Programming by Partial Examples

Kasra Ferdowsifard*
kferdows@eng.ucsd.edu
University of California, San Diego
La Jolla, California, USA

Ruanqianqian Huang*
r6huang@ucsd.edu
University of California, San Diego
La Jolla, California, USA

ABSTRACT

A common and intuitive user-synthesizer interaction model (IM) is Programming-by-Example, where the user provides output examples to specify the desired program's behavior on the inputs. However, providing large examples can be tedious. This paper presents Programming by Partial Examples (PBPE), a new IM where the user can provide partial output specifications which are completed by the synthesizer. We pose the following research question: *Would a PBPE synthesizer be more usable than one that requires full outputs?* A pilot study (N=4) on PopPy, a synthesizer that adopts PBPE, shows that it can be more usable because (1) partial output specifications were used extensively and did not increase confusion, and (2) users spent less time per synthesis call providing specifications with PopPy. Results from the pilot study also suggest refinement to the study design.

1 INTRODUCTION

Programming-by-Example (PBE) is a program synthesis paradigm where the user describes the intended program behavior through a set of input-output examples (*"specifications"*). The goal for the synthesizer is to find programs that successfully transform the inputs in each example into the corresponding output. Compared to other synthesis techniques, PBE is relatively low threshold [9] as it provides a more intuitive way of expressing user intent and does not require user understanding of the synthesizer.

User Driven Interaction [4] is the most common and intuitive user interaction model (IM) for PBE, where the user creates input-output examples and examines the program produced by the synthesizer. While this is a great IM, it does not scale well in the size of inputs and outputs, as manually constructing (large) examples can be time-consuming and error-prone [6]. Live Programming by Example (LPBE) synthesizers [3, 12, 13] incorporate Live Programming, a paradigm that constantly displays runtime values of a program, to mitigate the issue by reusing existing runtime values from the interface as inputs for each example, so that the user only needs to provide the outputs. Although such tools reduce the quantity of examples the user needs to provide, they do not assist users with writing complete, long outputs.

The requirement of complete specifications and the typo-prone nature of constructing long examples could exacerbate the *user-synthesizer gap* [3]: it is beyond the synthesizer's ability to distinguish specifications with typos from the rest and typos could lead to incomprehensible programs or synthesis failures, whereas the user might overestimate the synthesizer's ability to detect and recover from typos in specifications. As Ferdowsifard et al. [3] found in their user study on SnipPy, an LPBE synthesizer targeting general programming, incorrect mental models of the synthesizer restrained users from trying to understand the synthesized code, let alone confirming whether the synthesis results satisfied their input-output specifications.

Typos exist not only in program synthesis specifications but in all sorts of human inputs to computing systems. To address the problem, word completion was developed to complete the user input with a few suggestions as the user types in the initial part of a query [5] and can reduce the number of keystrokes and increase user engagement with the input system [1, 7]. This paper explores the possibility of leveraging word completion in PBE such that the synthesizer could accept and complete *partial* examples.

Prior studies reveal that partial examples might help bridge the user-synthesizer gap in PBE. Peleg and Polikarpova [11] developed a best-effort synthesis paradigm, Bester, that could return partially-valid programs using only the flawless part of the user-written specifications, and they found that users used such results for comprehension even when they were not completely valid. In fact, considerations for incorporating such user-synthesizer IMs based on partial input-output specifications into PBE tools have been already offered [8]. Wang et al. [14] also adopted partial input-output specifications in Viser, a synthesizer for data visualizations and corresponding scripts, by allowing users to specify part of the expected visualization outputs. Similarly, Wrex [2] synthesizes data wrangling code that applies to all rows of a data frame using a small number of example rows (part of the data frame), though its IM requires providing more example rows to synthesize different programs. As Viser and Wrex are both *domain-specific* PBE tools for non-programmers, what remains to be explored is whether a similar IM for general programming synthesis like SnipPy [3] would improve users' understanding and use of the synthesizer.

In this paper, we present Programming by Partial Examples (PBPE), a new IM where the user can provide partial output specifications which are completed by the synthesizer. We investigate whether a PBPE synthesizer could improve the specifications writing process and hence the overall usability of the synthesizer: *Would a PBPE synthesizer be more usable than one that requires full outputs?* We implemented PBPE in PopPy, an extension to SnipPy [3] that accepts partial output specifications. Through a between-subjects pilot study with four participants, we compared PopPy to the same tool without the partial output specification feature, and demonstrate that partial output specifications were used extensively and did not increase confusion, and users spent less time per synthesis call writing examples with partial output specifications. Our pilot study also suggests refinement to the design of a future large-scale study, including a quantitative assessment during recruitment and non-remote manipulation of the programming environment.

The main contributions of this paper are:

*Both authors contributed equally to this research.

(1) A novel user-synthesizer interaction model called Programming by Partial Examples, where the synthesizer accepts and completes partial output specifications written by the user.
(2) An implementation of this IM in a tool called PopPy, which to our knowledge is the first attempt to support partial outputs in the specifications for general-purpose inductive synthesis.
(3) A pilot study of PopPy on 4 programmers, which found that PopPy was extensively used, did not increase confusion, and reduced time spent in writing examples per synthesis call.

2 POPPY

2.1 Example Usage Scenario

A programmer named Alex is cleaning up a text file using Python, as a part of which they need to remove extra spaces and fix capitalization in each sentence in the file, e.g. going from "A strANGE sTRING" to "a strange string". Alex is an experienced programmer, so they quickly think of a solution: Split each sentence by whitespace, join the words back with single spaces, and make each word lowercase in the process. Unsure of how to do so in Python, a language new to them, Alex turns to PopPy to write the program.

They define a function `cleanup` with one parameter `line` and call it using the sentence above as the argument. They immediately see Projection Boxes appear in the function. Alex first needs to split the `line` argument into words, so they write `words = `. Since they don't know how to do so in Python, they write `??` instead of code (Fig. 1a). This places their cursor *inside* the box, where they can now enter the *value* `words` should have instead of the code. Alex is worried about mistyping one of the oddly capitalized words, so they write the list and its first element but leave the rest as `···` (Fig. 1b).

Once they press Enter they see Fig. 1c: the specification was accepted and PopPy is looking for matching programs. After a few seconds, they see that the message has changed, and the box is updated with the output in Fig. 1d. This is the output Alex had in mind, so they press Enter once more to accept it. This inserts the matching code and returns the cursor to the editor (Fig. 1f).

To finish this function definition, Alex needs to put single spaces between each word and turn them all lowercase, so they turn to PopPy again and see if it can do both in one line. After writing `return ??`, they provide a partial output again (Fig. 1h); however, the first available output `'a  strange string'` that appears in the box has an extra space and is not what they had in mind. They press → to see the next output and after a few outputs, they reach their goal `'a strange string'`. They again press Enter to accept it and get the final program:

```
def cleanup(line):
    words = line.split()
    return " ".join(words).lower()
```

2.2 Implementation

We implemented PopPy as an extension of SnipPy [3], which provided the framework for small-step live programming by example. To implement the new IM, we extended the user interface with the ability to display and select outputs, and extended the synthesizer with an optional new predicate for partial synthesis.

2.2.1 Enumerative Synthesis with Partial Specifications. A detailed description of the synthesis algorithm is beyond the scope of this paper. In brief, in regular Programming-by-Example using Bottom-up Enumerative Synthesis with Observational Equivalence, the synthesizer is presented with a list of input-output examples. It then enumerates programs and evaluates each program using the input values of each example. Once a program is found that evaluates to the output value for each example, the synthesizer outputs that program and exits. More generally, the synthesizer can be said to evaluate each program using a predicate and exits if the predicate evaluates to *true*.

We modified this algorithm in two notable ways. The first was to redefine the predicate from an exact match on each output, to

(1) A regular expression match on strings containing "`···`", with each "`···`" in the string replaced with the "`.+`" token,
(2) A list match on lists containing the Python `Ellipsis` object, where each `Ellipsis` was matched against one or more arbitrary elements, and
(3) An exact match as before otherwise.

Observational Equivalence guarantees that for an exact match predicate, the synthesizer will only find a single solution. This partial predicate, however, can be satisfied with multiple programs that are distinct under OE and therefore may be synthesized. This allowed us to also modify the synthesizer's exit condition. Previously, the synthesizer exited after finding a solution. With PopPy, we instead continue synthesis, outputting every solution until the user interface terminates the synthesizer or we reach programs of height 4, which (similar to SnipPy) we consider beyond PopPy's scope.

2.2.2 User Interface. To enable the interaction model described in Sec. 2.1, we converted the existing SnipPy interface with the ability to check for invalid specifications to prevent spurious failed synthesis calls, communicate with the synthesizer, and present new outputs interactively.

To check for invalid specifications, rather than sending the synthesis specification directly to the synthesizer, we first check that each value evaluates correctly in Python, and that it is of a type that the synthesizer supports. If there is an error or an invalid type, we present this as an error message to the user while keeping their specification (Fig. 1g), so that they may modify it before resubmitting.

Once the user's specification is accepted and synthesis starts, we fill the Projection Box column for the user's output with the message "`synthesizing`", and similarly add a message in code (Fig. 1c). Once a solution is found, the code message is replaced and the box column is updated with the values that the solution evaluates to in each example (Fig. 1d). Other solutions are kept in the background, and the user can move between the available outputs using the ← and → keys on the keyboard. If they reach the end of the list, we again use the "`synthesizing`" message to indicate that the end of the list is reached. If they try to go beyond this, a message appears stating that it is the end of available outputs.

If the user waits for more than 7 seconds at this output, we consider it a timeout and terminate the synthesizer to stay below a disruptive interruption, which was also the original timeout for SnipPy [10]. If the synthesizer exits or is terminated, the code

Figure 1: Solving the example scenario in PopPy. (a)-(f) shows synthesizing the first step with a partial list specification. (g) displays an error for an invalid specification. (h) shows the partial specification for strings.

message changes to Fig. 1e but the user is still able to move between the available outputs.

The user can stop the synthesis process at any point by pressing Esc. Once there is at least one output available, the user can select the current output by pressing Enter. After selecting an output, we insert the matching program in the user's code, and move the cursor back to the editor (Fig. 1f).

3 AN EXPLORATORY EVALUATION

Can PBPE help users: (a) write output specifications faster, (b) prevent typos in specifications, and (c) use the synthesizer more? To take initial steps towards answering these questions, we conducted a small between-subjects pilot study comparing PopPy to the same tool without the partial output specification feature.

We recruited 4 participants (3 male, 1 female) among the graduate students in the Computer Science and Engineering department. We asked potential participants to self-rate their Python proficiency on a Likert scale of 1 (Not at all familiar) to 5 (Extremely familiar), and selected those who rated themselves between 2 and 4 (inclusive). We then randomly assigned the participants into either the Experimental (using partial specifications) or Control (using complete specifications) group, with the constraint that we have exactly 2 participants per group.

3.1 Procedure

We conducted the studies remotely through a Zoom session, where an investigator shared their screen (with PopPy open on the screen) and participants used the tool through Zoom's remote control feature.

After a brief introduction to the study, each participant was asked to watch an approximately 5 minute tutorial video (see Sec. 3.2). This was followed by a loosely guided section where the participants were encouraged to replicate the tutorial steps and experiment with the tool. This was necessary to address any issues or questions users might have, and for them to get used to a slight delay in typing caused by Zoom's remote control.

Once the user declared that they were ready, they were presented with the first programming task. Once they declared that they had finished the first task, they were presented with the second, following the exact same procedure.

After finishing the second task, the participants were asked to complete a short survey involving long-answer questions about their experience with the tool and suggestions for improvements. This was optionally followed by follow-up questions by the investigator to clarify written responses.

3.2 Tutorial

Since PopPy requires instructions for productive use, we prepared a tutorial video demonstrating Projection Boxes using simple code examples, and explaining PopPy with and without partial output specifications using the task described in Sec. 2.1. We then edited the video to create two versions, one containing a partial specification example, and one with only complete specifications. The two videos were identical aside from this edit.

3.3 Tasks

Each participant solved two string and list processing tasks in Python that could be solved within 15 minutes. The tasks are:

- `Reverse`[1]: Reverse the words; keep the punctuation at the end.
- `Addition`[2]: Compute an addition expression given as a string.

3.4 Data Collection

Qualitative data for the pilot was collected by the post-study survey containing the following questions:

(1) In your own words, describe what you did today?
(2) Was the tool you used today useful? If so, how was it useful?
(3) Was the tool you used today confusing? If so, how was it confusing?
(4) Do you have any suggestions on how to improve the tool you used today?

We also collected quantitative data by recording timestamped logs of various interactions with PopPy, including each edit to the code, each specification sent to the synthesizer, and each result found by the synthesizer (including failed synthesis calls).

[1]http://shorturl.at/cyV05
[2]http://shorturl.at/pAHK2

3.5 Data Analysis

We used the recorded quantitative data to extract three pieces of information:

Time Spent on Providing Examples. measured by the timestamp difference between when the user typed ?? and when they sent the specification to the synthesizer, excluding cases where they quit without submitting the specification.

Number of Synthesized Requests. counted directly from the logs. This was further subdivided into "Successful" calls where the user accepted a solution, "Discontinued" where the synthesizer found solutions but the user quit without accepting any, and "Failed" where the synthesizer failed to find any solutions for the specification.

Percentage of Synthesized Code. measured by the percentage of tokens in the final code matching exactly to the output of a synthesis call from that code's file, divided by the total number of tokens added to the file by the user (excluding existing code boilerplate).

Since this was a small pilot study, we did not run any statistical analysis on the quantitative data, but visualizations of these can be found in Sec. 4.

3.6 Limitations

The main limitation of our pilot study is its scale. Although it was meant to refine the design of PopPy and of the main study, with only four participants from a limited pool, we cannot make any rigorous claims about the usability of PopPy's partial output specification feature. The small scale also resulted in high variance in participants' Python proficiency, the self-report nature of which is already inaccurate.

Also, participants solved two small programming tasks and used PopPy for less than 30 minutes; a more ecologically-valid study would be to have programmers use PopPy to solve more realistic problems with extended durations. As such, we did not observe cases of typos during the study (more in Sec. 4) and could not verify our hypothesis that PopPy could reduce typos.

4 RESULTS

This section reports the study results of individual subjects: S1 and S2 used PopPy with partial specifications, while S3 and S4 used PopPy with complete specifications only. All 4 subjects solved the programming tasks with some use of the synthesizer except for S4, who did not use the synthesizer for task `Addition`.

4.1 Time Spent on Providing Examples

Fig. 2 presents the distribution of time spent on providing examples per synthesis request of each subject by task. Overall, participants using the partial specifications-enabled synthesizer spent less time providing examples than participants using the synthesizer that requires complete specifications.

4.2 Using the Synthesizer

Fig. 3 illustrates the number of synthesis requests each participants made per task and the breakdown of the requests. The distribution of percentages of synthesized tokens in each participant's final solution per task (Fig. 4) resembles that of Fig. 3.

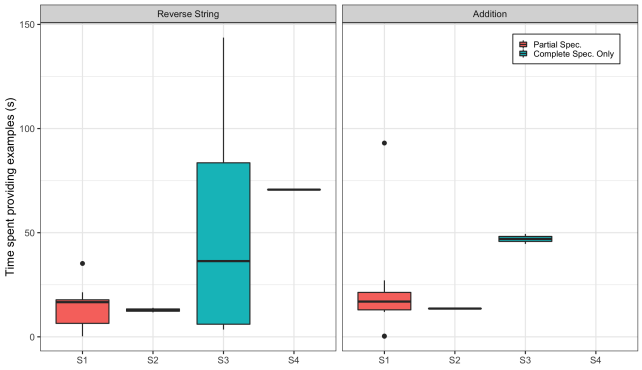

Figure 2: Time (seconds) spent on providing examples per synthesis request.

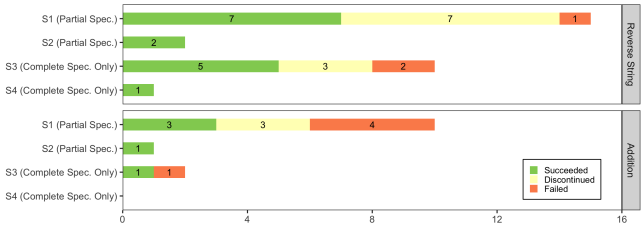

Figure 3: Number of synthesis requests per task.

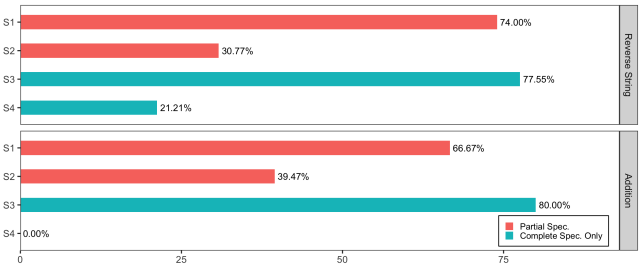

Figure 4: Percentage of synthesized tokens in the final solution.

Although less definitive, participants made more synthesis requests when allowed to write partial specifications. We also examined the Failed requests manually and found no typo-induced synthesis failures; all the failures were led by requests beyond the synthesizer's abilities.

For participants using PopPy with partial specifications, we calculated the percentages of synthesis requests using partial specifications per task. S1 used partial specifications in 80% and 71.43% of their synthesis requests in tasks `Reverse` and `Addition`, respectively, while S2 used partial specifications 100% of the time.

4.3 Qualitative Feedback

All participants described what they did in the study accurately. All found the synthesizer to be useful; 3 out of 4 participants considered

the synthesizer as a convenient lookup for built-in functions. We received no comments specifically on how examples were provided.

Regarding the confusing aspects of the tool, 2 participants expressed frustration with synthesizing the summation function in task `Addition`, which is not supported by PopPy, while 1 participant worried about potential frustration caused by synthesis failures in general. We found no complaints about writing specifications.

5 DISCUSSION

Through a pilot study, we explored the prospects of PBPE in speeding up specifications writing, preventing typos, and encouraging using the synthesizer. Although the study was small-scale and the data were not conclusive, our results indicated that PBPE would be preferred when available, make writing specifications faster, and increase the use of the synthesizer. All participants found the tool (not just the partial examples feature) useful particularly for library function lookup. No participants expressed concern with the feature or writing specifications in general, though there was frustration with the abilities of the synthesizer.

Contrary to our expectations, we observed no typos in the study. Kalfaoğlu and Stafford [6] suggested that typing errors are foreshadowed by a breakdown in performance. It is possible that we failed to observe typos in writing specifications because the programming tasks were too small and the intervention was too short to induce such a breakdown. Another possibility is that the delay caused by Zoom's remote control feature forced the participants to type more slowly and meticulously than they normally would and thus reduced the likelihood of typos. We need more controlled experiments with longer tasks and intervention through non-remote manipulation of the programming environment to better understand the effects of PBPE in preventing typos.

We also found `S3` intentionally used the synthesizer a lot despite having to write complete specifications, which was confirmed post-study: "I [actually] felt that coding [by hand] was easier." Since we designed the tasks open-ended without providing algorithmic guidance so that they resemble real problems, participants would have to pick between entirely using PopPy, writing the code by hand, and a mix of both to solve the tasks. Such lack of guidance might cause them to use the synthesizer more (or less) than they naturally would, which we found to be the case for `S3`.

6 FUTURE WORK AND CONCLUSION

We introduced PBPE and implemented it in PopPy, and discussed its preliminary evaluation in a small pilot study. We did not reach conclusive results about how it compares to full output specifications. However, we found that partial outputs are useful and widely used when available, decrease the time spent providing specifications, and do not increase confusion or frustration. These results lead us to conclude that partial output specifications are a promising direction, but that a larger user study is needed to better assess their effects on the usability of synthesis tools.

From the pilot studies, we also identified improvements that should be made to the study design. We saw that users' self-reported Python proficiency was not indicative of their performance on the tasks. A future study can account for this by replacing the self-reported question with an empirical measure, such as solving a programming task without the tool and measuring completion time and correctness. To avoid issues with remote control delays, PopPy could be offered as an extension to Visual Studio Code rather than a custom build to allow easy installation on participants machines. This could also enable running larger long-term studies in more natural settings, by asking participants to use PopPy in day-to-day tasks rather than strict study sessions with pre-selected tasks.

ACKNOWLEDGMENTS

We would like to thank Sorin Lerner, Hila Peleg, and Nadia Polikarpova for their helpful comments and suggestions.

REFERENCES

[1] Denis Anson, Penni Moist, Mary Przywara, Heather Wells, Heather Saylor, and Hantz Maxime. 2006. The Effects of Word Completion and Word Prediction on Typing Rates Using On-Screen Keyboards. *Assistive Technology* 18, 2 (Sept. 2006), 146–154. https://doi.org/10.1080/10400435.2006.10131913

[2] Ian Drosos, Titus Barik, Philip J. Guo, Robert DeLine, and Sumit Gulwani. 2020. Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems (CHI '20)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3313831.3376442

[3] Kasra Ferdowsifard, Allen Ordookhanians, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2020. Small-Step Live Programming by Example. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*. ACM, Virtual Event USA, 614–626. https://doi.org/10.1145/3379337.3415869

[4] Sumit Gulwani. 2012. Synthesis from Examples: Interaction Models and Algorithms. In *2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, Timisoara, Romania, 8–14. https://doi.org/10.1109/SYNASC.2012.69

[5] Sheri Hunnicutt and Johan Carlberger. 2001. Improving word prediction using Markov models and Heuristic methods. *Augmentative and Alternative Communication* 17, 4 (Jan. 2001), 255–264. https://doi.org/10.1080/aac.17.4.255.264

[6] Çığır Kalfaoğlu and Tom Stafford. 2014. Performance Breakdown Effects Dissociate from Error Detection Effects in Typing. *Quarterly Journal of Experimental Psychology* 67, 3 (March 2014), 508–524. https://doi.org/10.1080/17470218.2013.820762 Publisher: SAGE Publications.

[7] Jamie Klund and Mark Novak. 1997. If word prediction can help, which program do you choose? *Technology: Special Interest Section Quarterly* 7 (1997), 1–2. https://park.org/Guests/Trace/pavilion/ctg_wp1.htm

[8] 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 Number: 4.

[9] Brad Myers, Scott E Hudson, and Randy Pausch. 2000. Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction* 7, 1 (2000), 26.

[10] Antti Oulasvirta and Pertti Saariluoma. 2006. Surviving task interruptions: Investigating the implications of long-term working memory theory. *International Journal of Human-Computer Studies* 64, 10 (Oct. 2006), 941–961. https://doi.org/10.1016/j.ijhcs.2006.04.006

[11] Hila Peleg and Nadia Polikarpova. 2020. Perfect Is the Enemy of Good: Best-Effort Program Synthesis. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2:1–2:30. https://doi.org/10.4230/LIPIcs.ECOOP.2020.2 ISSN: 1868-8969.

[12] Mark Santolucito, William T. Hallahan, and Ruzica Piskac. 2019. Live Programming By Example. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, Glasgow Scotland Uk, 1–4. https://doi.org/10.1145/3290607.3313266

[13] Christopher Schuster and Cormac Flanagan. 2016. Live Programming by Example: Using Direct Manipulation for Live Program Synthesis. 7.

[14] Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. 2020. Visualization by example. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–28. https://doi.org/10.1145/3371117