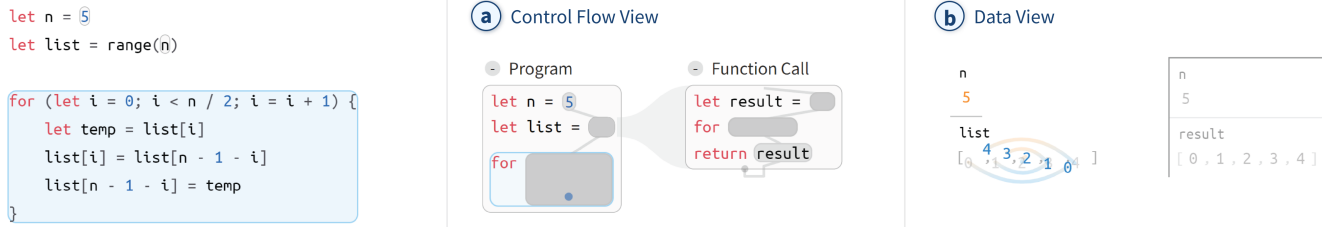haijunxia@ucsd.edu

Figure 1: Screenshot from CROSSCODE for code to reverse a list. (a) An overview of the execution with which users interact to navigate levels of detail; (b) Program state and changes to data indicated with animations, traces, and color encoding.

ABSTRACT

Program visualizations help to form useful mental models of how programs work, and to reason and debug code. But these visualizations exist at a fixed level of abstraction, e.g., line-by-line. In contrast, programmers switch between many levels of abstraction when inspecting program behavior. Based on results from a formative study of hand-designed program visualizations, we designed CROSSCODE, a web-based program visualization system for JavaScript that leverages structural cues in syntax, control flow, and data flow to aggregate and navigate program execution across multiple levels of abstraction. In an exploratory qualitative study with experts, we found that CROSSCODE enabled participants to maintain a strong sense of place in program execution, was conducive to explaining program behavior, and helped track changes and updates to the program state.

CCS CONCEPTS

• **Human-centered computing** → *Information visualization*; *Information visualization*; **Graphical user interfaces**.

KEYWORDS

program visualization, programming, debugging

1 INTRODUCTION

Computer programs translate human intent into machine-executable instructions. Their representations have largely remained the same: a set of strict textual instructions to generate a particular behavior. A programmer must translate their intent into precise instructions when writing code and then check execution details (e.g., observing an array update from $[5,1,2,3]$ to $[1,5,3,2]$) for alignment with high-level intent (e.g., *"sorting the first half of the array"*) when tracing and debugging code. Constantly ensuring congruence between the intended program behavior, the source code, and the execution demands excessive cognitive effort, becoming a learning barrier for beginners and a productivity impediment for experts.

Significant research has sought to reduce the cognitive effort of understanding computer programs by employing program visualizations. For example, tools like Python Tutor [17] and Projection Boxes [25] display the runtime state of a program at each line. They enable programmers to answer common tracing questions like *"What is the value of x at line 5?"*. However, they are less helpful in conveying overall program behavior, like *"How does insertion-sort develop a sorted list?"*, which must be pieced together by stepping through the code line-by-line. In contrast to line-by-line navigation, algorithm visualizations use specialized representations to communicate the key steps and behaviors in the program [18, 20, 37, 43]. However, these visualizations are not generically applicable: They need to be designed per algorithm or family of algorithms.

Therefore, existing systems either visualize *low-level* program states, which are *generically* applicable to many programs or represent *higher-level* program behavior, which needs to be *specifically* designed. In contrast, programmers and educators flexibly employ varying levels of abstraction to reason and communicate about programs. For example, a programmer may use breakpoints to inspect key program states and locate problematic sections, as well as hand-drawn diagrams to simulate program behavior and reason about program logic. Their reasoning processes can be bottom-up or top-down, depending on their expertise and tasks [26]. Similarly,

instructors often describe the same algorithm with gradually increasing levels of detail, starting with its purpose, then the main steps, and so on, moving between levels of abstraction. This research aims to fill the gap that exists in program visualization tools and can be summarized by the following problem statement: *Given that programmers reason about code across multiple levels of abstractions, how can we design a program visualization system that can change its level of abstraction according to the programmers' needs?*

Our goal is thus to identify and apply techniques to visualize program execution at multiple levels of abstraction and design supporting interactions to navigate between these levels. To inform our design, we surveyed 92 computer science diagrams and animations from well-known instructional media for consistent visual communication strategies. We identified three key design patterns performed by these visualizations: they aggregated operations (e.g., instead of *"x to temp to y,"* display *"x to y"*), abbreviated repetitive operations (e.g., summarizing a loop by showing the first few and the last iterations), and displayed an overview of the execution space (e.g., depicting the call graph of a recursive function).

Based on our findings, we developed CROSSCODE, a visualization system for a subset of JavaScript that takes advantage of the intrinsic features of a program, such as its syntactic structures, control flow, and data flow, to provide a flexible representation of its execution (Figure 1). CROSSCODE enables navigation between multiple levels of detail. Rather than navigate the runtime execution line-by-line, users can flexibly navigate to the desired levels of abstraction through the syntax nodes of the source code. CROSSCODE uses an intermediate mapping akin to a *'mental picture'* of the control flow to contextualize and navigate between levels of abstraction (Figure 1a). The data state is rendered with animations, trace paths, and color encoding, calculated from the data flow (Figure 1b). CROSSCODE[1] is implemented with a custom interpreter for a subset of JavaScript written in TypeScript and instantiated as a web front-end.

To evaluate the utility of CROSSCODE and its implications, we conducted an exploratory study with six expert programmers. We found that compared to PYTHON TUTOR, a line-by-line program visualization tool, participants were better oriented in the program's execution with CROSSCODE. Participants were also able to effectively navigate across repetitive operations and locate steps of interest. We contribute (a) identification of design patterns used to communicate program behavior, (b) a research prototype capable of generating and navigating program execution at varying levels of detail, and (c) results from a qualitative study, which shows that participants perceive CROSSCODE to be effective in locating errors and facilitating program understanding and communication.

2 RELATED WORK

This research draws on several threads in cognitive accounts of programming, program visualization, and dynamic representations.

2.1 Program Understanding

Researchers have proposed several cognitive models to describe how programmers develop an understanding of the source code [44]. These models suggest that programmers do not always parse source code line-by-line, but instead skip parts of code and regularly

---

[1]https://github.com/hayatpur/crosscode

scan back and forth. Bottom-up models propose that programmers incrementally chunk statements into higher-level abstractions [31]. Brooks suggests that programmers start with an overall hypothesis of a program and incrementally refine it [9]. Soloway et al. suggested that expert programmers rarely process each instruction individually, but instead think in terms of schemas, i.e., chunks of instructions, that achieve a particular behavior [41]. Mayrhauser et al. found evidence that when programs lacked familiar cues or expectations were violated, a bottom-up process started from the details of the source code to incrementally synthesize an overall understanding [50]. Still others suggest that bottom-up and top-down processes are used depending on the user's expertise and task [26].

A programmer must also emulate the computer, i.e., trace code. Tracing forms the foundation to enable a programmer to effectively read, write, and debug code but is difficult to learn to do well [27]. In a study that observed the effects of tracing on working memory, Crichton et al. found two primary strategies: linear tracing following the control flow from the beginning to the end, or on-demand tracing, starting from the end with variables of interest and tracing back through the code that affected the variable [12]. Their findings highlight the need to adapt to user tracing strategies. For example, a person using an on-demand strategy would want to follow the data dependencies of a variable, while those using a linear strategy would want to follow the control flow.

Ko et al. observed that, when debugging, developers anchor their search with the execution of the program to look for symptoms of failure [22]. After determining the relevant code, developers used its incoming and outgoing connections to locate errors. However, developers frequently lost context during tasks and were not able to keep track of their explorations. They hypothesized that this may be due to showing the source code for the entire file at once rather than at a more appropriate granularity.

The different accounts of how programmers form mental maps of code, trace code, and debug code, exemplify the need for flexible representations of computer programs.

2.2 Visualizations of Program Behavior

Displaying program states is a standard method that is used to understand, debug, and communicate code. For example, Python Tutor [17] uses a memory model graphic with variables, pointers, and memory values, while Thonny [6] overlays the results of expressions onto the source code to ground symbolic code with runtime values. To trace values over time, Victor demonstrated a live programming interface that annotated the values of relevant variables at each line of source code [48]. Lerner generalized Victor's visualization using projection boxes, which select a subset of the full semantics of the program to display, providing a customizable view of runtime information [25]. Other research has also investigated externalizing control flow. Reacher visualizes function calls as a graph to orient the user in debugging tasks [24], Debugger Canvas displays each function call in a spatial fragment [13], and Schematic Tables visualize a cross between decision tables and data flow [15].

Algorithm animations use specialized representations to communicate key behaviors of common algorithms. Starting with early animations like Sorting out sorting [7], these visualizations fluidly transition between relevant consecutive states of a program. They

help users understand how an algorithm works by focusing on specific key steps and invariants [20]. However, algorithm animations lack generalizability in their abstractions and visual encodings. Different visualizations must be configured and specified for different types of algorithms [43]. Similarly, conceptual diagrams and animations are a staple in computer science education. Due to the high degree of customization conceptual visualizations need, designers need to create such visualizations on a case-by-case basis by using graphic design tools to manually produce diagrams, which is a challenging and labor-intensive process [28].

We aim to bridge the gap between generic program visualizations and specialized visualizations like algorithm animations and conceptual diagrams.

2.3 Representing Multiple Levels of Abstraction

This research builds on work that sought to support the mental processes of users through flexible representations [47, 51]. For example, Victor explored the ladder of abstraction of data, procedures, and iterations from an automobile simulation that broadly represented the behaviors and patterns of a system [47]. The WritLarge system proposed flexible representational axes that allowed users to semantically, structurally, and temporally transition content representations to suit their needs rather than needing to conform to the single representation required by a user interface [51].

Within the domain of programming, Suh et al.'s Coding Strip system used comic strips to provide a conceptual abstraction of program behavior and ease the burden of learning complex program executions [46]. Omnicode, on the other hand, enabled abstraction over time by visualizing values of a variable over its lifetime with scatter plots [21]. Storey et al. proposed a top-down approach to algorithm animation, where information was gradually made more granular using steps that expanded it into more specific instructions. For example, a *'QuickSort'* step could expand into *'Divide'*, *'Conquer'*, and *'Combine'* [45]. (Their visualization and steps were pre-designed rather than generated procedurally.)

The present research builds on this work by generating different levels of abstraction of program execution for generic programs and by employing novel representation techniques that enable navigation and access to these levels of abstraction.

3 FORMATIVE STUDY

To derive design patterns for flexible representations of program execution, handmade diagrams and animations used in computer science instructional materials were studied. These were selected as they are often transformed and simplified using various visual communication techniques, making them a suitable source of potential design patterns.

3.1 Data Collection

We sampled 92 visualizations from well-known textbooks, online instructional articles, and videos [1, 4, 11, 29, 30, 34, 39], online instructional articles [8, 10, 38] and videos [32, 35]. Some assumptions were made to constrain the types of visualization considered. First, if two visualizations used the same style and technique, then only one was used. Second, visualizations that directly matched program execution step by step and used no visual encodings were

filtered out. This selection process resulted in 38 visualizations, 13 of which were animations.

Each visualization was then annotated with low-level visual attributes by the first author, which described its design, e.g., *"array represented as co-located boxes," "data flow represented with arrows,"* etc. The corresponding program source code described in the visualization was also annotated as it exemplified the deviations that the designer made from a potential implementation. The visual attributes and source code were then interpreted and grouped into broader design patterns to describe the process or transformation that could be used to communicate the program behaviors.

When deriving the design patterns, we followed Agrawala et al.'s visual design approach [3] and selected the following objectives each design pattern must satisfy: (a) *Program independence*, i.e., the pattern should generalize to a broad range of programs; (b) *Generative*, i.e., there should be a well-defined criterion for how and when to apply the rule based on a provided source code and execution trace; (c) *Cognitive rationale*, i.e., there should be a plausible hypothesis for how the rule leverages human perception and visual skills to aid in programming tasks.

3.2 Design Patterns

3.2.1 D1: Aggregate Operations into Key Steps. We found that designers communicated only key events of a program's execution rather than all of its details. What constitutes a key event depends on the purpose of the visualization and the behavior described. In most cases, key events aggregated lower-level events based on the structure and semantics of the source code. For example, a useful invariant of insertion-sort is that elements are inserted at the start of the list. To illustrate this, operations in the inner loop of the algorithm, where individual elements are being swapped, were aggregated into a single movement (Figure 2a). Animation and visual annotations, such as arrows, depict the data flow to communicate the overall action of the step (Figure 2a). Aggregations can allow viewers to focus on program execution milestones without being overwhelmed by low-level steps (e.g., Figure 2b). When instantiating this pattern, we presume these milestones to be aggregations based on syntax; this assumption may not always be the case (i.e., the implementation may be separate from visual intuition), but it provides generalizability.

3.2.2 D2: Abbreviate Repetitive Operations. Effort was made to reduce the amount of information for repetitive steps. For example, in loops with many iterations, only the first few and last are shown in detail, similar to how a large data structure is often abbreviated (e.g., writing a large list as [1, 2, ..., 13001]). This varies depending on the context, for example, in Figure 2c, only the first operation is shown in detail, and subsequent operations are aggregated into a single animation. Visualizations also exploited the symmetry of repeated operations with animations that sped up over time or diagrams that showed incrementally simpler representations. Abbreviations enable viewers to observe enough instances of a pattern to induct the general rule behind it, but without being overwhelmed by every instance of the pattern. We focus on two simple, but widespread repetitive structures when instantiating this pattern: linear loops and nested function-calls.
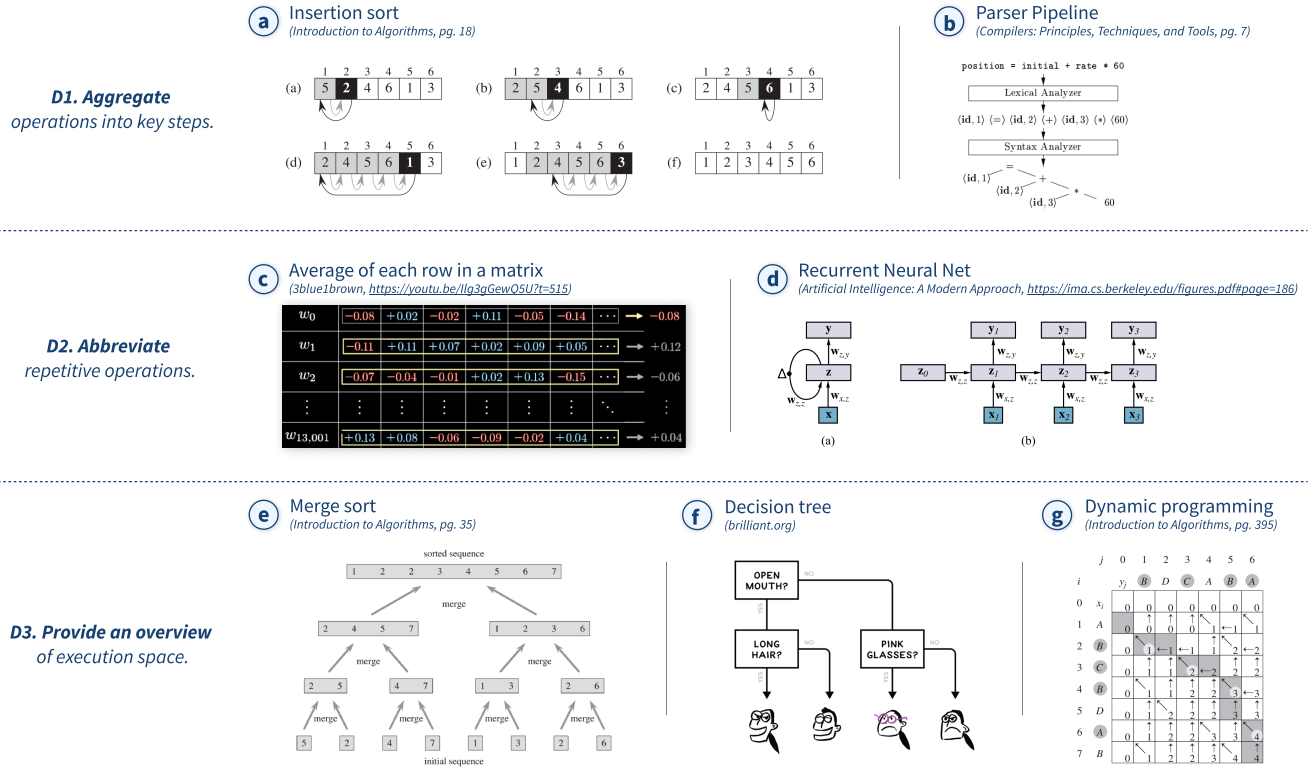
Figure 2: Examples of hand-designed illustrations that aggregated execution (a, b), abbreviate repetitive operations (c, d), and provide an overview of the execution space (e, f, g).

3.2.3 D3: Provide an Overview of Execution Space. An overview of the execution was often provided. For instance, execution of a recursive algorithm such as merge-sort algorithm is displayed through a call graph organized as a binary tree, where each node representing the state of the list (e.g., Figure 2e). Algorithms with double nested loops, such as for dynamic programming, often employed a grid, with one axis encoding iterations in the outer loop and the other encoding iterations of the inner loop (e.g., Figure 2g). In both cases, the control flow guides the layout, while the data flow and data values serve as landmarks: Figure 2e displays return value at each node, and 2g labels the values of the iterators on the axis. In other cases, the layout was guided mainly through the data flow, for example, a schematic of a neural network; where it better captures the underlying mathematical model (e.g., Figure 2d). An overview adds context to an execution step and enables visual deduction of overall patterns that would otherwise be difficult to synthesize in isolation. We choose to use the control flow to determine the layout, which can be readily mapped to multiple levels of abstraction.

3.3 Other Patterns

We uncovered patterns of data organization and encoding patterns that are not emphasized in our system, as they have been previously investigated and implemented, but are noted none-the-less:

(1) *Variables are annotated around key data structures.* Visualizations preserved the relationships of different variables in a strong visual hierarchy. They used a focal data structure, with data and control structures annotated on top. For example, variables that index into an array was annotated on top of the array. Support for this customization exists in algorithm animation systems, e.g., Alvis Live [18, 19], and customized placement of variables in Python Tutor [17].

(2) *Key data structures are temporally or spatially juxtaposed.* Data was frequently compared with itself or with other data by animations, providing an indication of change, or with small multiples to facilitate comparisons. Showing program states at multiple times has previously been explored by Omnicode [21] and Projection Boxes [25], and animations by algorithm animation systems [18, 19, 43].

3.4 Relation to Prior Visual Techniques

The three design patterns that we identified are not independent of prior research in visual communication techniques. In particular, Shneiderman's Visual Information-Seeking Mantra suggests that users should be able to (1) get an overview of the data, (2) zoom in on specific areas of interest, (3) filter the data to show only relevant information, and (4) access the details of the data on demand [40]. While Shneiderman originally intended these guidelines for understanding and interacting with complex data sets, our findings emphasize that the same underlying visual principles can be applied to understand and visualize code behavior.
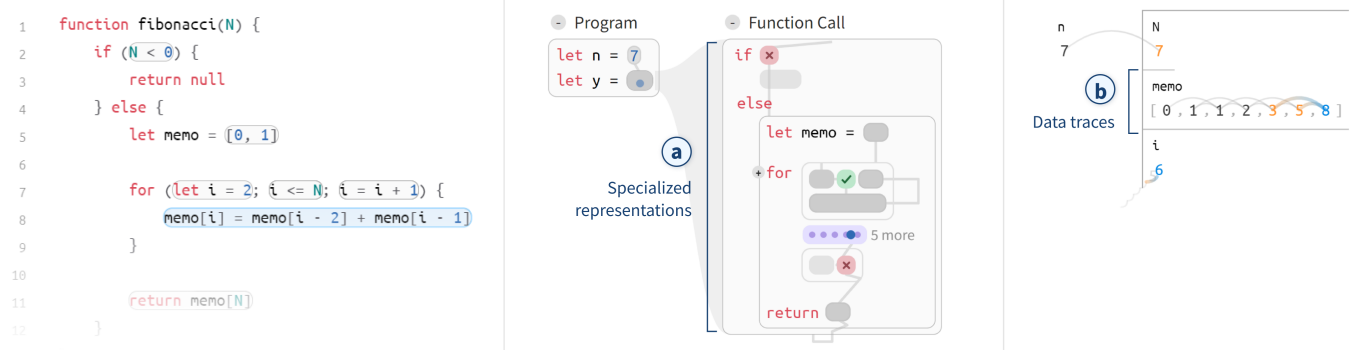
Figure 3: Execution of a Fibonacci algorithm. (a) Specialized representations are used for if-statements, for-loops, and function-calls; (b) Traces of data flow are rendered to help maintain a record of past *Steps* and infer patterns over time.

4 SYSTEM DESIGN

Our formative study and prior work on understanding program behavior stress the need to view program execution at higher levels of abstraction. Here, we describe the design of CrossCode, which groups individual runtime steps using the code's syntax tree.

4.1 Defining an Execution *Step*

The runtime of a JavaScript program can be modeled as registers, which store temporary values, and stack frames which map variable names to values. For example, `let x = 1 + 2` will first add `1 + 2` in a register, and then bind the output, `3`, to `x` in the current stack frame. A traditional program visualization may allow users to navigate through each of these steps one at a time, and view the memory state at each step. The key idea of CrossCode is to abstract sequences of individual steps, which are manipulation to registers and stack frames, into aggregate steps (or just *Steps* for short). Specifically, we group individual steps based on the nodes of a program's syntax tree. In doing so, we can query the execution based on the syntax, e.g., *"What is the effect of this for-loop?"* Since the syntax tree is hierarchical, this grouping is also hierarchical, we can query for further details, e.g. *"What is the effect of this if-statement in the for-loop?"*

For example, the loop: `for (let i = 0; i < 1; i++) {...}` contains five sub-*Steps*: (1) initialize: `let i = 0`; (2) the first test: `i < 1`, which succeeds; (3) the body: `{...}`; (4) the update: `i++`; and (5) the second test: `i < 1`, which fails and terminates the loop. These sub-*Steps* can be decomposed further, e.g., the test: `i < 1` is composed of two sub-*Steps*: (1) the identifier `i`, which reads from the stack frame and writes the value of `i` to a register, it cannot be decomposed further; and (2) the literal `1`, which writes the value `1` to a register and cannot be decomposed further.

Our goals are to design (a) an understandable visualization of a *Step* and (b) facilitate navigation across *Steps*, both over breadth, such as navigating to a different iteration of a loop, and over depth, such as navigating across a stack of a recursive function calls. We achieve these goals through three views: the *Control Flow View*, which visualizes the *Steps* themselves, the *Data View*, which shows the effect of the *Step* on the program state, and *Source Code View*, which grounds the *Step* in the source code.

4.2 Control Flow View

CrossCode visualizes a *Step* as a miniaturized block of source code from which it originated (Figure 3). This representation includes fragments of the corresponding source code which serve as landmarks. For example, a variable-declaration will partially reveal its left-hand side (e.g., `let x = ■`). *Steps* are organized according to their layout in the source code. The landmarks and consistent organization enable users to situate themselves in the control flow using their mental image of the source code.

The path control flow takes is annotated between each *Step* and visually connects adjacent *Steps*. A blue cursor on the path indicates where the program execution is currently (Figure 3a). Users can navigate the program execution with keyboard controls (that is, stepping with left and right arrow keys), by directly grabbing and moving the cursor, or by clicking on a *Step* to navigate to its end point. To navigate between levels of abstraction, a user can control-click on a *Step* to break it down to a lower level of abstraction (i.e., its sub-*Steps*). A control-click on an already broken-down step will aggregate it back to a higher level of abstraction.

4.2.1 Specialized Representations. Specialized representations of if-statements, for-loops, and function-calls are used to better reflect each structure's meaning. For example, if-statements and for-loops indicate the result of their condition directly in the *Control Flow View* using a checkmark for success or a cross for failure (Figure 3a). They also contain stubs for *Steps* that were not reached but exist in the source code (such as an unsuccessful body of an if-statement, for example, Figure 3a), which help maintain a close match between the source code and the control flow, even when parts of the source code are not reached. Unlike other control flow structures, a function-call is not expanded in place, but rather in a separate space, termed a *Frame* (Figure 3a). This reflects that each function call exists in its stack frame, analogous to existing call graphs visualizations [13, 24]. Each *Frame* has its own control flow cursor, which is synchronized across the other *Frames*. This helps situate oneself relative to other function-calls, such as to know if the execution is currently before, after, or during the *Frame*.

The *Control Flow* View works towards D1 by enabling users to select the appropriate level of abstraction, and D3 by providing an overview that orients and guides the user's navigation.
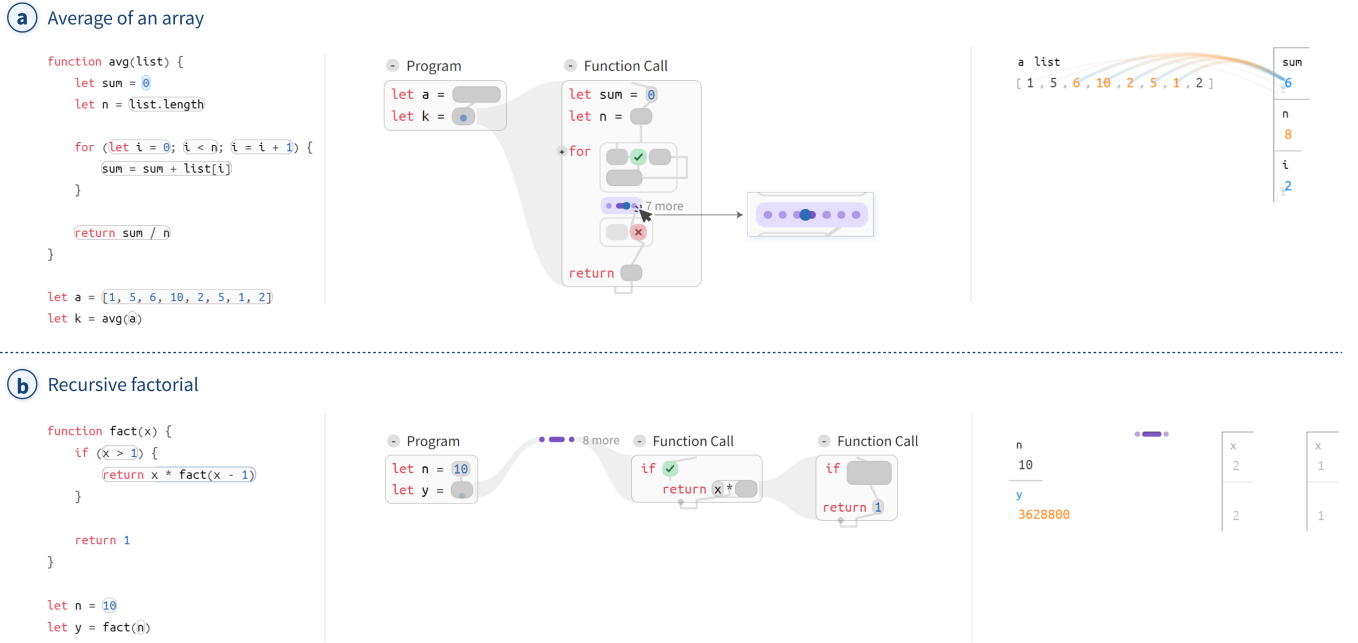
Figure 4: Abbreviations can be used for both loops and function-calls. (a) A program to compute the average of an array, subsequent iterations of the for-loop are abbreviated and their operations are aggregated, clicking on the abbreviation expands it to reveal a dot for each iteration individually. (b) A program to compute Factorial of *n*, the user has navigated to the base case of the function, and parent function-calls are abbreviated.

4.3 Data View

CROSSCODE externalizes the runtime state of a program through a view of the underlying memory. It displays columns of memory values with variable reference(s) to the value denoted above (Figure 3b). Data in different stack frames are spatially separated, corresponding to the position of their associated function-call *Frame* in the *Control Flow View* (Figure 3b).

4.3.1 Animations and Traces. To visualize transformations of data at multiple levels of abstraction, CROSSCODE derives an animation of a *Step* using its reads and writes. Animations not only provide a mechanism to maintain context when following execution but also to show aggregate data transformations (e.g., Figure 3b). A data read is colored orange, and a data write is colored blue, providing a lookup of how the current step modified the state. We employ the following animation types (**A**, **B**, and $\mathbf{X}_i$ denote memory values):

(1) *Create*(*A*) is used when *A* is written to with no reads from other existing visualized data. A fade-in animation for *A* is used to represent it.

(2) *Move*(*B* → *A*) is used when value *B* is copied to value *A*. The resulting animation is shown as *A* originating from *B* and gradually moving to its memory position. In addition, a trace is shown from data *A* to data *B*.

(3) *Cause*($X_0...X_N$ → *A*) occur when data is written to *A* from multiple reads, $X_0...X_N$, (e.g., in a binary expression). In this case, a trace is shown from each read, $X_0...X_N$, to *A*, along with a fade-in animation for *A*.

Animations serve to highlight the change between data states, whereas traces serve as a persistent visualization of the operation. Traces of past executions are displayed but faded, allowing users to infer and abstract over time while maintaining visual clarity (e.g., Figure 3b). Note that these animations cover a subset of possible data dependencies. Control flow dependencies such as conditional dependencies are not visualized (e.g., if value **B** being true causes value **A** to change, that dependency is not recorded).

4.3.2 Residuals. Data that is replaced (e.g., `let i = 1; i = 2`) is pushed back by fading it and giving it a positional offset, analogous to crossing out an older value when tracing a program with pen and paper. Residuals make it possible to visualize traces of data that no longer exist in the program state without adding visual noise. For example, in an in-place swapping algorithm (Figure 1b), the residuals disambiguate the origin of the traces.

The *Data View* instantiates D1 by indicating aggregated steps through animations and data traces.

4.4 Source Code View

The source code is synchronized with *Control Flow View*, in that the current *Steps* being executed is annotated on top and highlighted. So far, CROSSCODE's functionality has supported top-down methods of navigating abstractions, by starting from an overall *Step* and incrementally breaking it down. The source code provides a map to freely navigate across levels of abstraction. When selecting any

specific piece of the source code, the execution jumps to points at that level of abstraction. For example, a user can select a base case of a function-call to navigate to all points in the execution where that base case is executed.

4.5 Supporting Multiple Levels of Abstractions

Computer programs contain fundamental repetitive structures that can be impractical to show in their entirety. Rendering many iterations of a loop or displaying the entire control flow graph of a deeply nested function-call can easily become cluttered and difficult to reason about. CROSSCODE provides two mechanisms to manage this complexity: abbreviations and aggregations. Abbreviations instantiate D2: they provide structure to navigate and interpret repetitive procedures, and aggregations provide a zoomed-out overview of longer executions *Steps*, further facilitating D3.
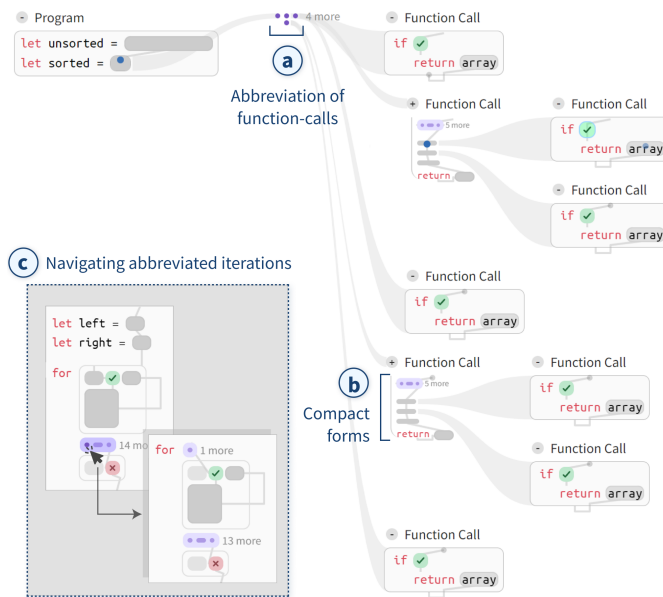

Figure 5: Execution of a recursive call to a Quicksort algorithm, expanded up till the base cases. (a) Progressive closure automatically abbreviates older function-calls; (b) Progressive disclosure only shows the last few steps of the Function, instead of showing all its sub-*Steps*. (c) Control-clicking on an abbreviated iteration expands it and progressive closure abbreviates the previously expanded iteration.

4.5.1 Abbreviations. Abbreviations aim to trim the amount of repetitive information by collapsing a *Step* into a dot, potentially grouped together with other dots (Figure 4a). By default, an abbreviation of N sub-*Steps* collapses its contained actions into three dots: the first represents sub-*Step* **0**, the second dot represents sub-*Steps* **1** to **N − 1**, and the last represents *Step* **N**. To revert an abbreviated *Step* to its normal level of detail, the user can control-click on it (Figure 5c). Clicking on an abbreviation group toggles it to an non-aggregated mode, where each sub-*Step* in the abbreviation is represented as a separate dot, that is, for N sub-*Steps*, there will be

N dots (Figure 4a, middle). We also use specialized abbreviations for for-loops and function-calls:

- For-loops do not abbreviate the initialization, the test, the body, and the update as separate dots, but instead aggregate them into iterations, i.e., a dot represents an entire iteration. By default, the first iteration is unabbreviated, and the rest of the iterations are abbreviated. In this way, users can view the first iteration in-depth and skim over subsequent iterations.
- Function-calls that are nested can be abbreviated (Figure 5a), which also abbreviates the function-call's corresponding scope in the *Data View* (Figure 4b). This makes it feasible to browse deeply nested structures, such as recursive function calls, by an overview of the call stack without viewing each function-call in isolation. For example, in a quick-sort algorithm, a user can easily read the path from the parent call to the base case (Figure 5a).

4.5.2 Compact Forms. In addition, we provide a toggle to view *Steps* in a compact form (Figure 5b). In their compact form, the landmarks, spacing, and overall correspondence of *Step* to the source code are minimized in favor of a compressed representation.

4.5.3 Progressive Disclosure and Closure (D3). Thus far, the two mechanisms to manage complexity, abbreviations, and compact forms, need to be applied manually. To help users maintain and establish context, we provide two modes in which these techniques are proactively applied.

(1) *Progressive disclosure* aims to show new information incrementally rather than presenting all sub-*Steps* at once. It does so by (a) automatically abbreviating for-loops, and (b) only showing the last four steps of a function-call body (Figure 5b). Note that if the user would like to see all iterations of a for-loop, we provide a toggle to show all the iterations of the loop, i.e. unroll the loop.

(2) *Progressive closure* collapses older *Steps* when a user expands a new *Step*. For example, given `f(g(h(x)))`, if the current level of abstraction is at `f(...)`, then when a user expands to the first child `g`, CROSSCODE automatically sets the parent, `f`, to be in a compact representation (Figure 5b). If the user expands to the second child `h`, it abbreviates the ancestor `f` entirely (Figure 5a). In doing so, this two-stage collapsing can help organize space while retaining context. This helps navigate recursive functions or complex call stacks (Figure 5a,b). Progressive closure also automatically closes older iterations of a loop as the user un-abbreviates new ones (Figure 5c).

In sum, CROSSCODE uses abbreviations and aggregations to manage complexity in larger executions. It invokes these techniques through progressive closure and disclosure of information, which we hope allow for effective navigation and viewing of multiple levels of abstraction.

4.6 Implementation

CROSSCODE is implemented with a custom interpreter and a web front-end written in TypeScript. For tooling, *Vite* is used to compile and bundle the project [49], and the *Monaco Editor* [49] is used

in the web front-end for the user to input source code. Herin, we describe the information pipeline:

4.6.1 Dynamic Analysis. The user's source code is parsed into an abstract syntax tree using *acorn* [2]. A custom interpreter walks through the syntax tree, executing on a stack-based, virtual memory model, and returns a structured trace of the execution which groups the operations based on the syntax tree. Our interpreter architecture was based on *Sval*, an open-source JavaScript interpreter [53]. Each node in the trace eventually decomposes into primitive operations, such as `CreateLiteral` (for creating a literal value in memory), `BinaryExpression` (for computing a binary expression), etc. Each primitive operation stores its reads and writes which are used to generate the data flow. In addition, all nodes store a precondition and a postcondition of the memory model before and after the execution of that node.

4.6.2 Rendering Control Flow View and Data View. The structured execution trace derived from the dynamic analysis drives *Control Flow View*. Each node in the trace is assigned a visual representation based on its type (e.g., a for-statement is assigned a different renderer than an if-statement). The visual representation is built and displayed through the DOM. Subsequently, the *Control Flow View* drives the *Data View*: based on the current node selected, the *Data View* renders its postcondition, and based on the position of the cursor, it interpolates an animation derived from the data traces. The data traces themselves are rendered using the *Perfect Arrows* library [33]. Users can edit the animation speed through an options panel rendered with *Tweakpane* [23].

4.7 Implementation Limitations

While the techniques in CrossCode can likely generalize to other imperative languages, we focus on a single language: JavaScript. Specifically, the subset of JavaScript with constants, function calls and definitions, binary operators, let-bindings, if-statements, and loops. Notable exceptions include minimal support for objects, no classes or prototype inheritance, no implementation of standard global objects outside of a limited subset of `Math` object, no exception handling, and no support for DOM manipulation. (See Section 7.1.3 and 7.2.3 for a discussion of ideas for supporting classes and synchrony respectively.)

A limitation to the adoption and generalizability of our current implementation is the use of a custom interpreter. Note that this is not a requirement: the information flows we use (i.e. control flow and data reads and writes) may be extracted reliably by instrumenting an existing runtime, such as of the browser, or by using a dynamic analysis library, e.g., Jalangi [36]. We opted to use a custom interpreter for its flexibility and ease of use in accessing the information we required during prototyping.

Finally, not all runtime operations are externalized in Cross-Code. For example, operations to create or pop scope are not made visible to the user as they are often aligned with another *Step* (e.g. a for-statement or a block-statement). Our goal was to aggregate execution; as such, we made design decisions to appropriately filter operations that were considered extraneous but that may be crucial to surface in other contexts.

5 USER STUDY

We conducted an exploratory user study to understand how the workflow supported by CrossCode differs from existing tools, and if the representations proposed are useful to understand and communicate computer programs.
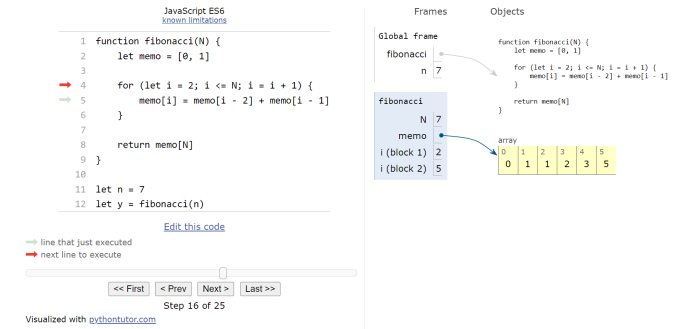

Figure 6: Screenshot of Python Tutor showing JavaScript code that iteratively computes the n-th number in the Fibonacci sequence.

We compared CrossCode against two other conditions in debugging tasks:

(1) Python Tutor, which is a widely online web app for visualizing program execution (Figure 6) [17]. It was chosen as it is the state-of-the-art in program visualization tools and a line-by-line visualization. It uses a two-column layout, with source code on the left, and a depiction of the current execution state on the right. It includes a global timeline through which users can step to different parts of the execution, and buttons to go to the next or previous steps. Clicking on a line of source code allows setting a breakpoint, which is visually indicated along the timeline.

(2) Drawing, instructors and programmers frequently draw out program execution (e.g., on the blackboard) to communicate or reason about program behavior themselves. This condition was chosen as it enables creation of completely flexible representations, which can highlight deficiencies in the representation of CrossCode or Python Tutor. Participants were allowed to use pen and paper or a digital drawing tool.

Our goal was not to compare the performance (i.e., the speed of completing the debugging task) but rather to elicit the representational affordances of each tool. The studies lasted between 90 minutes and 1 hour and 45 minutes each, and participants were compensated $40 USD for their time.

5.1 Participants

We recruited six participants with extensive programming experience who work and/or teach in computer science and related fields (Table 1).

5.2 Study Procedure

5.2.1 (5 minutes) Introduction. Participants were provided with a brief description of the research and study goal. It was emphasized

Table 1: Participant demographics, experience in years. Teaching includes experience as a teaching assistant.

ID	Title	Age	Gender	Programming Experience	Teaching Experience
P1	Assistant Professor	33	Male	10	3
P2	PhD Student	30	Male	10	3
P3	Software Engineer	26	Male	14	1
P4	PhD Student	27	Male	8	2.5
P5	Professor	54	Male	35	34
P6	PhD Student	24	Male	6	0.5

that the goal of the debugging tasks was not to reach the solution as fast as possible but to use the problem to reflect on the usefulness of the tool and its representation.

5.2.2 (15 minutes) Guided Walkthrough of CrossCode. Participants were presented with an introduction to the visualization techniques available within CrossCode and were guided through its various features. If participants were unfamiliar with Python Tutor, a brief tutorial on the use of the tool was provided.

5.2.3 (10 minutes x 6) Debugging Tasks. Participants were presented with six debugging tasks, with the goal of localizing and fixing a bug. Each task had a single bug, which included one of the following: off-by-one array index, incorrect loop bound, and incorrect base case. Two debugging tasks were performed per condition (i.e., one iterative and one recursive). The ordering of the conditions and the task chosen were appropriately counter-balanced, however, the iteration task always occurred first followed by the recursive task. As mentioned above, the goal of these tasks was not to be challenging, but to serve as a sandbox to explore these tools. After each task, participants were asked to reflect on whether the representation of the tool was useful for their own understanding of the algorithm and if they were to explain this algorithm and their solution to a novice, whether this tool would be conducive to that explanation. In doing so, it could reveal conflicts between each condition and the user's own mental model.

The six tasks involved debugging programs that:

(1) [Iterative] *Inserted into a sorted list such that it remains sorted.*
(2) [Iterative] *Returned the n'th Fibonacci number.*
(3) [Iterative] *Reversed a list in-place.*
(4) [Recursive] *Returned sorted list using out-of-place quick-sort.*
(5) [Recursive] *Returned sorted list using merge-sort.*
(6) [Recursive] *Returned index of a value in a sorted list with binary search.*

5.2.4 (15 minutes) Post-questionnaire and Interview. Participants were administered a questionnaire on the usability and usefulness of CrossCode through 7-point Likert Scale questions. During a semi-structured post-interview, participants were asked to indicate which tool they perceived to operate closer to the level that they understand code, explain how their workflow and approach to the

problem differed between tools, and which tool would best facilitate communication.

5.3 Study Limitations

This study has several notable limitations. The types of tasks were artificial and had far simpler bugs than those encountered in the wild, and as such, these results may not generalize to tasks outside typical algorithm procedures. There are notable sources of bias that make our findings subject to error: Participants likely knew Cross-Code was created by the researchers, all participants were male, the sample size was modest at best, and the analysis was performed only by the primary author. As all participants are experts, it is not directly evident whether CrossCode helps novices build better mental models and learn effective debugging processes.

6 STUDY RESULTS

The participants found CrossCode's representation to be understandable and effective. With CrossCode, participants rated themselves to maintain a strong sense of their place in the execution (1/6 Strongly Agree; 3/6 Agree; 1/6 Slightly Agree; 1/6 Neutral), color encoding and traces were easy to understand (5/6 Agree; 1/6 Slightly Agree) and data animations were easy to follow (4/6 Strongly Agree; 1/6 Agree; 1/6 Slightly Agree).

We conducted a thematic analysis of participant reflections during their task and interview focusing on the role of CrossCode as a tool for (1) debugging, (2) program understanding (that is, developing valid mental models of the code), and (3) visual aid to explanations. The first author coded the transcript, deriving 48 codes (e.g., *"Difficult to establish a convention with Paper & Paper"*), which were synthesized into seven primary themes. Below, we report the results of the thematic analysis.

6.1 CrossCode for Debugging

CrossCode added context and continuity to debugging, which was perceived to be helpful in orienting oneself, as well as locating and diagnosing bugs.

6.1.1 Control Flow View added necessary context to navigate the execution trace. We observed that instead of internalizing the control flow, participants exploited CrossCode's contextual cues to maintain a sense of place, for example, since *"CrossCode shows the history behind how it got there, I was able to go back in time"* (P2) unlike Python Tutor where *"you don't have any knowledge of how we got there because it's just showing the current frame."* P4 notes that this led to a slower debugging process:

> "I didn't feel confident in my ability to remember where I'd been or what to look at [with Python Tutor]... I just had to absorb all the information at each step... I had to step back several times and go to the beginning because I didn't really remember anything in between." -P4

Similarly, in Python Tutor, P1 *"kind of scrubbed [through execution] randomly... I don't know where to focus on but rather just scrolling back and forth to try and spot if there is something wrong on this line."* The *Control Flow View* in CrossCode also enabled

new workflows, such as allowing participants to visually deduce an erroneous program, e.g.,

> "So, I've got 1,2,3,4 iterations before I fail... I can see that I've only got three items left [in the array] and so just from this representation I can say, 'Why am I doing one more?' And of course I don't have that clue in any of the other tools... that requires some sophistication but that's huge." - P5

6.1.2 Animations, color encodings, and traces provided visible indicators of change. Animations enabled participants to follow the data flow of values. For example, P6 *"would just go back and see the animation again like this [grabbing the control flow cursor]... and say, 'Okay where did this [item in array] one go?' And I'll just slowly drag [the control flow cursor] and see"* (P6). Aggregations complemented these data views, P2 found it *"quicker and faster [to debug] in CROSSCODE [compared to PYTHON TUTOR] because I was able to quickly understand what was being changed, or quickly go through the whole for-loop to find something."* Unlike PYTHON TUTOR, which provided no indication of change, CROSSCODE's color encodings for reads and writes helped infer patterns, e.g.,

> "One of the great things about this animation here is that it shows which values are currently being modified, it shows that these two things [last two items of array] are being used... and based on those two things it's going to generate the next one [fibonacci number]." -P2

When DRAWING, participants indicated a change by crossing out previous values, which although not scalable to larger inputs, served as an ad hoc mechanism to abstract over time and make predictions of program behavior. While CROSSCODE's residuals also depicted past values, they were not sufficient when comparing values across substantially different points in time. P3 would have *"liked to be able to have a comparison, I could say 'I think this frame where we are right now is really interesting' take a snapshot and then go to the end and see the comparison"* (P3).

6.2 CrossCode for Program Understanding

CrossCode served as a visual aid to build valid and useful mental of the code. Aggregations provided the ability to synthesize key concepts, while abbreviations and the *Control Flow View* enabled navigation across repetitive structures and to key steps effectively.

6.2.1 Overview and abbreviations of control flow allows opportunistic navigation to interesting steps. When DRAWING, participants did not draw or simulate mental computations at every line, but instead opportunistically skipped repetitive steps (i.e. loops) or trivial steps (e.g. variable initialization). When investigating for-loops, participants frequently computed the first iteration of a loop manually, hypothesized a pattern, verified it over the next few iterations, and then skipped over subsequent iterations under all three conditions. CROSSCODE provided a structure for both forms of navigation. For P5, aggregations extended the functionality of breakpoints offered in traditional debuggers:

> "I found it natural to navigate the function by looking into it. I like being able to treat these [sub-steps]

atomically, in fact, these are a lot like breakpoints to me, except I don't have to set them manually." -P5

6.2.2 Aggregations alleviate the need to build mental models line-by-line. Participants found CROSSCODE's ability to chunk operations into *Steps* to be closer to their own mental model of code execution compared to stepping through the code line-by-line. For example, P1 shared their reasoning process as *"you can group all these steps together by just saying this creates a left array, and that creates a right array... like what you had [in CROSSCODE]."* Similarly, when DRAWING, P2, P3, and P4 explicitly drew chunks on top of the source code when tracing through the code by hand. P2 described their drawing process as:

> "I divided different parts of the code... based on what they are supposed to do. So, I've some feelings about some chunks that are correct, that are correct [which P2 explicitly annotated with a checkmark], and I'm suspicious about other chunks." -P2

6.3 CrossCode as an Instructional Visual Aid

CrossCode's aggregations were perceived as useful in facilitating explanation processes. However, visual aids are often specialized, and participants reported several opportunities to customize Cross-Code's representation to better direct explanations.

6.3.1 Aggregations map to explanation processes better than line-by-line tools. CROSSCODE's aggregations enabled a gradual increase in complexity; P1 noted that aggregations better mapped to concepts, which usually span many lines of code:

> "Line-by-line is the lowest level way to walk through what is going on... You wouldn't use that way to explain to people, but rather have concepts, like each concept maps to a function and then you create a link between those functions without relying on this [PYTHON TUTOR] linear representation." -P1

P6 found CROSSCODE to be *"the best in explaining, it's sort of like a pipeline, which visualized everything pretty well."* CROSSCODE was also found to be applicable to meta-thinking exercises. For example, when working through an algorithm with a student, P5 would want to ask:

> "Okay, if I were to execute this loop, what's the effect going to be like?'... and then if I treat it [the for-loop] as atomic it's easy to test that in the CROSSCODE tool... in PYTHON TUTOR I'd have to click next a bunch of times or set a breakpoint and then remember that I had set it." -P5

P4 found the ability to unabbreviate loops critical to explanation processes, *"for these loop-centric algorithms, of which there are many, it's really important to be able to roll and unroll loops... you want to be able to visualize them"* (P4).

6.3.2 Customizations are used for higher-order explanations. When DRAWING, participants frequently annotated on existing variable values (e.g., annotating the index i next to the item it indexes in the array) to both debug and explain code. P2 would have liked to *"visually show what [index] is list[n - 1 - i] pointing at, and what*

[index] is list[i] pointing at. What [index] is temp pointing at?." Annotations can also help direct attention when teaching, *"for the stuff I taught... like in data-structures, with rotating trees... you need to know where to look or you'll be lost... CROSSCODE is much closer to what I'd like, but I'd use paper or pen alongside it"* (P4).

6.4 Drawbacks and Limitations of CROSSCODE

The main overhead of CROSSCODE is in specifying the level of abstraction, which can slow debugging of simple examples compared to PYTHON TUTOR. While a user could have selected a part of the source code to navigate to the associated level of abstraction, it was scarcely used, and nearly all exploration was top-down. P4 and P3 also reported difficulties synchronizing between the *Control Flow View* and the *Data View*. And, P3, a practitioner, found it difficult to see the applicability of CROSSCODE as *"98% of the problems [they] have to explain to coworkers don't involve primitive data."*

6.4.1 Issues with Changes in Abstraction and Discoverability of the Appropriate Level of Abstraction. Participants reported several usability problems with CROSSCODE. P3 found that navigating the timeline in PYTHON TUTOR is straightforward and convenient, for example, *"as long as I can kind of get there"* even with *"trial and error, I don't feel that weird,"* (P3) and P6 found *"expanding and shrinking to be a little counterintuitive [in CROSSCODE]."* Selecting a piece of code to navigate to that level of abstraction was hard to discover and scarcely used. Adding computational suggestions to appropriate levels of abstraction may circumvent this: P2 found the resulting representation after *"select[ing] this [base case], that it shows me [expanded function graph], that's good. But it's not clear. I would have... descriptions or indicators on it [the base case]."* P3 suggested that previewing abstractions could make switching between them more approachable, *"Where you could hover over a ... checkbox [conditional check] it shows a literal representation of those numbers [being compared] at that moment"* (P3). CROSSCODE's default level of abstraction can also be deceptive. P1, for example, did not realize that a function-call was initially recursive, *"I just remember when I clicked that 'left', it just automatically did all the recursion functions, so I said, oh 'That's what it does.'"* P1-P5 also noted that they would need more time to become proficient and comfortable using CROSS-CODE compared to PYTHON TUTOR, for example, *"It's just that there are more features, I'd need more training"* (P5).

6.4.2 Disconnect between the Control Flow View and the Data View. P4 noted difficulties in tracking changes in *Control Flow View* and *Data View* simultaneously:

> "When I'm looking at [control flow] and unfolding stuff, I'm very much focused on it... the marble [cursor] is approaching solving the problem of what to look at. But there is nothing linking marble moving and the animations." -P4

Both P4 and P3 wished that the data state be better integrated with the control flow, for example, P4 *"appreciated like if you hover over this [control flow] it hovers over to the left [source code] as well, I wonder if it could hover over to the right [data]."*

7 DISCUSSION AND FUTURE WORK

By externalizing control flow, CROSSCODE allowed participants to maintain a sense of place in program execution, and the color encoding and animations provided strong indicators of change. CROSSCODE helped locate errors and was conducive to explanations.

7.1 Study Implications

There are significant implications of the workflows enabled by CROSSCODE which are explored below.

7.1.1 CROSSCODE encouraged a top-down program understanding strategy. CROSSCODE enabled new workflows for debugging and explaining computer programs; however, the overall interaction strategy was also singular because participants explored program execution top-down. Even though participants could have selected source code to navigate to a specific level of abstraction, they rarely did, possibly because there was not an indicator to select code nor a description of what selecting a particular piece of code would do. There was also a sudden change in the control flow representation, so context was difficult to preserve when switching between varying abstractions. Previewing a level of abstraction before committing to it may help preserve context. In addition, adding more navigation strategies, such as by selecting specific data of interest, can also help trace the code (that is, enabling program slicing [52]).

7.1.2 CROSSCODE's idealized representation can complement a programmer's internal model. CROSSCODE's does not display an accurate or complete model of the underlying computation, and this may sometimes be deceptive. However, a programmer's internal model of computation does not match with the computer's exact semantics either. Programmers do not code for a machine, but rather for an idealized execution environment [42], commonly referred to as notional machines (i.e., classes of abstractions over a computer's exact semantics). Notional machines have been used as a pedagogical device to help learners build valid mental models of a program, sometimes manifesting themselves through diagrams of memory models and code execution [14]. The notional machine proposed by CROSSCODE, if given enough time learn, may help users develop more useful and consistent mental models of code than existing systems since it is closer to how program behavior is communicated and explained than line-by-line tools.

7.1.3 CROSSCODE's model of navigation may scale to complex programs. Compared to existing line-by-line program visualization tools such as PYTHON TUTOR, CROSSCODE may have a greater potential to scale to more complex code as it aggregates operations rather than presenting them at a static level of detail. However, challenges remain in extending the current design to common programming paradigms such as object-oriented programming, which will require specialized abstractions (i.e., abstraction based on syntax tree will not solely capture models of classes and concepts such as inheritance) and representations (e.g., [16]). Other considerations, such as scaling across multiple files or to more complex states, will require specialized mechanisms for managing complexity in the *Control Flow View* and the *Data View*. Regardless, we believe that the guiding principle behind CROSSCODE, i.e., designing representations of execution based on how program behavior is visually explained, generalizes to these structures as well.

7.2 Future Work

The evaluation points to several critical directions for the design of future programming interfaces. Here, we outline three key areas of investigation for future work.

7.2.1 Aggregations beyond syntax. Our results suggest that top-down exploration of execution based on the program's syntax tree facilitates new and potentially more efficient program understanding and debugging processes. However, there are other appropriate abstraction strategies that were not explored. For one, CROSSCODE treats all source code the same: but there is a rich set of semantics in how the users write the source code and their documentation, e.g., users may implicitly chunk lines of code that serve a similar function. Programs are a form of human communication, akin to natural language: *Can statistical properties of the code (i.e., [5]) be leveraged to infer appropriate abstraction strategies?* The data flow may also be used to inform potential abstractions, for example, a loop that divides a list into two sub-lists, can be described as two aggregate steps: filling in the first sub-list and then filling in the second sub-list, a description which cannot be captured by CROSSCODE's aggregations.

7.2.2 Bridging the control flow and data representations. CROSSCODE separates a program's control flow from its data state. While doing so provides flexibility in the placement, participants found it difficult to track and synchronize between the two views. Various techniques from prior work in live programming and information visualization can help bridge the two views. For one, data can be visually annotated next to control flow structures, e.g., using similar visual techniques as Projection Boxes [25]. Control flow may be navigated by data (e.g., clicking on an item in the list could navigate execution to when that item was last modified). The same abbreviations that occur in the control flow can be extended to the data encoding: If the iterations of a loop are abbreviated, then the corresponding list items can be appropriately abbreviated in the data representation.

7.2.3 Externalizing control flow of synchronous executions and multi-threaded programs. Current representations of control flow in CROSSCODE support a single notion of time. Designing representations for synchronous operations such as web API calls or file I/O could help manage and communicate classes of bugs such as race conditions which are often difficult to form an accurate mental model of. This is especially true in distributed systems, where complex and time-sensitive operations can be difficult to diagnose with traditional debugging tools. The ability to communicate the behavior of concurrent systems to others could facilitate collaboration and improve the overall quality of instruction of distributed systems. By analyzing visual organizations and metaphors used to teach concurrency, future research can inform the design of appropriate representations of concepts such as shared resources, ownership, and collisions, which are otherwise difficult to surface.

8 CONCLUSION

The main conclusion from this research is that aggregation of a program's execution based on the syntax tree can better support program debugging and explanation workflows than line-by-line debuggers. Through a formative study of hand-designed depictions of program execution, we synthesized three design patterns: visualizations should aggregate execution, abbreviate repetitive operations, and present an overview of the execution space. We realized these patterns in CROSSCODE, a program visualization system capable of visualizing and navigating between multiple levels of abstractions. The results of an expert evaluation comparing CROSSCODE, PYTHON TUTOR, and DRAWING highlighted the usefulness of CROSSCODE's representation of control flow to maintain context, as well as its use of aggregations to explore execution beyond line-by-line methods.

ACKNOWLEDGMENTS

REFERENCES

[1] Harold Abelson and Gerald J. Sussman. 1996. *Structure and Interpretation of Computer Programs* (2nd ed.). MIT Press, Cambridge, MA, USA.

[2] acornjs. 2013. Acorn. Retrieved February 13, 2023 from https://github.com/acornjs/acorn

[3] Maneesh Agrawala, Wilmot Li, and Floraine Berthouzoz. 2011. Design Principles for Visual Communication. *Commun. ACM* 54, 4 (apr 2011), 60–69. https://doi.org/10.1145/1924421.1924439

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[5] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A Survey of Machine Learning for Big Code and Naturalness. *ACM Comput. Surv.* 51, 4, Article 81 (jul 2018), 37 pages. https://doi.org/10.1145/3212695

[6] Aivar Annamaa. 2015. Introducing Thonny, a Python IDE for Learning Programming. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '15)*. Association for Computing Machinery, New York, NY, USA, 117–121. https://doi.org/10.1145/2828959.2828969

[7] Ronald Baecker. 1998. Sorting out sorting: A case study of software visualization for teaching computer science. *Software visualization: Programming as a multimedia experience* 1 (1998), 369–381.

[8] Brilliant. 2011. Brilliant. Retrieved September 15, 2022 from https://brilliant.org/

[9] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* 18, 6 (1983), 543–554. https://doi.org/10.1016/S0020-7373(83)80031-5

[10] Wikimedia Commons. 2004. Retrieved September 15, 2022 from https://commons.wikimedia.org/

[11] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press, Cambridge, MA, USA.

[12] Will Crichton, Maneesh Agrawala, and Pat Hanrahan. 2021. The Role of Working Memory in Program Tracing. 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 56, 13 pages. https://doi.org/10.1145/3411764.3445257

[13] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. 2012. Debugger Canvas: Industrial Experience with the Code Bubbles Paradigm. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland) *(ICSE '12)*. IEEE Press, Zurich, Switzerland, 1064–1073.

[14] Paul E. Dickson, Neil C. C. Brown, and Brett A. Becker. 2020. Engage Against the Machine: Rise of the Notional Machines as Effective Pedagogical Devices. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 159–165. https://doi.org/10.1145/3341525.3387404

[15] Jonathan Edwards. 2007. No Ifs, Ands, or Buts: Uncovering the Simplicity of Conditionals. *SIGPLAN Not.* 42, 10 (oct 2007), 639–658. https://doi.org/10.1145/1297105.1297075

[16] Paul Gries, Vlad Mnih, J. Taylor, G M Wilson, and Lee Zamparo. 2005. Memview: a pedagogically-motivated visual debugger. *Proceedings Frontiers in Education 35th Annual Conference* (2005), S1J–11.

[17] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-Based Program Visualization for Cs Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education* (Denver, Colorado, USA) *(SIGCSE '13)*.

Association for Computing Machinery, New York, NY, USA, 579–584. https://doi.org/10.1145/2445196.2445368

[18] C. Hundhausen and S. Douglas. 2000. SALSA and ALVIS: a language and system for constructing and presenting low fidelity algorithm visualizations. In *Proceeding 2000 IEEE International Symposium on Visual Languages*. 67–68. https://doi.org/10.1109/VL.2000.874355

[19] Christopher D. Hundhausen and Jonathan L. Brown. 2007. What You See Is What You Code: A "Live" Algorithm Development and Visualization Environment for Novice Learners. *J. Vis. Lang. Comput.* 18, 1 (feb 2007), 22–47. https://doi.org/10.1016/j.jvlc.2006.03.002

[20] Christopher D. Hundhausen, Sarah A. Douglas, and John T. Stasko. 2002. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages & Computing* 13, 3 (2002), 259–290. https://doi.org/10.1006/jvlc.2002.0237

[21] Hyeonsu Kang and Philip J. Guo. 2017. Omnicode: A Novice-Oriented Live Programming Environment with Always-On Run-Time Value Visualizations. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, New York, NY, USA, 737–745. https://doi.org/10.1145/3126594.3126632

[22] Amy J. Ko and Brad A. Myers. 2009. Finding Causes of Program Output with the Java Whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Boston, MA, USA) *(CHI '09)*. Association for Computing Machinery, New York, NY, USA, 1569–1578. https://doi.org/10.1145/1518701.1518942

[23] Hiroki Kokubun. 2018. Tweakpane. Retrieved February 13, 2023 from https://cocopon.github.io/tweakpane/

[24] Thomas D. LaToza and Brad A. Myers. 2011. Visualizing call graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 117–124. https://doi.org/10.1109/VLHCC.2011.6070388

[25] Sorin Lerner. 2020. Projection Boxes: On-the-Fly Reconfigurable Visualization for Live Programming. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3313831.3376494

[26] Stanley Letovsky. 1987. Cognitive processes in program comprehension. *Journal of Systems and Software* 7, 4 (1987), 325–339. https://doi.org/10.1016/0164-1212(87)90032-X

[27] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between Reading, Tracing and Writing Skills in Introductory Programming. In *Proceedings of the Fourth International Workshop on Computing Education Research* (Sydney, Australia) *(ICER '08)*. Association for Computing Machinery, New York, NY, USA, 101–112. https://doi.org/10.1145/1404520.1404531

[28] Dor Ma'ayan, Wode Ni, Katherine Ye, Chinmay Kulkarni, and Joshua Sunshine. 2020. How Domain Experts Create Conceptual Diagrams and Implications for Tool Design. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3313831.3376253

[29] Bradley N Miller and David L Ranum. 2011. *Problem solving with algorithms and data structures using python Second Edition*. Franklin, Beedle & Associates Inc.

[30] Robert Nystrom. 2021. *Crafting interpreters*. Genever Benning.

[31] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19, 3 (1987), 295–341. https://doi.org/10.1016/0010-0285(87)90007-7

[32] Reducible. 2019. Reducible. Retrieved September 15, 2022 from https://www.youtube.com/c/Reducible

[33] Steve Ruiz. 2020. Perfect Arrows. Retrieved February 13, 2023 from https://github.com/steveruizok/perfect-arrows

[34] Stuart Russell and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach* (3rd ed.). Prentice Hall Press, USA.

[35] Grant Sanderson. 2017. 3Blue1Brown. Retrieved September 15, 2022 from https://www.youtube.com/c/3blue1brown

[36] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 488–498. https://doi.org/10.1145/2491411.2491447

[37] Clifford A. Shaffer, Matthew L. Cooper, Alexander Joel D. Alon, Monika Akbar, Michael Stewart, Sean Ponce, and Stephen H. Edwards. 2010. Algorithm Visualization: The State of the Field. *ACM Trans. Comput. Educ.* 10, 3, Article 9 (aug 2010), 22 pages. https://doi.org/10.1145/1821996.1821997

[38] Clifford A. Shaffer, Ville Karavirta, Ari Korhonen, and Thomas L. Naps. 2011. OpenDSA: Beginning a Community Active-EBook Project. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research* (Koli, Finland) *(Koli Calling '11)*. Association for Computing Machinery, New York, NY, USA, 112–117. https://doi.org/10.1145/2094131.2094154

[39] Daniel Shiffman, Shannon Fry, and Zannah Marsh. 2012. *The nature of code*. D. Shiffman.

[40] B. Shneiderman. 1996. The eyes have it: a task by data type taxonomy for information visualizations. In *Proceedings 1996 IEEE Symposium on Visual Languages*.

336–343. https://doi.org/10.1109/VL.1996.545307

[41] Elliot Soloway and Kate Ehrlich. 1984. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* SE-10, 5 (1984), 595–609. https://doi.org/10.1109/TSE.1984.5010283

[42] Juha Sorva. 2013. Notional Machines and Introductory Programming Education. *ACM Trans. Comput. Educ.* 13, 2, Article 8 (jul 2013), 31 pages. https://doi.org/10.1145/2483710.2483713

[43] J.T. Stasko. 1990. Tango: a framework and system for algorithm animation. *Computer* 23, 9 (1990), 27–39. https://doi.org/10.1109/2.58216

[44] M.-A.D. Storey, F.D. Fracchia, and H.A. Muller. 1997. Cognitive design elements to support the construction of a mental model during software visualization. In *Proceedings Fifth International Workshop on Program Comprehension. IWPC'97*. 17–28. https://doi.org/10.1109/WPC.1997.601257

[45] Margaret-Anne D Storey, F David Fracchia, and Sheelagh Carpendale. 1994. A Top-Down Approach to Algorithm Animation. *Techincal Report CMPT* (1994), 94–05.

[46] Sangho Suh, Celine Latulipe, Ken Jen Lee, Bernadette Cheng, and Edith Law. 2021. Using Comics to Introduce and Reinforce Programming Concepts in CS1. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (Virtual Event, USA) *(SIGCSE '21)*. Association for Computing Machinery, New York, NY, USA, 369–375. https://doi.org/10.1145/3408877.3432465

[47] Bret Victor. 2011. Ladder of Abstraction. Retrieved September 15, 2022 from http://worrydream.com/LadderOfAbstraction/

[48] Bret Victor. 2012. Inventing On Principle. Retrieved September 15, 2022 from https://vimeo.com/38272912

[49] vitejs. 2019. Vite. Retrieved February 13, 2023 from https://vitejs.dev/

[50] A. Von Mayrhauser and A.M. Vans. 1995. Program comprehension during software maintenance and evolution. *Computer* 28, 8 (1995), 44–55. https://doi.org/10.1109/2.402076

[51] Haijun Xia, Ken Hinckley, Michel Pahud, Xiao Tu, and Bill Buxton. 2017. Writ-Large: Ink Unleashed by Unified Scope, Action, & Zoom. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 3227–3240. https://doi.org/10.1145/3025453.3025664

[52] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A Brief Survey of Program Slicing. *SIGSOFT Softw. Eng. Notes* 30, 2 (mar 2005), 1–36. https://doi.org/10.1145/1050849.1050865

[53] Jimkwan Yau. 2018. Sval. Retrieved February 13, 2023 from https://github.com/Siubaak/sval