Exploring the Learnability of Program Synthesizers by Novice Programmers

Dhanya Jayagopal*
dhanyajayagopal@berkeley.edu
University of California, Berkeley
Berkeley, USA

Justin Lubin*
justinlubin@berkeley.edu
University of California, Berkeley
Berkeley, USA

Sarah E. Chasins
schasins@cs.berkeley.edu
University of California, Berkeley
Berkeley, USA

ABSTRACT

Modern program synthesizers are increasingly delivering on their promise of lightening the burden of programming by automatically generating code, but little research has addressed how we can make such systems learnable to all. In this work, we ask: What aspects of program synthesizers contribute to and detract from their learnability by novice programmers? We conducted a thematic analysis of 22 observations of novice programmers, during which novices worked with existing program synthesizers, then participated in semi-structured interviews. Our findings shed light on how their specific points in the synthesizer design space affect these tools' learnability by novice programmers, including the type of specification the synthesizer requires, the method of invoking synthesis and receiving feedback, and the size of the specification. We also describe common misconceptions about what constitutes meaningful progress and useful specifications for the synthesizers, as well as participants' common behaviors and strategies for using these tools. From this analysis, we offer a set of design opportunities to inform the design of future program synthesizers that strive to be learnable by novice programmers. This work serves as a first step toward understanding how we can make program synthesizers more learnable by novices, which opens up the possibility of using program synthesizers in educational settings as well as developer tooling oriented toward novice programmers.

KEYWORDS

learnability, program synthesis, novice programmers, qualitative, thematic analysis

ACM Reference Format:
Dhanya Jayagopal, Justin Lubin, and Sarah E. Chasins. 2022. Exploring the Learnability of Program Synthesizers by Novice Programmers. In *The 35th Annual ACM Symposium on User Interface Software and Technology (UIST '22), October 29-November 2, 2022, Bend, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3526113.3545659

*Authors contributed equally.

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

ACM ISBN 978-1-4503-9320-1/22/10.
https://doi.org/10.1145/3526113.3545659

1 INTRODUCTION

The promise of *program synthesis* is to lighten the burden of programming by automatically generating code that satisfies a user-provided specification. However, little work has studied how novice programmers learn and use synthesis tools. Our work draws on observations of early-stage programmers and identifies synthesizer design dimensions that affect synthesizer learnability. The end goal is to inform design guidelines so that the community can make synthesizers more approachable and ultimately boost their impact on a broader class of users.

We observed 22 novice programmers using five existing program synthesis tools (Blue-Pencil [48], Copilot [22], Flash Fill [23], Regae [76], and SnipPy [15]) and followed each session with a semi-structured interview.

We identified a number of influential design dimensions. One such dimension is that synthesizers can (i) require users to engage in a separate synthesis-specific specification mode or (ii) derive a specification as a byproduct of normal non-synthesis tool use. Another important dimension is whether users are in charge of triggering synthesis runs and the display of synthesis outputs or whether the tool is in charge. The size of the specification also matters, but seemingly not as much other dimensions—a surprising finding in light of design guidelines and goals from the synthesis literature, which emphasize specification size [8, 23, 37, 43, 56].

We also identified important user knowledge gaps and common strategies. Novices struggle with plan composition during synthesis in much the same way as during manual coding. Novice programmers struggle to figure out what kinds of specifications work well for a given synthesis tool. For synthesis tools embedded in familiar environments, novice programmers may also borrow behaviors from their pre-synthesizer use. Finally, novice programmers may engage more deeply with synthesis-written programs relative to teacher-written programs provided as exercise solutions.

Based on our findings, we provide a set of design opportunities to inform the design of future program synthesizers that aim to be learnable by novices.

No element of this paper is intended as an evaluation of the tools used in the study. In particular, we note that the tools we used in this study are not explicitly designed for learnability by novice programmers. Rather, we chose a stable of tools that exhibit different design choices for their synthesis algorithms, interfaces, and user interaction models as a means to uncover patterns in how these design choices affect users.

Learnability. A tool's *learnability* can refer either to its (i) first-encounter usability or (ii) how long its users take to gain proficiency. In this paper, we are exclusively concerned with the first definition.

Contributions. This paper presents the following contributions:

- An **observational study of novice programmers using program synthesis tools** for the first time.
- An analysis that identifies **key synthesizer design dimensions that affect learnability**; a preliminary analysis of particular design decisions that improve or reduce synthesizer learnability.
- A set of **opportunities for synthesizer designers**, to guide designers towards choices that make synthesis tools learnable by novice programmers.

2 BACKGROUND: PROGRAM SYNTHESIS

Program synthesis is the automatic generation of code based on some *specification* that communicates user intent. This expression of user intent can take many forms, including, for example, a logical formula, input-output examples, or task demonstrations. The term specification is typically used to mean a *formal specification*—an objective, machine-checkable predicate that gives yes or no answers about whether a given program meets the specification. For instance, we can automatically check whether a given program meets the logical specification $output = 2 * input$ or the input-output example specification $[3 \rightarrow 6, 5 \rightarrow 10]$. For this paper, we use a broader definition of specification that includes any expression of user intent to a synthesizer. In particular, the input to Copilot is a large portion of the content of an in-progress file in an IDE. Although this input does not offer a way of automatically and non-subjectively checking whether a synthesized program "meets" this specification, we will still use the term specification.

3 RELATED WORK

3.1 Learnability & Usability of Synthesis Tools

Recent work in the programming languages community and a variety of other subfields has produced tremendous advances in program synthesis technology [3, 16, 23–25, 33, 35, 45, 53, 56, 61, 63]. Synthesis approaches are now sufficiently mature that we are seeing them adopted within HCI [8, 13, 26, 31, 40, 47, 66, 70] and integrated into mainstream products [19, 50, 60, 73]. Although the early wave of program synthesis works from the programming languages community largely did not assess synthesizers' learnability or usability, the more recent wave of synthesis-augmented interfaces within the HCI community has started to offer answers about how humans interact with synthesizers.

Existing user studies of program synthesis tools fall into a few categories. Many works compare a novel synthesis tool against a manual programming condition [8, 15, 20, 30, 70]. These studies typically find that participants are faster—sometimes much faster—with synthesis than with manual coding, that they make fewer errors, that they are more likely to complete tasks, or some subset of these. Others focus exclusively on one novel synthesizer, with no control condition, and assess whether users can complete tasks with the tool at all [14, 41, 68]. These studies typically find that participants can complete tasks with the synthesizer under test.

Comparison of synthesizer variants. More recently we have begun to see user studies that test a novel synthesizer against a tweaked versions of the same synthesizer. Such studies begin to answer a question similar to ours: what *aspects* of program synthesizers affect their usability? To date, we know of two such studies.

The first of these studies [55] compares how successful participants were using (i) a traditional programming-by-example synthesizer in which users provide input-output examples versus (ii) a variant of the synthesizer using a paradigm the authors dub "programming not only by example" that also allows users to select subexpressions of a candidate program to either keep or reject. The authors found that the subexpression selection mechanism was faster to use than standard input-output examples and strongly preferred to examples overall except for with expert users.

The second of these studies (which investigates one of the tools we use in our study, Regae) [76] compares how successful participants were using (i) the second condition of the above study versus (ii) a further enhancement on the programming-by-example paradigm that allows users to mark parts of their regex as either "general" or 'literal" (so-called *semantic augmentation*) as well as receive automatic example generation of additional input examples (so-called *data augmentation*). The authors found that participants were significantly faster in the second condition.

User studies of synthesizers in our study. We highlight in particular that three of the five tools in our protocol have already been studied in published user studies: Regae (as we discussed in previous subsection), SnipPy, and Blue-Pencil.

The authors of SnipPy ran their user study [15] as a controlled experiment between (i) a live programming environment with no synthesis features and (ii) the same live programming environment with SnipPy activated. Neither correctness nor speed differences were statistically significant between the two groups, but the authors did find that when the participants broke the task down in a way that the synthesizer understood, the synthesizer produced substantial and helpful parts of the overall solution.

The authors of Blue-Pencil did a field study [48] that included follow-up interviews to collect qualitative feedback of their tool in an uncontrolled environment. Their study revealed two key usability barriers: (i) low discoverability and (ii) frustration with lacking a mechanism to preview the effects of running the synthesized code before accepting or rejecting it.

Our approach. All the aforementioned studies of synthesizer usability and learnability are highly relevant to the questions we address in our work. However, in contrast to the existing foundational studies in this space—which explore either one synthesis tool at a time, one synthesis tool and one traditional programming language, or two very close variants of a single synthesizer—our work draws on observations of participants using five diverse synthesis tools. Comparing across a variety of divergent tools lets us explore a somewhat broader swath of the synthesis design space and situate our observations in the broader landscape of that design space.

3.2 Interactive Program Synthesis

Recent work in the programming languages and program synthesis community has begun to emphasize *interactive program synthesis*. Within the program synthesis community, interactive program synthesis usually refers to a synthesizer in which either (i) if the user

updates an existing specification, the synthesis algorithm works differently than if the user had provided this specification initially, or (ii) the synthesis tool can guide the user in how to augment an ambiguous specification. For instance, tools may show partially completed programs at multiple stages during synthesis and request user feedback before completing more [4], or they may show users examples of program behaviors and ask if they are correct [21]. Some recent work even presents a framework for reasoning about how user inputs can be used to reduce specification ambiguity [54]. More than the first wave of program synthesis research, this recent wave is wrestling with the role of human users in the synthesis process and recognizing synthesis as an interactive human-machine collaboration. However, the work in this space still currently emphasizes advances in algorithm design or performance over advances in usability; user studies are not currently more common in the space of interactive program synthesis techniques than in the more traditional techniques.

3.3 Learnability & Usability of Novice-Focused Programming Environments

Research on programming environments has studied novice programmers to explore the learnability of a variety of tools from Codecademy-style [10] tutorial systems with built-in programming environments [51, 62] to inquisitive IDEs that prompt users with questions about program behavior during development [27]. Because program synthesizers often display synthesized programs in a specialized or general-purpose programming environment, many of the key insights from existing work on programming environment usability can be directly applied to the programming environment elements of program synthesizers. In this study, we focus specifically on the aspects of the program synthesizers themselves that impact their learnability by novice programmers.

3.4 Learnability & Usability of AI Tools

With at least two decades of work in the space of human-AI interaction, there has been substantial work that addresses the learnability and usability of AI tools. Some works offer guidelines for improving human-AI collaboration [2, 5, 28, 29, 36, 42, 52]; others detail the usability or learnability of particular classes of AI tools [17, 44, 49, 58, 74]. We are not aware of any usability studies to date from the human-AI interaction community that investigate the usability of AI tools that produce computer code.

3.5 Think-Aloud and Observational Studies with Novices

The observational think-aloud design used in our work connects closely with prior studies [7, 39, 46, 71] that use observations of novice programmers to uncover patterns in a variety of processes, from novices' plan composition strategies to how they track program state. Although existing observational studies of novice programmers have yielded important insights in CS education [7, 39, 71] as well as language and IDE design [46], this approach has not yet been applied to synthesis-augmented programming environments.

4 PROGRAM SYNTHESIZERS

We selected a range of program synthesis tools for participants to use based on two main criteria: (i) the tool could plausibly support novice programmers, and (ii) the tool is publicly available and functional in their publicly available form. We selected five such tools based on the additional goal of exploring a diversity of program synthesis algorithms and interfaces: Blue-Pencil, Copilot, Flash Fill, Regae, and SnipPy.

Blue-Pencil [48] is a plugin for Microsoft's Visual Studio Code that aims to automate repetitive code edits, such as changing the name of a variable. GitHub Copilot [22] is an editor plugin powered by OpenAI's Codex model [9] that automatically synthesizes code on the fly based on the code and comments the user has already written. Flash Fill [23] is a feature of Microsoft Excel that uses programming-by-example to automatically populate data cells based on existing patterns in other data cells. Regae [76] is a standalone tool that implements programming-by-example synthesis for regular expressions (regexes). SnipPy [15] is a fork of Visual Studio Code that supports "small-step live programming by example" for Python; it lets users supply input-output examples via *projection boxes* [38] that dynamically show the values of variables at different points in the program. Table 1 describes the input and output of each tool, and Figure 1 shows screenshots.

5 METHOD

Participants and Recruitment. We conducted virtual study sessions with 22 participants (with identifiers P0–P21), all of whom were enrolled in a second-semester computer science course on data structures at an R1 university in the United States. We screened participants (recruited from Piazza, Facebook groups, and Slack workspaces) via a survey for little to no external programming experience beyond their first semester of computer science. We compensated participants with a 30 USD Amazon gift card.

Study Protocol. Study sessions consisted of one recorded Zoom call between the first author of this paper and the participant. We scheduled these calls for 1.5 hours, but most lasted approximately 45 minutes to 1 hour. We divided each session into two sections.

In the first section, participants narrated their thought processes aloud as they used various program synthesizers to complete tasks that we assigned. We selected tools to be shown to each participant at random such that an equal number of participants saw the five tools. The tasks that we assigned depended on the program synthesis tools that the participants used, and we detail these tasks in Table 2. In most cases, we designed the tasks to be similar to the tasks that were used as examples in the associated publications for each of the tools. We additionally provided participants with a minimal verbal explanation of the tools to get started.

In the second section of the study, the investigator asked participants in a semi-structured interview to talk about their experience with the program synthesis tools that they used, and to elaborate on certain topics that came up during their use of the tools.

Analysis. We used thematic analysis to analyze the recorded sessions of novice programmers working with program synthesizers. The first author reviewed each recorded session and tagged any relevant participant actions or narration with a low-level textual

Table 1: ***Characteristics of the tools in the study, including the input and output for the underlying synthesizer.***

Synthesizer	Input	Output	Communicating Input	Communicating Output	Communicating Failure
Blue-Pencil	Trace of user's code edit actions	Program for automating detected repetitive program edits	Automatically collects synthesizer input in the background	Light bulb icon that, when clicked, reveals options to execute the code transformations	No communication
Copilot	Arbitrary text, such as a partially-written program	Arbitrary text, ideally a program	Automatically collects synthesizer input in the background	Grayed-out text inserted at cursor, as in other text suggestion interfaces	No communication
Flash Fill	Input-output pairs of strings	String transformation program that transforms each example input into its output	User fills an empty column adjacent to an "input" column that serves as the corresponding "outputs," then clicks the Flash Fill button	Tool fills remaining cells of the output column by running synthesized string transformation	Pop-up error message
Regae	(i) Strings a regex should accept, (ii) strings a regex should not accept, (iii) literal/generalizable labels applied to substrings of accepted strings, (iv) desirable/undesirable labels for subexpressions of candidate regexes	A regex satisfying the user-provided examples	Custom standalone interface with specific slots for each type of input	Displays set of candidate regexes	Progress bar turns red and displays "Synthesis Failed."
SnipPy	A Python program with a hole (??) and a set of input-output examples	A Python expression satisfying the user-provided examples	(i) User types hole (??) in program text, (ii) a box displays values of variables in scope as inputs for examples, (iii) user provides output for each input, (iv) user presses enter key	Synthesizer replaces hole with synthesized expression	Synthesizer replaces hole with "Synthesis Failed" comment

summary. After accumulating many such summary tags, the first author grouped these tags into loose categorizations (the beginnings of the themes in this paper), and re-watched existing footage to annotate any missing occurrences of these categorizations. This inductive, open coding process continued, with the themes becoming more concrete and robust over time via discussion with the other authors on this paper until arriving at the set of themes that form the subsection headers in Section 6. Finally, all authors collaborated to explore and refine these themes even further while the first author reviewed the recorded session for any additional evidence and nuance for the final themes.

6 RESULTS

Our analysis of novice programmers using program synthesizers revealed seven main themes:

(§6.1) Synthesizers can either (i) require users to engage in a separate process to produce a specification or (ii) derive a specification as a byproduct of normal non-synthesis tool use; participants struggled more with the former.
(§6.2) Synthesizers give users varying amounts of control over *when* to trigger synthesis-related activities; overall, synthesizers that exposed more control were more confusing to participants, although this choice had drawbacks too.
(§6.3) Participants sometimes struggled with synthesizers that required large specifications, but this pattern was not as pronounced as the others we identified.
(§6.4) Participants did not always know what constitutes meaningful progress in the context of a given synthesizer.
(§6.5) Participants did not always know what constitutes a good specification for a given synthesizer.
(§6.6) In familiar programming environments, participants often borrowed behavior from their pre-synthesizer use of the environment.
(§6.7) Participants engaged with synthesis-written programs in ways that they might not engage with teacher-written programs provided as exercise solutions.

Based on these results, we provide a set of design opportunities that that we place in outlined, yellow boxes throughout this section; each design opportunity appears immediately after the relevant observations.

6.1 Voluntary and Incidental Specifications

Synthesizers vary in how they have the user communicate a specification, and we observed that participants faced more learnability barriers when they had to use an entirely new process to craft the specification separately from the primary artifact they were

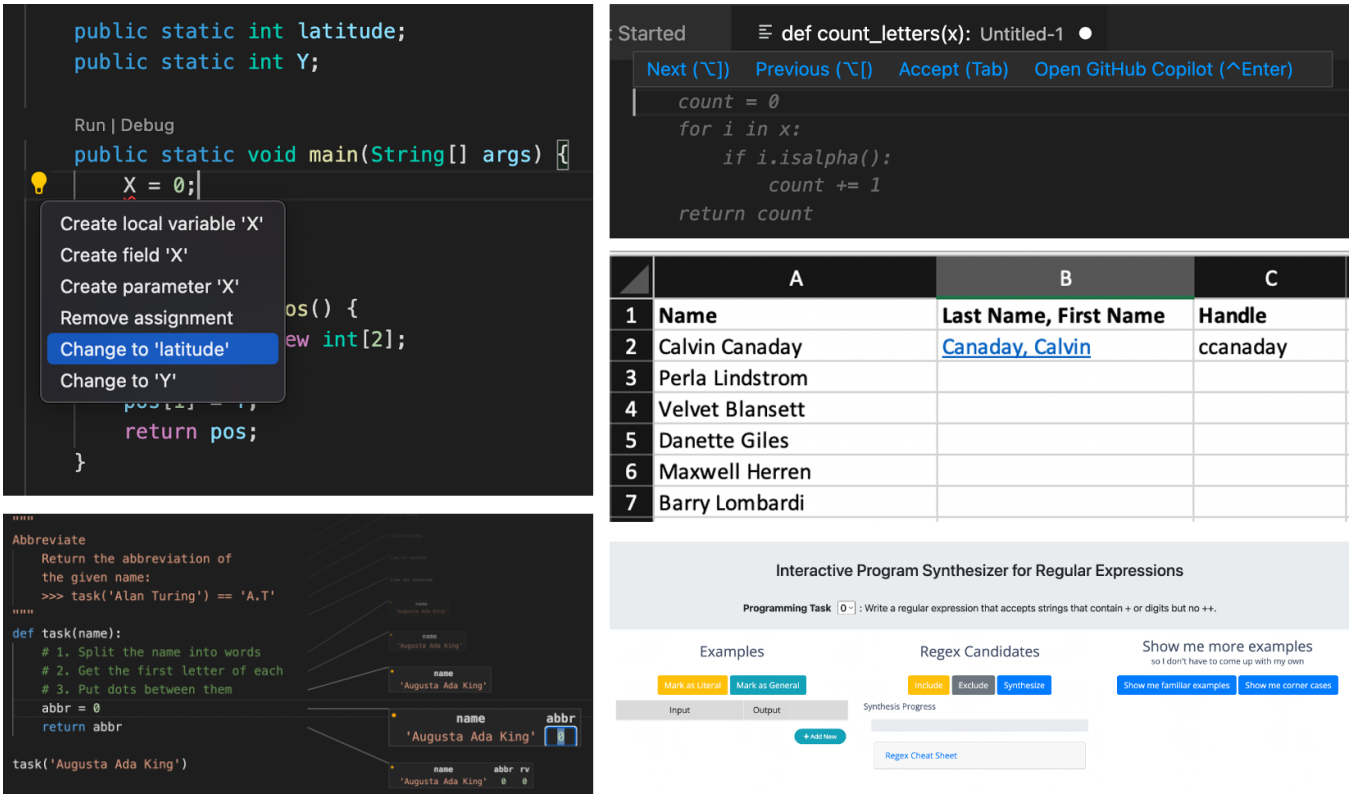

Figure 1: ***Screenshots of each of the program synthesizers used in the study.*** **Clockwise from the top left: Blue-Pencil, Copilot, Flash Fill, Regae, and SnipPy.**

developing. We call these separate, standalone specifications *voluntary specifications*. In contrast, some tools create specifications behind the scenes based on users' existing, pre-synthesizer behaviors. In such cases, when a specification is a byproduct of proceeding toward an existing goal, we call the specification an *incidental specification*. The following table presents a summary classification, which we expand upon in the next two subsections:

Voluntary Specifications:	Regae, SnipPy
Incidental Specifications:	Blue-Pencil, Copilot, Flash Fill

6.1.1 Voluntary Specifications: Specifications created only as input to a synthesizer. Two of the synthesizers in our study—SnipPy and Regae—require participants to craft specifications beyond what they would have produced during their normal programming process. All participants who used tools that required such a specification—what we call a *voluntary specification*—asked for some form of clarification or assistance before successfully eliciting a synthesis output.

SnipPy offers a traditional programming environment in which users *could* author code manually, but in which manual coding does *not* produce a specification. Instead, participants use a special synthesis mode if they want to take advantage of the synthesizer. Four of the six SnipPy participants attempted to write code manually to complete SnipPy-Reverse-2 instead of entering input-output examples to make a specification for the synthesizer. P6 jumped to thinking about the structure of the code rather than input-output examples, remarking, "I was thinking I could use a for loop on the input. Would that make sense?" Similarly, P5 was surprised that they were not supposed to manually write the code, remarking,

> So I'm not allowed to type my own code, I have to do it this way?

In contrast to SnipPy, Regae offers an entirely new programming environment which does not resemble the programming environments that study participants had seen before. Regae gives users functionality for entering (i) strings that their target regular expression should reject or accept and (ii) hints for shaping how the synthesizer searches the space of programs; notably, it does *not* allow users to author regular expressions manually. However, despite lacking visual similarities with traditional programming environments (see Figure 1), participants still attempted to use the environment as a site to write programs manually; four of the six Regae participants all tried to write partial regular expressions rather than input-output examples. Looking at a synthesized (but incorrect) regular expression, P2 wondered aloud, "Is there anywhere I can write the regex directly then? I can't find the expressions I want in [these suggestions]."

Even before looking at Regae's synthesis results, participants were still inclined to write code rather than examples. P15's immediate reaction to Regae was to inspect the interface for about 15 seconds and subsequently ask, "So where should I write the code?

Table 2: *Participant tasks.* Tasks are ordered by difficulty and were assigned in order. Throughout this paper, we refer to tasks by the identifier Synthesizer-Name-#.

Synthesizer	Name	#	Description
Blue-Pencil	Point	1	Change the program to use `Point` objects to represent position rather than a pair of integers `(x, y)`.
Blue-Pencil	Rename	2	Change the name of variable `X` to `latitude`. Change the name of variable `Y` to `longitude`.
Blue-Pencil	LinkedList	3	Change a list to be a `LinkedList` (built-in Java class) instead of a primitive array.
Copilot	Abbreviate	1	Write a program to return the abbreviation of a given name.
Copilot	Occurrences	2	Write a program to turn a list into a dictionary that counts the number of occurrences of each character.
Copilot	Subsequence	3	Write a program to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order. For example, the output for `[10, 22, 9, 33, 21, 50, 41, 60, 80]` should be `6` because the longest sorted subsequence is `[10, 22, 33, 50, 60, 80]`.
Flash Fill	Names	1	Using the data from Column A, populate Column B with `<Last Name>, <First Name>` and populate Column C with `<First Initial><Last Name>` in lower case.
Flash Fill	Emails	2	Populate Column B with the prefixes of the email addresses in Column A.
Flash Fill	Characters	3	Populate Column B with all of the upper case letters from Column A. Populate Column C with all of the lower case letters from Column A. Populate Column D with all of the numbers from Column A.
Regae	Plus	1	Write a regular expression that accepts strings that contain `+` or digits but no `++`.
Regae	ABC	2	Write a regular expression that accepts strings that only have `A`, `B`, `C`, or any combinations of them.
Regae	Phone	3	Write a regular expression that accepts phone numbers that start with one optional `+` symbol and follow with a sequence of digits. For example, `+91` and `91`, but not `91+`.
SnipPy	Abbreviate	1	Write a program to return the abbreviation of a given name.
SnipPy	Reverse	2	Write a program to reverse a given string.
SnipPy	Filter	3	Write a program to return a given string without a specified letter.

I don't see anywhere for me to write the expressions." Similarly, while using Regae, P1 explained, "Basically I am trying to exclude two plus signs from being next to each other, and the regex guide said that the 'not' expression would work for that." After struggling for about 90 seconds to enter a regex directly rather than an input-output example, we informed P1 that the interface expected an input-output example, to which they responded: "Oh! I'm sorry. I thought I had to write the code directly in the box. Let me try entering an example instead."

Design Opportunity. Novice programmers may prefer program synthesis tools where they can *also* complete tasks using non-synthesis strategies or pre-existing expertise, such as by writing code manually.

Although voluntary specifications proved to be a barrier to initial learnability, participants did not necessarily view them as a long-term barrier to usability. P5 later remarked,

> I'm so used to writing code—not writing output of it—but now that I got the hang of it … if I hadn't done an intro to Python course, I think it would actually be really helpful.

And P4 commented,

> When you first showed me how to use [it], I was still pretty confused because I have never used anything like that pop-up before. Once I saw it a few times though, I was able to get used to it.

6.1.2 Incidental Specifications: Specification as a byproduct of normal tool use. The remaining three synthesizers in our study elicit specifications as a byproduct of normal tool use without requiring synthesizer-specific specification input. We refer to these specifications as *incidental specifications*.

- In Excel, users build Flash Fill inputs by filling a subset of cells in a column. This is the same first step that a user would take to fill a whole column manually.
- In Visual Studio Code, users build Copilot inputs by writing part of a program. This is the same first step that a user would take to write a program manually.
- In Visual Studio Code, users build Blue-Pencil inputs by editing a part of a program. This is the same first step that a user would take to perform the corresponding edits across the entire program manually.

Table 3: *A summary of which synthesizers feature triggerless and user-triggered initiation and result communication.*

	Triggerless Result Communication	**User-Triggered Result Communication**
Triggerless Initiation	Copilot	Blue-Pencil
User-Triggered Initiation	*quadrant empty by construction*	Flash Fill, Regae, SnipPy

Incidental specifications generally appeared to be the more exciting and less confusing for participants. For instance, after P19 was introduced to Copilot, they responded "Oh, cool! Okay, so I just have to [write code] normally." Similarly, for Flash Fill, P5 read the Flash Fill instructions, then began to complete the first Flash Fill task by filling in two of the cells manually. They clicked the Flash Fill button and, upon seeing that it completed the rest of the task automatically, exclaimed, "Oh that is super useful! I didn't realize it was going to do [that]." P4 was similarly delighted with the incidental of nature of Flash Fill's specification, commenting "Oh! That is very convenient. I didn't have to type anything except for the first cell."

Incidental specifications also enabled participants to ignore synthesis features when they did not feel compelled to rely on them. For example, P9 started BluePencil-Rename-2 by manually editing code. Then, while they were doing so, the contextual button to run Blue-Pencil's synthesis appeared three separate times—all while P9 was writing the first line of code alone. Although this contextual button is arguably not very discoverable (P14, for example, mentioned that "[they] barely noticed the light bulb when it showed up"), the incidental nature of Blue-Pencil's specification (simply writing code as usual) did not diminish P9's pre-existing ability to complete the task.

Moreover, while SnipPy and Regae users were reluctant to shift from authoring code manually to providing voluntary specifications built up by alternative interactions, Flash Fill users were comfortable providing a specification by filling spreadsheet cells rather than by writing cell formulas manually, which suggests that a key learnability stumbling block is *not necessarily* asking for non-program specifications, but rather asking users to take actions that do not visibly move the user towards the target output. These findings resonate with broader research on the relatively high cognitive costs of task-switching [1, 32, 59] and the usability issues around modes [12, 57].

Design Opportunity. Incidental specifications are a promising avenue for on-ramping novice programmers to using program synthesizers.

6.2 Triggerless and User-Triggered Initiation and Result Communication

We identified two important design decisions for program synthesizers regarding the mechanism and timing for synthesis initiation and result communication:

- Does the tool decide when it should run synthesis, or does the user decide?
- Does the tool decide when it should communicate synthesis results, or does the user decide?

We use the terms *triggerless initiation* and *triggerless result communication* to describe when the tool automatically makes these decisions; conversely, we use the terms *user-triggered initiation* and *user-triggered result communication* when the user makes these decisions. We categorize the program synthesizers that participants used into quadrants in Table 3 representing all possible combinations of the aforementioned design choices. We note, however, that only three of the four quadrants are inhabitable by tools: because synthesizer outputs can only be displayed after the synthesizer runs, a tool that relies on user-triggered synthesis necessarily also relies on user-triggered result communication.

6.2.1 Triggerless Initiation + Triggerless Result Communication. Copilot uses triggerless initiation and triggerless result communication: it proactively provides code suggestions as the user types normally. Although participants who used Copilot did not always accept the synthesis suggestions without modification (as we discuss later in Section 6.7.1), these participants all initiated synthesis rapidly and were able to see synthesis results immediately. For example, P20 started Copilot-Abbreviate-1 by writing `def abbrev(name):`, to which Copilot quickly showed a synthesis suggestion. P20 then used their mouse to hover over the suggested line of code and asked, "So should I just accept this suggestion? It looks correct to me." P18 had a similar experience, and simply pressed the tab key to accept the synthesis suggestion when presented with it. P19 summarized their experience by remarking, "I really liked how I could see the suggestion without having to do anything." This behavior may violate a maxim of *polite software* [72]—in particular, that "polite software respects, and does not preempt, rightful user choices." However, these findings findings resonate with the polite software observation that '[impolite software] may help novices … who may trade politeness for usefulness' [72].

On the other hand, a downside to an entirely triggerless experience was that, after relying on synthesis for parts of a task, participants sometimes wanted to continue to rely on the synthesizer by *manually* triggering it. For example, when using Copilot to synthesize a solution for Copilot-Subsequence-3, P20 noticed a bug in the synthesized code. When starting to debug, P20 wondered aloud, "How do I get the suggestions again? Can I use the tool to help find the error?" Ultimately, P20 could not figure out how to manually trigger Copilot suggestions for what they wanted to do, so they finished the task by fixing the bugs manually.

6.2.2 Triggerless Initiation + User-Triggered Result Communication. Unlike Copilot, Blue-Pencil has triggerless initiation but relies on user-triggered result communication. It identifies repetitive edit actions as the user types normally, and it automatically synthesizes program transformation scripts to automate the edits. However, the user must go out of their way to request to view synthesis outputs by clicking on a contextual light bulb icon that appears when synthesis has completed.

Overall, participants were confused about *when* they should click on this contextual light bulb. As P14 described, " I didn't really know

when to click on it, because I didn't know how it could help." In some cases, Blue-Pencil's contextual light bulb appeared during a time when the participant struggled to complete an aspect of a task we assigned them, and, if the participant had clicked on it, they would have seen a suggestion from Blue-Pencil that would have automatically completed the task successfully. For example, P9 tried to manually make modifications to the starter code in BluePencil-Point-1, but their edits resulted in errors because P9 did not initialize a `Point` object correctly. (Blue-Pencil would have initialized the `Point` correctly.) Similarly, P10 mentioned they were confused about a syntactic construct in Java that was relevant to the edit they needed to make in BluePencil-Point-1, and thus they could not initially complete one of the repetitive edits. (Blue-Pencil would have completed the edit with the correct syntax.)

Design Opportunity. Synthesis with triggerless initiation may be more learnable to novice programmers, but to reap the full benefits of triggerlessness, synthesis tools will likely need to support *both* triggerless initiation as well as triggerless result communication.

6.2.3 User-Triggered Initiation + User-Triggered Result Communication. Regae, Flash Fill, and SnipPy all rely on user-triggered initiation: users must manually decide when they have built a sufficient synthesizer input, then trigger synthesis themselves. As a result, these tools necessarily require a user trigger for the synthesis to communicate output results, which places them all in the bottom-right quadrant of Table 3: user-triggered initiation with user-triggered result communication.

Placing the burden of deciding when a specification is sufficient into the hands of novice programmers produced significant learnability obstacles for the synthesizers in this category.

On a basic level, simply remembering how to trigger synthesis was a learning obstacle for participants. Four of the six participants who used SnipPy struggled significantly to remember how to trigger synthesis (which is done by introducing a hole, ??, in the program text). P3 even asked the same question twice after they saw the ?? work as a trigger for the synthesis pop up: "How do I open the box to put in examples again?" and "Sorry, I forgot how to get the box again, can you remind me?" Working with Regae, P1 kept adding input-output examples because they did not know how to trigger synthesis; only after adding twelve input-output examples did they ask,"Is there a maximum number of examples I can add? What do I do next?"

On a deeper level, participants often built substantially larger specifications than the synthesizer required—a mode of failure that should never occur in triggerless synthesis initiation. For example, four of the six Regae participants spent significantly more time than necessary adding exhaustive input-output examples in the Regae interface and only clicked the synthesize button when we prompted them to do so; until that time, the synthesizer did not run. After 3 minutes, we nudged P6 to stop adding examples, to which they remarked:

> Oh that makes sense. I wasn't sure when to stop adding examples because I thought I had to add one for every possible input.

P12 had a similar experience, adding ten annotated examples during Regae-Plus-1 before asking, "Is this too many inputs? I don't know how many more to add."

Design Opportunity. Synthesizers with user-triggered initiation may need to provide guidance or feedback to users on *how many* examples are likely to be necessary for a given task.

Even more fundamentally, user-triggered synthesis initiation creates a setting that is ripe for user misconceptions about specifications to manifest. For example, the participant behavior we described in the previous paragraph indicates that participants often seemed to hold the belief that the synthesizers they worked with would always perform better if the synthesizers were given more information. We call this belief about synthesizer specifications the *monotonicity belief*. In fact, the performance of synthesis algorithms may degrade as the sizes of specifications grow large, even when parts of the larger specification are redundant and the specifications are satisfied by the same program [16, 69].

This monotonicity belief led to counterproductive behaviors when working with, for example, Regae. As a synthesizer with user-triggered initiation, Regae required participants to decide—without guidance—how many examples to provide before triggering synthesis. P2 initially entered a set of five redundant examples that caused the synthesizer to return an incorrect result. Instead of modifying these examples to be more representative of the problem domain, P2 immediately jumped to adding ten more examples, but the total number of examples P2 entered simply caused Regae to time out.

Design Opportunity. User-triggered synthesis lets novice programmers construct incorrect theories about when to trigger synthesis. Designers of such synthesizers may want to consider identifying misconceptions about their tool (such as the belief that larger specifications will improve synthesis performance) via user studies, then refine their tool to proactively combat these misconceptions.

6.3 Small Specification Size

Overall, participants more easily learned to use—and were delighted by—synthesizers that required relatively small specifications, such as Copilot, Blue-Pencil, and Flash Fill. For example, once Copilot suggested synthesized code for Copilot-Abbreviate-1 after P19 typed only a function declaration (`def abbrev(name):`), P19 exclaimed: "Oh wow! Can I just press tab and accept the suggestion?" Similarly, after P14 made just one program edit for BluePencil-Rename-2, Blue-Pencil synthesized the remaining edits needed to complete the task, to which P14 responded: "I mean, I think it did all of it for me." Lastly, after completing FlashFill-Characters-3 by running Flash Fill on just one example, P6 remarked, "Oh, cool! That was easy. I didn't know for sure if it could figure out the pattern from [one example]."

While participants did seem to appreciate a smaller specification size, it is worth noting that we observed that other factors—such as voluntary versus incidental specifications (Section 6.1) and triggerless versus user-triggered interactions (Section 6.2)—seemed to

have a more substantial impact on the learnability of the program synthesizers. Recommended practices for synthesis systems [37] and many existing program synthesizers make small specification size an explicit goal [8, 23, 43, 56], so this finding may suggest a change in emphasis for future algorithmic work.

Design Opportunity. Synthesis designers may want to explore factors other than size to reduce the burden of specification, such as collecting incidental specifications, which could elicit large specifications from novice programmers with relative ease.

6.4 Interpreting Synthesis Outputs

Participants struggled to use synthesis outputs to figure out if the synthesizer had made useful progress, in particular (i) thinking the synthesizer made progress on a task when it had not and (ii) not realizing that the synthesizer made progress on a task.

This struggle with assessing progress may reflect difficulties decomposing a task into smaller subproblems, understanding when synthesis outputs are solving those subproblems, or some combination of both. If decomposition is at fault, this echoes the difficulties novice programmers face with *plan composition* (putting together fragments of programming knowledge that accomplish particular kinds of tasks [64]) and *task decomposition* in programming more broadly [6, 18, 65]. However, they may also reflect a lack of necessary guidance from the synthesis tool.

6.4.1 Incorrectly Believing the Synthesizer Made Progress. In one case, P14 made an incorrect edit in BluePencil-Point-1 (changing a return statement to return an entire point instead of an x-coordinate). Blue-Pencil then generalized this edit, and P14 accepted Blue-Pencil's suggestion to carry the edit through the rest of the program. However, these edits took P14 farther from the task's solution and ultimately P14 had to make the necessary edits manually, without Blue-Pencil's help.

In another case, P3 overgeneralized their experience working on the first SnipPy task (SnipPy-Abbreviate-1) to the second SnipPy task (SnipPy-Reverse-2) by reusing the step in SnipPy-Abbreviate-1 of splitting a space-separated string into a list of words. As in SnipPy-Abbreviate-1, synthesis succeeded for this query and produced code to separate the string properly. P3 appeared to interpret this result as an encouraging sign and continued by providing input-output examples to reverse the string, not recognizing that the first "successful" result was not a meaningful step towards solving SnipPy-Reverse-2. Due to not having enough of the task broken down properly, SnipPy was unable to synthesize code to satisfy the input-output examples that P3 specified, which led P3 to scrutinize their second query to the synthesizer and completely ignore the code generated by their first query, which was actually at fault.

Design Opportunity. Because novice programmers may be confused about the difference between synthesis succeeding on a query and making meaningful progress on a task, synthesis designers may need interventions like (i) documentation stressing the distinction or (ii) scaffolding for problem decomposition.

6.4.2 Incorrectly Believing the Synthesizer Did Not Make Progress. When working on Regae-Plus-1, P17 observed some results that were almost (but not exactly) correct, noting "This one is kind of correct because [the examples] can't contain ++, but it still accepts letters." However, they did not understand that they could mark these partial solutions to be included in a full solution in Regae; instead, they completely threw away the synthesis results and went back to add and augment examples in their original specification to try to get a more correct synthesis result.

Overall, participants often missed when the synthesizer made useful suggestions or even full solutions, even when the synthesis outputs were visible. For instance, P19 noticed that the autocomplete suggestion for Copilot-Occurrences-2 was incorrect, so they started to rename a function in an effort to trigger Copilot to synthesize something different. Although the Copilot pane which shows additional suggestions (beyond the autocomplete one) displayed two correct synthesis outputs, the participant did not realize and continued with their strategy of renaming the function.

6.5 What Kind of Specification Does My Synthesizer Want?

Participants did not always know how best to specify their intent to the synthesizers they used. In particular, participants did not know what *kinds* of specification were best for a given synthesizer. For example, when attempting Copilot-Occurrences-2, P20 was not satisfied with the first synthesis suggestion, but seemed unsure how much of an impact changing their function declaration would have: "I feel like this is wrong, and I am a bit confused. If I change the name of the function, would this change the results that Copilot would output?" P19 faced a similar situation, receiving an incorrect Copilot autocomplete suggestion, then struggling to figure out how to communicate to Copilot what they wanted to change. They came up with the same strategy of changing a function name, tweaking a portion of their program text to read `def listToDictionary(list)` instead of `def count(list)`. These observations reinforce a pattern identified in [34], that lacking a formal programming language, users try to teach themselves the informal "language" accepted by the language model, largely via trial and error.

For triggerless tools, sometimes users could not even figure out how to change their tool use behaviors in order to *trigger synthesis*. For example, P9 could not figure out what to include in their specification to trigger the contextual light bulb in Blue-Pencil, remarking "It feels like the light bulb pops up randomly."

The specification issue was most prominent with participants using example-based synthesizers. As we discussed in Section 6.2.3, participants using user-triggered example-based synthesizers often had the quantitative misconception that more examples would always result in a better outcome (the monotonicity belief); however, participants also had qualitative misconceptions about what *kinds* of examples to provide as well. For example, P17 provided an extensive set of correctly-annotated input-output examples to Regae, ran the synthesizer, and received only an indication of synthesis failure after a few minutes of waiting. Despite their large synthesis specification, P17 had missed some edge cases while providing

an overabundance of redundant examples. P17's response to this situation was to wonder aloud:

> Am I missing any cases? I feel like I covered all of the edge cases. Do I need to add a different example for every letter?

Similarly, P15 initially entered small, underspecified set of input-output examples into Regae. The synthesizer returned an incorrect regex, and P15's reaction was to add significantly more positive and negative input-output examples (approximately 15), but they did not provide the additional, optional annotations on the input-output examples that the synthesizer needed to complete the task. Their endeavor ultimately resulted in another synthesis failure.

Lack of feedback upon synthesis failure. Participants' lack of knowledge about what constitutes a good specification was exacerbated by the lack of informative feedback upon synthesis failure, which echoes findings in the explainability of recommendation systems [67], the learnability of AI systems more broadly (Section 3.4), the notion of the *user-synthesizer gap* [15] that previous usability studies of synthesizers have explored [14, 15], and findings from follow-up work on the Regae synthesizer [75].

For example, after Regae timed out on a synthesis request from P2, P2 asked, "Is there a way to check why it failed?" Similarly, when attempting SnipPy-Reverse-2, P2 entered "Augusta Ada King" and the same string backward as an input-output example which (for a reason we do not know) caused the synthesizer to fail. When the participant later changed the example to "John Doe" and kept all other aspects of the specification the same, the synthesis query succeeded, and P2 asked "Oh that's weird. Do you know why it didn't work before?"

This lack of feedback on specifications upon failure was particularly problematic in synthesizers with complex, multi-part specifications. For example, the Regae interface requires the user to take several distinct steps: (i) enter input-output examples, (ii) annotate input-output examples as either "general" or "literal," and (iii) decide when there are enough examples to manually trigger the synthesizer. Aside from the fact that longer sequences of actions mean that users will need to wait longer to view the results of synthesis, we observed that tools that require more independent specification components make it harder to debug synthesis failures. For example, a user may wonder: Are the input-output examples correct? Are they correctly annotated? Are there enough examples? Too many? P15 exemplified this confusion: Having completed the steps above for Regae, P15 received an incorrect result from the synthesis engine. Upon receiving the incorrect result, P15 attempted to use the "Mark as Literal" and "Mark as General" buttons more extensively in order to improve the synthesized results. They also tried to add several more input-output examples that were similar to the ones they had already written. Neither of these approaches helped the synthesizer succeed; the specification bug was actually that the examples did not cover all aspects of the requirements.

Design Opportunity. Novice programmers may benefit from (i) feedback about why a given synthesis run has failed and (ii) interactions that help them understand an incorrect synthesis output and why it was produced.

In summary, participants often did not know what constitutes a good specification, so they invented theories, some of which were incorrect. Without feedback about how to improve their specification when synthesis failed, these wrong theories persisted.

6.6 Borrowing Existing Behaviors in Familiar Environments

Three of the five synthesizers in our study were instantiated in existing, mainstream environments: Flash Fill in Excel, and BluePencil and Copilot in Visual Studio Code. When participants used synthesizers in these previously familiar environments, they may have borrowed behaviors and intuitions from their previous exposures to these tools.

For example, participants using Flash Fill were inclined to enter multiple examples and drag the cell or simply manually type in the formula themselves, rather than provide just one example and use the Flash Fill button. P6 remarked, "I thought I had to enter in a lot of examples and then drag down on the cell, but this is way easier." Similarly, P5 explained,

> I would have probably right clicked on the cell or done a Google search to figure out how to fill in the spreadsheet if you hadn't showed me the Flash Fill button. I don't use Excel too much but that approach usually works for other [programming environments].

When completing BluePencil-Rename-2, P9 knew they could change all variable occurrences to a new name in Visual Studio Code by right clicking on a variable and selecting the "Change All Occurrences" option. They chose this approach that they already knew how to do over using the contextual Blue-Pencil light bulb, which was already present beside the line of code in question.

Finally, P19 did not open the Copilot tab in Visual Studio Code at all prior to the last task. They explained:

> Usually when I use the helper tools in VSCode they don't show anything useful, so I don't use any of the pop ups or features anymore.

Design Opportunity. Because novice programmers may apply pre-existing strategies from a particular environment to synthesis-augmented variants of the environment, designers may benefit from being aware of common pre-existing user behaviors.

6.7 More Effort + Less Trust = More Engagement

Synthesis tools provide novice programmers different ways to engage with coding and debugging relative to a traditional educational setting. In particular, participants in our study quickly saw code they had not written themselves, worked to understand and debug code that is partially or entirely incorrect, and used the output of the synthesizer as a tool to refine their understanding of the problem domain. In some ways, synthesizer output is comparable to an instructor-provided exercise solution in a traditional computer science course—both may show the student an example of a working program. But we see a few key differences:

- **More effort.** Program synthesizers require the user to come up with some form of specification.
- **Less trust.** Program synthesizers may return code that satisfies the given specification but does not do what the user intends.

While these points could be viewed as negative attributes of program synthesis, we observed some positive impacts of these differences. Namely, they may result in more engagement than we expect from students looking at, for example, teacher-written exercise solutions.

6.7.1 Code Engagement. In nearly all cases that the synthesizers returned code, participants investigated the code to some degree to ensure its correctness rather than immediately accepting the code and moving on. For example, P0 completed SnipPy-Abbreviate-1 and manually traced an example input out loud before coming to the conclusion that the synthesized code was incorrect. Similarly, P1 attempted to identify any counterexamples that would imply that the synthesized code was incorrect, ultimately concluding that the code was in fact the correct answer to Regae-Plus-1. On various occasions, P18, P19, and P20 all took at least 60 seconds walking through synthesized code in Copilot before accepting a suggestion. After doing so, both P18 and P20 accepted code they knew did not fully meet the specification for the task at hand and subsequently spent time debugging and tweaking the code to match their needs. P17 describes how this process of engagement unfolded for them in a similar situation:

> This semi walks you through the process of how to get there, and, like, makes you think about what are the edge cases—what do I think about in order to get to that end goal.

While we cannot say for sure that these participants would not have done the same for instructor-provided solution code, these examples suggest that because synthesizers engender less trust, they increase the engagement with output code.

More generally, many participants expressed signs of engagement such as surprise or delight at the code the synthesizer returned, sometimes learning new ways to tackle the problem. For example, after P3 completed SnipPy-Abbreviate-1, they expressed their understanding of the synthesized code, which included a list comprehension. P3 remarked that they had never seen a list comprehension before, but their stated understanding of the code was correct: "I have never seen it done like this before. Does this just mean that it automatically loops over every element without a `for` loop?" Similarly, P4 remarked of SnipPy,

> After the synthesizer delivered the code for me, I kinda went over it and was like "Ahh! That's a really smart way to do it"—I would have thought of a much longer way to do it.

Design Opportunity. Novices spend time reading and tracing synthesized programs rather than accepting them without question. Synthesis may be a useful tool in the toolbox for designers targeting code reading or understanding.

6.7.2 Specification Engagement. Participants spent substantial time and energy on refining specifications, shaping them to eventually match their true intent. After engaging with a synthesis output that satisfied their input specification but did not reach their goal, participants had to reexamine their input specification. For example, although the participants described in Section 6.5 did not have a correct mental model of what kinds of examples would be useful for the synthesizer at hand, they did spend a significant amount of time working and reworking their input specifications.

We observed that Regae participants P2 and P7 found that Regae responded to their synthesis queries by producing results that allowed expressions with an alphabetical letter even though they added a negative example as an input. It was only after they saw this synthesis error that they refined their specifications by annotating examples as general or literal.

We observed that participants produced better specifications over time, during interactions with the synthesizers. We speculate that because synthesis tools can interactively generate code that satisfies a given specification, they may support novice programmers in writing complete and correct specifications. Asking students to ensure that a synthesizer arrives at the correct solution when given their specification may be a plausible educational intervention for teaching students how to reason about correctness of programs in general, and important correctness concepts such as covering corner cases.

Design Opportunity. Novice programmers using synthesis tools spend time wrestling with and refining specifications for the programs they have in mind. Synthesis may be a useful tool in the toolbox for designers targeting specification skills.

7 LOOKING BACK: WHY PBD SYSTEMS FAIL

Tessa Lau's classic article "Why PBD systems fail: Lessons learned for usable AI" [37] offers five design guidelines for crafting usable programming-by-demonstration synthesizers, shaped by formal and informal feedback on the various synthesizers that she and her teams invented. Given the influence of these guidelines in the usable synthesis community, we now briefly conclude our analysis by discussing how our findings relate to her guidelines thirteen years later.

1. Detect failure and fail gracefully. In Section 6.4.1, we discuss how participants sometimes incorrectly believed synthesizers had made progress when they had not. Detecting that a synthesizer is failing or unlikely to succeed given further queries may address this issue. We also discuss synthesizers' lack of feedback upon synthesis failure in Section 6.5 and how it prompted participants to become confused and engage in counterproductive behavior.

2. Make it easy to correct the system. In Section 6.2.1, we discuss how participants sometimes wanted to manually trigger a triggerless synthesizer to refine the synthesis output, but had difficulty doing so, echoing Lau's call for ease of correction. In Section 6.5, we discuss how participants did not always know what specification to provide to the system to best achieve their desired outcomes, leading to extended counterproductive specification modifications in an attempt to correct the synthesizers.

3. Encourage trust by presenting a model users can understand. We discuss participants' difficulties with interpreting synthesis

progress in Section 6.4, although we did not observe instances of participants fundamentally misunderstanding the output of the synthesizer itself (e.g., the synthesized code, regexes, or code transformations). We speculate this stems from the fact that our participants were novice programmers rather than non-programmers.

4. Enable partial automation. This guideline reflected the observation that users may not always want the synthesizer to do all the work (total automation). Participants reported appreciating tools in which they could make progress towards their goal without the synthesizer's help—e.g., writing code manually or filling spreadsheet cells (Section 6.1). This reinforces the value of partial automation. However, Lau's discussion particularly emphasizes the importance of letting users modify synthesized programs manually, after synthesis. We saw evidence for this too, with Regae users wanting to manually write regexes based on synthesis outputs (Section 6.1.1).

5. Consider the perceived value of automation. This guideline emphasized that helping the synthesizer is work, and this new work is only worthwhile if it is faster or easier than completing the task another way. Lau suggested integrating synthesis into existing tools to reduce the work required of the user. Participants appeared to prefer incidental specifications (Section 6.1.2) over voluntary specifications (Section 6.1.1), which may support Lau's suggestion. On the other hand, we discuss in Section 6.6 that participants' pre-existing ideas about editor features could interfere with their judgment about the utility of the integrated synthesis tool.

Summary. Our analysis contributes multiple themes that go beyond the discussion in "Why PBD Systems Fail," including: (i) axes for synthesis authors to explore (Sections 6.1 and 6.2), (ii) the impact of collecting specifications in the background and how it nuances the idea that tools "must learn accurately from an absurdly small number of user-provided training examples" [37] (Sections 6.1.2 and 6.3), and (iii) that some of the "failures" of these systems (namely, that they require more effort and instill less trust, as discussed in Section 6.7) may actually produce novel design opportunities for synthesis authors. However, we also find support for some of Lau's existing guidelines, especially for the importance of guidelines (1) detecting and communicating failure, (2) making it easy for users to refine specifications, and (4) letting users do some amount of the work manually. In addition to serving as a standalone set of themes for future research to explore, we hope putting this study in conversation with existing works like Lau's seminal guidelines can help the community to revisit and re-prioritize existing guidance.

8 LIMITATIONS AND FUTURE WORK

Disentangling effects. As a consequence of using real tools that were not designed to vary one design dimension at a time, we cannot definitively attribute every pattern to a particular design decision. A randomized controlled experiment would be required to achieve definitive conclusions about varying degrees of learnability along the axes that we identified. However, more than attributing success to a particular dimension versus another, this study was aimed at *identifying* and *exploring* dimensions of interest in the first place. We hope future research on synthesizer learnability will contribute experiments that vary a given synthesizer across the dimensions we identified. We expect this style of research will eventually help the field develop guidelines for how we should design synthesis tools, rather than only evaluating a given tool in isolation or against manual coding.

Effects of think-aloud and investigator presence. Participants completed tasks and thought aloud simultaneously, which could increase cognitive load relative to using tools outside a lab setting [11]. Further, the first author was present throughout the video meeting and recorded the session for future analysis. If participants felt pressured, their interactions with the synthesizers may have been harder. Conversely, the presence of the first author may have had the opposite effect if it assisted their learning in any way.

Non-evaluative. Some of the tools in our study were not created with the intention of being learnable for novice programmers. We selected the tools in our study with the goal of uncovering design decisions that do and do not work for novices. However, the fact that serving novice programmers is not an explicit goal for some of them is one of several reasons that this study should *not* be read as an evaluation of either the synthesizers we used or whether existing synthesis tools for novices are serving that audience.

External validity. Although the tools used in the study are real and publicly available, the research team selected the tasks that participants completed in sessions. The learnability of synthesis tools for these tasks may or may not resemble the learnability for users' real tasks. Moreover, our participants were drawn from a pool of undergraduates attending an R1 university. While the behavior of this population has generated a useful starting point for understanding the learnability of program synthesizers by novice programmers, it almost certainly does not reflect the behavior of *all* novice programmers. We hope future work in this area explores the behaviors of additional groups of novice programmers.

Learning outcomes. Our study explored whether novices found it easy to learn to use synthesizers during first-use exposure, but it did *not* offer insights into whether these tools help them learn programming or computing concepts. Importantly, showing these tools are learnable is *not* the same as showing they improve students' learning outcomes. In fact, learnable synthesis tools may damage CS education outcomes. Exploring the impact of synthesizers on novices' learning outcomes in CS education contexts would help us understand more about roles for synthesizers for novices, perhaps via a longitudinal study using validated concept inventories.

Ultimately, even if synthesizers are eventually shown to have a negative impact on CS students' learning outcomes, we would remain interested in the goal of making synthesizers learnable to novice programmers. Crucially, not all programming novices are CS students. Non-technical domain experts like journalists or social scientists may have a specific task in mind—e.g., a programmatic data analysis—without wanting to learn computer science.

Incompleteness. We expect the insights from this study will be useful for the designers of future program synthesis tools. However, we are certain our study is not the final word on the learnability of program synthesis tools for novice programmers. This study drew on observation of 22 sessions with novice programmers, all of which (i) followed a single, fixed structure and (ii) we analyzed via qualitative analysis alone. Regarding (i), we are confident that there

is much more to learn from additional qualitative and need-finding work; regarding (ii), our observations offer plausible hypotheses for future, quantitative follow-on work.

9 CONCLUSION

Program synthesis tools have the potential to make programming tasks easier for a variety of audiences. This work explores key barriers that stand between novice programmers and this vision of learnable, helpful synthesizers. We hope reasoning explicitly about design dimensions like voluntary versus incidental specifications, triggerless versus user-triggered synthesis, and triggerless versus user-triggered output will eventually support our community in making synthesis tools more learnable. The common misconceptions we observed may also play an important role in helping future synthesizer authors design specifications that work for real users. Although this work is only a first step toward understanding how to make program synthesizers more learnable by novices, we believe insights from this work can ease the design process for designers who aim to make synthesizers for use in novice-targeted development tools.

REFERENCES

[1] D. Alan Allport, Elizabeth A. Styles, and Shulan Hsieh. 1994. Shifting Intentional Set: Exploring the Dynamic Control of Tasks. In *Attention and Performance 15: Conscious and Nonconscious Information Processing*. The MIT Press, 421–452.

[2] Saleema Amershi, Dan Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi Iqbal, Paul N. Bennett, Kori Inkpen, Jaime Teevan, Ruth Kikin-Gil, and Eric Horvitz. 2019. Guidelines for Human-AI Interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems (CHI '19)*. Association for Computing Machinery, 1–13. https://doi.org/10.1145/3290605.3300233

[3] Matej Balog, Alexander Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2017. DeepCoder: Learning to Write Programs. In *Proceedings of ICLR'17*.

[4] Osbert Bastani, Xin Zhang, and Armando Solar-Lezama. 2019. Synthesizing Queries via Interactive Sketching. *CoRR* abs/1912.12659 (2019). arXiv:1912.12659 http://arxiv.org/abs/1912.12659

[5] Victoria Bellotti and Keith Edwards. 2001. Intelligibility and Accountability: Human Considerations in Context-Aware Systems. *Human–Computer Interaction* 16, 2-4 (Dec. 2001), 193–212. https://doi.org/10.1207/S15327051HCI16234_05

[6] Francisco Enrique Vicente Castro and Kathi Fisler. 2016. On the Interplay Between Bottom-Up and Datatype-Driven Program Design. In *SIGCSE Technical Symposium*. https://doi.org/10.1145/2839509.2844574

[7] Francisco Enrique Vicente Castro and Kathi Fisler. 2020. *Qualitative Analyses of Movements Between Task-Level and Code-Level Thinking of Novice Programmers*. Association for Computing Machinery, 487–493. https://doi.org/10.1145/3328778.3366847

[8] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology (UIST '18)*. Association for Computing Machinery, 963–975. https://doi.org/10.1145/3242587.3242661

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374 [cs.LG]

[10] Codecademy. 2022. Learn to Code - for Free. https://www.codecademy.com. Accessed: 2022-07-26.

[11] Simon P. Davies and Adrian M. Castell. 1994. From Individuals to Groups Through Artifacts: The Changing Semantics of Design in Software Development. In *User-Centred Requirements for Software Engineering Environments*, David J. Gilmore, Russel L. Winder, and Françoise Détienne (Eds.). https://doi.org/10.1007/978-3-662-03035-6_2

[12] Asaf Degani. 1996. *Modeling Human-Machine Systems: On Modes, Error, and Patterns of Interaction*. Ph.D. Dissertation. Georgia Institute of Technology.

[13] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. *Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists*. Association for Computing Machinery, 1–12. https://doi.org/10.1145/3313831.3376442

[14] Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. LooPy: Interactive Program Synthesis with Control Structures. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 153 (Oct. 2021), 29 pages. https://doi.org/10.1145/3485530

[15] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. *Small-Step Live Programming by Example*. Association for Computing Machinery, 614–626. https://doi.org/10.1145/3379337.3415869

[16] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. *ACM SIGPLAN Notices* 50, 6 (June 2015), 229–239. https://doi.org/10.1145/2813885.2737977

[17] Leah Findlater and Joanna McGrenere. 2004. A Comparison of Static, Adaptive, and Adaptable Menus. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*. Association for Computing Machinery, 89–96. https://doi.org/10.1145/985692.985704

[18] Kathi Fisler and Francisco Enrique Vicente Castro. 2017. Sometimes, Rainfall Accumulates: Talk-Alouds with Novice Functional Programmers. In *International Conference on Computing Education Research (ICER)*. https://doi.org/10.1145/3105726.3106183

[19] Nat Friedman. 2021. Introducing GitHub Copilot: your AI pair programmer. https://github.blog/2021-06-29-introducing-github-copilot-ai-pair-programmer/. Accessed: 2022-04-04.

[20] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. Association for Computing Machinery, 653–663. https://doi.org/10.1145/2568225.2568250

[21] Ivan Gavran, Eva Darulova, and Rupak Majumdar. 2020. Interactive Synthesis of Temporal Specifications from Examples and Natural Language. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 201 (Nov. 2020), 26 pages. https://doi.org/10.1145/3428269

[22] GitHub Inc. 2021. GitHub Copilot: Your AI pair programmer. https://copilot.github.com/. Accessed: 2022-03-31.

[23] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. Association for Computing Machinery, 317–330. https://doi.org/10.1145/1926385.1926423

[24] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-Free Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. Association for Computing Machinery, 62–73. https://doi.org/10.1145/1993498.1993506

[25] Sumit Gulwani and Ramarathnam Venkatesan. 2009. *Component Based Synthesis Applied to Bitvector Circuits*. Technical Report MSR-TR-2010-12. Microsoft Research.

[26] Brian Hempel and Ravi Chugh. 2016. Semi-Automated SVG Programming via Direct Manipulation. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology* (Tokyo, Japan) *(UIST '16)*. Association for Computing Machinery, New York, NY, USA, 379–390. https://doi.org/10.1145/2984511.2984575

[27] Austin Z. Henley, Julian Ball, Benjamin Klein, Aiden Rutter, and Dylan Lee. 2021. *An Inquisitive Code Editor for Addressing Novice Programmers' Misconceptions of Program Behavior*. IEEE Press, 165–170. https://doi.org/10.1109/ICSE-SEET52601.2021.00026

[28] K. Höök. 2000. Steps to Take before Intelligent User Interfaces Become Real. *Interacting with Computers* 12, 4 (2000), 409–426. https://doi.org/10.1016/S0953-5438(99)00006-5

[29] Eric Horvitz. 1999. Principles of Mixed-Initiative User Interfaces. In *Proceedings of CHI '99, ACM SIGCHI Conference on Human Factors in Computing Systems, Pittsburgh, PA, ACM Press*. 159–166.

[30] Thibaud Hottelier, Ras Bodik, and Kimiko Ryokai. 2014. Programming by Manipulation for Layout. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST '14)*. Association for Computing Machinery, 231–241. https://doi.org/10.1145/2642918.2647378

[31] Jingmei Hu, Priyan Vaithilingam, Stephen Chong, Margo Seltzer, and Elena L. Glassman. 2021. Assuage: Assembly Synthesis Using A Guided Exploration. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST '21)*. Association for Computing Machinery, 134–148. https://doi.org/10.1145/3472749.3474740

[32] A. T. Jersild. 1927. Mental Set and Shift. *Archives of Psychology* 14, 89 (1927).

[33] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. 215–224. https://doi.org/10.1145/1806799.1806833

[34] Ellen Jiang, Edwin Toh, Alejandra Molina, Kristen Olson, Claire Kayacik, Aaron Donsbach, Carrie J. Cai, and Michael Terry. 2022. Discovering the Syntax and Strategies of Natural Language Programming with Generative Language Models. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22)*. Association for Computing Machinery, New York, NY, USA, Article 386, 19 pages. https://doi.org/10.1145/3491102.3501870

[35] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis modulo Recursive Functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. Association for Computing Machinery, 407–426. https://doi.org/10.1145/2509136.2509555

[36] Todd Kulesza, Margaret Burnett, Weng-Keen Wong, and Simone Stumpf. 2015. Principles of Explanatory Debugging to Personalize Interactive Machine Learning. In *Proceedings of the 20th International Conference on Intelligent User Interfaces (IUI '15)*. Association for Computing Machinery, 126–137. https://doi.org/10.1145/2678025.2701399

[37] Tessa Lau. 2009. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. *AI Magazine* 30, 4 (Oct. 2009), 65–65. https://doi.org/10.1609/aimag.v30i4.2262

[38] Sorin Lerner. 2020. *Projection Boxes: On-the-Fly Reconfigurable Visualization for Live Programming*. Association for Computing Machinery, 1–7. https://doi.org/10.1145/3313831.3376494

[39] Colleen M. Lewis. 2012. The Importance of Students' Attention to Program State: A Case Study of Debugging Behavior. In *Proceedings of the Ninth Annual International Conference on International Computing Education Research (ICER '12)*. Association for Computing Machinery, 127–134. https://doi.org/10.1145/2361276.2361301

[40] Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. 2017. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. Association for Computing Machinery, 6038–6049. https://doi.org/10.1145/3025453.3025483

[41] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M. Mitchell, and Brad A. Myers. 2019. PUMICE: A Multi-Modal Agent That Learns Concepts and Conditionals from Natural Language and Demonstrations. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. Association for Computing Machinery, 577–589. https://doi.org/10.1145/3332165.3347899

[42] Brian Y. Lim and Anind K. Dey. 2009. Assessing Demand for Intelligibility in Context-Aware Applications. In *Proceedings of the 11th International Conference on Ubiquitous Computing (UbiComp '09)*. Association for Computing Machinery, 195–204. https://doi.org/10.1145/1620545.1620576

[43] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 109:1–109:29. https://doi.org/10.1145/3408991

[44] Ewa Luger and Abigail Sellen. 2016. "Like Having a Really Bad PA": The Gulf between User Expectation and Experience of Conversational Agents. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems (CHI '16)*. Association for Computing Machinery, 5286–5297. https://doi.org/10.1145/2858036.2858288

[45] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*. Association for Computing Machinery, 48–61. https://doi.org/10.1145/1065010.1065018

[46] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind Your Language: On Novices' Interactions with Error Messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Portland, Oregon, USA) *(Onward! 2011)*. Association for Computing Machinery, New York, NY, USA, 3–18. https://doi.org/10.1145/2048237.2048241

[47] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST '15)*. Association for Computing Machinery, 291–301. https://doi.org/10.1145/2807442.2807459

[48] Anders Miltner, Sumit Gulwani, Vu Le, Alan Leung, Arjun Radhakrishna, Gustavo Soares, Ashish Tiwari, and Abhishek Udupa. 2019. On the Fly Synthesis of Edit Suggestions. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 143 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360569

[49] Chelsea Myers, Anushay Furqan, Jessica Nebolsky, Karina Caro, and Jichen Zhu. 2018. Patterns for How Users Overcome Obstacles in Voice User Interfaces. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. Association for Computing Machinery, 1–7. https://doi.org/10.1145/3173574.3173580

[50] Sujata Narayana. 2020. Power BI Desktop August 2020 Feature Summary: Text/CSV By Example (preview). https://powerbi.microsoft.com/en-us/blog/power-bi-desktop-august-2020-feature-summary/#_text_csv. Accessed: 2022-04-05.

[51] Greg L. Nelson, Benjamin Xie, and Amy J. Ko. 2017. Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research (ICER '17)*. Association for Computing Machinery, 2–11. https://doi.org/10.1145/3105726.3106178

[52] Donald A. Norman. 1994. How Might People Interact with Agents. *Commun. ACM* 37, 7 (July 1994), 68–71. https://doi.org/10.1145/176789.176796

[53] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. *SIGPLAN Not.* 50, 6 (June 2015), 619–630. https://doi.org/10.1145/2813885.2738007

[54] Hila Peleg, Shachar Itzhaky, and Sharon Shoham. 2018. Abstraction-Based Interaction Model for Synthesis. In *Verification, Model Checking, and Abstract Interpretation*, Isil Dillig and Jens Palsberg (Eds.). Springer International Publishing, 382–405. https://doi.org/10.1007/978-3-319-73721-8_18

[55] Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming Not Only by Example. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, 1114–1124. https://doi.org/10.1145/3180155.3180189

[56] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. Association for Computing Machinery, 522–538. https://doi.org/10.1145/2908080.2908093

[57] Jef Raskin. 2000. *The Humane Interface: New Directions for Designing Interactive Systems*. Addison-Wesley Professional.

[58] Ruth Ravichandran, Sang-Wha Sien, Shwetak N. Patel, Julie A. Kientz, and Laura R. Pina. 2017. Making Sense of Sleep Sensors: How Sleep Sensing Technologies Support and Undermine Sleep Health. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems (CHI '17)*. Association for Computing Machinery, 6864–6875. https://doi.org/10.1145/3025453.3025557

[59] Robert D. Rogers and Stephen Monsell. 1995. Costs of a Predictible Switch between Simple Cognitive Tasks. *Journal of Experimental Psychology: General* 124, 2 (1995), 207–231. https://doi.org/10.1037/0096-3445.124.2.207

[60] Chad Rothschiller. 2012. Flash Fill. https://www.microsoft.com/en-us/microsoft-365/blog/2012/08/09/flash-fill/. Accessed: 2022-04-04.

[61] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. Association for Computing Machinery, 305–316. https://doi.org/10.1145/2451116.2451150

[62] Steven C. Shaffer. 2005. Ludwig: An Online Programming Tutoring and Assessment System. *SIGCSE Bull.* 37, 2 (June 2005), 56–60. https://doi.org/10.1145/1083431.1083464

[63] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. *SIGARCH Comput. Archit. News* 34, 5 (Oct. 2006), 404–415. https://doi.org/10.1145/1168919.1168907

[64] Elliot Soloway and Kate Ehrlich. 1986. Empirical Studies of Programming Knowledge. In *Readings in Artificial Intelligence and Software Engineering*, Charles Rich and Richard C. Waters (Eds.). https://doi.org/10.1016/B978-0-934613-12-5.50042-2

[65] James C. Spohrer and Elliot Soloway. 1986. Novice Mistakes: Are the Folk Wisdoms Correct? *Communications of the ACM* (July 1986). https://doi.org/10.1145/6138.6145

[66] Amanda Swearngin, Chenglong Wang, Alannah Oleson, James Fogarty, and Amy J. Ko. 2020. *Scout: Rapid Exploration of Interface Layout Alternatives through High-Level Design Constraints*. Association for Computing Machinery, 1–13. https://doi.org/10.1145/3313831.3376593

[67] Nava Tintarev and Judith Masthoff. 2015. Explaining Recommendations: Design and Evaluation. In *Recommender Systems Handbook*, Francesco Ricci, Lior Rokach, and Bracha Shapira (Eds.). Springer US, 353–382. https://doi.org/10.1007/978-1-4899-7637-6_10

[68] Priyan Vaithilingam and Philip J. Guo. 2019. Bespoke: Interactively Synthesizing Custom GUIs from Command-Line Applications By Demonstration. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology (UIST '19)*. Association for Computing Machinery, 563–576. https://doi.org/10.1145/3332165.3347944

[69] Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. 2019. Visualization by Example. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019), 49:1–49:28. https://doi.org/10.1145/3371117

[70] Chenglong Wang, Yu Feng, Rastislav Bodik, Isil Dillig, Alvin Cheung, and Amy J Ko. 2021. Falx: Synthesis-Powered Visualization Authoring. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI '21)*. Association for Computing Machinery, Article 106, 15 pages. https://doi.org/10.1145/3411764.3445249

[71] Jacqueline Whalley and Nadia Kasto. 2014. A Qualitative Think-Aloud Study of Novice Programmers' Code Writing Strategies. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education (ITiCSE '14)*. Association for Computing Machinery, 279–284. https://doi.org/10.1145/2591708.2591762

[72] Brian Whitworth. 2005. Polite Computing. *Behaviour & Information Technology* 24, 5 (Sept. 2005), 353–363. https://doi.org/10.1080/01449290512331333700

[73] Mark Wilson-Thomas. 2019. Refactoring made easy with IntelliCode! https://devblogs.microsoft.com/visualstudio/refactoring-made-easy-with-intellicode/. Accessed: 2022-04-04.

[74] Rayoung Yang, Eunice Shin, Mark W. Newman, and Mark S. Ackerman. 2015. When Fitness Trackers Don't 'Fit': End-User Difficulties in the Assessment of Personal Tracking Device Accuracy. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp '15)*. Association for Computing Machinery, 623–634. https://doi.org/10.1145/2750858.2804269

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

[76] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. 2020. *Interactive Program Synthesis by Augmented Examples*. Association for Computing Machinery, 627–648. https://doi.org/10.1145/3379337.3415900