Debugging into Existence with Program Synthesis

***Abstract*—When modifying an existing codebase to handle new functionality, programmers will often debug the code until the insertion point for the new code. This method, termed *Debugging into Existence*, helps programmers familiarize themselves with the surrounding code and runtime state. Despite its real-world usage, it is limited by the inability to test potential code past the first time the location is called, since added functionality would change the future state making it irrelevant.**

Prior work has pioneered *Live Execution* over partial programs, with extensions using the provided values for synthesis by programming by example. In this work, we present DESYNT, a debugger extension that integrates live execution and program synthesis to extend the Debugging into Existence interaction model. DESYNT grants programmers meaningful runtime information across many executions, by allowing them to manipulate program state according to the desired functionality. Based on the state provided by the programmer, DESYNT then synthesizes programs that capture this functionality. We evaluated DESYNT in a between-subjects study on 10 users, and found that in tasks that do not involve complex fault localization, DESYNT reduces time to completion and concentrates programmer effort into fewer code locations. In addition, we found that users that used DESYNT spent more of their task time debugging, indicating DESYNT supports Debugging into Existence for those that already use it.

I. INTRODUCTION

Programmers often begin writing a program or a feature with only a vauge idea of what the target program should eventually do. In essence, they will work through the precise specification of the code *by writing the code*. In 1983 the name *exploratory programming* was coined for this phenomenon [42].

A specific instance of this has emerged in data-intensive environments. A programmer needs to make use of unknown (and perhaps underspecified) data that is easily made available by running the program. For example, they are writing a bookshelf application that queries a cataloging website such as HARDCOVER.APP [3] and want to extract information from the response JSON, but they are not certain of the structure of the response, nor of the API functions used to manipulate it. To that end, they use exceptions or mechanisms like assigning an expression of type `Nothing` (e.g., Scala's `???`) to create a compiling program that can run up to the point where the data is known and can be examined. In our example, the programmer would set a breakpoint on line 6 to examine `responseJson`.

```
3   runningMax = [0] runningMax = [0]
4   def handleBook(isbn: str): isbn = '597394'
5       responseJson = queryServer(isbn) responseJson =
6       raise Exception("for breakpoint")
7       runningMax.append(max(runningMax[-1],length))
```

Some developers call this workflow "debugger driven development" [2], [4], but in the research literature it is formally known as *Debugging into Existence* [39]: programmers use breakpoints during debugging to inspect variable values and the internal interpreter to test out expressions that can then be copied over to the code.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
→ responseJson["data"]
> {'books': [{...}]}
→ responseJson["data"]["books"]
> [{'id': 597394, 'title': "The Hitchiker's Guide to the Gal
→ len(responseJson["data"]["books"])
  1
> responseJson["data"]["books"][0]
```

The shortcoming of Debugging into Existence as a technique is that it only gives the programmer access to the first iteration. If the `handleBook` function runs within a larger loop that, for example, fetches books from a list fetched from elsewhere, then only the first response will be inspected. If the JSON is different between the first and second books, e.g., some books do not have a `"pages"` field, then the code resulting from the inspection of the list may be unsuitable or crash, which means the error must be understood before the exploration can continue.

When the runs of `handleBook` are independent, the user can use a `print` or some other conceptual *nop* as the place for their breakpoint, and break again at the next calls to inspect additional JSONs. However, the result may be imporant for getting to the next iteration. This is shown here as the trivial case where the next line depends on it, but can be more involved, e.g., the next iteration might use data that is not yet extracted here.

Live execution. We extend the Debugging into Existence workflow using the concepts of *live evaluation* [33], [36] and *live execution* [14], designed to evaluate around program holes. While live evaluation evaluates around *program holes* whenever posisble, pausing when a hole cannot be evaluated, live execution collects values from the programmer, which allows the run of the program to continue as usual. By replacing the exception with a hole, which we mark as `??`, and employing live execution, the programmer can assign desired values to a variable and continue past the first iteration of the program's run.

Programming by Example. The LOOPY synthesizer employed live execution to collect output values for program synthesis by examples in a live programming environment [14]. The idea is that variable values comprise the example's input, and provided values provide the output. However, LOOPY's live programming environment is limited in the depth of executions

it can track, and requires constant evaluation of the code. This makes it hard or impossible to use for large codebases or for code that interacts with the environment, e.g., reading files or calling web services. For functions like `handleBook`, the programmer would still need to use the debugger.

However, once live execution is employed to execute past the hole, the same base insight of LOOPY still holds: an input-output example can be constructed from the values of variables before the hole and values assigned to the hole by the programmer when debugging past it. These values can then be sent to a *Programming by Example* (PBE) synthesizer [6], [18], [20], [45], [46] and programs that satisfy the examples can be suggested to the programmer.

The* DESYNT *system. We present DESYNT, a debugger extension to leverage program synthesis during Debugging into Existance. DESYNT combines live execution past a predetermined hole within the debugger with a PBE synthesizer, that can suggest hole completions to the programmer We include in DESYNT two *views*: a *user-triggered* view where it is up to the programmer to request a program after any number of iterations, and a *triggerless* view that starts trying to generate programs once a minimum number of input-output pairs is reached, and any time a program is available it is suggested to the programmer. Once a program is available, its output value can be used to step over the hole, as long as the programmer is happy with it.

Our results show that in tasks where fault localization is not a major component, DESYNT improves the time to task completion, as well as concentrating the programmer's effort on the specific point of the change. Moreover, tool usage is correlated with spending more time debugging than performing other activities—manually writing code and code exploration. While we do not have enough information to show causation, this at least indicates that DESYNT can support programmers whose work style favors debugging. There was no real difference in how users reacted to the different views of DESYNT or in their utility, indicating that both can be useful to different participants in different scenarios. However, we did observe a behavioral difference that appears driven by the differences between the views: programmers in the *user-triggered* view provided more examples spent more iterations validating a synthesis result after a successful synthesis call.

Contributions. The contributions of this paper are:

- ▷ Extending the Debugging into Existence workflow past the first iteration using live execution.
- ▷ Integration of Programming by Example into this workflow to generate the new code.
- ▷ DESYNT, an implementation of our interaction model in VSCode, with two different views that trigger synthesis differently.
- ▷ Analysis of user behaviour across two views of DESYNT.

II. DESYNT BY EXAMPLE

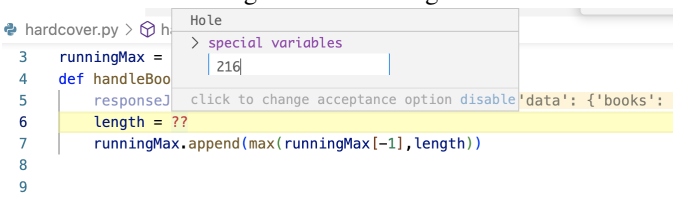
The programmer sets out to write `handleBook`, which is called for every book in a series of books, finding the maximum length *so far* at each point. This time, they approach this task with DESYNT.

Instead of the exception they added to the code in Sec. I, they now set a *hole*, denoted `??`, which indicates an expression assigned to the variable `length`:

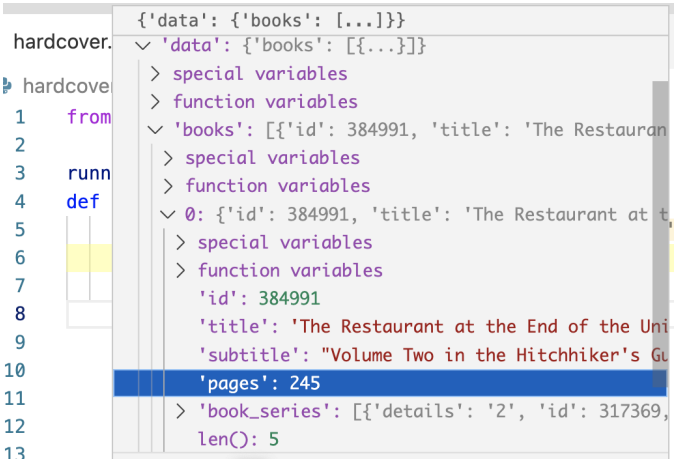
```
3   runningMax = [0]
4   def handleBook(isbn: str):
5       responseJson = queryServer(isbn)
6       length = ??
7       runningMax.append(max(runningMax[-1],length))
8
```

When DeSynt is installed, `??` is a valid token in Python, though it cannot be left in the file later. The programmer starts the debugger, which automatically adds a breakpoint to the line with the hole, and breaks the run there.

First change, in* user-triggered *view. Once the breakpoint is hit, the programmer can use the debugger to inspect the state of the code, as before, and once they've settled on a value for the assigned variable `length` they can write or copy-paste its value which will change it in the debug view:



They can then run or step past the breakpoint. Since `handleBook` is called multiple times during the run, the breakpoint will trigger each time. The second time will look like so:



Here, the JSON is similar, and they again set the value of the hole and continue.

```
3   runningMax = [0]
4   def handleBook(isbn: str): isbn = '384991'
5       responseJson = queryServer(isbn) responseJson = {'dat
6       length = ?? length = 245
7       runningMax.append(max(runningMax[-1],length)) length
```

At this point, the programmer can stop the debugger and enter code that they tested out in the debug console, but they can also press the *Generate* button to task DESYNT for a program.

The DESYNT synthesizer suggests the expression `responseJson["data"]["books"][0]["pages"]` to fetch the number of pages out of the JSON by inserting it in gray in place of the hole in the editor.

The programmer can keep running to see how this suggestion fares on additional data, or accept it immediately and stop the debugger. In this case, they accept it, adding it to the code and stopping the run.

```
3   runningMax = [0]
4   def handleBook(isbn: str):
5       responseJson = queryServer(isbn)
6       length = responseJson["data"]["books"][0]["pages"]
7       runningMax.append(max(runningMax[-1],length))
```

Second change, in* triggerless *view. The programmer now runs their program again, and gets an exception:

```
TypeError: '>' not supported between instances of '
    NoneType' and 'int'
```

The reason for this, unbeknownst to the programmer, is that the JSON for the fourth book has no value for `"pages"`, so their code populates the `length` variable with `None`.

The programmer knows they will probably need to modify the `length` variable based on data they have not yet seen, so they move the raw value from the JSON to an intermediate variable and insert a new hole to assign to `length`:

```
3   runningMax = [0]
4   def handleBook(isbn: str):
5       responseJson = queryServer(isbn)
6       numPages = responseJson["data"]["books"][0]["pages"]
7       length = ??
8       runningMax.append(max(runningMax[-1],length))
```

The programmer runs DESYNT in its *triggerless* view. Initially, they only see iterations with the correct value in `numPages` which they transfer to `length`. After three iterations, the DESYNT synthesizer tries behind the scenes to find a program that satisfies all the provided input/outputs. At this point, the program is trivial: `numPages`, and it is suggested to the programmer, so when the breakpoint is triggered for the fourth iteration, the program suggestion is displayed, as is the value it computes as a suggested value to assign to `numPages`.

```
3   runningMax = [0] runningMax = [0, 216, 245, 245]
4   def handleBook(isbn: str): isbn = '373728'
5       responseJson = queryServer(isbn) responseJson = {'dat
6       numPages = responseJson["data"]["books"][0]["pages"]
7       length = numPages length = None
8       runningMax.append(max(runningMax[-1],length))
```

Here the programmer notices that the suggested value is `None`, and upon closer inspection of the JSON see that this is because there is no value there. They now need to make a decision: what happens when the number of pages is missing? It is the on-the-go understanding of the problem space and the need to make decisions like this that make Debugging into Existence exploratory.

The programmer decides to take the current book out of the running for longest book, and the easiest way to do that is to set its length to 0. They then set this as the value of `length` in this iteration.

```
7       length = numPages length = 0
8       runningMax.append(max(runningMax[-1],length))
```

Since the value of `length` and the suggested value are different, this indicates to DESYNT that the programmer has ruled out the suggested program, `numPages`. DESYNT calls the synthesizer again. By the next time the program breaks, DESYNT suggests a new program:

```
3   runningMax = [0]
4   def handleBook(isbn: str):
5       responseJson = queryServer(isbn)
6       numPages = responseJson["data"]["books"][0]["pages"]
7       length = numPages if numPages else 0
8       runningMax.append(max(runningMax[-1],length))
```

which the programmer accepts, and DESYNT adds to their code.

III. The DESYNT System

DESYNT is an extension of the Python debugger in VSCODE [35].

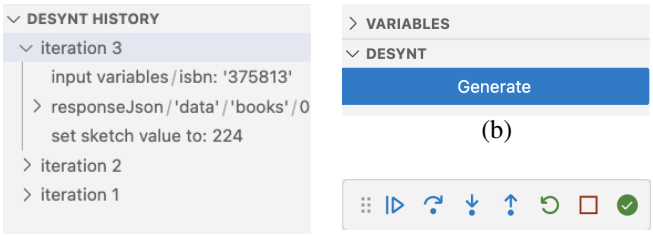
A. *The* DESYNT *interaction model*

Adding holes to Python. DESYNT extends Python's syntax with a new expression, a hole denoted `??`. Holes can be assigned to a variable or returned from a function. A program with a hole, called a *sketch* [43], cannot be executed by the Python runtime, but can be run in DESYNT via the debugger. `??` denotes an expression whose actual code is not known, and so cannot be executed past. Placing a hole indicates the programmer's expectation that it will be replaced with a program snippet. While there is no technical limitation on the number of holes a program can have and that DESYNT can support, this can be quite confusing to users. We therefore limit DESYNT to handling program sketches with a single hole. Additional holes will trigger an error.

Running programs with holes. As shown in Sec. II, once the programmer sets a hole in a program, they can use DESYNT to start the debugger and run the program normally up to the hole. When debugging, holes act as breakpoints that cannot be removed, i.e., the program will always break before the hole's evaluation as it does not have a program to evaluate.

At the breakpoint for a hole, the IDE behaves as in any other breakpoint: the stack and variable states can be inspected, the VSCode debug console can be used to evaluate expressions, and the run of the program can be stopped. This allows the user to explore the program as they would in a debugging into existence session in a regular debugger. If the hole does not have a value, running past the current statement is disabled.

Evaluating holes. Holes are evaluated by *live execution*: the user is asked to provide a value for the hole, which is then set into the assigned variable or returned from the function. This is similar to the ability to set the value of a variable while debugging, a feature that is available in many IDEs.



(a) (b) (c)

Fig. 1: (a) The DESYNT history side panel. (b) The *Generate* button for the *user-triggered* view. (c) The VSCode debug tooltip including an accept button on the right.

If a candidate program snippet to replace the hole already exists, DESYNT will evaluate the candidate program and automatically set the value of the hole to be the evaluation result. However, it will still break before the hole, allowing the programmer to examine the result and decide whether the value is correct or if it needs to be changed. The user can keep running with a candidate program for as long as they wish, until the run of the program terminates. If the run ends when there is a candidate program, the user will be asked whether they want to accept the current candidate. If the programmer replaces the value from a candidate snippet, this *invalidates* the candidate, which means there will not be a candidate snippet until synthesis is run again.

Program snippet candidates. DESYNT generates program suggestions by calling a PBE synthesizer. Each input-output pair provided to the synthesizer as specification comprises the state of available variables before the hole's breakpoint as input and the value provided by the user to step over the hole as output. All input-output pairs collected from the beginning of the debug session are stored in an IDE side panel named "DESYNT history" (Fig. 1a), where they can be inspected, and are given to the synthesizer every time it is called, so that the resulting suggestion will be consistent with everything the programmer entered so far.

Once a candidate program is returned by the synthesizer, it replaces the hole in the editor, but is grayed out to denote it is a debug-time value. And, as for any other variable while debugging, the editor will also show the debug-time value of the variable the hole is assigned to at the breakpoint. This allows the user to quickly assess whether the current candidate is correct for the current inputs without having to run it separately in the debug console.

Accepting a snippet. If the programmer is happy with a suggested snippet, they can use an accept button that is added to the VSCode debug tooltip (Fig. 1c). The program is then added to the editor window replacing the hole, and the file is saved. If the user stops the debugger, no change is made to the file even if there is a current candidate.

B. The different views of DESYNT

DESYNT offers two views that differ in when a program is suggested to the user. Their names, *user-triggered* and *triggerless*, are derived from the terms coined by Jayagopal et al. [23].

The* user-triggered *view. When DESYNT is run in *user-triggered* view, a second DESYNT panel is added to the IDE window above "DESYNT history", which contains a *Generate* button (Fig. 1b). No attempt to generate a program will be made until the user presses the *Generate* button, no matter how many iterations have passed.

When the user presses *Generate*, the DESYNT synthesizer is called with every input-output pair accumulated so far, and if a program is found, it is set as the current candidate. Even if the current candidate is invalidated, the synthesizer will not be called again unless the user asks for it explicitly.

The* triggerless *view. When DESYNT is in *triggerless* view, the synthesizer is called automatically. It is first called after the programmer provides three examples. After that, it is called every time the user invalidates the current candidate—since it is called with all accumulated input-output pairs, it is always called with at least three examples.

C. The DESYNT PBE Synthesizer

The synthesizer behind DESYNT is a bottom-up synthesizer based on concrete finite tree automata [46].

The synthesizer runs with a timeout of 15 seconds. This is longer than other interactive synthesizers (e.g., [14]), but in the context of the interaction the extra time will not be felt by the user. The synthesizer is launched once the programmer runs past the breakpoint, but the result is not needed until the program breaks again and the debug view is populated, which can take seconds and sometimes more.

While recent work [29], as well as our own experiments, have shown that LLM-based synthesis from examples struggles with correctness, DESYNT itself does not depend on a specific code-generation system. This means that our symbolic synthesizer can be replaced with a generate-and-verify loop against an LLM or with a more mature model when one is available.

IV. STUDY

We evaluated DESYNT with a 2-hour lab study, with two treatment groups, comparing a *no-tool* control to both the *user-triggered* and *triggerless* views of DESYNT. This research was performed under the oversight of the institutional review board of the authors' home institution.

Participants. We recruited 11 participants (10 M/1 F), all third-year undergraduate students at the authors' home institution. We recruited students out of the "Introduction to AI" course at the authors' home institution as a means to ensuring basic knowledge of Python and debugging. We advertised via course mailing lists and texting groups. Participants were compensated $13/hour. Participants were given a pretest for minimal competency in Python and in using the VSCode debugger. One participant (P4, M) failed the pretest and was excluded from the remainder of the study.

Protocol. We performed a between subjects study with a control condition (*no-tool*) where participants used VSCode with its built-in debugger and no synthesizer, and two experimental conditions, *user-triggered* and *triggerless* for the two views of the tool. Participants were randomly assigned to a condition, using online counterbalancing. Participants in the experimental conditions were shown a video about DESYNT, and all participants were then guided through an example task, where control group participants were guided through debugging into existence, and experimental group participants were guided through debugging into existence and then solving the same task using DESYNT. Participants were then asked to solve three tasks, all carried out in the same order, each with its assigned timeout and with a way for the paticipants to self-validate their solution. Finally, participants were asked to fill out a survey about their experience. Participants in the *user-triggered* or *triggerless* groups were shown a video of the other view of the tool and asked for their opinion about it. The protocol was approved by the ethics committee of the authors' home institution.

Tasks. After a guided training task (deriving an arithmetic expression from examples), participants performed three tasks in this order:

Task 1: Bounded circles (new feature). Participants were given code that visualizes circles moving freely on a canvas, and asked to add a restriction on circles leaving the canvas. Baseline code used an API that was likely unfamiliar to participants, and changes to one frame directly impact the state of the next frame (iteration). The provided code was one file with 70LOC. Task timeout was 20 minutes.

Task 2: VIN—Vehicle Identification Number (fault localization and bug fixing task). Participants were given a codebase that verifies vehicle identification numbers. The codebase included tests for the code, five of which are failing. Not all failures result from the same bug, and some of the code is superfluous to the errors. Participants were asked to locate and fix the code that fails the tests. The provided code was 1921LOC spanning 24 files. Task timeout was 25 minutes.

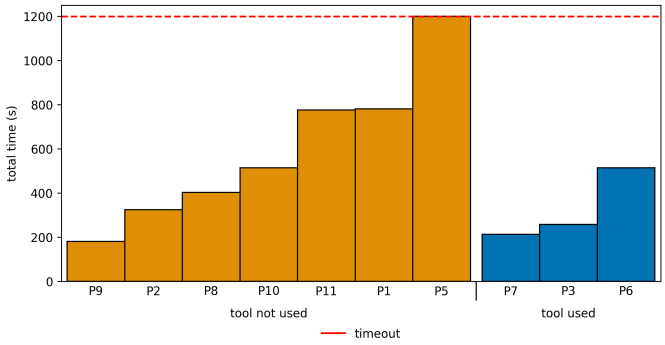
Task 3: Scheduler (reverse engineering). Participants were given a blackbox implementation of a scheduler and asked to recreate it in their code. Specifically, they were given an event ordering supported by the blackbox implementation that they should support. The scheduling loop is dependent as earlier decisions impact later ones. The provided code was two files, totaling 141LOC. Task timeout was 20 minutes.

V. RESULTS

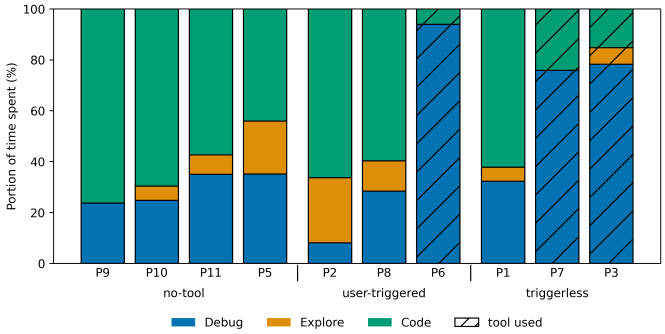
A. Task 1: Bounded circles

Only three of the six participants in the two tool groups (P3, P6, and P7) used DESYNT for the Bounded circles task. Therefore in this task we do not consider the participants in their original groups, combining the three as DESYNT users, regardless of view, and have seven *tool not used* users.

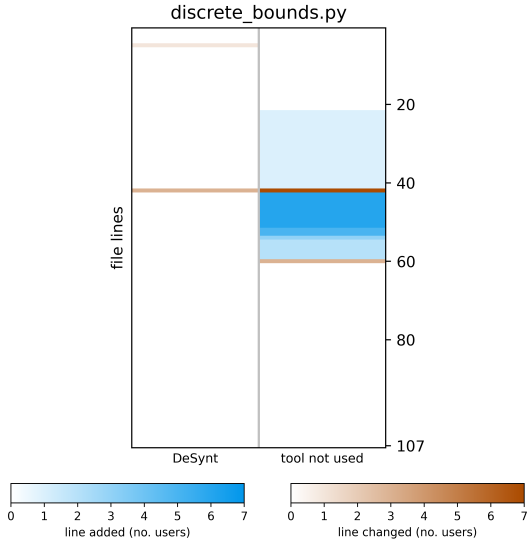
Fig. 2a shows the time to solution for participants when solving the Bounded Circles task. Solving the task without DESYNT took an average of 597s (496s without timeouts), whereas the participants who did use the tool took 212s–513s.



(a) Time to complete the Bounded Circles task. Because of low tool usage in this task, we consider participants as tool used/not used rather than their original study groups.



(b) Portion of the participant sessions spent exploring, writing code, and debugging.



(c) How many participants edited each file line (files are aligned). `discrete_bounds.py` is the only file in the task.

Fig. 2: Results for Task 1: Bounded Circles

One participant from the control group timed out.

The provided code for this task used `numpy`, DESYNT participants all solved the task with a single line of vectorized `numpy` code, while those that did not use the tool added the new feature using `for` loops. There were a number of *no tool used* participants (P10, P12) that commented about knowing there was a better solution but not having the time or understanding to write it.

We coded participant sessions into three activity types, *exploration*, *code editing*, and *debugging*, where using DESYNT is considered debugging, as it takes place in debug time. Fig. 2b shows the percentage of each session spent in each activity type.

In the Bounded Circles task the portion of the time spent exploring was low, and the majority was divided between code editing and debugging. As Fig. 2b shows, tool use is the determining factor as to where the majority of the participant's time was spent. Participants that used DESYNT spent a far larger amount of time debugging when compared with those that did not; the DESYNT user with the smallest fraction of time debugging debugged twice as much, proportionally, compared to the *no tool used* participant with the largest fraction debugging.

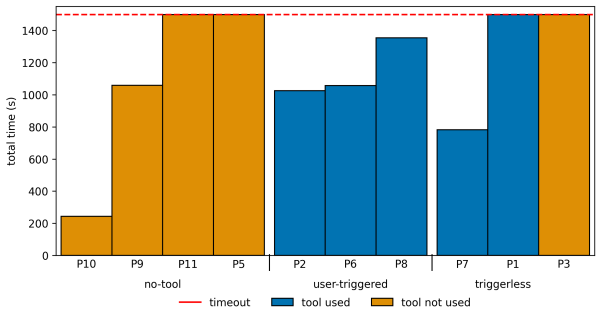
In order to understand the locality of changes made by participants, we counted the number of users that edited each line. We assigned new line numbers to align all versions of each file as follows: as in a line-based diff, any insertion moved both sides by the number of lines inserted; when aligning all participant tasks to each other, this moved the next original line number by the size of the maximum insertion at that location. This resulted in a line numbering where each line could be either changed (i.e., an original line) or added (i.e., an inserted line), but never both. The number of users that edited each line is shown in Fig. 2c.

In the Bounded Circles task, DESYNT users added no new lines, with almost all changes being localized to a single location, using `numpy` vector operations. Users without the tool, on the other hand, inserted a number of lines before and after that line; these are `for` loops with varying lengths of loop bodies.

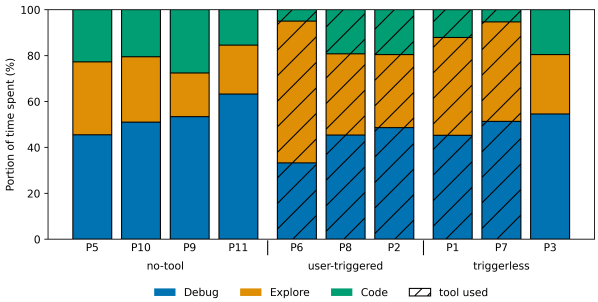
B. Task 2: VIN

Time to complete the VIN task is shown in Fig. 3a. Five out of six participants in the experimental groups used DESYNT in this task. One tool user (with the *triggerless* view) timed out, as did two *no-tool* participants and the one participant in the *triggerless* group who did not use the tool. On average, DESYNT users completed the task in 1144s, (1055s when excluding participants who timed out). Participants who did not use the tool took 1161s on average, (651s excluding timeouts). The fastest time to completion—just over four minutes, and almost 10 minutes faster than any other participant—was a *no-tool* user.

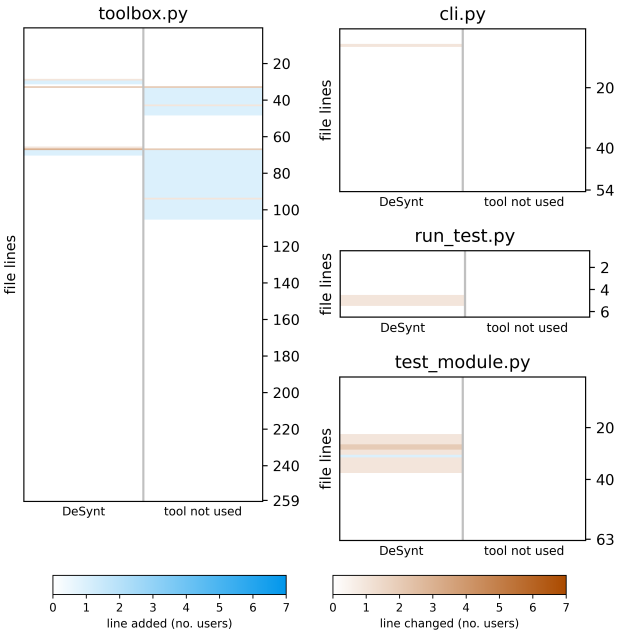
This task involved fixing a bug in a multi-file codebase. This impacts the portion of the time spent exploring the code, as seen in Fig. 3b: this was the task where participants spent



(a) Time to complete the VIN task.



(b) Portion of participant sessions spent exploring, writing code, and debugging.



(c) How many participants edited each file line (files are aligned). These four files were the only ones edited out of 24 files provided with the task.

Fig. 3: Results for Task 2: VIN.

the largest portion of their time time exploring, regardless of their group and of whether they used DESYNT or not.

Participants that used DESYNT spent more time exploring compared to participants that did not use DESYNT. This is likely correlated with DESYNT participants having trouble with the tool in this task: only 2 of the DESYNT users placed

the hole (??) in a useful place in the code.

Two *no-tool* participants (P5, P9) not familiar with the `strip` functionality in Python used a **`for`** loop to remove whitespaces, while no DESYNT users did this.

Moreover, the heatmap of file modification locations (Fig. 3c) show that both users with and without DESYNT made more widespread modifications. Participants without DESYNT edited the target file, `toolbox.py`, in more locations. Interestingly, DESYNT users interacted with a number of other files, while participants without the tool users did not. These were test files, and the modifications were specifically to better run DESYNT, e.g., modifying `test_module.py` to run specific tests and reach the breakpoint with relevant data faster (P8) or changing the order of tests (P2).

C. Task 3: Scheduler

Fig. 4a shows times to complete the Scheduler task. All participants given DESYNT used it in this task. Moreover, this task had the most stark difference between participants with and without DESYNT; participants with DESYNT completed the task in 349s on average with no timeouts, whereas all *no-tool* participants timed out.

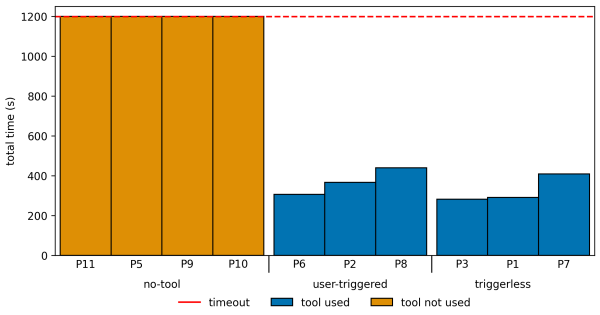
Fig. 4b shows that participants in the *no-tool* group spent a larger portion of their time writing code than DESYNT participants, but perhaps more importantly, they spent a smaller portion of their time debugging; the largest portion spent debugging in the *no-tool* group (P5) is still less than the least for a DESYNT user (P7).

DESYNT users modified the code in fewer locations, compared to *no-tool* users (Fig. 4c). Code additions performed by DESYNT users were to facilitate their use of DESYNT; specifically, P6 and P8 used **`print`** and `assert` statements used to ensure the generated code satisfied the desired behavior. *no-tool* users added a larger number of lines compared to DESYNT users. Additionally, P10 changed the `jobs.json` file that defines the data read by the code in order to better understand the role of each parameter in the function they reverse engineered.

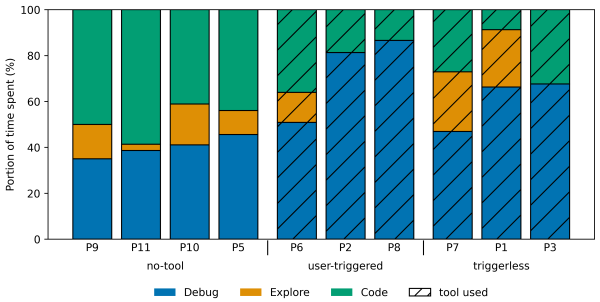
D. Additional results

The left side of Fig. 5 shows the number of examples users provided before synthesis. Participants using the *user-triggered* view provided more examples before synthesis for all tasks compared to *triggerless* users, where synthesis was always attempted after three examples. In the *triggerless* view one synthesis call is indicated as happening with four examples; this is the only case in our study where synthesis ran after three examples and the user immediately invalidated the candidate snippet with an additional example, which launched synthesis again. One participant in the *user-triggered* view (P8) worked in short bursts of two examples for most of the study. The scheduler had the largest number of examples provided before synthesis while, on average, VIN had the least.

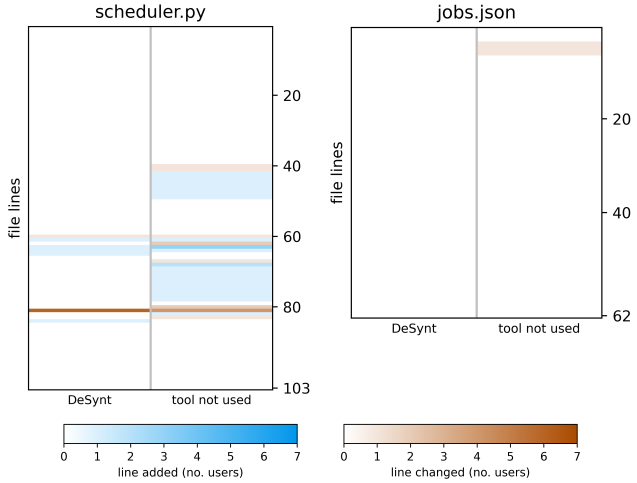
The right side of Fig. 5 shows the number of validation iterations participants performed after synthesis and before accepting a snippet, where a validation iteration consists of the



(a) Time to complete the Scheduler task.



(b) Portion of participant sessions spent exploring, writing code, and debugging.



(c) How many participants edited each file line (files are aligned). These two files are the only ones provided with the tasks.

Fig. 4: Results for Task 3: Scheduler.

user checking that the synthesized snippet produces the desired result for the variable values at the breakpoint. The trend here is similar to the examples provided: although *triggerless* users had no set number of validation examples they needed to provide, on average they provided less across all tasks apart from VIN. Additionally, in *user-triggered* users we see large variance across all tasks.

E. Post-study survey

After completing the tasks, participants from the treatment groups (*user-triggered* and *triggerless*, 6 participants) filled

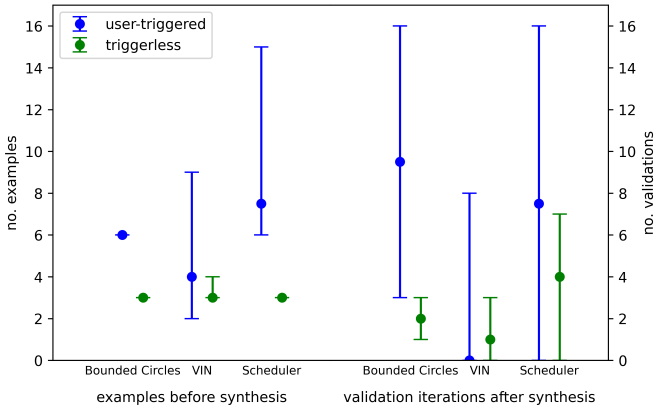


Fig. 5: Left: Minimum, maximum, and medians of the number of examples provided before every synthesis event in each task. Since the *triggerless* view of DESYNT synthesizes first at three examples, this is the median, but one user changed the suggested output after synthesis in VIN, raising the maximum to four. Right: Minimum, maximum, and medians of the number of times the participant continued running to the breakpoint to validate the result.

out a survey about their experience, which had 17 statements to rank from "strongly disagree" to "strongly agree". The questions and the distribution of answers are shown in Tab. I.

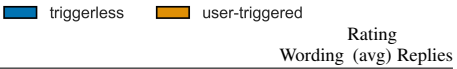
In general, DESYNT was well-received by users, who indicated for both views that the interaction with DESYNT provided users with the means to explore and understand solutions easily, helped users understand complex behavior and generated easy to understand code. Furthermore, DESYNT helped users better translate their mental model.

The responses to two sets of questions stand out. First, in both positive and negative wording of the statement, participants rated the ease of entering values approximately neutral.

Second, the questions indicating trust in DESYNT had a mixed response. Users with the *user-triggered* view trusted the suggested code less than *triggerless* participants, even though all participants indicated that they understood suggested snippets well.

Finally, *triggerless* participants indicated that the amount of examples before the synthesizer is launched is not intuitive. This is shown in the survey, and P3 also indicated it in their additional verbal comments after the survey.

TABLE I: Post study survey of participants. Similar questions that were asked both negatively and positively are grouped regardless of their order in the survery.



Question	Wording	Rating (avg)	Replies
DESYNT was easy to use.	Positive	4.0	
DESYNT was difficult to use.	Negative	2.0	
Code suggestions are well integrated in the IDE.	Positive	4.5	
DESYNT helped me to understand complex behavior.	Positive	4.8	
DESYNT helped me understand how to translate expected behavior into code.	Positive	4.3	
It was not easy to enter values as examples.	Negative	2.5	
Entering values as examples was easy.	Positive	3.7	
Placing a hole with ?? was easy.	Positive	4.8	
Placing a hole with ?? was difficult.	Negative	1.2	
The number of examples required to get a code suggestion aligned with my intuition.†	Positive	3.3	
DESYNT's functionality is well integrated in the IDE.	Positive	4.8	
Code suggestions were easy to understand in the context of the program.	Positive	4.5	
DESYNT is a useful addition to the methods I use for software development.	Positive	3.5	
I trusted code snippets that I accepted to fulfill my intent.	Positive	3.7	
I understood suggested code snippets.	Positive	4.8	
Verifying a suggested code snippet was easy.	Positive	4.3	
DESYNT is a useful addition to my debugging arsenal.	Positive	3.7	

†only asked of participants in the *triggerless* group.

VI. DISCUSSION

A. DESYNT's utility depends on type of task

Our results hint that the type of task has an impact on how useful DESYNT is. Like many other program synthesis tools, DESYNT thrives as a tool for *API discovery* [12], [22], i.e., in cases where the functionality is simple but the implementation is difficult or uses an unfamiliar API. This was especially relevant in the Bounded Circles task, where P6 and P7 both noted that they would not have produced a solution without the tool. Furthermore, P10 and P2, who both solved the task without DESYNT and used a loop, remarked that they felt there was a better solution that they did not know how to write.

It is also apparent that DESYNT's utility depends on the type of example the user is required to enter; P3 pointed out that a problem with DESYNT is the need to enter a textual representation of outputs, which might be objects. In the post-study survey (Tab. I), participants on average scored the ease of entering examples as fairly low, as well. This problem shows up in existing testing tools [1], [32] as well as in code generation tools [14], [15].

For *user-triggered* participants, who could provide any number of examples before synthesizing, that number appears to be affected by the difficulty of the task, with difficulty

measured by the success rate of the *no-tool* group. Fig. 5 shows that participants in the *user-triggered* group provided fewer examples in VIN task, and more in Bounded Circles and Scheduler tasks. Similar dynamics are also present in the number validation iterations a participant performed before accepting a candidate solution. There does appear to be a discrepancy on the Bounded Circles task, which has higher examples and validation iterations than VIN even though *no-tool* users faired better in Bounded Circles. We believe this is a result of the localization required in VIN, with the code required to solve Bounded Circles being more complex then in VIN.

Linking to this discrepancy, an interesting hypothesis stemming from our results is that DESYNT's utility depends on knowing where the new code will go; this was pointed out by participants, e.g., P7. Tasks that have a large fault localization component, e.g., VIN, are ones where DESYNT made no significant difference, whereas in Bounded Circles and Schedulers DESYNT let participants make very local edits and insertions and have far better outcomes in completing the task.

B. *Using* DESYNT *correlates with debugger use*

We find that when working with DESYNT users spent a larger portion of their time in the debugger, this is simply the result of DESYNT running inside the debugger. However, for users whose normal development style heavily relies on the debugger, DESYNT's additional functionality of running partial programs, on top of debug console code inspecting variables, means they are able to stay in the debugger more than they could without DESYNT. This is consistent with the varying responses to the survey questions "DESYNT is a useful addition to the methods I use for software development" and "DESYNT is a useful addition to my debugging arsenal" (Tab. I).

Further work is needed to understand how effective DESYNT is for users with differing development methods. We speculate that DESYNT enhances the developer experience when the task requires implementing functionality that does not easily translate to a clear mental model of the required code.

C. *Differences between* user-triggered *and* triggerless

The view of the tool participants were assigned to does not affect participants' success as much as other factors, e.g., task type. However, there are apparent differences based on the differences in the interaction model.

The most notable differences between the two views are seen in Fig. 5. *user-triggered* users provided more examples before triggering synthesis when compared to the fixed three of *triggerless*—across all tasks more than three examples were provided on average before synthesizing. This is consistent with the results of Jayagopal et al. [23], who found users often provide much larger specifications than are needed to complete the task.

Additionally, *user-triggered* users also used more verification steps. This appears to correlate with the difference in trust of the synthesized snippet between *user-triggered* and *triggerless* users seen in Tab. I. Interestingly, this suggests that when given control, users trust their choice less than when choices are made for them.

This may also provide a reason for the paradoxical answers in Tab. I, where, for *user-triggered* users, generated snippet understandability and verifiability are high but trust is low. This could be because in more difficult problems users may have been unsure that the number of provided examples covered corner cases and hence, when put in control and with an uncertain mental model, understandability and verifiability do not translate to trust.

On the other hand, for *triggerless* users we find that, although trust is high, the number of example required did not effectively align with the participants intuition. This appears to be linked with subsection VI-A; for easier problems, three examples may be sufficient while for harder tasks users may prefer to provide more examples to ensure they have captured the complete behavior. However, users still place a large amount of trust in DESYNT, when it decides when to generate.

D. *Classification of* DESYNT *as a synthesis interaction model*

Jayagopal et al. [23] introduced a framework for understanding program synthesis (code generation) interaction models with three axes in the design space: i) whether specifications are *incidental* or *voluntary*, ii) whether the initiation of synthesis is *user-triggered* or *triggerless*, and iii) whether the communication of results to the user is *user-triggered* or *triggerless*. In Table 3, they show the matrix of options for (ii) and (iii), noting that the quadrant for user-triggered initiation and triggerless result communication is "empty by construction", as triggering the initiation of synthesis implies triggering the communication of results. We argue, however, that DESYNT ostensibly fits in the missing quadrant: while setting a sketch and running the debugger is an initiation of the synthesis "mode" of DESYNT, when in its *triggerless* view, the decision to run the synthesizer behind the scenes is not actively triggered, and a result will only be communicated to the programmer when one is available.

Interestingly, once participants in the *triggerless* group gained some experience with DESYNT, they would wait for a synthesis event to happen after hitting the breakpoint three times and entering output values; reminiscent of the triggered result communication interaction model. This suggests that, when a tool behaves predictably, it "moves" to a *user-triggered* result communication.

However, this may change with a different synthesis engine: in our current engine, synthesis cannot fail with three examples then succeed with four examples, so if a program does not appear after three examples, one will not appear later in the session. A different code generation backend may behave differently, yielding different results.

VII. Related Work

A. Exploratory Programming

Exploratory programming began as a description of programs in AI research where there is no specific intended output, but the result can be "eyeballed" by the programmer [42]. It has since been extended to more generally describe cases where programmers start writing code before its behavior is fully specified, thinking about their code *as they write it* [26], [40], [41], [44].

Several specialized tools that target specific exploratory behaviors have been suggested, with a special focus on allowing the programmer to backtrack or to maintain multiple versions of the code that can be switched to efficiently [9], [24], [25], [34], [47].

B. Debugging into Existence

Debugging into Existence was identified by Rosson and Carroll [39] as the dynamic and incremental implementation process that stems from edits driven by testing and debugging. It is also weakly related to *Babylonian-style Programming* [38] that helps users view live examples for smaller parts of the code, allowing them to examine (and then edit) an inner part of the code directly.

Debugging into Existence has been criticized [49] as a an ineffective bug-fixing method, in particular in regards to fault localization. This criticism is in accordance with our results: the underlying workflow to DeSynt worked well when the location of new code or broken code was known, but not when actual, manual fault localization had to take place.

C. Interaction models for code generation

Code generation tools for programmers take many different approaches. Copilot [16] suggests potential program snippets that can be tab-completed into the code editor based on only textual file context. The exploratory nature of Copilot use was explored by Barke et al. [8], and was further highlighted by CoLadder [48] and its interaction focused on refining intent and creating sub-goals.

Several code generation tools [37], [50] use an iterative refinement loop where more specifications are provided to rule out intermediate program candidates.

SnipPy [15], LooPy [14], and Leap [13] are a family of tools for code generation within a *live programming* environment—an environment that provides continuous feedback on the state of the code as the code is being written—and assisting the programmer in formulating specifications as well as in validating the code generation result.

The closest work to ours is CodeHint [17], where the programmer sets a breakpoint in the code, debugs to it, then uses types and examples to demonstrate the intended value. CodeHint also uses an observational equivalence synthesis engine with variable valuations at the breakpoint as inputs. However, CodeHint does not alleviate the issue of access to dependent calls to the same code location: a user of CodeHint can run the code again with different inputs, but it does not continue the execution with the user provided values in the way that Hazel [33], [36] and LooPy do.

D. Programming by Example

Programming by Example (PBE) is a paradigm of program synthesis where intent is specified by pairs of input to output. FlashFill [18], [19] is a PBE synthesizer integrated into Microsoft Excel, where inputs are provided by table rows and outputs by values added manually to a new column. The FlashFill synthesis engine is powered by Version Space Algebra, and in FlashFill++ [10] is improved with guarded DSLs. Another common search method for PBE is Observational Equivalence [6], [45], and its implementation using Finite Tree Automata [46] is the one used by DeSynt.

PBE has been used for domain specific programs: Excel formulas in `FlashFill` and regular expressions in tools like Regel [11] and Regae [50]. Other synthesizers [14], [15], [17], [37] target general programming, mainly centring on primitive types as they are easier to provide examples for.

E. Interaction Models for Debugging

Interaction model work for debugging is mainly centered around aiding comprehension and fault localization, i.e., the hypothesis generation phase of debugging [5], [49], helping the programmer better utilize debug-time information such as execution traces, variable values, and location in the call stack. The WhyLine [27], [28] allows the programmer to ask "why" and "why not" questions about the state of the program at a breakpoint, including about the history of the execution trace. Microbat [31] recommends suspicious steps for the user to manually inspect, and Enlighten [30] uses statistical fault localization to improve the results of a similar process. Hypothesizer [5] uses a database of hypotheses and tries to match them with demonstrated faulty behavior. Robin [7] employs an LLM agent that has access to the debugger to perform assisted "conversational debugging". UNFOLD [21] lifts the debugging experience into *live programming*, to help programmers track buggy states in event-driven UI code.

Unlike these, DeSynt does not center on fault localization, and as shown in Sec. V, is in fact less useful when mixed with tracking a fault. CodeHint [17], discussed above, is the most similar to DeSynt in employing the debugger mainly as an inspector in a development task, but DeSynt's ability to access the code multiple times throughout a single run makes it a tool for inspection and comprehension even if code generation is not employed.

References

[1] Accessed 2025/05/11. [Online]. Available: https://hypothesis.works/

[2] "debug-here 0.1.0 - docs.rs," https://docs.rs/crate/debug-here/0.1.0, docs version 0.6.0 (17de230 2022-01-09).

[3] "Hardcover," accessed 2025/03/26. [Online]. Available: https://hardcover.app/

[4] "Railsconf 2014 - debugger driven developement with Pry by Joel Turnbull," https://www.youtube.com/watch?v=4hfMUP5iTq8, 2014.

[5] A. Alaboudi and T. 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*, 2023, pp. 1–14. [Online]. Available: https://www.doi.org/10.1145/3586183.3606781

[6] A. Albarghouthi, S. Gulwani, and Z. Kincaid, "Recursive program synthesis," in *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*. Springer, 2013, pp. 934–950. [Online]. Available: https://www.doi.org/10.1007/978-3-642-39799-8_67

[7] Y. Bajpai, B. Chopra, P. Biyani, C. Aslan, D. Coleman, S. Gulwani, C. Parnin, A. Radhakrishna, and G. Soares, "Let's fix this together: Conversational debugging with GitHub Copilot," in *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2024, pp. 1–12. [Online]. Available: https://www.doi.org/10.1109/VL/HCC60511.2024.00011

[8] S. Barke, M. B. James, and N. Polikarpova, "Grounded Copilot: How programmers interact with code-generating models," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, Apr. 2023. [Online]. Available: https://doi.org/10.1145/3586030

[9] T. Beckmann, J. Bergsiek, E. Krebs, T. Mattis, S. Ramson, M. C. Rinard, and R. Hirschfeld, "Probing the design space: Parallel versions for exploratory programming," *Art Sci. Eng. Program.*, vol. 10, no. 1, 2025. [Online]. Available: https://doi.org/10.22152/programming-journal.org/2025/10/5

[10] J. Cambronero, S. Gulwani, V. Le, D. Perelman, A. Radhakrishna, C. Simon, and A. Tiwari, "FlashFill++: Scaling programming by example by cutting to the chase," *Proceedings of the ACM on Programming Languages*, vol. 7, no. POPL, pp. 952–981, 2023. [Online]. Available: https://www.doi.org/10.1145/3571226

[11] Q. Chen, X. Wang, X. Ye, G. Durrett, and I. Dillig, "Multi-modal synthesis of regular expressions," in *Proceedings of the 41st ACM SIGPLAN conference on programming language design and implementation*, 2020, pp. 487–502. [Online]. Available: https://www.doi.org/10.1145/3385412.3385988

[12] Y. Feng, R. Martins, Y. Wang, I. Dillig, and T. W. Reps, "Component-based synthesis for complex apis," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017, pp. 599–612.

[13] K. Ferdowsi, R. Huang, M. B. James, N. Polikarpova, and S. Lerner, "Validating AI-generated code with live programming," in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–8. [Online]. Available: https://www.doi.org/10.1145/3613904.3642495

[14] K. Ferdowsifard, S. Barke, H. Peleg, S. Lerner, and N. Polikarpova, "LooPy: interactive program synthesis with control structures," *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–29, 2021. [Online]. Available: https://www.doi.org/10.1145/3485530

[15] K. Ferdowsifard, A. Ordookhanians, H. Peleg, S. Lerner, and N. Polikarpova, "Small-step live programming by example," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, 2020, pp. 614–626. [Online]. Available: https://www.doi.org/10.1145/3379337.3415869

[16] N. Friedman, "Introducing GitHub Copilot: Your AI pair programmer," Feb 2022, accessed 2025/05/11. [Online]. Available: https://github.blog/news-insights/product-news/introducing-github-copilot-ai-pair-programmer/

[17] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen, "CodeHint: Dynamic and interactive synthesis of code snippets," in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 653–663. [Online]. Available: https://www.doi.org/10.1145/2568225.2568250

[18] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," *ACM Sigplan Notices*, vol. 46, no. 1, pp. 317–330, 2011. [Online]. Available: https://www.doi.org/10.1145/1925844.1926423

[19] S. Gulwani, W. R. Harris, and R. Singh, "Spreadsheet data manipulation using examples," *Communications of the ACM*, vol. 55, no. 8, pp. 97–105, 2012. [Online]. Available: https://www.doi.org/10.1145/2240236.2240260

[20] S. Gulwani, O. Polozov, R. Singh *et al.*, "Program synthesis," *Foundations and Trends® in Programming Languages*, vol. 4, no. 1-2, pp. 1–119, 2017. [Online]. Available: https://www.doi.org/10.1561/2500000010

[21] R. L. Huang, P. J. Guo, and S. Lerner, "UNFOLD: Enabling live programming for debugging GUI applications," in *2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2024, pp. 306–316. [Online]. Available: https://www.doi.org/10.1109/VL/HCC60511.2024.00041

[22] M. B. James, Z. Guo, Z. Wang, S. Doshi, H. Peleg, R. Jhala, and N. Polikarpova, "Digging for fold: synthesis-aided api discovery for haskell," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.

[23] D. Jayagopal, J. Lubin, and S. E. Chasins, "Exploring the learnability of program synthesizers by novice programmers," in *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3526113.3545659

[24] M. B. Kery, "Tools to support exploratory programming with data," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2017, pp. 321–322. [Online]. Available: https://www.doi.org/10.1109/VLHCC.2017.8103490

[25] M. B. Kery, A. Horvath, and B. A. Myers, "Variolite: Supporting exploratory programming by data scientists." in *CHI*, vol. 10, 2017, pp. 3 025 453–3 025 626. [Online]. Available: https://www.doi.org/10.1145/3025453.3025626

[26] M. B. Kery and B. A. Myers, "Exploring exploratory programming," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2017, pp. 25–29. [Online]. Available: https://www.doi.org/10.1109/VLHCC.2017.8103446

[27] A. J. Ko and B. A. Myers, "Designing the Whyline: a debugging interface for asking questions about program behavior," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2004, pp. 151–158. [Online]. Available: https://www.doi.org/10.1145/985692.985712

[28] ——, "Debugging reinvented: asking and answering why and why not questions about program behavior," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 301–310. [Online]. Available: https://doi.org/10.1145/1368088.1368130

[29] W.-D. Li and K. Ellis, "Is programming by example solved by LLMs?" *arXiv preprint arXiv:2406.08316*, 2024. [Online]. Available: https://www.doi.org/10.48550/arXiv.2406.08316

[30] X. Li, S. Zhu, M. d'Amorim, and A. Orso, "Enlightened debugging," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 82–92. [Online]. Available: https://doi.org/10.1145/3180155.3180242

[31] Y. Lin, J. Sun, Y. Xue, Y. Liu, and J. Dong, "Feedback-based debugging," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 393–403. [Online]. Available: https://www.doi.org/10.1109/ICSE.2017.43

[32] Y. Liu, P. Nie, O. Legunsen, and M. Gligoric, "Inline tests," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3551349.3556952

[33] J. Lubin, N. Collins, C. Omar, and R. Chugh, "Program sketching with live bidirectional evaluation," *Proceedings of the ACM on Programming Languages*, vol. 4, no. ICFP, pp. 1–29, 2020. [Online]. Available: https://www.doi.org/10.1145/3408991

[34] T. Mattis, P. Rein, and R. Hirschfeld, "Edit transactions: Dynamically scoped change sets for controlled updates in live programming," *Art Sci. Eng. Program.*, vol. 1, no. 2, p. 13, 2017. [Online]. Available: https://doi.org/10.22152/programming-journal.org/2017/1/13

[35] Microsoft, "Visual Studio Code," https://github.com/microsoft/vscode, 2015, accessed: 2025-05-05.

[36] C. Omar, I. Voysey, R. Chugh, and M. A. Hammer, "Live functional programming with typed holes," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–32, 2019. [Online]. Available: https://www.doi.org/10.1145/3290327

[37] H. Peleg, R. Gabay, S. Itzhaky, and E. Yahav, "Programming with a read-eval-synth loop," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020. [Online]. Available: https://www.doi.org/10.1145/3428227

[38] D. Rauch, P. Rein, S. Ramson, J. Lincke, and R. Hirschfeld, "Babylonian-style programming," *The Art, Science, and Engineering of Programming*, vol. 3, no. 3, pp. 9–1, 2019. [Online]. Available: https://www.doi.org/10.48550/arXiv.1902.00549

[39] M. B. Rosson and J. M. Carroll, "The reuse of uses in Smalltalk programming," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 3, no. 3, pp. 219–253, 1996. [Online]. Available: https://www.doi.org/10.1145/234526.234530

[40] D. W. Sandberg, "Smalltalk and exploratory programming," *ACM Sigplan Notices*, vol. 23, no. 10, pp. 85–92, 1988. [Online]. Available: https://www.doi.org/10.1145/51607.51614

[41] M. Shaw, "Myths and mythconceptions: What does it mean to be a programming language, anyhow?" 2021, keynote talk: HOPL IV: History of Programming Languages. [Online]. Available: https://www.pldi21.org/prerecorded_hopl.K1.html

[42] B. Sheil, "Environments for exploratory programming," *Datamation*, vol. 29, no. 2, pp. 131–144, 1983.

[43] A. Solar-Lezama, "Program sketching," *Int. J. Softw. Tools Technol. Transf.*, vol. 15, no. 5-6, pp. 475–495, 2013. [Online]. Available: https://doi.org/10.1007/s10009-012-0249-7

[44] M. Taeumel, P. Rein, and R. Hirschfeld, "Toward patterns of exploratory programming practice," *Design Thinking Research: Translation, Prototyping, and Measurement*, pp. 127–150, 2021. [Online]. Available: https://www.doi.org/10.1007/978-3-030-76324-4_7

[45] A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. Martin, and R. Alur, "TRANSIT: specifying protocols with concolic snippets," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 287–296, 2013. [Online]. Available: https://www.doi.org/10.1145/2499370.2462174

[46] X. Wang, I. Dillig, and R. Singh, "Synthesis of data completion scripts using finite tree automata," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–26, 2017. [Online]. Available: https://www.doi.org/10.1145/3133886

[47] N. Weinman, S. M. Drucker, T. Barik, and R. DeLine, "Fork It: Supporting stateful alternatives in computational notebooks," in *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, ser. CHI '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3411764.3445527

[48] R. Yen, J. S. Zhu, S. Suh, H. Xia, and J. Zhao, "CoLadder: Manipulating code generation via multi-level blocks," in *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*, 2024, pp. 1–20. [Online]. Available: https://www.doi.org/10.1145/3654777.3676357

[49] A. Zeller, *The Debugging Book*. CISPA Helmholtz Center for Information Security, 2024, ch. Introduction to Debugging, retrieved 2024-07-01 16:49:37+02:00. [Online]. Available: https://www.debuggingbook.org/

[50] T. Zhang, L. Lowmanstone, X. Wang, and E. L. Glassman, "Interactive program synthesis by augmented examples," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, 2020, pp. 627–648. [Online]. Available: https://www.doi.org/10.1145/3379337.3415900