Translations Alone Do Not Help Programmers Work With Unfamiliar Abstractions

Jacob Yim

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-224
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-224.html

December 19, 2024

Acknowledgement

First, I want to thank my advisor, Sarah, and my mentor, Justin–thank you so much not only for your guidance on this project, but also for teaching me how to do research! I am immensely grateful to have the opportunity to do this kind of work, and I owe it all to you both. I would also like to thank Kevin and Laila, my collaborators on this project, without whom this would never have been possible. A huge thank you goes out to the many, many participants who took part in our study, and to friends who joined pilot studies, gave insightful feedback, and recruited participants. Special thanks to Sami for encouraging me every step of the way, and finally, to my parents for their unconditional love and support.

Translations Alone Do Not Help Programmers Work With Unfamiliar Abstractions

by Jacob Yim

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of Master of Science, Plan II.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Sarah E. Chasins
Research Advisor

December 17, 2024
(Date)

* * * * * * *

Professor Marcia C. Linn
Second Reader

December 18, 2024
(Date)

Translations Alone Do Not Help Programmers Work With Unfamiliar Abstractions

by

Jacob Yim

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Master of Science, Plan II

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sarah E. Chasins, Chair
Professor Marcia C. Linn

Fall 2024

Translations Alone Do Not Help Programmers Work With Unfamiliar Abstractions

Abstract

Translations Alone Do Not Help Programmers Work With Unfamiliar Abstractions

by

Jacob Yim

Master of Science, Plan II in Computer Science

University of California, Berkeley

Professor Sarah E. Chasins, Chair

When programmers edit other programmers' code or computer-generated code, they often need to work with unfamiliar abstractions—e.g., when adopting a new library or language, or entering a preexisting codebase. Prior work has hypothesized that showing a translation from unfamiliar abstractions into familiar abstractions will help. We explored this question in a 98-participant user study. We asked participants to edit Python programs that used an unfamiliar library, with or without access to a translation into vanilla Python. Participants with access to the translation were neither faster nor less error-prone. We used a set of interfaces that augment translations in a range of ways to further explore the question of whether translations can help programmers work with unfamiliar abstractions. Our results suggest design opportunities for the problem of supporting programmers in working with new libraries and languages.

Acknowledgments

First, I want to thank my advisor, Sarah, and my mentor, Justin–thank you so much not only for your guidance on this project, but also for teaching me how to do research! I am immensely grateful to have the opportunity to do this kind of work, and I owe it all to you both. I would also like to thank Kevin and Laila, my collaborators on this project, without whom this would never have been possible. A huge thank you goes out to the many, many participants who took part in our study, and to friends who joined pilot studies, gave insightful feedback, and recruited participants. Special thanks to Sami for encouraging me every step of the way, and finally, to my parents for their unconditional love and support.

1.1 Introduction

Programmers are often tasked with reading, editing, and reusing code written by other programmers and, increasingly, automatic code generators. With the recent rise of LLM programming, there is a particularly urgent need to support programmers in working with code they didn't write themselves. While we expect that programmers know the abstractions they use in their own code, machine-written or peer-written code may use abstractions they do not know. Updating or adapting programs with unfamiliar abstractions can be difficult. These unfamiliar abstractions may include unknown libraries or even unknown programming languages. In this work, we explore automatic tooling for supporting programmers in modifying code that uses unfamiliar libraries or languages.

Prior work has speculated that automatically generated programs can help programmers use unfamiliar abstractions [54, 23, 116, 12], in particular by showing *translations* [92, 18, 4, 37]—that is, by juxtaposing code using unfamiliar abstractions with equivalent code using abstractions the programmer already knows. However, this intervention has not yet been tested. We are not aware of any empirical evidence about whether program translations are helpful to programmers working with unfamiliar abstractions. To fill this gap, we conducted a user study of 98 participants performing modifications to Python code with unfamiliar functions from the TensorFlow library. We use this study to ask: *Do translations help programmers work with unfamiliar abstractions?*

Our study reveals that translations alone do not help programmers work with unfamiliar abstractions. We use a series of custom-designed interfaces to explore whether presenting additional information about translations *can* make translations helpful; for the four custom translation interfaces we tested, the interfaces do make successful participants faster but do not increase the number of successful participants.

This study is certainly not the final word on whether translations help programmers work with unfamiliar abstractions. It may be that translations are helpful if they are particularly readable or are designed for a particular domain. The utility of translations may also depend on the task the user is trying to accomplish, the user's expertise, the perceived distance between source and target language, or other factors entirely. However, it is clear that, in contrast to prior speculation, we do not have reason to believe that translations alone—in absence of other factors or augmentations—help programmers work with unfamiliar abstractions.

Specialized translation interfaces did speed the editing process relative to the no-translation control. This observation leads us to a set of design opportunities. We expect future work can uncover other ways to make translations useful for unfamiliar abstractions.

Contributions This short paper presents the following contributions:

1. **A 98-participant user study assessing participants' success in editing programs that used unfamiliar abstractions.** We compared seven conditions, including five conditions that presented translations into familiar abstractions.

2. **A set of design opportunities based on the findings from this study.** In particular, we offer implications for HCI researchers as well as HCI practitioners regarding tool support for program modification tasks.

Overall, our findings confirm that supporting users in program modification tasks remains an open problem in HCI research. While prior work has speculated that translations may help, our findings suggest that future HCI work cannot necessarily rely on translations alone to fill this gap. Building on these findings, we conclude with lessons and open questions and for the HCI community on how to support program modification tasks as opposed to program authoring tasks.

1.2 Related Work

Despite the repeated speculation that translations may help programmers, there are no existing works that study users to identify whether translations help them with programming tasks. We therefore split the landscape of related works into categories according to whether they (i) study users' use of program translations, (ii) study users doing program modification, program reading, or program comprehension tasks, or (iii) less relatedly, offer tool support for users doing program modification, program reading, or program comprehension tasks. We are not aware of any studies that cover both (i) translations and (ii) their effect on a programming task.

Studies of Users Seeing Translations

We are aware of one study showing translated programs to programmers. Transfer Tutor [92] "guides programmers through code snippets of two programming languages and highlights reusable concepts from a familiar language to learn a new language. Transfer Tutor also warns programmers about potential misconceptions carried over from the previous language" [92]. From qualitative analysis of a thinkaloud study of Transfer Tutor users, they concluded that users did use "learning transfer" as a cognitive strategy. This study did not aim to shed light on the effects of translation on any programming tasks, so there was no non-translation control condition.

Aside from the above, the works that come closest to touching on the topic of showing users translations are other studies related to transfer of learning. Comprehending code that uses unfamiliar abstractions can be thought of as a small-scale instance of transfer of programming languages. Transfer is a topic of ongoing interest in the computer science education research community [88, 87, 104, 9, 75, 99, 100, 42, 98, 62] with some perspectives from software engineering researchers as well [93]. Such research typically focuses on transferring programming skills from one language to another over a relatively long period of time, often in the context of a classroom.

Studies of Users Doing Program Modification, Program Reading, or Program Comprehension

Since the literature includes few studies on the specific task that interests us—working with and modifying code that includes unfamiliar abstractions—we here widen our net to consider works that may touch on related themes. For instance, program reading and program comprehension may be part of the process of program modification.

Working With Others' Code Is Hard

The literature offers clear evidence that working with code that one has not written oneself is difficult [96, 17, 47, 32, 68, 107]. Studies identify that the process is time-consuming [17, 47, 68, 107], and even that professional developers spend 58% of their working hours on code comprehension tasks [107], which may be one component of work with others' code. Especially relevant to our own context, a study of developers performing small code modification tasks found that participants spent 35% of their time understanding unfamiliar code [47].

Making it easier to understand code written by others may be especially important as large language models (LLMs) are increasingly being used for code generation. While many studies have found that LLMs help experienced programmers write code [8, 59, 70], others have found that programmers struggle to use them effectively, in large part because of issues around understanding the LLM-written code [101, 35, 83, 102, 15, 26, 74]. Existing work highlights issues around understanding code enough to check whether it is correct [26, 74] and struggling with unfamiliar abstractions in the LLM-generated code [61, 26, 74].

Studies of Program Comprehension

Since at least the 1970s, there has been a long line of work studying how both novice and expert programmers comprehend code in a familiar language [91, 11, 96, 106, 57, 82, 48, 27, 103, 47, 51, 52, 86, 63, 94, 49, 2]. In contrast to these works, we are interested in exploring how programmers work with code using unfamiliar abstractions.

A smaller corpus of work has investigated how programmers comprehend code when working with new languages [89, 46] or domains [90], including the resources they turn to for learning [1, 4]. Additionally, Gross and Kelleher [30] studied how *non*-programmers comprehend programs. Most relevantly, Shaft and Vessey [90] identify that program comprehension can be easier in a familiar domain with an unfamiliar language than an unfamiliar domain with a familiar language, and Ko and Uttl [46] identify domain knowledge as the best predictor of debugging success. Neither work touches on translation.

Tool Support for Program Modification, Program Reading, or Program Comprehension

Finally, we conclude with existing work on tool support for a variety of tasks related to program modification.

Debugging

Program modification tasks can be thought of as debugging tasks. A wide variety of debugging support tools have been introduced, including tools based on program slicing [105, 21, 31, 108, 115, 45, 44, 7, 58] and tools that characterize failing tests [84, 39, 5, 29, 33, 60, 78, 3]. Researchers have previously highlighted the difficulty in creating tools that can measurably improve debugging outcomes [19, 79].

Pseudocode Generation

Prior work has investigated how to automatically generate pseudocode, with a major application in improving comprehension of programs using unfamiliar abstractions [76, 28, 16]. In one case, Oda et al. [76] evaluate their pseudocode generation system with a user study, providing evidence that pseudocode can improve code comprehension.

Natural Language Explanations

Another related approach is to generate program explanations in natural language. Many researchers have found large language models (LLMs) promising for this purpose. Several researchers found LLMs useful for explaining worked examples to students in computer science classrooms [55, 65, 40], while Balse et al. [6] achieved similar results for explaining student errors. Yan et al. [110] found that lightweight in-situ natural language explanations from LLMs improve code understanding. LLM-powered conversational tools for code understanding have also been a topic of new research: GILT [72] is an IDE plugin that produces code explanations without user prompting, while IntelliExplain [109] enables users to conversationally explain and write code in natural language. Both systems were found to be generally helpful for explaining code to programmers during user studies. Researchers have also reported success with less automated (non-LLM) approaches to explain snippets of code—such as subparts of a program—using natural language summaries [41, 34, 67].

Understanding Large Codebases

Many prior tools have aimed to improve comprehension of large-scale codebases for developers working in a familiar language [71, 97, 56, 38, 85, 95, 36, 10, 20, 43, 50], sometimes to understand changes over time [112, 111]. We focus on smaller code snippets with unfamiliar abstractions, as might be produced by an automated code generator or program synthesizer.

Interpretable Program Synthesis

One line of work that seeks to explain small snippets of code with unfamiliar abstractions is interpretable program synthesis. To explain their output, existing tools have used a variety of techniques such as graphically depicting outputs in forms like comics [69] or block-based programs [14], disambiguation interactions [67], presentation of corner cases [113], or communication of intermediate results and provenance [116]. Another line of synthesis tools aim to explain their work—that is, how they arrived at their solutions. Peleg et al. [81], Hu et al. [37] and Zhang et al. [114] allow the user to guide synthesis, thus requiring the user to understand the synthesizer's work as it operates. Le et al. [53] and Peleg et al. [80] introduce frameworks for modeling these kinds of tools. Nazari et al. [73] introduce a system by which a synthesizer can explain subcomponents of its outputs based on top-level specification.

1.3 Research Questions

Here we briefly describe our research questions and how they connect to our experimental design.

Throughout this section, we use the term *starter code* to refer to a program that includes uses of unfamiliar abstractions. We use the term *translation* to refer to a program with the same input-output behavior, but using only familiar abstractions. We explore the following research questions about translations:

RQ1 Do translations help programmers work with unfamiliar abstractions?

RQ2 How does translation compare to an alternative intervention—natural language explanation—in helping programmers work with unfamiliar abstractions?

RQ3 How can translations be augmented to be more helpful to programmers working with unfamiliar abstractions?

To answer **RQ1**, we compare programmer performance on a program editing task when given two kinds of information:

1. The program using unfamiliar abstractions (starter code).
2. The program using unfamiliar abstractions and a translation using familiar abstractions.

To answer **RQ2**, we compare programmer performance on a program editing task when given:

(3) The program using unfamiliar abstractions and a non-translation natural-language explanation.

To answer **RQ3**, we started with a set of four hypotheses about information that might make translations more helpful to programmers. **RQ3** explores these hypotheses in particular:

H1 A mapping between familiar and unfamiliar program components helps.

H2 Translation of individual program components in isolation helps.

H3 Seeing fine-grained translation steps helps.

H4 Natural-language explanation of the translation helps.

We designed a set of four interfaces to explore these hypotheses (Section 1.4), each of which augments translations in pointed ways. Each one embodies a particular hypothesis about what may affect translations' usefulness. If the hypothesis associated with one of these interfaces is true, we expect to see improved programmer performance with that interface compared to seeing the translation alone.

1.4 Interfaces

In this section, we describe the programming interface associated with each condition our study. In addition to the RQ-relevant features highlighted below, all interfaces featured the ability to read, edit, and run Python programs

Control and Translation Interfaces

We first describe the two interfaces we used to assess **RQ1**, Basic-Control. The Basic-Control interface displays only the original TensorFlow program (Figure 1.1a). The Basic-Translation interface displays the original TensorFlow program and a translation to vanilla Python code produced by an automatic translation tool (Figure 1.1b).

Natural Language Interface

We now describe the interface we used to assess **RQ2**, Alt-NL. The Alt-NL interface (Figure 1.1c) displays the original TensorFlow program alongside a natural language explanation. We produced natural language explanations using the latest version of OpenAI's GPT-4 large language model [77]. We prompted the model using the following text: "Explain this TensorFlow program concisely:", followed by the TensorFlow starter code. We ran each prompt once, before the start of the study, so all participants saw the same explanation for a given program.

Pointed Interfaces

We now describe the four pointed interfaces we developed and used to assess hypotheses **H1–H4** of **RQ3**. Recall that **RQ3** is designed to explore four hypotheses about information that might make translations more helpful to programmers.

We generated the pointed interfaces for each program automatically, based on extracting translation-related information from an automatic translation tool. We intentionally constrained our pointed interfaces to information that can be collected automatically from a translation tool, so that we can expect these kind of interfaces to be automatically generatable rather than relying on human guidance or insights. However, our interface designs are *not* specific to the particular translation tool we instrumented. We therefore follow the description of each interface design with a description of what information a tool must export in order to produce the interface in question.

Pointed Interface 1 (Pointed-Highlight): Highlighting Correspondence Between Translated Components

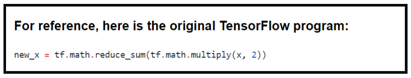
We first describe Pointed-Highlight, the pointed interface designed to explore **H1: A mapping between familiar and unfamiliar program components helps.**

This hypothesis centers on the idea of connecting regions of the source program and translated program. For example, highlighting the correspondence between components (such as uses of a particular variable) in a source and translated program may support programmers. By emphasizing mapping, this interface lets us explore whether making the mapping between components salient may play a role in programmers' usage of translations.

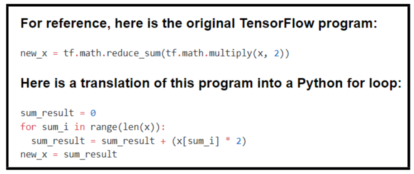
Pointed-Highlight displays the source program, the translated program, and a set of buttons. When clicked, each button highlights one of the TensorFlow functions in the source program and the corresponding code in the translation. It also highlights each function argument in its own color, in both the source and translated programs.

For example, Figure 1.1d shows Pointed-Highlight after clicking the `tf.math.reduce_sum` button. This function computes the sum of the elements in a tensor and is highlighted in yellow in the original TensorFlow program. The corresponding translated code is also highlighted in yellow. The argument of the function, `tf.math.multiply(x, 2)`, doubles `x` and is highlighted in red in the original TensorFlow program. This argument determines both the number of iterations of the translated `for` loop and the value added to `sum_result` at each iteration. Consequently, these parts of the translation (and nothing else) are highlighted in red. These highlighted visuals make explicit the mappings between function calls and function arguments, even when they look different in the source and translated programs.

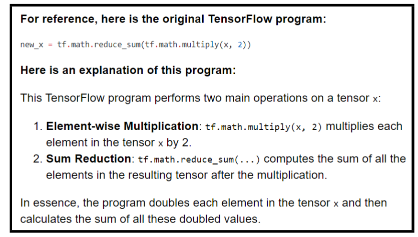
Tool Requirements for Implementation To generate a Pointed-Highlight instance, we need annotations in both the input and output programs. A tool that translates between the source and target language substitutes abstractions from the target language in for components of the source language. Whenever a substitution occurs, the tool must annotate



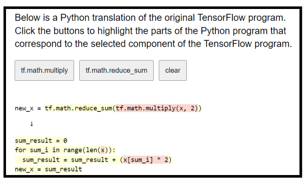
(a) Basic-Control (**RQ1**): shows just the starter code.



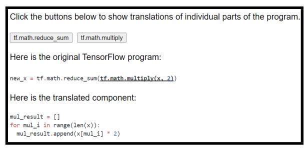
(b) Basic-Translation (**RQ1**): shows the starter code and a translation.



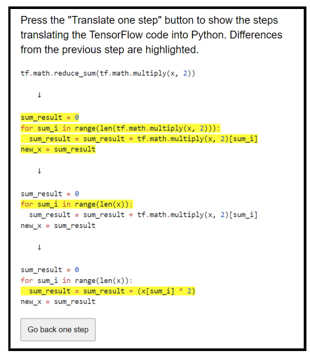
(c) Alt-NL (**RQ2**): shows the starter code and a natural language explanation.



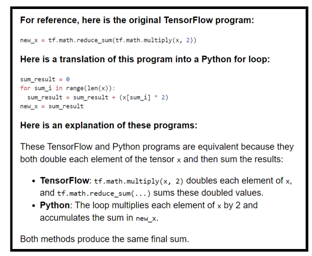
(d) Pointed-Highlight (**RQ3**): shows the starter code and a translation, highlighting corresponding components.



(e) Pointed-Individual (**RQ3**): shows the starter code and buttons that translate individual components.



(f) Pointed-StepByStep (**RQ3**): shows the starter code and a translation as well as a sequence of intermediate translation steps.



(g) Pointed-Translation+NL (**RQ3**): shows the starter code, a translation, and a natural language explanation of why the translation is equivalent.

Figure 1.1: **The seven interfaces we use to answer our research questions.** Each interface specifically helps answer one of our research questions from Section 1.3. **RQ1** asks whether translations help programmers work with unfamiliar abstractions; we compare Basic-Control (a) and Basic-Translation (b) to answer it. **RQ2** asks whether natural language explanations help; we additionally compare Alt-NL (c) to answer it. **RQ3** asks whether translations can be augmented to be more helpful; we compare the pointed interfaces (d–g) to answer it.

the start and end of the slice of the input program that was replaced; the tool must also annotate the start and end of the slice of the output program that was produced by the substitution. All annotations must remain, even as multiple substitutions are applied. The tool must use this same process if *part* of a translated program is drawn from an annotated portion of the source program. For example, in the example above, the red-highlighted argument was `tf.math.multiply(x, 2)`; when `x` was drawn from the argument to be used as the input to `len`, the tool must add the annotations to maintain the red highlighting, even for this partial use.

Pointed Interface 2 (Pointed-Individual): Translating Individual Components

We designed POINTED-INDIVIDUAL to explore **H2: Translation of individual program components in isolation helps.**

POINTED-INDIVIDUAL does not display a whole-program translation of the original TensorFlow program. Instead, it provides a set of buttons that show translations of individual components of the original TensorFlow program. Each button corresponds to a TensorFlow function in the original program. When these buttons are clicked, POINTED-INDIVIDUAL underlines the function call in the original program and displays a translation of the underlined component. POINTED-INDIVIDUAL provides a translation of the entire program only when the user clicks the button that corresponds to the outermost TensorFlow function call.

For example, Figure 1.1e shows POINTED-INDIVIDUAL after clicking the `tf.math.multiply` button. POINTED-INDIVIDUAL underlines this function call in the source program and displays only this function's translation.

Tool Requirements for Implementation To produce POINTED-INDIVIDUAL, we need translations not only for the whole program, but also for each individual use of an unfamiliar abstraction. For a translation tool, a straightforward way of generating the requisite information is as follows: During a translation, whenever the tool encounters a use of an unfamiliar abstraction, rerun the tool on just the use of the unfamiliar abstraction. The results of these smaller component translations can be stored alongside the whole-program translation.

Pointed Interface 3 (Pointed-StepByStep): Step-by-Step Translations

We use POINTED-STEPBYSTEP for exploring hypothesis **H3: Seeing fine-grained translation steps helps**.

Rather than a single transformation, translations may also be viewed as a sequence of more granular transitions between programs. If step-by-step translation helps, then displaying the individual steps that form the full translation (rather than a single, all-at-once transformation) may support programmers working with unfamiliar abstractions.

POINTED-STEPBYSTEP initially shows only the original TensorFlow program and a "Translate one step" button. When the user clicks the step button, POINTED-STEPBYSTEP displays one step of the translation process, with differences from the previous step highlighted in yellow. The user can click the step button repeatedly until the program is fully translated. POINTED-STEPBYSTEP also displays a "Go back one step" button to hide the latest step.[1]

[1] In our implementation, these translation steps correspond to underlying rewrite rules in the tool that powers the interface. POINTED-STEPBYSTEP thus exposes the internals of the underlying tool.

For example, Figure 1.1f shows POINTED-STEPBYSTEP after clicking the "Translate one step" button repeatedly until the program is fully translated. The final block of code is a full translation, and all intermediate blocks are partial translations en route to the final translation. POINTED-STEPBYSTEP thus emphasizes fine-grained steps of an incremental translation process.

Tool Requirements for Implementation POINTED-STEPBYSTEP works naturally for a translation tool that proceeds via a sequence of smaller translation steps. In addition to the source program and translated program, this interface requires a sequence of intermediate programs en route to the translated program. For the common case of a translation tool that uses rewrite rules, the tool can log the program before and after each transformation is applied. The "diff" between any given pair of programs can either be computed after the fact at interface time (our approach), or packaged with the sequence of programs as an output from the translation tool.

Pointed Interface 4 (Pointed-Translation+NL): Translations Supported by Natural Language Explanations

We use POINTED-TRANSLATION+NL to explore hypothesis **H4: Natural-language explanation of the translation helps**.

We posit that an explanation of the translation written in natural language may make translations more useful to programmers. POINTED-TRANSLATION+NL thus shows the original TensorFlow program, a translation, and a natural language explanation of why the translation is equivalent.

We produced natural language explanations using the same large language model as ALT-NL. We prompted the model using the text: "Concisely explain why the following TensorFlow and Python programs are equivalent:", followed by the original TensorFlow program and then the Python translation. As with ALT-NL, we ran all prompts ahead of time so that all participants saw the same explanation.

Tool Requirements for Implementation To be compatible with this interface, a translation tool only needs to provide the translation itself. No additional information from the internals of the translation process is required.

1.5 Study

We describe the study protocol and our findings.

Study Structure

We conducted a seven-condition between-subjects study with 98 participants.

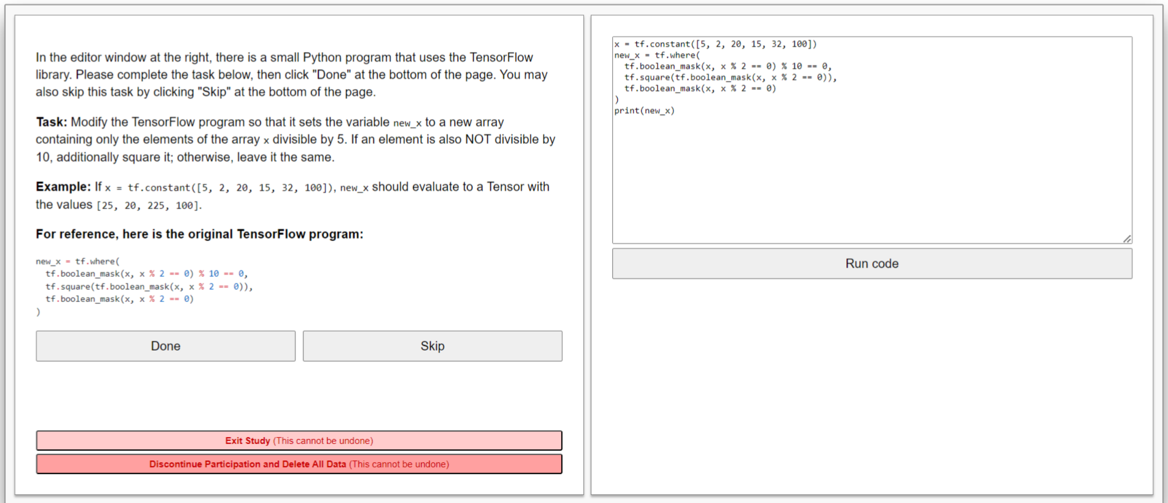


Figure 1.2: **The main part of the web interface for the study.** Participants used this interface (extended in a variety of ways we describe in Sections 1.4, 1.4, and 1.4 and show in Figure 1.1) to complete program modification tasks. On the left is a (i) description of the interface, (ii) a program modification task, (iii) the starter code, and (iv) buttons to submit the current code, skip the task, exit the study, and discontinue participation and delete all data. On the right is (i) a code editor initialized with the starter code and (ii) a run button that runs the current code and displays the result.

Participants and Recruiting We recruited via a screening survey about respondents' prior experience with Python and a variety of Python libraries (including TensorFlow), as well as asking them to predict the outputs of three snippets of Python code. We recruited participants who (i) self-identified as having experience with Python, (ii) did not self-identify as having experience with TensorFlow, and (iii) correctly predicted the output of all three snippets of Python code. We recruited 98 participants (14 per condition), primarily through university-affiliated mailing lists, newsletters, and forums, and through snowball sampling. We ran one-on-one study sessions over Zoom. We compensated each participant with a 30 USD Amazon gift card. This study was approved by our institution's Institutional Review Board.

We balanced self-identified programming experience levels across the seven conditions using a survey question proposed and validated by Feigenspan et al. [25]. Figure B1 in Appendix B shows the distribution of self-identified programming experience by condition. Tables B1 and B2 in Appendix B show that the secondary metrics of familiarity with other libraries such as NumPy and years of experience are roughly balanced among the conditions.

Session Protocol We asked each participant to complete a tutorial task, then three program editing tasks (shown in Table A1 in Appendix A). Each participant used only a single interface. Each task included starter code that uses some abstractions from the TensorFlow high-performance computing library [66] and a request to modify the starter code to match a new described behavior, as we show in Figure 1.2. We ran a between-subjects study in which participants completed these tasks in a fixed order in one of seven conditions, corresponding to the seven interfaces described in Section 1.4: two for studying **RQ1**, one for **RQ2**, and four for **RQ3**. All interfaces featured the ability to read, edit, and run Python programs, as well as to skip the task entirely.

After informed consent, we guided participants through the tutorial. During the tutorial, the researcher explained the type of modification tasks participants would perform. The researcher then explained the features of the study interface, including the interface elements specific to the participant's assigned condition. At the end of the tutorial, participants completed the tutorial task, a simpler modification task than the main study tasks. After completing this task, participants proceeded to the main study tasks.

During the main study tasks, participants were permitted to use all internet resources and software tools except those capable of code generation (e.g. ChatGPT, Copilot, program synthesizers). Study sessions were capped at 90 minutes, after which participants "timed out" and were asked to close the study webpage. Text written in the code editor and interactions on the webpage (e.g. button presses) were periodically logged by the study interface.

Following the tasks, participants were asked to provide feedback on the study during a short debriefing session.

Results

To answer our research questions, we examine two measurements:

1. **Success rate**, the number of tasks for which a participant performed the program modification task successfully (within the 90-minute time limit) divided by the number of tasks they were assigned (3).[2] Higher is better.

2. **Time taken**, the total time taken by a participant who successfully completes all tasks (not including the tutorial or tutorial task). Lower is better.

We do not examine success rates or time taken for individual tasks, as these may be subject to learning effects that are not controlled for by our study design.

Success rate has a discrete distribution with possible values of 0%, 33%, 67%, and 100% (as shown in Figure B2 in Appendix B), so small distributional changes can shift the median by 16.5% or more. Thus, we use the sample mean as a measure of central tendency

[2]We used the Hypothesis [64] property-based testing library to assess correctness over the domain of integers from −1000 to +1000.

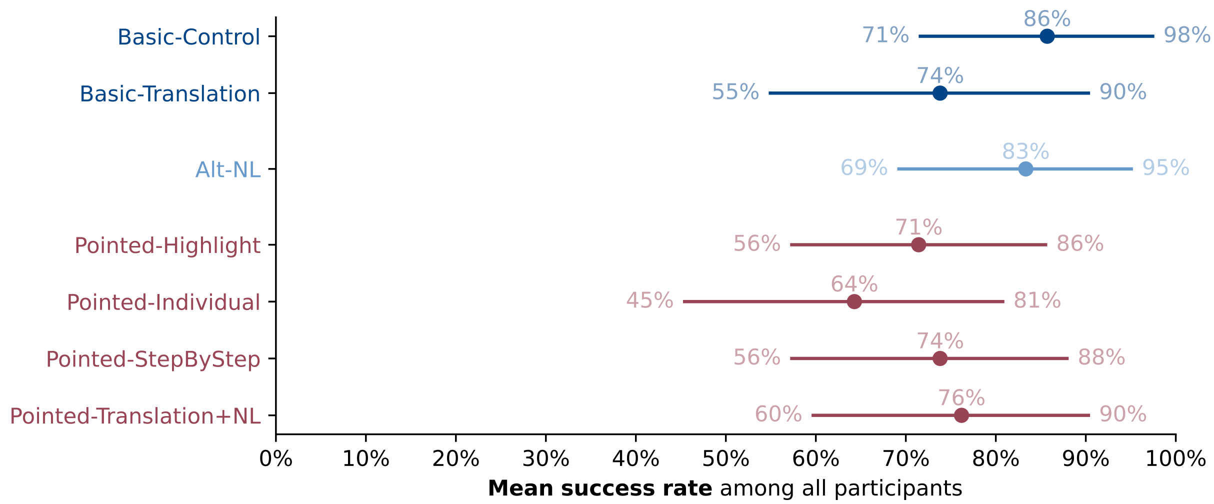


Figure 1.3: **Mean success rate among all participants from our study broken down by interface.** **Success rate** is defined as the number of tasks for which a participant performed the program modification task successfully (within the 90-minute time limit) divided by the number of tasks they were assigned (3). Error bars are 95% bootstrap confidence intervals for the estimator and do not directly correspond to dispersion of the data nor to statistical significance. Chart coloring indicates which research question analyzes the interface in question.

for **success rate**. On the other hand, **time taken** has a continuous distribution. As **time taken** is always positive, we expect it to be right-skewed, possibly with outliers. Figure B3 in Appendix B confirms this expectation. Therefore, we use the sample median as a measure of central tendency for **time taken**. Moreover, we report information about **time taken** only for successful participants (those who finished all three tasks successfully), as the time a participant takes to skip a task or provide an incorrect answer does not help answer our research questions. Figures 1.3 and 1.4 display estimates of these measures from our dataset with two-sided 95% confidence intervals (CIs) computed via the BCa bootstrap [24] with 99,999 resamples. We avoid *p*-values and instead quantify the uncertainty in our measurements with CIs in light of best practices for fair statistical communication in HCI [22].

Translations Alone Do Not Help Programmers Work With Unfamiliar Abstractions

To answer **RQ1**, we compare Basic-Control and Basic-Translation. Figure 1.3 shows Basic-Control has a mean **success rate** of 86% (CI: 71–98%) and Basic-Translation has a mean **success rate** of 74% (CI: 55%–90%). Figure 1.4 shows Basic-Control has a median **time taken** for successful participants of 32.0 minutes (CI: 18.4–45.7 minutes)

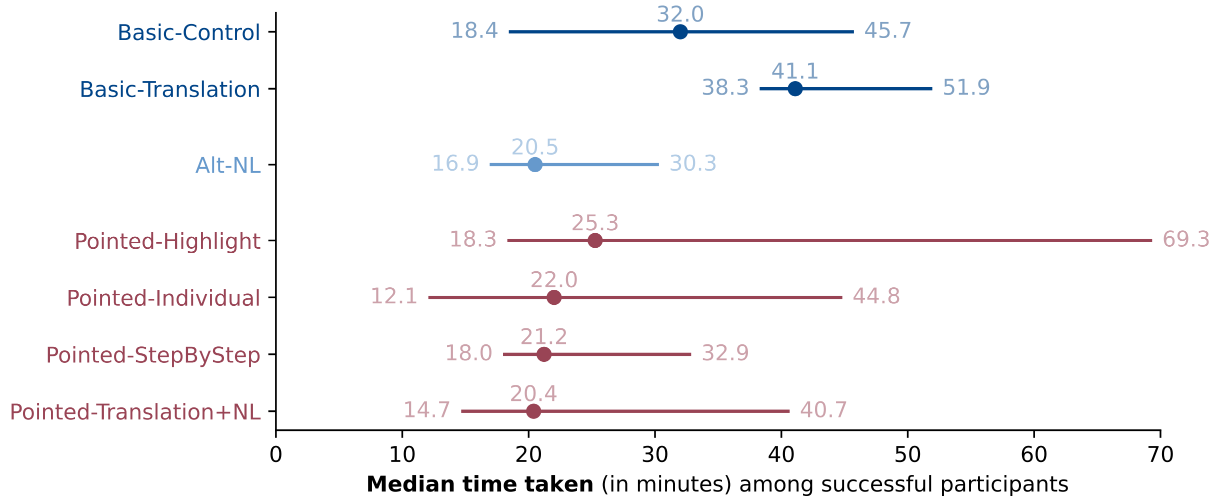


Figure 1.4: **Median time taken among successful participants from our study broken down by interface. Time taken** is defined as the total time taken by a participant to complete all tasks, and we consider only participants who successfully completed all three tasks for this chart. Error bars are 95% bootstrap confidence intervals for the estimator and do not directly correspond to dispersion of the data nor to statistical significance. Chart coloring indicates which research question analyzes the interface in question.

and Basic-Translation has a median **time taken** for successful participants of 41.1 (CI: 38.3–51.9 minutes). Participants using Basic-Translation thus did not perform better than Basic-Control on either measure.

Natural Language Explanations Moderately Speed Work With Unfamiliar Abstractions

To answer **RQ2**, we additionally compare to Alt-NL. Figure 1.3 shows Alt-NL has a mean **success rate** of 83% (CI: 69–95%), and Figure 1.4 shows Alt-NL has a median **time taken** for successful participants of 20.5 minutes (CI: 16.9–30.3 minutes). Participants using Alt-NL thus performed substantially better than Basic-Translation and moderately better than Basic-Control, having a similar mean **success rate** and a lower **time taken** compared to the latter.

Augmented Translations Moderately Speed Work With Unfamiliar Abstractions

To answer **RQ3**, we additionally compare to our four pointed interfaces. Figure 1.3 indicates all four pointed interfaces have moderately lower mean **success rate** than Basic-Control or Alt-NL, roughly on par with Basic-Translation. Figure 1.4 indicates all four pointed interfaces have median **time taken** substantially better than Basic-Translation and moderately better than Basic-Control, on par with Alt-NL. One caveat is the large CI for Pointed-Highlight (18.3–69.3 minutes), which indicates substantially less certainty in our measurement for this condition.

Threats to Validity and Limitations

Threats to Validity Comparing **time taken** only among successful participants may introduce survivorship bias. An alternative study design could be to ask participants to keep re-submitting their answer until it was correct. This design would assume participants have access to a correctness oracle, which may reduce ecological validity.

The particular interfaces, translations, and large language model prompts we used in this study are not representative of all possibilities that test our research questions. While we do see consistent results across the four pointed interfaces, it is possible that other designs would yield different outcomes. Moreover, our interfaces and others may perform differently on different tasks or task domains. We also used one particular library (TensorFlow) and language (Python) for our study, and our results may not generalize to libraries or languages that are substantially different from these choices, such as those relying heavily on static types or manual memory management. Additionally, our findings may not generalize to populations beyond the one we studied, which consisted of programmers already familiar with Python (but not TensorFlow) who primarily were students and recent alumni of R1 universities in the United States.

Lastly, even with a small effect size (such as improving **time taken** by 1 second), increasing the number of participants in each condition would eventually yield confidence intervals that detect these differences. It is therefore possible that we fail to detect true differences between the conditions due to small effect size.

Limitations This study is limited in that we did not conduct qualitative data analysis, which could provide nuanced insights into participants' experience of each interface. We also cannot make comparisons among the individual tasks due to possible learning effects from one task to the next. This could have been mitigated by a between-subjects counterbalanced study design, but we chose not to make the additional assumption that aggregating among different task orders would yield a meaningful estimate of the **success rate** and **time taken** on each interface.

Lastly, we emphasize that, although it may be tempting to use our data about program modification to draw conclusions about program comprehension or learning outcomes, the ability to work with and modify programs does not necessarily require a deeper understanding of the programs or the abstractions it relies on.

Exploratory Observations About Additional Resources

We did not formally collect or analyze qualitative data, but we did note some exploratory observations during our study sessions that point to how participants used additional resources. We observed that many participants used internet resources, including official TensorFlow documentation, Google search, and StackOverflow. Additionally, many participants used the text editor to iteratively run, modify, and rerun code. Some participants also used print statements, commented code, or, when provided translations, copied translated programs into the editor to run or modify them. Finally, during post-study debriefing, many participants expressed that running the code in the text editor was helpful, and some participants mentioned that looking at library documentation was unhelpful, citing its verbosity or the difficulty of finding relevant information.

1.6 Design Opportunities

Here we take up the question of what our results should mean for future work in HCI.

Our results confirm that supporting programmers in program modification tasks remains a huge open problem. With interventions including state-of-the-art LLM-written explanations, translation into familiar abstractions, and interface-augmented translations, program editing times were still in the above-15-minutes range. We have seen program drafting tools that improve *program authoring* times dramatically, sometimes by an order of magnitude. We see an opportunity for making progress towards delivering the same speedups for *program modification*. Here we suggest four directions for future design work to advance us on that path.

For Synthesizers, Compilers, and Other Tools that Make Translations: Translation Augmentations May Speed Editing Our findings offered evidence that translations alone are not enough to help participants complete the particular tasks we assigned. For user-facing tools that produce translations—e.g., some program synthesizers—designers may be tempted to provide translations as an aid to users. Although this may increase our confidence as tool builders that our tool really explains what it's doing, it may not actually help users. Designers should consider: (i) Assessing whether translations help. (ii) Designing their tools to produce the kinds of supplementary information described in Section 1.4; all four pointed interfaces made participants faster than the control or translation-only conditions, which suggests a possible role for those translation augmentations.

For Situations Where Users Must Understand Translations: Translation Augmentations May Affect Understanding Setting aside the goal of supporting modification tasks, all four pointed interfaces changed participants' behaviors relative to the translation-only condition. It is therefore possible that translation augmentations may affect programmers' *understanding* of the translation, not just their ability to work with the unfamiliar code. This suggests one direction for researchers, and another for practitioners. For researchers: Does the additional information surfaced in the four pointed interfaces support programmers' understanding of a translation? For practitioners: For situations where understanding or interpreting a translation is a key goal, consider adding and assessing the interactions we prototyped in the pointed interfaces we describe in Section 1.4.

For Low Data Regimes: Classical Methods as an Alternative Our results offer evidence that classical methods—specifically, mechanical program translations, paired with information extracted from the translation process—are competitive with natural language explanations. For situations where LLMs do not work well—e.g., brand new abstractions, low data regimes [13]—designers can consider translation-based programming aids. With LLMs driving a surge of research on automatic programming aids, but often limited to high-resource programming languages, we posit that translation-backed techniques suggest a design opportunity: For programming tools where LLM-generated explanations fail for low-resource languages or niche problems, is there an opportunity to combine both approaches?

How should we support modification tasks? Finally, although our pointed interfaces allowed us to explore four hypotheses about what kind of information makes translations more helpful, this study should not be the last entry on this question. Of all seven conditions in our study, participants completed the highest number of editing tasks in the control condition! Natural language explanations, translations, and augmented translations all lowered the success rate, even as explanations and augmented translations improved completion times. Seeing *only* the program with the unfamiliar abstractions produced the highest rate of successful task completions even though—because?—it was far from the fastest. This suggests clear design opportunities: (i) What interventions can designers invent that will lower task times without lowering completion rates? (ii) Further, what interventions can designers invent that will lower program modification times even more, delivering the order-or-magnitude speedups we've seen for program authoring?

1.7 Conclusion

Prior work has hypothesized that showing a translation from unfamiliar abstractions into familiar abstractions will help programmers work with unfamiliar libraries and languages. In a 98-participant user study, we found that participants with access to a translation were neither faster nor less error-prone than participants without access to a translation. We found that programmatically-generated supplementary information, in a variety of different interfaces, made successful participants faster at using translations—but did not make more participants successful.

Our results suggest an open problem: Beyond the four pointed interfaces introduced here, how can tools use translations to help programmers work with new libraries and languages?

Beyond this design opportunity, the work suggests a broad set of open translation-related research questions for future work. Here we list a few:

1. If we are constrained to show only a translation with no additional information, what makes a translation better or worse?

2. We used a fixed translated program for all conditions, including the translation-only condition. Is there a tradeoff between making a translated program easy to understand in isolation versus making the translation itself—the mapping between input and output—easy to understand?

3. Can we characterize situations in which translations are more or less helpful? Based on the domain, the kind of task, the conceptual distance between source and target language, the background or experience levels of the programmers, or other features?

We hope future work will take up these questions so that, as a community, we can build even better tools to aid programmers in the increasingly common situation of working with code they did not write themselves.

Appendix A

Study Tasks

Table A1: **Program modification tasks for our study.** Data from the tutorial task was not included in any analysis.

<table>
<tr><th>Name</th><th>Task Description</th><th>Starter Code</th></tr>
<tr><td>Tutorial</td><td>Modify the TensorFlow program so that it sets the variable <code>new_x</code> to the sum of the product of each entry of <code>x</code> with 3.

Example: If <code>x = tf.constant([1, 1, 2, 3, 5])</code>, <code>new_x</code> should evaluate to a Tensor with the value <code>36</code>.</td><td><pre>new_x = tf.math.reduce_sum(tf.math.multiply(x, 2))</pre></td></tr>
<tr><td>Task 1</td><td>Modify the TensorFlow program so that it sets the variable <code>new_x</code> to a new array containing only the elements of the array <code>x</code> divisible by 5. If an element is also NOT divisible by 10, additionally square it; otherwise, leave it the same.

Example: If <code>x = tf.constant([5, 2, 20, 15, 32, 100])</code>, <code>new_x</code> should evaluate to a Tensor with the values <code>[25, 20, 225, 100]</code>.</td><td><pre>new_x = tf.where(
  tf.boolean_mask(x, x % 2 == 0) % 10 == 0,
  tf.square(tf.boolean_mask(x, x % 2 == 0)),
  tf.boolean_mask(x, x % 2 == 0)
)</pre></td></tr>
<tr><td>Task 2</td><td>Modify the TensorFlow program so that it sets the variable <code>new_x</code> to a rolling weighted sum of array <code>x</code> with a window size of 2. The weights should be 2 and 1. That is, the first element of the new array should be the first element of <code>x</code> multiplied by 2 plus the second element of <code>x</code> multiplied by 1; the second element of the new array should be the second element of <code>x</code> multiplied by 2 plus the third element of <code>x</code> multiplied by 1; and so on.

Example: If <code>x = tf.constant([4, 12, 8, 8])</code>, <code>new_x</code> should evaluate to a Tensor with the values <code>[20, 32, 24]</code>.</td><td><pre>new_x = tf.squeeze(
  tf.nn.conv1d(
    tf.reshape(x, [1, int(x.shape[0]), 1]),
    tf.constant([[[1]], [[1]], [[1]]]),
    1,
    'VALID'
)
)</pre></td></tr>
<tr><td>Task 3</td><td>Modify the TensorFlow program so that it sets the variable <code>new_x</code> to the maximum of the minimum of every other sub-array in the array <code>x</code> (that is, the array at index 0, index 2, index 4, etc.).

Example: If <code>x = tf.constant([[1, 2, 3], [101, 102, 103], [11, 12, 13]])</code>, <code>new_x</code> should evaluate to a Tensor with the value <code>11</code>.</td><td><pre>new_x = tf.math.reduce_min(
  tf.math.reduce_max(
    tf.boolean_mask(x, tf.range(len(x)) % 3 == 0),
    axis=1
)
)</pre></td></tr>
</table>

Appendix B

Distributions of Study Data

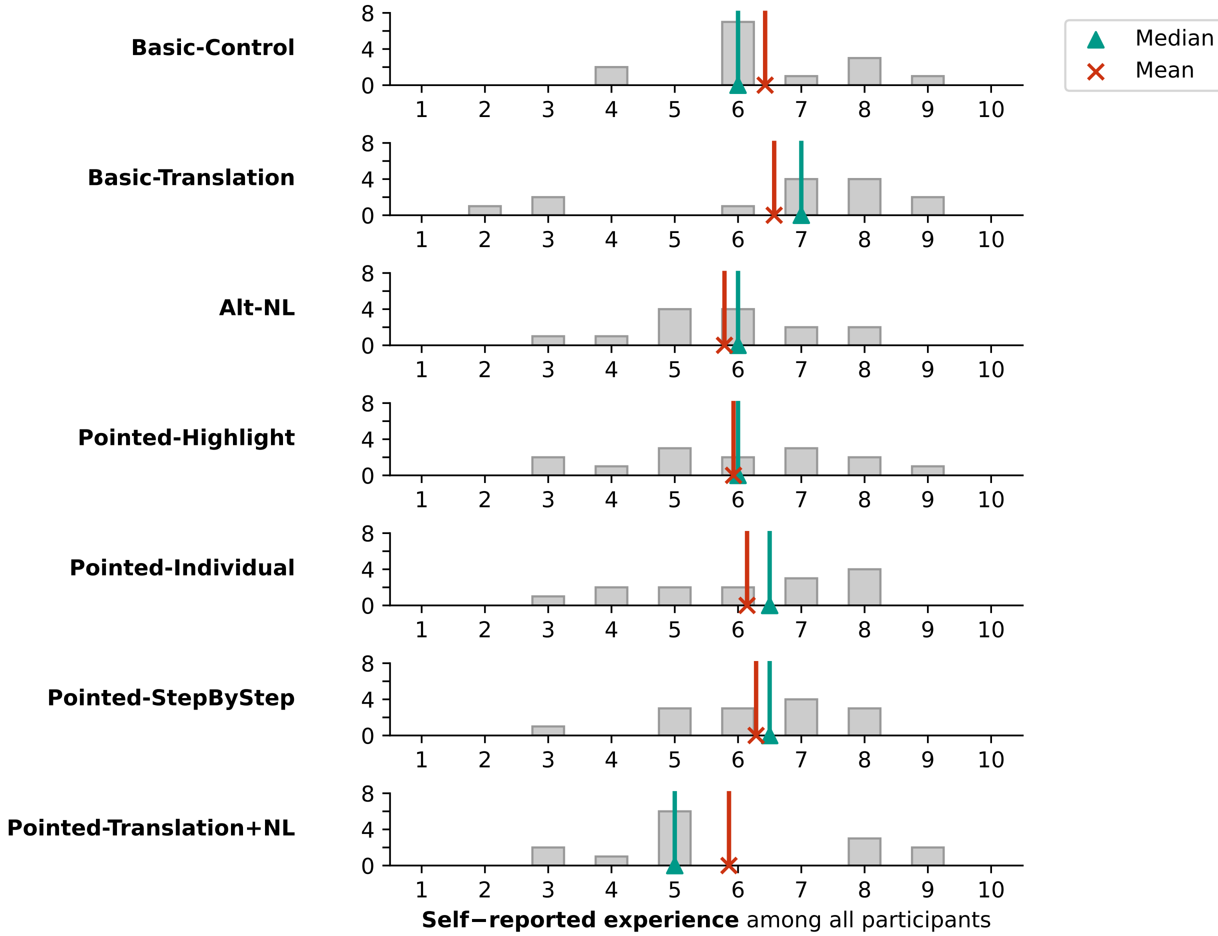


Figure B1: **Distribution of self-reported experience (on a scale of 1–10) among all participants broken down by interface.** Self-reported experience is discrete data, so we use a bar chart (with a proportional x-axis) to display it; the widths of the bin do not carry meaning.

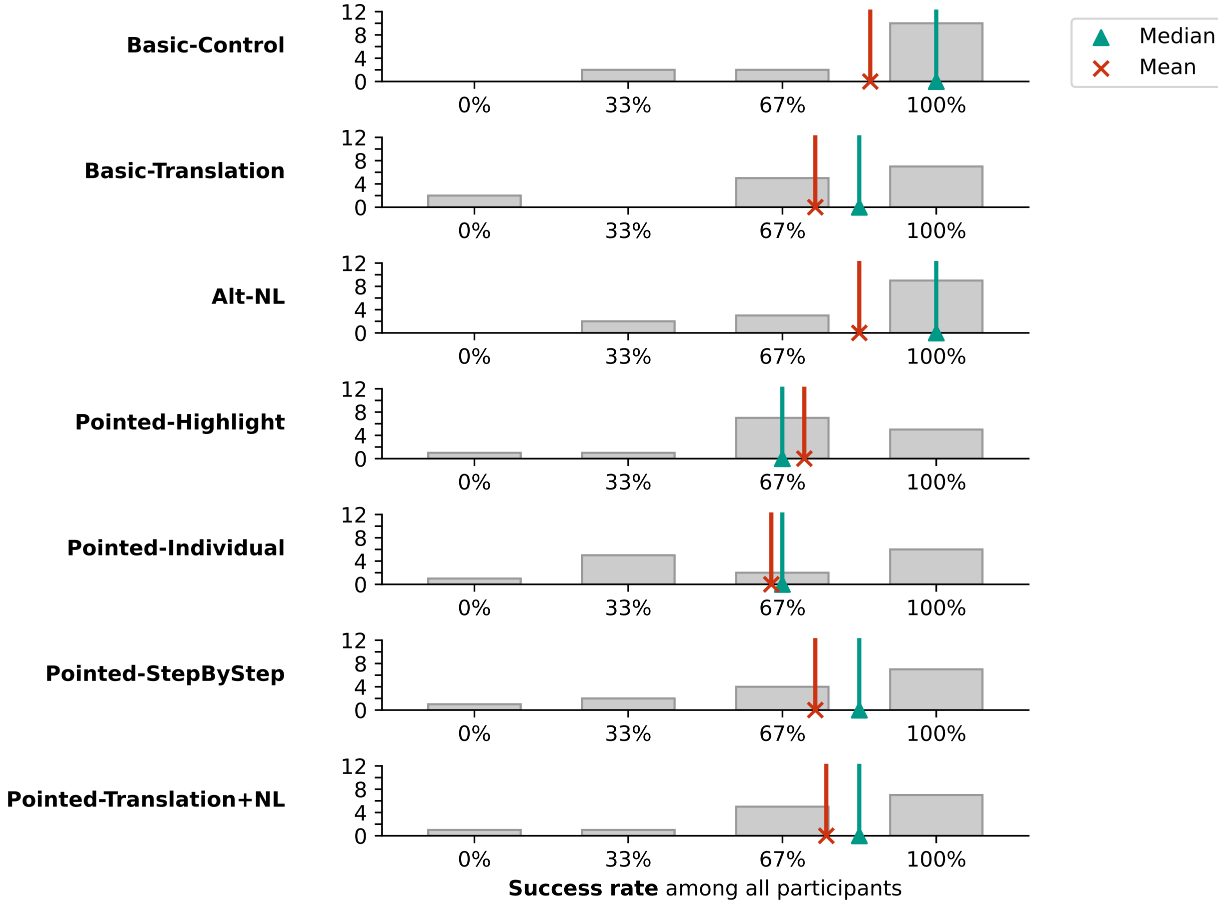


Figure B2: **Distribution of success rate among all participants broken down by interface. Success rate** is defined as the number of tasks for which a participant performed the program modification task successfully (within the 90-minute time limit) divided by the number of tasks they were assigned (3). Each interface was used by $98/7 = 14$ participants. **Success rate** is discrete data, so we use a bar chart (with a proportional x-axis) to display it; the widths of the bin do not carry meaning.

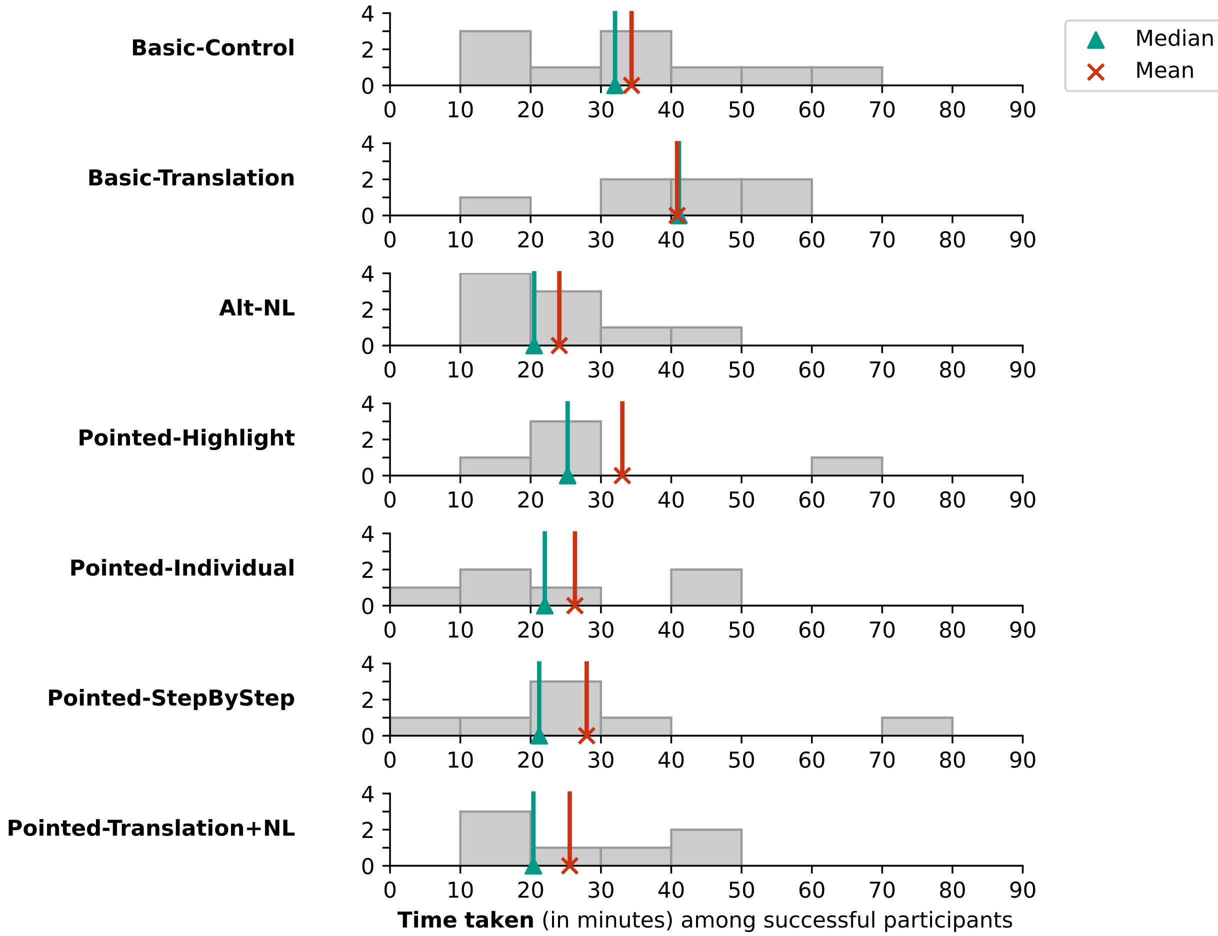


Figure B3: **Distribution of time taken among correct participants broken down by interface. Time taken** is defined as the total time taken by a participant to complete all tasks. **Time taken** is continuous data, so we use a histogram to display it.

	`beautiful_soup`	`flask`	`matplotlib`	`nltk`	`numpy`	`pandas`	`pytorch`
Basic-Control	3	0	12	2	14	11	1
Basic-Translation	3	0	9	0	13	10	3
Alt-NL	3	3	10	0	12	10	2
Pointed-Highlight	4	3	8	2	14	10	4
Pointed-Individual	3	5	9	1	12	11	3
Pointed-StepByStep	4	2	11	0	11	12	3
Pointed-Translation+NL	2	3	9	0	11	10	2

Table B1: **The number of participants in each condition that had worked with a variety of existing libraries.** All participants indicated that they had **not** worked with the library we used in this study, TensorFlow. The three most similar libraries, `numpy`, `pandas`, and `pytorch`, had roughly the same number of participants that had worked with it in each condition.

	YoE	**Weekly YoE**
Basic-Control	4.0	2.5
Basic-Translation	3.5	2.5
Alt-NL	4.0	2.0
Pointed-Highlight	5.5	2.0
Pointed-Individual	4.0	3.0
Pointed-StepByStep	4.5	4.0
Pointed-Translation+NL	5.0	2.0

Table B2: **The median years of programming experience participants had in each condition.** The YoE column indicates median years of overall experience programming, and the Weekly YoE column indicates median years of experience programming on a weekly basis. The overall median YoE is 4 and Weekly YoE is 3.

Bibliography

[1] Parastoo Abtahi and Griffin Dietz. "Learning Rust: How Experienced Programmers Leverage Resources to Learn a New Programming Language". In: *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*. Chi Ea '20. New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1–8. ISBN: 978-1-4503-6819-3. DOI: 10.1145/3334480.3383069. (Visited on 03/30/2024).

[2] Abdulaziz Alaboudi and Thomas D. LaToza. "Using Hypotheses as a Debugging Aid". In: *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Aug. 2020, pp. 1–9. DOI: 10.1109/vl/hcc50065.2020.9127273. (Visited on 03/30/2024).

[3] Abdulaziz Alaboudi and Thomas D. Latoza. "Hypothesizer: A Hypothesis-Based Debugger to Find and Test Debugging Hypotheses". In: *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*. Uist '23. New York, NY, USA: Association for Computing Machinery, Oct. 2023, pp. 1–14. ISBN: 9798400701320. DOI: 10.1145/3586183.3606781. (Visited on 03/30/2024).

[4] Gina R. Bai, Joshua Kayani, and Kathryn T. Stolee. "How Graduate Computing Students Search When Using an Unfamiliar Programming Language". In: *Proceedings of the 28th International Conference on Program Comprehension*. Icpc '20. New York, NY, USA: Association for Computing Machinery, Sept. 2020, pp. 160–171. ISBN: 978-1-4503-7958-8. DOI: 10.1145/3387904.3389274. (Visited on 03/30/2024).

[5] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. "From symptom to cause: localizing errors in counterexample traces". In: *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Popl '03. New York, NY, USA: Association for Computing Machinery, Jan. 2003, pp. 97–105. ISBN: 978-1-58113-628-9. DOI: 10.1145/604131.604140. URL: https://dl.acm.org/doi/10.1145/604131.604140 (visited on 09/11/2024).

[6] Rishabh Balse et al. "Evaluating the Quality of LLM-Generated Explanations for Logical Errors in CS1 Student Programs". In: *Proceedings of the 16th Annual ACM India Compute Conference*. Compute '23. New York, NY, USA: Association for Computing Machinery, Dec. 2023, pp. 49–54. ISBN: 9798400708404. DOI: 10.1145/3627217.3627233. URL: https://dl.acm.org/doi/10.1145/3627217.3627233 (visited on 09/08/2024).

[7] David Binkley et al. "ORBS: language-independent program slicing". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. New York, NY, USA: Association for Computing Machinery, Nov. 2014, pp. 109–120. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635893. URL: https://dl.acm.org/doi/10.1145/2635868.2635893 (visited on 09/11/2024).

[8] Christian Bird et al. "Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools". In: *Queue* 20.6 (Jan. 2023), pp. 35–57. ISSN: 1542-7730. DOI: 10.1145/3582083. URL: https://doi.org/10.1145/3582083.

[9] Matt Bower and Annabelle McIver. "Continual and Explicit Comparison to Promote Proactive Facilitation during Second Computer Language Learning". In: *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*. ITiCSE '11. New York, NY, USA: Association for Computing Machinery, June 2011, pp. 218–222. ISBN: 978-1-4503-0697-3. DOI: 10.1145/1999747.1999809. (Visited on 09/12/2024).

[10] Andrew Bragdon et al. "Code Bubbles: A Working Set-Based Interface for Code Understanding and Maintenance". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Chi '10. New York, NY, USA: Association for Computing Machinery, Apr. 2010, pp. 2503–2512. ISBN: 978-1-60558-929-9. DOI: 10.1145/1753326.1753706. (Visited on 03/30/2024).

[11] Ruven Brooks. "Towards a Theory of the Comprehension of Computer Programs". In: *International Journal of Man-Machine Studies* 18.6 (June 1983), pp. 543–554. ISSN: 0020-7373. DOI: 10.1016/s0020-7373(83)80031-5. (Visited on 03/28/2024).

[12] José Cambronero et al. "FlashFill++: Scaling Programming by Example by Cutting to the Chase". In: *Proc. ACM Program. Lang.* 7.POPL (Jan. 2023), 33:952–33:981. DOI: 10.1145/3571226. (Visited on 09/12/2024).

[13] Federico Cassano et al. "MultiPL-E: A Scalable and Polyglot Approach to Benchmarking Neural Code Generation". In: *IEEE Transactions on Software Engineering* 49.7 (July 2023), pp. 3675–3691. ISSN: 1939-3520. DOI: 10.1109/TSE.2023.3267446. (Visited on 09/12/2024).

[14] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. "Rousillon: Scraping Distributed Hierarchical Web Data". In: *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. Uist '18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 963–975. ISBN: 978-1-4503-5948-1. DOI: 10.1145/3242587.3242661. (Visited on 03/31/2022).

[15] John Chen et al. "Learning Agent-based Modeling with LLM Companions: Experiences of Novices and Experts Using ChatGPT & NetLogo Chat". In: *Proceedings of the CHI Conference on Human Factors in Computing Systems*. Chi '24. Honolulu,

HI, USA: Association for Computing Machinery, 2024. ISBN: 9798400703300. DOI: 10.1145/3613904.3642377. URL: https://doi.org/10.1145/3613904.3642377.

[16] Heetae Cho and Seonah Lee. "Java2Pseudo: Java to Pseudo Code Translator". In: *CEUR Workshop Proceedings*. Iseapsec '22'. 2022. URL: https://www.semanticscholar.org/paper/Java2Pseudo%3A-Java-to-Pseudo-Code-Translator-Cho-Lee/ce1774ad621bc5127e7a9e026761b86fec0a08e6 (visited on 09/08/2024).

[17] T. A. Corbi. "Program understanding: Challenge for the 1990s". In: *IBM Systems Journal* 28.2 (1989). Conference Name: IBM Systems Journal, pp. 294–306. ISSN: 0018-8670. DOI: 10.1147/sj.282.0294. URL: https://ieeexplore.ieee.org/document/5387570 (visited on 09/11/2024).

[18] Will Crichton. *Human-Centric Program Synthesis*. en. Sept. 2019. URL: https://arxiv.org/abs/1909.12281v1 (visited on 09/09/2024).

[19] Brian de Alwis, Gail C. Murphy, and Martin P. Robillard. "A Comparative Study of Three Program Exploration Tools". In: *15th IEEE International Conference on Program Comprehension (ICPC '07)*. June 2007, pp. 103–112. DOI: 10.1109/icpc.2007.6. (Visited on 03/30/2024).

[20] Robert DeLine and Kael Rowan. "Code Canvas: Zooming towards Better Development Environments". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. Icse '10. New York, NY, USA: Association for Computing Machinery, May 2010, pp. 207–210. ISBN: 978-1-60558-719-6. DOI: 10.1145/1810295.1810331. (Visited on 03/30/2024).

[21] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. "Critical slicing for software fault localization". In: *Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*. Issta '96. New York, NY, USA: Association for Computing Machinery, May 1996, pp. 121–134. ISBN: 978-0-89791-787-2. DOI: 10.1145/229000.226310. URL: https://dl.acm.org/doi/10.1145/229000.226310 (visited on 09/11/2024).

[22] Pierre Dragicevic. "Fair Statistical Communication in HCI". In: *Modern Statistical Methods for HCI*. Ed. by Judy Robertson and Maurits Kaptein. Cham: Springer International Publishing, 2016, pp. 291–330. ISBN: 978-3-319-26633-6. DOI: 10.1007/978-3-319-26633-6_13. (Visited on 09/10/2024).

[23] Ian Drosos et al. "Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists". en. In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. Honolulu HI USA: Acm, Apr. 2020, pp. 1–12. ISBN: 978-1-4503-6708-0. DOI: 10.1145/3313831.3376442. URL: https://dl.acm.org/doi/10.1145/3313831.3376442 (visited on 09/10/2024).

[24] Bradley Efron. "Better Bootstrap Confidence Intervals". In: *Journal of the American Statistical Association* 82.397 (Mar. 1987), pp. 171–185. ISSN: 0162-1459. DOI: 10.1080/01621459.1987.10478410. (Visited on 09/10/2024).

[25] Janet Feigenspan et al. "Measuring programming experience". en. In: *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. Passau, Germany: Ieee, June 2012, pp. 73–82. ISBN: 978-1-4673-1216-5 978-1-4673-1213-4 978-1-4673-1215-8. DOI: 10.1109/icpc.2012.6240511. URL: http://ieeexplore.ieee.org/document/6240511/ (visited on 09/09/2024).

[26] Molly Q Feldman and Carolyn Jane Anderson. "Non-Expert Programmers in the Generative AI Future". In: *Proceedings of the 3rd Annual Meeting of the Symposium on Human-Computer Interaction for Work*. Chiwork '24. Newcastle upon Tyne, United Kingdom: Association for Computing Machinery, 2024. ISBN: 9798400710179. DOI: 10.1145/3663384.3663393. URL: https://doi.org/10.1145/3663384.3663393.

[27] Vikki Fix, Susan Wiedenbeck, and Jean Scholtz. "Mental Representations of Programs by Novices and Experts". In: *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*. Chi '93. New York, NY, USA: Association for Computing Machinery, May 1993, pp. 74–79. ISBN: 978-0-89791-575-5. DOI: 10.1145/169059.169088. (Visited on 03/27/2024).

[28] Walaa Gad et al. "DLBT: Deep Learning-Based Transformer to Generate Pseudo-Code from Source Code". en. In: *Computers, Materials & Continua* 70.2 (2021). Publisher: Tech Science Press, pp. 3117–3132. ISSN: 1546-2218, 1546-2226. DOI: 10.32604/cmc.2022.019884. URL: https://www.techscience.com/cmc/v70n2/44664 (visited on 09/08/2024).

[29] Alex Groce, Daniel Kroening, and Flavio Lerda. "Understanding Counterexamples with explain". en. In: *Computer Aided Verification*. Ed. by Rajeev Alur and Doron A. Peled. Berlin, Heidelberg: Springer, 2004, pp. 453–456. ISBN: 978-3-540-27813-9. DOI: 10.1007/978-3-540-27813-9_35.

[30] Paul Gross and Caitlin Kelleher. "Non-Programmers Identifying Functionality in Unfamiliar Code: Strategies and Barriers". In: *Journal of Visual Languages & Computing*. Part Special Issue on Selected Papers from VL/HCC'09 21.5 (Dec. 2010), pp. 263–276. ISSN: 1045-926x. DOI: 10.1016/j.jvlc.2010.08.002. (Visited on 03/30/2024).

[31] Tibor Gyimóthy, Árpád Beszédes, and Istán Forgács. "An efficient relevant slicing method for debugging". In: *SIGSOFT Softw. Eng. Notes* 24.6 (Oct. 1999), pp. 303–321. ISSN: 0163-5948. DOI: 10.1145/318774.319248. URL: https://dl.acm.org/doi/10.1145/318774.319248 (visited on 09/11/2024).

[32] Michael Hansen, Robert L. Goldstone, and Andrew Lumsdaine. *What Makes Code Hard to Understand?* arXiv:1304.5257 [cs]. Apr. 2013. DOI: 10.48550/arXiv.1304.5257. URL: http://arxiv.org/abs/1304.5257 (visited on 09/11/2024).

[33] Dan Hao et al. "A similarity-aware approach to testing based fault localization". In: *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. Ase '05. New York, NY, USA: Association for Computing Machinery, Nov. 2005, pp. 291–294. ISBN: 978-1-58113-993-8. DOI: 10.1145/1101908.1101953. URL: https://dl.acm.org/doi/10.1145/1101908.1101953 (visited on 09/11/2024).

[34] Andrew Head et al. "Tutorons: Generating Context-Relevant, on-Demand Explanations and Demonstrations of Online Code". In: *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Oct. 2015, pp. 3–12. DOI: 10.1109/VLHCC.2015.7356972. (Visited on 09/12/2024).

[35] Arto Hellas et al. "Exploring the Responses of Large Language Models to Beginner Programmers' Help Requests". In: *Proceedings of the 2023 ACM Conference on International Computing Education Research - Volume 1*. Icer '23. Chicago, IL, USA: Association for Computing Machinery, 2023, pp. 93–105. ISBN: 9781450399760. DOI: 10.1145/3568813.3600139. URL: https://doi.org/10.1145/3568813.3600139.

[36] Emily Hill, Lori Pollock, and K. Vijay-Shanker. "Exploring the Neighborhood with Dora to Expedite Software Maintenance". In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. Ase '07. New York, NY, USA: Association for Computing Machinery, Nov. 2007, pp. 14–23. ISBN: 978-1-59593-882-4. DOI: 10.1145/1321631.1321637. (Visited on 03/30/2024).

[37] Jingmei Hu et al. "Assuage: Assembly Synthesis Using A Guided Exploration". In: *The 34th Annual ACM Symposium on User Interface Software and Technology*. Uist '21. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 134–148. ISBN: 978-1-4503-8635-7. DOI: 10.1145/3472749.3474740. (Visited on 03/26/2024).

[38] Doug Janzen and Kris De Volder. "Navigating and Querying Code without Getting Lost". In: *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*. Aosd '03. New York, NY, USA: Association for Computing Machinery, Mar. 2003, pp. 178–187. ISBN: 978-1-58113-660-9. DOI: 10.1145/643603.643622. (Visited on 03/30/2024).

[39] James A. Jones, Mary Jean Harrold, and John Stasko. "Visualization of Test Information to Assist Fault Localization". In: *Proceedings of the 24th International Conference on Software Engineering*. Icse '02. New York, NY, USA: Association for Computing Machinery, May 2002, pp. 467–477. ISBN: 978-1-58113-472-8. DOI: 10.1145/581339.581397. (Visited on 03/30/2024).

[40] Breanna Jury et al. "Evaluating LLM-generated Worked Examples in an Introductory Programming Course". In: *Proceedings of the 26th Australasian Computing Education Conference*. Ace '24. New York, NY, USA: Association for Computing Machinery, Jan. 2024, pp. 77–86. ISBN: 9798400716195. DOI: 10.1145/3636243.3636252. URL: https://dl.acm.org/doi/10.1145/3636243.3636252 (visited on 09/08/2024).

[41] Sean Kandel et al. "Wrangler: interactive visual specification of data transformation scripts". en. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Vancouver BC Canada: Acm, May 2011, pp. 3363–3372. ISBN: 978-1-4503-0228-9. DOI: 10.1145/1978942.1979444. URL: https://dl.acm.org/doi/10.1145/1978942.1979444 (visited on 09/10/2024).

[42] Yvonne Kao, Bryan Matlen, and David Weintrop. "From One Language to the Next: Applications of Analogical Transfer for Programming Education". In: *ACM Trans. Comput. Educ.* 22.4 (Nov. 2022), 42:1–42:21. DOI: 10.1145/3487051. (Visited on 09/12/2024).

[43] Holger M. Kienle and Hausi A. Müller. "Rigi—An Environment for Software Reverse Engineering, Exploration, Visualization, and Redocumentation". In: *Science of Computer Programming*. Experimental Software and Toolkits (EST 3): A Special Issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT 2008) 75.4 (Apr. 2010), pp. 247–263. ISSN: 0167-6423. DOI: 10.1016/j.scico.2009.10.007. (Visited on 03/30/2024).

[44] Amy J. Ko and Brad A. Myers. "Extracting and Answering Why and Why Not Questions about Java Program Output". In: *ACM Transactions on Software Engineering and Methodology* 20.2 (Sept. 2010), 4:1–4:36. ISSN: 1049-331x. DOI: 10.1145/1824760.1824761. (Visited on 03/30/2024).

[45] Amy J. Ko and Brad A. Myers. "Finding Causes of Program Output with the Java Whyline". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Chi '09. New York, NY, USA: Association for Computing Machinery, Apr. 2009, pp. 1569–1578. ISBN: 978-1-60558-246-7. DOI: 10.1145/1518701.1518942. (Visited on 03/30/2024).

[46] Amy J. Ko and Bob Uttl. "Individual Differences in Program Comprehension Strategies in Unfamiliar Programming Systems". In: *11th IEEE International Workshop on Program Comprehension, 2003.* May 2003, pp. 175–184. DOI: 10.1109/wpc.2003.1199201. (Visited on 03/27/2024).

[47] Amy J. Ko et al. "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks". In: *IEEE Transactions on Software Engineering* 32.12 (Dec. 2006), pp. 971–987. ISSN: 1939-3520. DOI: 10.1109/tse.2006.116. (Visited on 03/30/2024).

[48] Jürgen Koenemann and Scott P. Robertson. "Expert Problem Solving Strategies for Program Comprehension". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Chi '91. New York, NY, USA: Association for Computing Machinery, Mar. 1991, pp. 125–130. ISBN: 978-0-89791-383-6. DOI: 10.1145/108844.108863. (Visited on 03/27/2024).

[49] Naveen Kulkarni and Vasudeva Varma. "Supporting Comprehension of Unfamiliar Programs by Modeling Cues". In: *Software Quality Journal* 25.1 (Mar. 2017), pp. 307–340. ISSN: 1573-1367. DOI: 10.1007/s11219-015-9303-5. (Visited on 03/30/2024).

[50] Thomas D. LaToza and Brad A. Myers. "Visualizing Call Graphs". In: *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Sept. 2011, pp. 117–124. DOI: 10.1109/vlhcc.2011.6070388. (Visited on 03/30/2024).

[51] Thomas D. LaToza et al. "Program Comprehension as Fact Finding". In: *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*. Esec-fse '07. New York, NY, USA: Association for Computing Machinery, Sept. 2007, pp. 361–370. ISBN: 978-1-59593-811-4. DOI: 10.1145/1287624.1287675. (Visited on 03/30/2024).

[52] Joseph Lawrance et al. "Using Information Scent to Model the Dynamic Foraging Behavior of Programmers in Maintenance Tasks". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Chi '08. New York, NY, USA: Association for Computing Machinery, Apr. 2008, pp. 1323–1332. ISBN: 978-1-60558-011-1. DOI: 10.1145/1357054.1357261. (Visited on 03/30/2024).

[53] Vu Le et al. *Interactive Program Synthesis*. Mar. 2017. DOI: 10.48550/arXiv.1703.03539. arXiv: 1703.03539 [cs]. (Visited on 03/26/2024).

[54] Mina Lee, Sunbeom So, and Hakjoo Oh. "Synthesizing regular expressions from examples for introductory automata assignments". In: *SIGPLAN Not.* 52.3 (Oct. 2016), pp. 70–80. ISSN: 0362-1340. DOI: 10.1145/3093335.2993244. URL: https://doi.org/10.1145/3093335.2993244.

[55] Juho Leinonen et al. "Comparing Code Explanations Created by Students and Large Language Models". en. In: *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. Turku Finland: Acm, June 2023, pp. 124–130. ISBN: 9798400701382. DOI: 10.1145/3587102.3588785. URL: https://dl.acm.org/doi/10.1145/3587102.3588785 (visited on 09/09/2024).

[56] Timothy C. Lethbridge and Francisco Herrera. "Assessing the Usefulness of the TkSee Software Exploration Tool". In: *Advances in Software Engineering: Comprehension, Evaluation, and Evolution*. Ed. by Hakan Erdogmus and Oryal Tanir. New York, NY: Springer, 2002, pp. 73–93. ISBN: 978-0-387-21599-0. DOI: 10.1007/978-0-387-21599-0_4. (Visited on 03/30/2024).

[57] Stanley Letovsky. "Cognitive Processes in Program Comprehension". In: *Journal of Systems and Software* 7.4 (Dec. 1987), pp. 325–339. ISSN: 0164-1212. DOI: 10.1016/0164-1212(87)90032-x. (Visited on 03/28/2024).

[58] Xiangyu Li and Alessandro Orso. "More Accurate Dynamic Slicing for Better Supporting Software Debugging". In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. ISSN: 2159-4848. Oct. 2020, pp. 28–38. DOI: 10.1109/ICST46399.2020.00014. URL: https://ieeexplore.ieee.org/abstract/document/9159078 (visited on 09/12/2024).

[59] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. "A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Icse '24. Lisbon, Portugal: Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: 10.1145/3597503.3608128. URL: https://doi.org/10.1145/3597503.3608128.

[60] Ben Liblit et al. "Scalable statistical bug isolation". In: *SIGPLAN Not.* 40.6 (June 2005), pp. 15–26. ISSN: 0362-1340. DOI: 10.1145/1064978.1065014. URL: https://dl.acm.org/doi/10.1145/1064978.1065014 (visited on 09/11/2024).

[61] Michael Xieyang Liu et al. ""What It Wants Me To Say": Bridging the Abstraction Gap Between End-User Programmers and Code-Generating Large Language Models". In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. Chi '23. Hamburg, Germany: Association for Computing Machinery, 2023. ISBN: 9781450394215. DOI: 10.1145/3544548.3580817. URL: https://doi.org/10.1145/3544548.3580817.

[62] Kuang-Chen Lu et al. "What Happens When Students Switch (Functional) Languages (Experience Report)". In: *Proceedings of the ACM on Programming Languages* 7.Icfp (Aug. 2023), 215:796–215:812. DOI: 10.1145/3607857. (Visited on 03/30/2024).

[63] Walid Maalej et al. "On the Comprehension of Program Comprehension". In: *ACM Transactions on Software Engineering and Methodology* 23.4 (Sept. 2014), 31:1–31:37. ISSN: 1049-331x. DOI: 10.1145/2622669. (Visited on 03/27/2024).

[64] David MacIver, Zac Hatfield-Dodds, and Many Contributors. "Hypothesis: A new approach to property-based testing". In: *Journal of Open Source Software* 4.43 (Nov. 21, 2019), p. 1891. ISSN: 2475-9066. DOI: 10.21105/joss.01891. URL: http://dx.doi.org/10.21105/joss.01891.

[65] Stephen MacNeil et al. "Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book". In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. Sigcse 2023. New York, NY, USA: Association for Computing Machinery, Mar. 2023, pp. 931–937. ISBN: 978-1-4503-9431-4. DOI: 10.1145/3545945.3569785. URL: https://dl.acm.org/doi/10.1145/3545945.3569785 (visited on 09/08/2024).

[66] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[67] Mikaël Mayer et al. "User Interaction Models for Disambiguation in Programming by Example". In: *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. Uist '15. New York, NY, USA: Association for Computing Machinery, Nov. 2015, pp. 291–301. ISBN: 978-1-4503-3779-3. DOI: 10.1145/2807442.2807459. (Visited on 03/26/2024).

[68] Roberto Minelli, Andrea Mocci, and Michele Lanza. "I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time". In: *2015 IEEE 23rd International Conference on Program Comprehension*. Issn: 1092-8138. May 2015, pp. 25–35. DOI: 10.1109/icpc.2015.12. URL: https://ieeexplore.ieee.org/abstract/document/7181430 (visited on 09/11/2024).

[69] F. Modugno and B. A. Myers. "Visual Programming in a Visual Shell—A Unified Approach". In: *Journal of Visual Languages & Computing* 8.5 (Dec. 1997), pp. 491–522. ISSN: 1045-926x. DOI: 10.1006/jvlc.1997.0049. (Visited on 03/27/2024).

[70] Konstantinos Moratis et al. "Write me this Code: An Analysis of ChatGPT Quality for Producing Source Code". In: *Proceedings of the 21st International Conference on Mining Software Repositories*. Msr '24. Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 147–151. ISBN: 9798400705878. DOI: 10.1145/3643991.3645070. URL: https://doi.org/10.1145/3643991.3645070.

[71] Hausi A. Müller et al. "A Reverse-Engineering Approach to Subsystem Structure Identification". In: *Journal of Software Maintenance: Research and Practice* 5.4 (1993), pp. 181–204. ISSN: 1096-908x. DOI: 10.1002/smr.4360050402. (Visited on 03/30/2024).

[72] Daye Nam et al. "Using an LLM to Help With Code Understanding". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Icse '24. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 1–13. ISBN: 9798400702174. DOI: 10.1145/3597503.3639187. URL: https://dl.acm.org/doi/10.1145/3597503.3639187 (visited on 09/08/2024).

[73] Amirmohammad Nazari et al. "Explainable Program Synthesis by Localizing Specifications". In: *Proceedings of the ACM on Programming Languages* 7.Oopsla2 (Oct. 2023), 298:2171–298:2195. DOI: 10.1145/3622874. (Visited on 03/30/2024).

[74] Sydney Nguyen et al. "How Beginning Programmers and Code LLMs (Mis)read Each Other". In: *Proceedings of the CHI Conference on Human Factors in Computing Systems*. Chi '24. Honolulu, HI, USA: Association for Computing Machinery, 2024. ISBN: 9798400703300. DOI: 10.1145/3613904.3642706. URL: https://doi.org/10.1145/3613904.3642706.

[75] Casey O'Brien, Max Goldman, and Robert C. Miller. "Java Tutor: Bootstrapping with Python to Learn Java". In: *Proceedings of the First ACM Conference on Learning @ Scale Conference*. L@S '14. New York, NY, USA: Association for Computing Machinery, Mar. 2014, pp. 185–186. ISBN: 978-1-4503-2669-8. DOI: 10.1145/2556325.2567873. (Visited on 12/06/2024).

[76] Yusuke Oda et al. "Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation". In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Nov. 2015, pp. 574–584. DOI: 10.1109/ase.2015.36. URL: https://ieeexplore.ieee.org/document/7372045/?arnumber=7372045 (visited on 09/08/2024).

[77] OpenAI et al. *GPT-4 Technical Report*. arXiv:2303.08774 [cs]. Mar. 2024. DOI: 10.48550/arXiv.2303.08774. URL: http://arxiv.org/abs/2303.08774 (visited on 09/11/2024).

[78] Mike Papadakis and Yves Le Traon. "Metallaxis-FL: mutation-based fault localization". en. In: *Software Testing, Verification and Reliability* 25.5-7 (2015). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1509, pp. 605–628. ISSN: 1099-1689. DOI: 10.1002/stvr.1509. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1509 (visited on 09/12/2024).

[79] Chris Parnin and Alessandro Orso. "Are Automated Debugging Techniques Actually Helping Programmers?" In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. Issta '11. New York, NY, USA: Association for Computing Machinery, July 2011, pp. 199–209. ISBN: 978-1-4503-0562-4. DOI: 10.1145/2001420.2001445. (Visited on 03/30/2024).

[80] Hila Peleg, Shachar Itzhaky, and Sharon Shoham. "Abstraction-Based Interaction Model for Synthesis". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Isil Dillig and Jens Palsberg. Cham: Springer International Publishing, 2018, pp. 382–405. ISBN: 978-3-319-73721-8. DOI: 10.1007/978-3-319-73721-8_18.

[81] Hila Peleg et al. "Programming with a Read-Eval-Synth Loop". In: *Proceedings of the ACM on Programming Languages* 4.Oopsla (Nov. 2020), 159:1–159:30. DOI: 10.1145/3428227. (Visited on 04/05/2024).

[82] Nancy Pennington. "Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs". In: *Cognitive Psychology* 19.3 (July 1987), pp. 295–341. ISSN: 0010-0285. DOI: 10.1016/0010-0285(87)90007-7. (Visited on 09/07/2020).

[83] James Prather et al. ""It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers". In: *ACM Trans. Comput.-Hum. Interact.* 31.1 (Nov. 2023). ISSN: 1073-0516. DOI: 10.1145/3617367. URL: https://doi.org/10.1145/3617367.

[84] Thomas Reps et al. "The use of program profiling for software maintenance with applications to the year 2000 problem". In: *Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*. Esec '97/fse-5. Berlin, Heidelberg: Springer-Verlag, Nov. 1997, pp. 432–449. ISBN: 978-3-540-63531-4. DOI: 10.1145/267895.267925. URL: https://dl.acm.org/doi/10.1145/267895.267925 (visited on 09/11/2024).

[85] Martin P. Robillard. “Automatic Generation of Suggestions for Program Investigation”. In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Esec/fse-13. New York, NY, USA: Association for Computing Machinery, Sept. 2005, pp. 11–20. ISBN: 978-1-59593-014-9. DOI: 10.1145/1081706.1081711. (Visited on 03/30/2024).

[86] Tobias Roehm et al. “How Do Professional Developers Comprehend Software?” In: *Proceedings of the 34th International Conference on Software Engineering*. Icse ’12. Zurich, Switzerland: IEEE Press, June 2012, pp. 255–265. ISBN: 978-1-4673-1067-3. (Visited on 04/16/2021).

[87] Jean Scholtz and Susan Wiedenbeck. “Learning New Programming Languages: An Analysis of the Process and Problems Encountered”. In: *Behaviour & Information Technology* 11.4 (July 1992), pp. 199–215. ISSN: 0144-929x. DOI: 10.1080/01449299208924339. (Visited on 03/30/2024).

[88] Jean Scholtz and Susan Wiedenbeck. “Learning Second and Subsequent Programming Languages: A Problem of Transfer”. In: *International Journal of Human–Computer Interaction* 2.1 (Jan. 1990), pp. 51–72. ISSN: 1044-7318. DOI: 10.1080/10447319009525970. (Visited on 03/30/2024).

[89] Jean Scholtz and Susan Wiedenbeck. “Using Unfamiliar Programming Languages: The Effects on Expertise”. In: *Interacting with Computers* 5.1 (Mar. 1993), pp. 13–30. ISSN: 0953-5438. DOI: 10.1016/0953-5438(93)90023-m. (Visited on 03/27/2024).

[90] Teresa M. Shaft and Iris Vessey. “Research Report—The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension”. In: *Information Systems Research* (Sept. 1995). DOI: 10.1287/isre.6.3.286. (Visited on 03/27/2024).

[91] Ben Shneiderman. “Exploratory Experiments in Programmer Behavior”. In: *International Journal of Computer & Information Sciences* 5.2 (June 1976), pp. 123–143. ISSN: 1573-7640. DOI: 10.1007/bf00975629. (Visited on 03/28/2024).

[92] Nischal Shrestha, Titus Barik, and Chris Parnin. “It’s Like Python But: Towards Supporting Transfer of Programming Language Knowledge”. In: *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Oct. 2018, pp. 177–185. DOI: 10.1109/VLHCC.2018.8506508. (Visited on 09/12/2024).

[93] Nischal Shrestha et al. “Here We Go Again: Why Is It Difficult for Developers to Learn Another Programming Language?” In: *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. Oct. 2020, pp. 691–701. (Visited on 09/12/2024).

[94] Janet Siegmund. "Program Comprehension: Past, Present, and Future". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 5. Mar. 2016, pp. 13–20. DOI: `10.1109/saner.2016.35`. (Visited on 03/27/2024).

[95] Vineet Sinha, David Karger, and Rob Miller. "Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases". In: *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*. Eclipse '05. New York, NY, USA: Association for Computing Machinery, Oct. 2005, pp. 21–25. ISBN: 978-1-59593-342-3. DOI: `10.1145/1117696.1117701`. (Visited on 03/30/2024).

[96] Elliot Soloway and Kate Ehrlich. "Empirical Studies of Programming Knowledge". In: *IEEE Transactions on Software Engineering* Se-10.5 (Sept. 1984), pp. 595–609. ISSN: 1939-3520. DOI: `10.1109/tse.1984.5010283`. (Visited on 03/27/2024).

[97] Tarja Systä, Kai Koskimies, and Hausi Müller. "Shimba—an Environment for Reverse Engineering Java Software Systems". In: *Software: Practice and Experience* 31.4 (2001), pp. 371–394. ISSN: 1097-024x. DOI: `10.1002/spe.386`. (Visited on 03/30/2024).

[98] Ethel Tshukudu. "Understanding Conceptual Transfer in Students Learning a New Programming Language". PhD thesis. University of Glasgow, 2022. DOI: `10.5525/gla.thesis.82984`. (Visited on 03/30/2024).

[99] Ethel Tshukudu and Quintin Cutts. "Semantic Transfer in Programming Languages: Exploratory Study of Relative Novices". In: *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. ITiCSE '20. New York, NY, USA: Association for Computing Machinery, June 2020, pp. 307–313. ISBN: 978-1-4503-6874-2. DOI: `10.1145/3341525.3387406`. (Visited on 03/30/2024).

[100] Ethel Tshukudu and Quintin Cutts. "Understanding Conceptual Transfer for Students Learning New Programming Languages". In: *Proceedings of the 2020 ACM Conference on International Computing Education Research*. Icer '20. New York, NY, USA: Association for Computing Machinery, Aug. 2020, pp. 227–237. ISBN: 978-1-4503-7092-9. DOI: `10.1145/3372782.3406270`. (Visited on 03/30/2024).

[101] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. "Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models". In: *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*. Chi Ea '22. New Orleans, LA, USA: Association for Computing Machinery, 2022. ISBN: 9781450391566. DOI: `10.1145/3491101.3519665`. URL: `https://doi.org/10.1145/3491101.3519665`.

[102] Priyan Vaithilingam et al. "Towards More Effective AI-Assisted Programming: A Systematic Design Exploration to Improve Visual Studio IntelliCode's User Experience". In: *Proceedings of the 45th International Conference on Software Engineering: Software Engineering in Practice*. Icse-seip '23. Melbourne, Australia: IEEE Press, 2023,

pp. 185–195. ISBN: 9798350300376. DOI: 10.1109/icse-seip58684.2023.00022. URL: https://doi.org/10.1109/ICSE-SEIP58684.2023.00022.

[103] Anneliese von Mayrhauser and A. Marie Vans. "Program Comprehension During Software Maintenance and Evolution". In: *Computer* 28.8 (Aug. 1995), pp. 44–55. ISSN: 0018-9162. DOI: 10.1109/2.402076. (Visited on 04/15/2021).

[104] Karen P. Walker and Stephen R. Schach. "Obstacles to Learning a Second Programming Language: An Empirical Study". In: *Computer Science Education* 7.1 (Jan. 1996), pp. 1–20. ISSN: 0899-3408. DOI: 10.1080/0899340960070101. (Visited on 03/30/2024).

[105] Mark Weiser. "Program Slicing". In: *IEEE Transactions on Software Engineering* Se-10.4 (July 1984). Conference Name: IEEE Transactions on Software Engineering, pp. 352–357. ISSN: 1939-3520. DOI: 10.1109/tse.1984.5010248. URL: https://ieeexplore.ieee.org/document/5010248/?arnumber=5010248 (visited on 09/11/2024).

[106] Susan Wiedenbeck. "Beacons in Computer Program Comprehension". In: *International Journal of Man-Machine Studies* 25.6 (Dec. 1986), pp. 697–709. ISSN: 0020-7373. DOI: 10.1016/s0020-7373(86)80083-9. (Visited on 03/27/2024).

[107] Xin Xia et al. "Measuring Program Comprehension: A Large-Scale Field Study with Professionals". In: *IEEE Transactions on Software Engineering* 44.10 (Oct. 2018). Conference Name: IEEE Transactions on Software Engineering, pp. 951–976. ISSN: 1939-3520. DOI: 10.1109/tse.2017.2734091. URL: https://ieeexplore.ieee.org/document/7997917 (visited on 09/11/2024).

[108] Xiangyu Zhang, R. Gupta, and Youtao Zhang. "Precise dynamic slicing algorithms". en. In: *25th International Conference on Software Engineering, 2003. Proceedings.* Portland, OR, USA: Ieee, 2003, pp. 319–329. ISBN: 978-0-7695-1877-0. DOI: 10.1109/icse.2003.1201211. URL: http://ieeexplore.ieee.org/document/1201211/ (visited on 09/11/2024).

[109] Hao Yan, Thomas D. Latoza, and Ziyu Yao. *IntelliExplain: Enhancing Interactive Code Generation through Natural Language Explanations for Non-Professional Programmers.* en. arXiv:2405.10250 [cs]. May 2024. URL: http://arxiv.org/abs/2405.10250 (visited on 09/09/2024).

[110] Litao Yan et al. "Ivie: Lightweight Anchored Explanations of Just-Generated Code". In: *Proceedings of the CHI Conference on Human Factors in Computing Systems.* CHI '24. New York, NY, USA: Association for Computing Machinery, May 2024, pp. 1–15. ISBN: 9798400703300. DOI: 10.1145/3613904.3642239. (Visited on 09/12/2024).

[111] YoungSeok Yoon and Brad A. Myers. "Semantic Zooming of Code Change History". In: *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).* Oct. 2015, pp. 95–99. DOI: 10.1109/vlhcc.2015.7357203. (Visited on 03/30/2024).

[112] YoungSeok Yoon, Brad A. Myers, and Sebon Koo. "Visualization of Fine-Grained Code Change History". In: *2013 IEEE Symposium on Visual Languages and Human Centric Computing*. Sept. 2013, pp. 119–126. DOI: 10.1109/vlhcc.2013.6645254. (Visited on 03/30/2024).

[113] Tianyi Zhang et al. "Interactive Program Synthesis by Augmented Examples". In: *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. Uist '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 627–648. ISBN: 978-1-4503-7514-6. DOI: 10.1145/3379337.3415900. (Visited on 03/26/2024).

[114] Tianyi Zhang et al. "Interpretable Program Synthesis". In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. Chi '21. New York, NY, USA: Association for Computing Machinery, May 2021, pp. 1–16. ISBN: 978-1-4503-8096-6. DOI: 10.1145/3411764.3445646. (Visited on 03/26/2024).

[115] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. "Pruning dynamic slices with confidence". In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Pldi '06. Ottawa, Ontario, Canada: Association for Computing Machinery, 2006, pp. 169–180. ISBN: 1595933204. DOI: 10.1145/1133981.1134002. URL: https://doi.org/10.1145/1133981.1134002.

[116] Zhanhui Zhou et al. "INTENT: Interactive Tensor Transformation Synthesis". In: *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. Uist '22. New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 1–16. ISBN: 978-1-4503-9320-1. DOI: 10.1145/3526113.3545653. (Visited on 03/26/2024).