Small-Step Live Programming by Example

Kasra Ferdowsifard
UC San Diego
kferdows@eng.ucsd.edu

Allen Ordookhanians
UC San Diego
aordookh@ucsd.edu

Hila Peleg
UC San Diego
hpeleg@eng.ucsd.edu

Sorin Lerner
UC San Diego
lerner@cs.ucsd.edu

Nadia Polikarpova
UC San Diego
npolikarpova@eng.ucsd.edu

ABSTRACT
Live programming is a paradigm in which the programming environment continually displays runtime values. Program synthesis is a technique that can generate programs or program snippets from examples. Previous works that combine the two have taken a holistic approach to the way examples describe the behavior of functions and programs. This paper presents a new programming paradigm called Small-Step Live Programming by Example that lets the user apply Programming by Example locally. When using Small-Step Live Programming by Example, programmers can change the runtime values displayed by the live visualization to generate local program snippets. We implemented this new paradigm in a tool called SnipPy, and performed a user study on 13 programmers. Our study finds that Small-Step Live Programming by Example with SnipPy helps users solve harder problems faster, and that for certain types of queries, users prefer it to searching the web. Additionally, we identify the *user-synthesizer gap*, in which users' mental models of the tool do not match its ability, and needs to be taken into account in the design of future synthesis tools.

Author Keywords
Live Programming, Program Synthesis

CCS Concepts
•Human-centered computing → Graphical user interfaces; •Software and its engineering → Automatic programming;

INTRODUCTION
Live programming is a paradigm where the programming environment continually displays runtime values. While live programming provides immediate feedback about the current state of execution, it does not explicitly help the programmer to discover the next line of code they need to write to accomplish their goal.

UIST'20, October 20–23, 2020, Minneapolis, MN, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-6708-0/20/04. . . $15.00

DOI: https://doi.org/10.1145/3313831.XXXXXXX

On the other hand, program synthesis is a technique that helps programmers by generating code automatically. There are many approaches to program synthesis, but in this paper we focus on a class of techniques called Programming-by-Example (PBE), where the programmer provides input-output examples, and the synthesizer produces candidate programs that satisfy these examples. While program synthesis can generate code that accomplishes a given goal, traditional synthesizers are stand-alone and not integrated tightly into the development work-flow, which makes it hard for the programmer to formulate the goal for the synthesizer to solve.

As such, live programming and PBE are perfectly suited for each other: the live programming environment provides all the values needed for the programmer to easily provide examples without a significant break in workflow; and synthesis from examples helps address a limitation of live programming, which is that it does not explicitly generate statements.

Because of this symbiotic relationship, there has been prior work on combining Live Programming with PBE, sometimes called Live Programming by Example [39, 38] or synthesis from Direct Manipulation Interfaces [8, 31, 19]. However, broadly speaking, examples in this prior work describe behavior holistically, meaning that each example impacts either the entire program, or a large part of the program (e.g., an entire function). In the literature on program semantics, this kind of specification is usually referred to as a big-step semantics [49].

In this paper, we describe a different approach to Live Programming by Example, which we call *Small-Step Live Programming by Example* (SSL-PBE). In contrast to prior work, SSL-PBE allows the programmer to specify examples in a live programming environment, but only for a single missing statement. Synthesis in SSL-PBE starts in a live programming environment where program state is displayed after each statement. While in traditional live programming the displayed state is read-only, in SSL-PBE the runtime values can be modified. When the programmer edits values in the state, a program synthesizer runs to generate a local program snippet that satisfies the new data. SSL-PBE is unique in that it enables a new programming paradigm where the programmer "leads" the generation of the program with data.

To understand the viability of this new paradigm, we implemented SSL-PBE for the Python programming language, in a tool named SnipPy. SnipPy uses the live programming environment of Projection Boxes [27] and a custom-made enumerative program synthesizer to generate Python statements.

Through a user study, we demonstrate that SSL-PBE is easy to use, and has an impact on task time and correctness on more difficult tasks. Our study also shows that the synthesizer can generate between 18% to 66% of the code, thus demonstrating that the synthesizer and the human can work together to form a complete solution. Finally, our user study also shows that almost all our participants preferred SnipPy over searching the internet in some cases, the main reasons being that compared to the internet searches SnipPy incurs a lower cognitive burden, automatically connects the snippet with the surrounding code, and provides a more compact solution.

The main contributions of this paper are:

- We present a novel paradigm called Small-Step Live Programming by Example, in which programmers can modify live data to generate code snippets.
- We present an implementation of this paradigm in a tool called SnipPy.
- A user study of SnipPy on 13 programmers found that SnipPy complements web searches in bridging certain types of knowledge gaps, and that SnipPy helped users solve harder problems
- We identify the *user-synthesizer gap*, where a mismatch between the user's mental model of the synthesizer and the abilities of the synthesizer hinders the user's ability to use the synthesizer effectively. We believe that the *user-synthesizer gap* needs to be addressed as synthesis tools begin to target programmers rather than end-users.

MOTIVATING EXAMPLE

A programmer named Kayla is processing a text file using Python. One part of the processing involves reducing a name (e.g., "Augusta Ada King") to a non-standard form of initials[1] (e.g., "A.A.K"). Kayla has a lot of programming experience, but is only a casual Python user, which means Kayla does not immediately know how to achieve this task in Python. Being an experienced programmer, Kayla breaks down the task into two components: getting the first letter of each word, and reconnecting them in the desired format.

Opportunistic programming

Kayla turns to searching for the answer using a search engine. Unsure of the precise string terminology of Python, but used to relying on search engines, she tries the natural language search, "Python first letter of every word". The first result is a link to the Stack Overflow question "How can I get the first letter of each word in a string?" which has no accepted answer but upon further reading has code which looks suitable in the comments discussing the question. Copying the code and modifying the variable names, Kayla now has the following:

```
letters = [w[0] for w in s.split(' ')]
```

[1]The task is taken from this competitive programming exercise: https://www.codewars.com/kata/57eadb7ecd143f4c9c0000a3

This code returns a list of initials. Now all that remains is to format it. Since the required formatting is trickier than standard initials punctuation, Kayla knows that a loop is not the easiest way to go in this case. Kayla recalls that Python has a `join` method to convert arrays to strings, so she tries:

```
letters = [w[0] for w in s.split(' ')]
res = letters.join('.')
```

However, this code produces the runtime error `AttributeError: 'list' object has no attribute 'join'`, which Kayla now looks up online as well. After some digging, Kayla realizes that `join` is a method on strings, so the correct code is in fact:

```
letters = [w[0] for w in string.split(' ')]
res = '.'.join(letters)
```

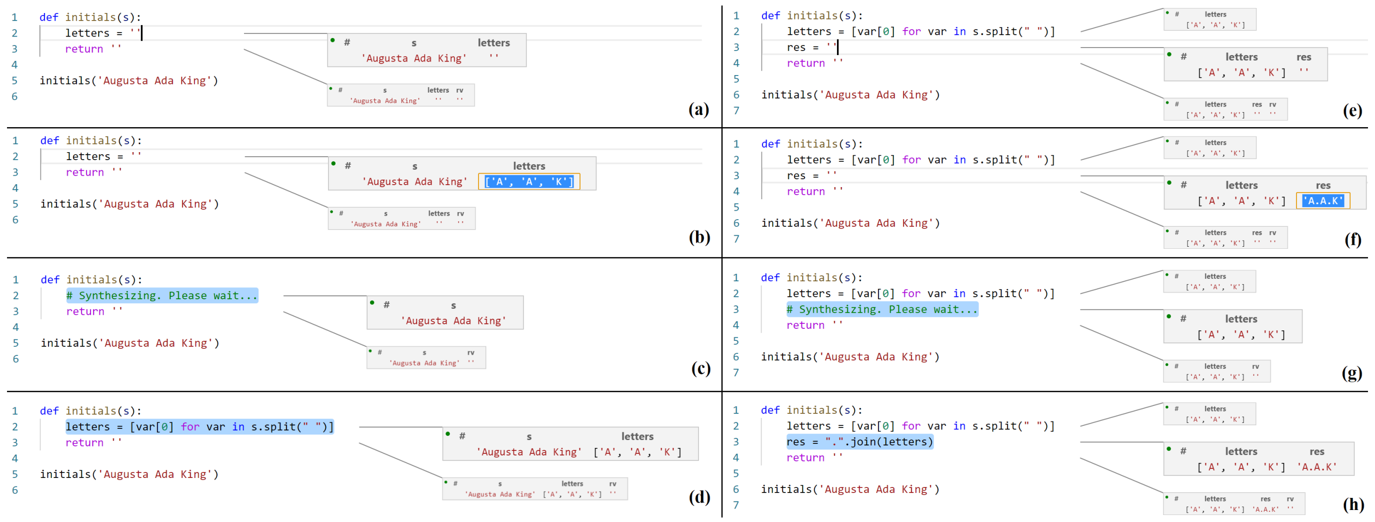
Small-Step Live Programming by Example

Let us consider the same task again, but instead Kayla will use our proposed approach, Small-Step Live Programming by Example, as reified in our SnipPy tool. Kayla starts in a live programming environment, in this case Projection Boxes [27], as seen in Figure 1(a). The visualization shows at each line a projection box with the values of all variables at that line. However, while in a traditional live programming environment the visualization is read-only, in our approach the values of variables in the visualization can be edited to show the programmer's intent. Thus, Kayla edits the value of `letters` in the projection box to enter the desired value, as shown in Figure 1(b). By this, Kayla is stating that she wants SnipPy to generate a code snippet that will produce `['A','A','K']` in the `letters` variable when `s` is `'Augusta Ada King'`. In the synthesis literature this is called an *input-output example*. If the statement Kayla wanted to generate was executed multiple times (i.e., it is inside a loop or in a function that is called multiple times), the projection box would have one line per execution and Kayla would be able to provide one or more input-output examples, one for each execution of the statement.

Once Kayla has provided some input-output examples by changing the live visualization, these examples are sent to a Programming by Example (PBE) synthesis engine, while the user is told that the synthesizer is working in the background Figure 1(c). In previous PBE tools aimed at programmers, providing examples is often a weak point of the interaction model, sometimes requiring a break in the workflow that could be as severe as switching to a different tool and editing its configuration files. Through direct manipulation of live data, our approach makes the specification process seamless. Furthermore, focusing the user's attention on the value assigned to a single variable turns the synthesizer into a helper utility in a larger task, which harnesses the still-limited power of synthesis to solve specific sub-tasks for the user.

Within seconds, the SnipPy synthesizer finds a solution and adds the generated code snippet, as shown in Figure 1(d).

Next, Kayla creates a new variable, called `res`, as shown in Figure 1(e) – for brevity of the figure, at this point we configured Projection Boxes to not display the variable `s` anymore. Kayla changes the projection box value of `res` to be `'A.A.K'`, as shown in Figure 1(f). This provides the synthesizer with an input-output example stating the output in `res` should be `'A.A.K'`



(a)

```
def initials(s):
    letters = ''
    return ''

initials('Augusta Ada King')
```

```
#  s                   letters
   'Augusta Ada King'  ''

#  s                   letters  rv
   'Augusta Ada King'  ''       ''
```

(b)

```
def initials(s):
    letters = ''
    return ''

initials('Augusta Ada King')
```

```
#  s                   letters
   'Augusta Ada King'  ['A', 'A', 'K']

#  s                   letters  rv
   'Augusta Ada King'  ''       ''
```

(c)

```
def initials(s):
    # Synthesizing. Please wait...
    return ''

initials('Augusta Ada King')
```

```
#  s
   'Augusta Ada King'

#  s                   rv
   'Augusta Ada King'  ''
```

(d)

```
def initials(s):
    letters = [var[0] for var in s.split(" ")]
    return ''

initials('Augusta Ada King')
```

```
#  s                   letters
   'Augusta Ada King'  ['A', 'A', 'K']

#  s                   letters          rv
   'Augusta Ada King'  ['A', 'A', 'K']  ''
```

(e)

```
def initials(s):
    letters = [var[0] for var in s.split(" ")]
    res = ''
    return ''

initials('Augusta Ada King')
```

```
#  letters
   ['A', 'A', 'K']

#  letters          res
   ['A', 'A', 'K']  ''

#  letters          res  rv
   ['A', 'A', 'K']  ''   ''
```

(f)

```
def initials(s):
    letters = [var[0] for var in s.split(" ")]
    res = ''
    return ''

initials('Augusta Ada King')
```

```
#  letters
   ['A', 'A', 'K']

#  letters          res
   ['A', 'A', 'K']  'A.A.K'

#  letters          res  rv
   ['A', 'A', 'K']  ''   ''
```

(g)

```
def initials(s):
    letters = [var[0] for var in s.split(" ")]
    # Synthesizing. Please wait...
    return ''

initials('Augusta Ada King')
```

```
#  letters
   ['A', 'A', 'K']

#  letters
   ['A', 'A', 'K']

#  letters          rv
   ['A', 'A', 'K']  ''
```

(h)

```
def initials(s):
    letters = [var[0] for var in s.split(" ")]
    res = ".".join(letters)
    return ''

initials('Augusta Ada King')
```

```
#  letters
   ['A', 'A', 'K']

#  letters          res
   ['A', 'A', 'K']  'A.A.K'

#  letters          res      rv
   ['A', 'A', 'K']  'A.A.K'  ''
```

Figure 1. Writing code using SNIPPY: (a)-(d) generates the first statement and (e)-(h) generates the second statement.

when `s` is `'Augusta Ada King'` and `letters` is `['A', 'A', 'K']`. Within a few seconds, the SNIPPY synthesizer generates a statement, as shown in Figure 1(g)-(h). To finish the code, Kayla changes the return statement to return the `res` variable.

In summary, this example showed how Kayla was able to finish writing the code, without leaving the IDE, without searching online, and without bearing the cognitive load of thinking about the details of list comprehension, `split` or `join`.

RELATED WORK

Live programming

Live programming is a paradigm that provides immediate visualizations of a program's behavior. This research area dates back to the seminal work of Hancock [18]. Live programming environments have been developed for Python [15, 22], Java [5], Javascript [37, 1], Lisp [2] and ML-like languages [32]. There is also work categorizing different kinds of liveness [43, 47], and studies exploring the benefits of live programming [48, 23]. Traditionally, live programming only provides feedback to the programmer, and not the ability to edit the output or synthesize new pieces of code. Our work distinguishes itself from traditional live programming by contributing a new paradigm where changing data in the visualization can produce small code snippets for the programmer.

Big-step program synthesis and repair

Program synthesis takes a specification of behavior and returns a program satisfying the specification. Automatic program repair (APR) takes a program and a specification, and minimally changes the program to satisfy the specification. Both commonly take specifications that describe the full program, or as previously desribed, "big-step" specifications. Most synthesis and APR literature assumes a user that provides the specifications, but does not focus on the experience of the user when creating the specifications or processing the results.

Programming by Example

Programming by Example (PBE) is a field of program synthesis where behavior specifications are provided as input-output pairs. This approach has been applied to string transformations of Excel data [13] and file renaming [14], data extraction [25], and transforming tabular data [51], all synthesis solutions for end-users. In APR and synthesis aimed at developers, examples can also take the form of unit tests [29, 30, 50, 26].

Since examples are partial specifications, often resulting in too-specific programs that require more examples to generalize, [36] question their sufficiency as a specification tool. Entering examples is also error-prone, a problem tackled by [35].

Live programming by example

Two approaches for synthesis for developers are named "Live Programming by Example". The first [39] allows users to edit HTML output causing changes to its rendering JavaScript code. The second [38] consists of two connected files, a code file with functions and an examples file with unit tests for the functions. Unit tests are executed for live programming data, and modifying a unit test will attempt to synthesize a new function where the tests pass. Both approaches shift the audience of PBE from non-programmer end-users to programmers who interact with the synthesized code, but both still use examples as a holistic mode of specification. In SNIPPY, the separation between code and data is not as severe: live values are adjacent to the code, not in another file. Additionally, SNIPPY uses examples as small-steps on intermediate values, rather than just the output of the entire execution, giving users access to localized synthesis from local examples.

Domain specific synthesis tools

Several recent projects focus on *domain-specific* example-driven tools for non-programmers: Rousillon [7] generates web scrapers, Bespoke [45] GUI applications, and Wrex [9] Python wrangling code in Jupyter Notebooks. SNIPPY targets general programming, rather than one domain-specific task.

As such, our intended users are programmers, not data scientists or power-users, and we augment a different workflow.

Direct manipulation interfaces as specification

Direct manipulation user interfaces [40] are visualizations of the system state that can be changed, and are prevalent as editors of graphical representations. Sketch-n-Sketch [8, 31, 19] is a direct-manipulation editor for *programmable* graphical formats such as SVG, HTML, and LaTeX, where direct manipulation changes to the visual output become repair operations on the generating code and inputs. [19] takes this further by adding refactoring tools and turning the direct manipulation editor into a visual programming environment. The direct manipulation workflow is aimed at modifying visual objects in languages with little separation between code and input. SSL-PBE is a technique for more expressive langauges.

Small-Step Program Synthesis

Several existing synthesis and APR techniques rely on a small-step specification, or specifications that tackle only local behavior within a larger program.

Sketch and sketching

Sketching (popularized by Sketch [42] and modified into a variant used by many synthesis works [11, 41, 21, 34]) is a method for specifying to the synthesizer a partial implementation of the target program with *holes*, initially missing numeric values but now missing expressions or statements. Sketches isolate local steps for the synthesizer to fill, and are usually accompanied by a big-step specification (e.g., examples) from which a local specification for the holes is derived.

SnipPy differs from these in two ways: first, synthesis queries in SnipPy are themselves locally-specified, and do not require the potentially-lossy transformation from a larger specification, and second, and enabled by the previous, SnipPy can be used without end-to-end specifications for the full program, which means it can be used to perform exploration.

Single statement synthesis

Small-step synthesis has been worked into IDEs to attempt predicting the next statement. For example, InSynth [17] code-completes assignments using the type of the assigned variable as the specification. As another example, CodeHint [11] allows the programmer to stop the program at a breakpoint and generate a new version of the next statement by providing a specification on the current program state (typically a typef-based specification, though examples are possible). The user can trigger the breakpoint again to provide specifications for more inputs, but those only serve to rule out candidates generated for the first input state. While in CodeHint providing specifications for multiple inputs requires running to the breakpoint again and again, in SnipPy the Projection Boxes display flattens both loops and multiple inputs into a single list, allowing the user to specify any and all inputs at once.

Both CodeHint and InSynth return several ranked options for the completion, a different interaction model than that of SnipPy, which returns a single result. While returning a single result is an all-or-nothing approach, it also means that users do not need to select out of multiple options, a task that previous work has shown is difficult for users to perform correctly without in-depth inspection [36].

Program states as specifications

Another form of local specifications is *direct state manipulation*, or changing the program's internal state to create repair specifications, as in Wolverine [46] and JDial [20]. Both provide ways for users to modify the values of variables along an execution trace of the program on a test input, then create a repair to the program. Wolverine, which has a gdb-like interface, also hot-patches the repair and allows debugging to continue.

Unlike both projects, which are APR tools, SnipPy is a synthesis tool. This means that SnipPy's user is not working on an already fully-written program and only correcting a bug found in the course of the execution.

Both projects also tackle a common problem: APR can find a trivial fix that satisfies a specification by removing desired functionality. Wolverine's solution is to let the programmer, while debugging the program, mark an intermediate state as specification, declaring the *unmodified state* should appear in the execution trace of the final program. SnipPy faces a similar problem, declaring certain behavior as already-correct while not forcing the user to specify all inputs. Its solution, discussed in the next section, is inspired by Wolverine's solution, adapted from repair to synthesis.

USER INTERFACE

We implemented SnipPy on top of Projection Boxes [27] by making the variable values editable in each projection box. After the programmer modifies some values in a projection box, every modified row becomes an example that is sent to a Programming by Example (PBE) synthesizer. Recall that the projection box for a line in the program contains multiple rows if that line is executed multiple times.

Activating SnipPy

In order to allow SnipPy to naturally become part of the development workflow, it can be activated and specified entirely via keyboard operations. To assign a synthesized value to a variable, the user begins typing an assignment statement, but instead of a concrete value, they assign ??:

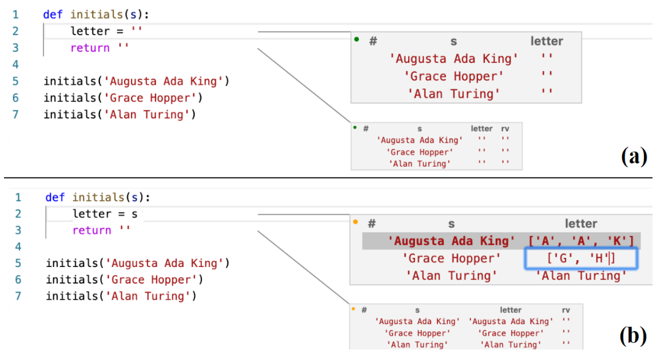
```
newVar = ??
```

This special token, which is not valid Python code, temporarily becomes the temporary assigned value 0 (since Python variables are not typed, a specific initial value by type cannot be generated), and the focus is moved to the now-editable value of the variable, as in Figure 1(b). The user can then enter a new value, and start the synthesizer by hitting Enter.

Modified values are then packaged as examples to the synthesizer, where each input state includes all variables that are in scope at the line that is being synthesized, namely all variables that appear in the projection box at that line.

Editing examples

Some examples are inconvenient to type, but easier to edit. For example, if the user wants to turn the string `'Augusta Ada King'` in the variable `s` into a list of words `['Augusta','Ada','King']`, they can start with a blank variable value and enter in every



(a)

```
def initials(s):
    letter = ''
    return ''

initials('Augusta Ada King')
initials('Grace Hopper')
initials('Alan Turing')
```

```
#           s               letter
     'Augusta Ada King'      ''
       'Grace Hopper'        ''
        'Alan Turing'        ''
```

```
#          s              letter  rv
    'Augusta Ada King'      ''    ''
      'Grace Hopper'        ''    ''
       'Alan Turing'        ''    ''
```

(b)

```
def initials(s):
    letter = s
    return ''

initials('Augusta Ada King')
initials('Grace Hopper')
initials('Alan Turing')
```

```
#           s                    letter
     'Augusta Ada King'   ['A', 'A', 'K']
       'Grace Hopper'        ['G', 'H']
        'Alan Turing'      'Alan Turing'
```

```
#          s                     letter          rv
    'Augusta Ada King'   'Augusta Ada King'      ''
      'Grace Hopper'        'Grace Hopper'       ''
       'Alan Turing'         'Alan Turing'       ''
```

Figure 2. Using SnipPy with multiple values. (a) Projection Boxes showing multiple values for the same line, and (b) providing only some of the values as examples to SnipPy.

word into the list, but it is easier and far less error prone to start with the value of the string and edit it into a list. To this end, the user can activate SnipPy with an expression reference:

```
newVar = s??
```

This will put the user in edit mode for the values of `newVar` as before, but instead of a default initial value will populate `newVar` with the value of `s`.

Multiple examples

Projection Boxes can show multiple values for a line of code. This can happen in one of two cases: if the function is called multiple times with multiple values, as seen in Figure 2(a), or if the current line of code is inside a loop. For SnipPy, this means that there are different values for the in-scope variables that can be sent to the synthesizer as multiple examples for the same expression. When the Projection Boxes contain multiple rows, the edited variable becomes editable in all rows, as seen in Figure 2(b), and the user can travel between the values using the Tab key.

Sometimes the user does not want to provide an output value for every row in the Projection Boxes. A simple example for this would be inside a loop with many iterations, where two or three examples will suffice to demonstrate the desired functionality. SnipPy allows users to only edit some of the output values. Rows where the value was changed are submitted to the synthesizer as examples, and unchanged rows are ignored. Figure 2(b) shows highlighted rows that were edited by the user and will be sent as examples to the synthesizer.

Occasionally, the value of some of the rows already exhibits correct behavior. These already-correct rows will not be changed, but if they are not sent to the synthesizer along with changed rows, a program that changes their behavior may be synthesized. To avoid this, SnipPy lets the user specify that the output value in a row is correct as-is by selecting it with Shift+Enter. A variant of this problem was previously solved in Wolverine [46], where the user can mark a state as "specification" the execution must still pass through, but with no assurance which line of code will pass through it.

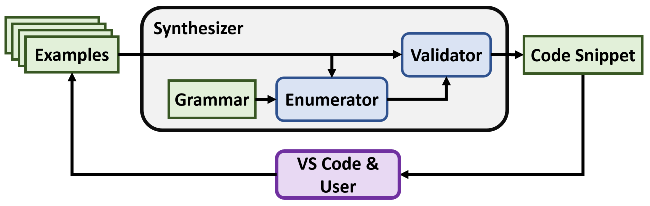


Figure 3. The structure of our *generate-and-test* synthesizer. Programs are enumerated and passed to the validator to be tested against the user-provided examples.

A Synchronous Modality

SnipPy synthesis calls are synchronous. This means that once the user calls the synthesizer, a wait message (shown in Figure 1(c)) appears, and the user waits for synthesis to quickly finish or fail. An asynchronous workflow could have been explored, but it would not allow SnipPy to be used in a more exploratory manner. The synchronous workflow motivates us to use a timeout that is as brief as possible, which means our synthesizer must be extremely efficient to find meaningful expressions within that timeout. We chose a timeout of seven seconds, enough to synthesize programs of up to height 3 (zero-based) on several desktop and laptop architectures, while staying well below a distruptive interruption [33]. In the next section, we discuss the design considerations when building such a synthesizer.

SYNTHESIZER IMPLEMENTATION

We designed and implemented a custom synthesizer to generate the one-line Python snippets requested by SnipPy users. The synthesizer that we built is known as an *enumerating generate-and-test* synthesizer: broadly speaking, *enumerating* means that the synthesizer enumerates programs by expanding a grammar that represents the space of programs to search, and *generate-and-test* means that the synthesizer evaluates each enumerated program to test whether it fits the given examples. Figure 3 shows an overview of SnipPy as a system, including details of our enumerating generate-and-test synthesizer.

As with any synthesizer, the astronomical size of the search space is the main challenge, which we begin to mitigate using the known technique of *observational equivalence* [44, 3], which unifies all programs that behave the same on inputs from the example. However, our setting adds several additional challenges: (i) Python is a dynamically typed language, so types cannot be used out of the box to constrain the search, (ii) our synthesizer generates Python list comprehensions, which are loops, a hard problem for synthesis, (iii) our synthesizer needs to generate string constants to enable the string manipulation code that Python programmers write, and (iv) our synthesizer should work with as few examples as possible, despite PBE's propensity for trivial solutions for such tasks.

Yet, despite all these challenges, we need to create a synthesizer that operates in an interactive setting, able to generate useful snippets in seconds. We discuss how we address each of these four challenges, while maintaining interactive speeds, in each of the following subsections.

$E \ ::= I \mid S \mid L \mid \ldots$
$I \ ::= x \mid I{+}I \mid S$`.find(`$S$`)` $\mid$ `max(`IL`)` $\mid$ `int(`S`)` $\mid \ldots$
$S \ ::= s \mid S{+}S \mid S$`[`$I$`]` $\mid S$`[`I`:`I`]` $\mid$ `str(`I`)` $\mid S$`.join(`SL`)` $\mid \ldots$
$L \ ::= SL \mid IL$
$SL ::= sl \mid S$`.split(`S`)` $\mid$ `[`S `for var in` L`]` $\mid \ldots$
$IL ::= il \mid$ `[`I `for var in` L`]`
$x \ ::=$ vars with only int values in examples
$s \ ::=$ vars with only string values in examples
$sl \ ::=$ vars with only list-of-string values in examples
$il \ ::=$ vars with only list-of-int values in examples

Figure 4. Fragment of the expression grammar supported by SNIPPY. *E* is the root expression; *I* are integer expressions; *S* are string expressions; *SL* are list-of-strings expressions; *IL* are list-of-ints expressions.

Python as a typed language

Python is a dynamically typed language, i.e., type-checking is deferred until runtime. This means a lot of flexibility, even allowing a variable at one statement in the program to take on different types each time the statement is executed. However, our synthesizer runs statically (i.e., before runtime), at which point type information in Python is not available. Without type information for the generated expressions, any possible operation could be applied to a given expression, which makes the search space intractably large. Instead, we treat Python as a typed language with integers, strings, booleans, and lists and dictionaries that are homogeneous in type.

We do this by inferring variable types from the examples, and designing a grammar that has productions for each type, restricting the parameters of functions to certain types. To provide a sense of what this grammar looks like, Figure 4 shows a fragment of the expression grammar used in SNIPPY (note it is far from the full grammar). This grammar only allows for homogeneous lists (lists of all integers or lists of all strings), and also restricts certain functions to operate only on certain types, e.g., `max` is restricted to lists of integers even though `max` can run on iterable types of all kinds. While this limits the SNIPPY synthesizer, it also greatly reduces the number of programs that must be searched, which lets SNIPPY find large and useful programs within seconds.

List and Dictionary Comprehensions

Our synthesizer is a *bottom-up* synthesizer, meaning that it uses production rules from the grammar to combine previously discovered expressions into larger expressions. For example, using the grammar in Figure 4 and the synthesis call in Figure 1(c), the enumerator starts with the terminal production rules, enumerating constants, including `' '`, and the variables, in this case the string `s`. The enumerator then applies the rule $S ::= S$`.split(`S`)` to create all expressions of this form, with S replaced by any of the current string expressions, which are the string constants and `s`, generating (among others), `' '.split(s)` and `s.split(' ')`. The process continues, iteratively building larger and larger expressions. Each expression is tested against the examples, and if an expression that satisfies all examples is found, it is returned to the user.

Unfortunately, this process breaks down for list and dictionary comprehensions that SNIPPY must support. Consider the production $SL ::=$ `[`S `for var in` L`]` for making a list of strings from another list. The nonterminal S (the "body" of the comprehension) can include the new variable `var`, so S in this production is actually derived from a different grammar than L (since L cannot access `var`). To account for this, our synthesizer uses the approach proposed in [34]: expressions are generally built using a standard bottom-up approach, except bodies of comprehensions, which are built using *nested* bottom-up enumerations. For $SL ::=$ `[`S `for var in` L`]`, the enumerator builds L bottom-up, and for each generated L it fixes L in the expression `[`S `for var in` L`]`, and then starts a nested enumeration for S.

Performing this new *nested* enumeration for comprehension bodies has three benefits: (1) the nested enumeration can include the comprehension variable `var` (2) the nested enumeration can omit the production rules for comprehensions, preventing nested comprehensions and reducing the search space, and (3) most importantly, this nested enumeration can incorporate the inputs from the examples into *observational equivalence*, as the external enumeration does, with the help of another technique called *input extension* [34], and by doing so drastically reduce the search space of the nested enumeration.

Discovering string literals

Bottom-up enumeration has to start with a set of constant literals like `0` and `1`. To make synthesis efficient this set must be small. SNIPPY supports the constants `-1`, `0`, `1`, and `' '`, though more constants can be enumerated with simple post-processing. E.g., the expression `1 + 1 + 1` (generated by the grammar) can be simplified via post-processing to `3`.

However, for string problems, particularly string wrangling, there is often a need for string literals that *cannot* be discovered by the grammar, even with the above post-processing. One could, in theory, ensure the grammar contained string concatenation and every ASCII character, which would let the synthesizer construct any ASCII string, but this would make the search unusably slow, and still only cover English strings.

Another approach to this problem, taken by the benchmark suite of the competition for syntax-guided solvers [4], is to adapt the synthesizer's grammar for every problem, adding only the string literals needed on a per-task basis. However, this does not suit a live system like SNIPPY where the user can ask for arbitrary problems to be solved, since we do not want to burden the user with specifying string literals each time.

SNIPPY implements a middle ground between these approaches. Its grammar is initialized with a single string literal, `' '`, but is extended as necessary. Before enumeration, the synthesizer searches the outputs of the provided examples for substrings that do not appear in any of the inputs or the grammar, and adds them to the grammar. For instance, in our motivating example, the output initials are separated by the character `'.'`, which is not part of the SNIPPY grammar. If it does not appear in any of the input variables in the example, then it will be added to the grammar. However, if the user had added a new variable `dot` with the value `'.'`, then a new constant is no longer needed, and it will not be added.

Variable usage

Synthesizers often require that the synthesized expression use all variables available, a property called *relevancy* [16]. However, this requirement does not make sense for SnipPy. The example inputs include every variable in scope during the assignment statement being synthesized, including inputs to previous steps and intermediate results. Forcing the synthesizer to use all these variables can lead to unintuitive and hard to explain results, and will likely cause no result to be found.

However, removing the relevancy requirement entirely is also problematic. Let us assume the user gives the variable `out` the value `-2`. Several programs will evaluate to `-2` on the inputs from the examples, but the first one found by the synthesizer is `-1 + -1`, which is then post-processed into `-2` and returned to the user. PBE tools approach this scenario in one of two ways: by requiring the user to add another example to show `-2` is not *always* the output, or by biasing the synthesis process heavily against constants. This bias is sometimes so severe that, for instance, if `-2` is needed within a larger expression like `s[-2]` (the second to last character of the string `s`), the synthesizer would prefer a program where `-2` is generated with as few constants as possible, such as `a.find(b) + a.find(b)` when `b` is not a substring of `a`. The result, `s[a.find(b) + a.find(b)]`, is both less general—it works for the current inputs, but may not work for others—and makes little sense unless the user understands this biased model.

SnipPy does not bias its search against constants in general, but applies a reduced relevancy requirement by not returning a result program that does not use variables. For example, it will construct the expression `-2`, and use it to construct larger programs such as `s[-2]`. However, if the example output is `-2`, the synthesizer will not return `-2` as the target program, and instead continue searching for a more suitable program.

STUDY METHODOLOGY

To evaluate SnipPy, we conducted a within-subjects user study comparing Python development using SnipPy to developing with Projection Boxes.

We focused our study on the following research questions:

RQ1: Does SSL-PBE make a difference in speed and correctness compared to an unaided development process?
RQ2: How useful is SSL-PBE, as measured for example by the percentage of the final code that is written by SnipPy vs by the user?
RQ3: Do users report positive experiences with SSL-PBE?
RQ4: How does SSL-PBE compare to searching the internet for help?

We recruited 13 participants, 9 male, 4 female, with between 3 and 23 years of programming experience (average 8.7) for a two-hour user study. We asked potential participants to self-rate their Python experience on a scale of 1 (not familiar at all) to 5 (extremely familiar), and selected participants with experience between 2 and 4.

Tasks

Each participant solved 4 Python tasks from the competitive programming website codewars.com. The tasks are:

A: `abbreviate`[2]: Convert full name to lowercase initials separated by periods
B: `count-duplicates`[3]: return number of characters that appear in a given list more than once
C: `max-min`[4]: compute min and max of a list
D: `palindrome`[5]: compute whether a string can be a palindrome if rotated by one ore more characters

We grouped the tasks into two sets that provided the same level of difficulty: (A,B) and (C,D). A and C were easier tasks, while B and D were harder tasks. We used two order of the tasks: (A,B);(C,D) and (C,D);(A,B).

Control and Test Conditions

We use two tool configuration, one control and one test. The control is called Projection Boxes, which in this case will refer to the live visualization *without* SnipPy. The test condition is SnipPy. Since users had never seen Projection Boxes before, we randomized the order of the control/test to prevent any advantage to SnipPy users from being more experienced with Projection Boxes.

Since we have two orders of the control/test, and two orders of the tasks, we have four groups:

1. SnipPy: (A,B) ; Projection Boxes: (C,D) (4 users)
2. Projection Boxes: (C,D) ; SnipPy: (A,B) (2 users)
3. SnipPy: (C,D) ; Projection Boxes: (A,B) (4 users)
4. Projection Boxes: (A,B) ; SnipPy: (C,D) (3 users)

Participants were randomly assigned into the above groups, maintaining even group sizes, divided by level of expertise. Participants were then asked to solve the first two tasks with the first tool and the second two tasks with the second tool.

Study Session

The study was conducted remotely via video conferencing. Because SnipPy requires installing and setting up a runtime environment, the study was also conducted via remote control.

Users were first given a survey about their background as programmers. Additionally, users were asked whether they have experience with other synthesis tools, either by prior use of academic tools or using smart code completion products.

We developed two instructional videos, one for Projection Boxes, and one for SnipPy. The SnipPy video assumes Projection Boxes had been introduced. Participants starting with Projection Boxes were shown the Projection Boxes video before using Projection Boxes, then the SnipPy video before using SnipPy. Participants starting with SnipPy were shown both the Projection Boxes and the SnipPy video before starting with SnipPy, and no additional video before using Projection Boxes. After the instructional video for a tool participants were given a demo task not

[2] https://www.codewars.com/kata/554b4ac871d6813a03000035, an additional step asking for a lowercase abbreviation was added to make the task more difficult.
[3] https://www.codewars.com/kata/54bf1c2cd5b56cc47f0007a1
[4] https://www.codewars.com/kata/554b4ac871d6813a03000035
[5] https://www.codewars.com/kata/5a8fbe73373c2e904700008c

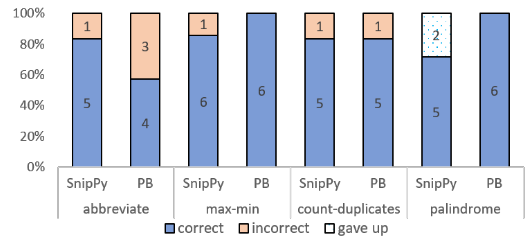


Figure 5. Percentage and number of correct answers for each task.

		Easy		Hard	
		`abbreviate`	`max-min`	`count-duplicates`	`palindrome`
All	avg	22%	33%	-7%	2%
	med	21%	139%	-18%	-21%
Correct	avg	21%	13%	-25%	-1%
	med	28%	80%	-36%	-21%

Table 1. Changes in session times in SNIPPY compared to PROJECTION BOXES. Negative percentage indicates a speedup.

related to the study tasks for a few minutes of guided exploration of the tool. Users were also given an opportunity to ask questions about the tool after the demo tasks.

Participants then performed the tasks. When using PROJECTION BOXES, participants were given a web browser and free internet access to search for code, whereas SNIPPY users were only given SNIPPY. Participants were instructed to use SNIPPY as much (or as little) as they wish. Tasks included suggested examples to help users check their answers. Participants determined when a task ended, either by saying they completed it or by giving up on the current task and moving to the next task. Each task was capped at 35 minutes.

After all four tasks, users were given a final survey asking them to reflect on ways SNIPPY helped them to write code.

RESULTS

Session times and correctness (RQ1)

Figure 5 shows the number and percentage of correct answers to each task, determined via 10 unit tests for each task that were run after the session ended.. Our study is too small to show statistical significance, but we examine the tasks with the most notable differences: `abbreviate` and `palindrome`.

In `abbreviate`, participants who did not have SNIPPY made more mistakes. Most of the mistakes in `abbreviate` had to do with using an incorrect separator between initials. Users who used SNIPPY to synthesize the code that combines the first letters of the names did not make this mistake, as they used the given expected output to generate the correct code. Also, of the two participants who gave up on `palindrome` (P1, and P12), P12 did not use the synthesize function feature of SNIPPY for the entire 20 minute session, essentially reducing the session to a PROJECTION BOXES session.

Table 1 shows the change in session times from PROJECTION BOXES to SNIPPY, with PROJECTION BOXES as a baseline. A negative percentage indicate a reduction in session time (speedup), whereas a positive percentage indicates an increase in session time (slowdown). The numbers are provided both

		`abbreviate`	`max-min`	`count-duplicates`	`palindrome`
Useful calls / Total calls	avg	61%	36%	36%	27%
	med	58%	25%	20%	20%
Synthesized / All code	avg	47%	66%	28%	18%
	med	47%	65%	23%	15%

Table 2. Synthesis calls in relation to the final solution

for all sessions, and for all sessions where the participants found correct solutions.

While our study is not large enough to provide statistically significant results, broadly speaking, our preliminary numbers suggest a possible pattern based on how hard the task is. Indeed, recall that `abbreviate` and `max-min` were easier tasks, while `count-duplicates` and `palindrome` were harder tasks. In the two easier tasks, SNIPPY appears to make the sessions longer, whereas for the two harder tasks, SNIPPY appears to make the sessions shorter.

There are two factors that could explain this. First, for easier tasks, writing the code directly can be faster than using a synthesizer, especially if using the synthesizer requires multiple round trips (e.g., if the first example is insufficient, and a second example is needed).

Our results echo those of previous studies such as Galenson et al. [11], where using synthesis in a freeform manner more than doubled the time to completion of the task. However, we are encouraged by the fact that our slowdown is not as severe, indicating the live programming aspect of SSL-PBE helps mitigate some of the overhead of using the tool.

Usefulness of Synthesis (RQ2)

We measured how many synthesis calls were useful to the programmer. We counted as useful any synthesis call where a non-trivial part of the synthesized code was used in the participant's final program. As not useful we counted all other calls, including calls where synthesis timed out. The results are in the top part of Table 2.

In general, we see that a sizeable number of synthesis calls are not useful. Still, for every task but `palindrome` at least one participant had 100% useful synthesis calls. P8 in `count-duplicates` and P12 in `palindrome` had no useful calls in the course of solving their task.

Additionally, we measured how much of each user's final program originated from the synthesizer. To do this, we computed the proportion of tokens (using Python's own code tokenizer) in the user's answer that came from synthesis. We did not count things not generated by SNIPPY, such as the assignment into a variable or the return statement of the function, and in the case the user renamed a variable in the synthesized code, the variable name was counted as user code, while the rest of the snippet was counted as synthesizer code. In short, we measured the manual effort that was performed by the user and how much was delegated to the synthesizer. The average and median results are in the bottom part of Table 2.

Because SNIPPY does not generate things like the return statement or assignments, and because `palindrome` required a

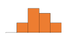
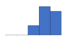
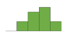
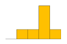
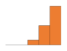
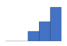
	Average	Median	Dist.
SnipPy helped me write my code	3.46	3	
SnipPy was easy to use	4.23	4	
I would use SnipPy again	3.54	4	
SnipPy would be useful beyond today's tasks	3.69	4	
I would like to have Projection Boxes	4.54	5	
I would like to have SnipPy available	4.38	5	

Table 3. Survey Results. All questions are on a likert scale where 1 is "Disagree" and 5 is "Agree".

loop to be manually written, 100% synthesized was not a possible result. Broadly speaking, programmers tackling the harder tasks (`count-duplicates` and `palindrome`) wrote more of the program manually. These tasks are harder to break up into synthesis-ready chunks, and in some approaches to the task, the synthesizer will no longer help. All tasks except `palindrome` *could* be solved almost entirely by synthesis, and the largest portion synthesized by one user was 83% in `count-duplicates` (P3, 39 of 47 tokens). The way a problem is deconstructed for synthesis is crucial to how much of it can be synthesized. Users whose breakdown of the task meshed with SnipPy could synthesize every step and write almost no code, whereas users who did not come up with such a breakdown were still able to synthesize code, but to a lesser extent. Overall, we see that although a lot of synthesis calls were not successful, calls that were successful provided users with substantial parts of the solution.

SnipPy *and data-dependent loops*

A very frequent cause of failed synthesis calls was an attempt to synthesize statements inside loops that cause a data dependency between the iterations, or a loop where a variable is written to in one iteration, then used in the next, a simple example of which is `sum = sum + i`. Dependent loops are a known hard problem in program synthesis [34], and are notoriously hard to specify correctly even under the best conditions. Attempting to synthesize these was a gap in the participants' understanding of the synthesizer limitations (even for participants who were previously familiar with synthesis tools). We discuss the implications of this gap in the next section.

User survey (RQ3)

Table 3 shows the results of our survey, including the average, median and the distribution of scores. In the "Discussion" section below we will discuss in more detail the factors that affect the utility of SnipPy, and explain these results. For now, we do note that, even though the scores on utility are lower than others, because SnipPy can be invoked as needed, users still overall said they would like to have SnipPy available.

Comparison to Searching the Internet (RQ4)

One of the questions in our post-study questionnaire asked participants to compare SnipPy to searching the internet. Overall, 23.1% of participants said that they preferred SnipPy to the internet, 15.4% said they preferred the internet, and the remainder said that it depends and explained the trade-offs.

P1, P10, and P12 stated that SnipPy can work well even if one does not have a clear picture of what they should search for online. P1 also said that SnipPy solutions are more concise.

One recurrent theme we observed is that searching the internet and SnipPy supplement each other, each having different strengths. (In fact P7 said that they would first try SnipPy and if that didn't work they would search the internet.) For the kinds of code snippets that SnipPy can generate, SnipPy is better, for several reasons that were explicitly mentioned by our participants. First, SnipPy can find a solution quickly without imposing the cognitive burden of switching to another window or tool. Second, SnipPy can find compact solutions. Third SnipPy correctly connects the generated snippet to the surrounding code – in contrast solutions from the internet often need to be adapted and correctly glued into the surrounding context, a non-trivial and error prone task. Finally SnipPy can work well even when the programmer does not have a clear picture of what to search for on the internet.

On the other hand, however, SnipPy (as with any synthesis tool) has limitations in what it can do, and this affects its utility compared to searching the internet.

DISCUSSION

Usage of Small-Step Live Programming by Example

Through our study, we identify three predominant ways in which SnipPy helped programmers.

First, some participants used SnipPy in precisely the way we anticipated: decomposing the problems into smaller steps, then editing the live data to make SnipPy generate code snippets for those smaller steps. In these cases, SnipPy does not help *algorithmically*, but instead provides help with individual steps of a larger algorithm. The most successful uses of SnipPy were ones where the programmer came up with the high-level strategy, and SnipPy helped with the individual steps.

Second, some participants used SnipPy "on the side": they would stop coding the main task they were working on, and start writing code separately to get SnipPy to generate a useful snippet. For example, P1 used this approach to generate code for rotating a string by a constant number—3 characters. Once the code for rotating a string by 3 was generated, P1 took the snippet, generalized it to an arbitrary rotation by `k` and placed it inside a loop. This interruption in the flow of programming leads to a less fluid process, but still uses SnipPy effectively.

Third, some participant used SnipPy to recall details about Python syntax or Python libraries they had forgotten. In this situation, the programmer might know how to do something, but forgot (or possibly is not fully familiar with) the details of expressing it in Python. Examples of such easily forgotten details, especially for those with less Python experience (but even for programmers with a lot of Python experience) include: the order of parameters to certain methods, like `split`; the exact syntax of dictionary comprehension; the exact syntax of list/dictionary comprehension with an embedded filter; the name of library functions, e.g., for converting characters to lower case, for returning the keys *and* values of a dictionary, or for returning the elements *and* indices of a list.

Understanding Synthesized Code

When a synthesizer generates code, there is a question of how well the programmer understands the code. In our study, programmers checked that the code appeared reasonable but did not try to understand the details. In some cases, participants remarked on the synthesized code being simpler than they would have written. In other cases, participants explicitly said that the code worked, but they did not fully understand it.

One may be concerned about correctness when programmers use code that they do not fully understand, but in our study we observed that this did not drive programmers to an incorrect solution. We also observed users sometimes take code snippets they do not fully understand from the internet in our control setting, and are not the first to document this [6].

However, we observed a much more interesting problem when programmers do not understand the synthesized code: it leads to the mindset that the synthesizer is all-or-nothing: either the synthesizer eventually generates code that works, or if not, then the programmer just gives up on the synthesizer altogether. Unfortunately, this can prevent the programmer from using an almost-correct solution generated by the synthesizer.

This happened to P2 who used SNIPPY in `palindrome` to generate an almost-correct solution. Given the setup that the programmer used, the synthesized code only worked for lists of size 4. Had the programmer generalized 4 to an expression for the list's length, the problem would have been solved. Instead they tried unsuccessfully to generalize the examples and re-synthesize, and eventually gave up on the problem.

More generally, this leads us to the following takeaway:

Because programmers do not try to understand the code generated by the synthesizer, they unnecessarily shy away from trying to use partial results from the synthesizer.

This in turn points to a possible direction for future research, namely on understandability and usability of partial results in synthesizer-generated code (something that has already started being explored, for example in Wrex [9] and Bester [35]).

Mental Model of the Synthesizer

We have noticed that the mental model that the programmer has of the synthesizer is very important. We start by framing our discussion in terms of the well-known gulfs of evaluation and execution. The gulf of evaluation captures how well a user can understand the internal state of the system. The gulf of execution captures how well a user can discover how to make the system take steps toward an ulterior goal. In the setting of programming, the gulf of evaluation relates to understanding the program state and its result; the gulf of execution relates to understanding what kinds of statements should be written next to finish a task. E.g., at a command line prompt, showing the current directory and the computer name eases the gulf of evaluation (exposing internal state); making commands at the prompt discoverable via auto-complete or command-line searches may ease the gulf of execution (making it easier to choose the next step toward a goal).

PROJECTION BOXES help with the gulf of evaluation, since they make the internal state of the program visible at all times. However, they do not help explicitly with the gulf of execution, since they do not help directly with writing the code.

SNIPPY provides this missing aspect of PROJECTION BOXES, easing the gulf of execution with explicit help discovering the next statement toward a broader end goal. However, this oversimplification misses an important subtlety. While SNIPPY does ease the gulf of execution in some ways, it introduces a different kind of burden that also relates to the gulf of execution: the programmer must now pick between SNIPPY and one of three other approaches: (1) writing the code by hand, (2) searching the internet, or (3) manually decomposing the problem into smaller pieces to try with SNIPPY.

So, in essence, we have shifted the gulf of execution from one kind of gulf to another: from figuring out what statement to write next, to figuring out if SNIPPY should be used for the next statement. This new gulf of execution is particularly interesting because for programmers to make the choice between SNIPPY and other approaches, we have observed that they must have an *accurate* mental model of the synthesizer's abilities. If a programmer broadly understands (through trial and error) what kinds of tasks the synthesizer can do, they will know when to invoke the synthesizer and when to try something else. However, if the programmer has a poor mental model of the synthesizer's ability (e.g., one that *overestimates* the synthesizer's ability), then the programmer might waste time and energy trying to get the synthesizer to do something that it simply cannot. This leads to frustration, making it less likely that the synthesizer will be used the next time around. Furthermore, if the programmer *underestimates* the synthesizer's ability, they will under-utilize the tool.

We introduce the term **user-synthesizer gap** *to refer to this gap between the user's mental model of the synthesizer's abilities and the actual abilities of the synthesizer.*

We are not the first to notice this kind of effect. Lau [24] explored the related topic of a user's *trust* of the synthesizer, concluding that the adoption of Programming by Demonstration tools is held back by tool behaviors that undermine that trust. Gero et al. [12] explored mental models of AI agents in an interactive game, strengthening the conclusion that mistrusting the system is detrimental to success of the user, but also finding that: (1) users with a generally good understanding of AI systems developed a better mental model of the AI agent and (2) people tended to overestimate the AI's abilities.

The way the *user-synthesizer gap* manifested itself in our study shows that the overestimation of the AI system's capabilities documented by Gero et al. also occurs for synthesizers, but that it may involve *underestimating* the synthesizer's ability instead. Because state-of-the-art general-purpose synthesizers still cannot generate *all* the necessary code in a real setting, the only way a synthesizer can help a programmer is on subproblems to a larger task. In this situation, the *user-synthesizer gap* will inevitably come into play. This is less pronounced in domain-specific tools, as the limits of the domain act as an accurate mental and actual model for the synthesizer's limits.

We observed three properties about this gap. First, it is a much bigger problem if the user over-estimates the synthesizer's ability than underestimates it. Second, the larger the gap is, the more difficult it becomes for the programmer to make choices about how to incorporate the synthesizer into programming tasks. Third, this gap is self-correcting in some ways, in that if the gap is large, programmers eventually understand this, and adjust their mental model to reduce the gap. Consequently, as the programmer learns more about the synthesizer through trial and error, the gap can decrease over time, but this is a non-trivial learning curve that takes time, and can be a significant impediment to the adoption of synthesizers.

All the above observations lead us to believe that reducing the *user-synthesizer gap* represents an impactful future research arc that has the potential to further unlock the potential of state-of-the-art synthesis techniques.

LIMITATIONS

While the results of the study are promising, our study has certain limitations that remain to be addressed in future work.

Our study compares SSL-PBE with live programming augmented with searching the internet in a browser. Further studies would be needed to compare SSL-PBE to other interaction models, such as web searches or knowledge bases embedded in the IDE [28, 10], big-step synthesis tools in and out of the IDE [21], and smart code completion [17].

There are also several threats to the validity of our results. Our survey was conducted in the presence of one of the authors, which could lead to a social desirability bias. Additionally, the phrasing of questions was not neutral (e.g. "SNIPPY helped me write my code" instead of "How helpful was SNIPPY in writing your code").

There may also be a bias in our findings on users' understanding of the synthesized code. Our tasks each included one or more examples participants could input into the live programming environment, which could limit users' view of the code to those inputs and discourage them from examining the synthesized code further.

Finally, the small sample size and short length of tasks could be a threat to the internal validity of the study. Individual differences in coding speed could affect the conclusions we drew from the length of the programming sessions.

CONCLUSION

We introduced a new paradigm called *Small-Step Live Programming by Example* and discussed its implementation in SNIPPY. Through a within-subjects study we demonstrated that this paradigm is easy to use, and is most effective in non-trivial tasks. We also found that almost all participants preferred SNIPPY over searching the internet in some cases. Furthermore, our study showed that most users did not attempt to understand the code deeply, which resulted in an all-or-nothing approach to using SNIPPY's output. Finally, we identified the "*user-synthesizer gap*", which describes the gap between the user's mental model of the synthesizer's capabilities and its actual capabilities. We believe that reducing this gap represents an important direction for future research.

ACKNOWLEDGMENTS

This work was supported in part by NSF grants 1943623 and 1911149.

REFERENCES

[1] 2019. Alfie. `https://alfie.prodo.ai/`. (2019). Accessed: 2019-09-01.

[2] 2019. LightTable. `http://lighttable.com/`. (2019). Accessed: 2019-09-01.

[3] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive program synthesis. In *International Conference on Computer Aided Verification*. Springer, 934–950.

[4] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. 2017. Sygus-comp 2017: Results and analysis. *arXiv preprint arXiv:1711.11438* (2017).

[5] Benjamin Biegel, Benedikt Lesch, and Stephan Diehl. 2015. Live object exploration: Observing and manipulating behavior and state of Java objects. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 581–585. DOI: `http://dx.doi.org/10.1109/ICSM.2015.7332518`

[6] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. Association for Computing Machinery, New York, NY, USA, 1589–1598. DOI: `http://dx.doi.org/10.1145/1518701.1518944`

[7] Sarah E Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*. 963–975.

[8] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 341–354. DOI: `http://dx.doi.org/10.1145/2908080.2908103`

[9] 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. DOI: `http://dx.doi.org/10.1145/3313831.3376442`

[10] Ethan Fast, Daniel Steffee, Lucy Wang, Joel R. Brandt, and Michael S. Bernstein. 2014. Emergent, Crowd-Scale Programming Practice in the IDE. In *Proceedings of the*

SIGCHI Conference on Human Factors in Computing Systems (CHI '14). Association for Computing Machinery, New York, NY, USA, 2491–2500. DOI: http://dx.doi.org/10.1145/2556288.2556998

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

[12] Katy Ilonka Gero, Zahra Ashktorab, Casey Dugan, Qian Pan, James Johnson, Werner Geyer, Maria Ruiz, Sarah Miller, David R Millen, Murray Campbell, and others. 2020. Mental Models of AI Agents in a Cooperative Game Setting. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–12.

[13] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330.

[14] Sumit Gulwani, Mikaël Mayer, Filip Niksic, and Ruzica Piskac. 2015. StriSynth: synthesis for live programming. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 701–704.

[15] Philip J. Guo. 2013. Online Python Tutor: Embeddable Web-based Program Visualization for Cs Education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE '13)*. ACM, New York, NY, USA, 579–584. DOI: http://dx.doi.org/10.1145/2445196.2445368

[16] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2019. Program synthesis by type-guided abstraction refinement. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–28.

[17] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 27–38.

[18] Christopher Michael Hancock. 2003. *Real-time Programming and the Big Ideas of Computational Literacy*. Ph.D. Dissertation. Cambridge, MA, USA. AAI0805688.

[19] Brian Hempel, Justin Lubin, and Ravi Chugh. 2019. Sketch-n-Sketch: Output-Directed Programming for SVG. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 281–292.

[20] Qinheping Hu, Roopsha Samanta, Rishabh Singh, and Loris D'Antoni. 2019. Direct Manipulation for Imperative Programs. In *International Static Analysis Symposium*. Springer, 347–367.

[21] Jinru Hua and Sarfraz Khurshid. 2017. EdSketch: Execution-driven sketching for Java. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*. ACM, 162–171.

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

[23] Jan-Peter Krämer, Joachim Kurz, Thorsten Karrer, and Jan O. Borchers. 2014. How live coding affects developers' coding behavior. *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2014), 5–8.

[24] Tessa Lau and others. 2008. Why PBD systems fail: Lessons learned for usable AI. In *CHI 2008 Workshop on Usable AI*.

[25] Vu Le and Sumit Gulwani. 2014. FlashExtract: a framework for data extraction by examples. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 542–553.

[26] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: semantics-based repair of Java programs via symbolic PathFinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 376–379.

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

[28] Hongwei Li, Zhao Xuejiao, Zhenchang Xing, Lingfeng Bao, Xin Peng, Dongjing Gao, and Wenyun Zhao. 2015. AmAssist: In-IDE ambient search of online programming resources. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015 - Proceedings* (04 2015), 390–398. DOI: http://dx.doi.org/10.1109/SANER.2015.7081849

[29] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 166–178.

[30] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. *ACM SIGPLAN Notices* 51, 1 (2016), 298–312.

[31] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional Evaluation with Direct Manipulation. *Proc. ACM Program. Lang.* 2, OOPSLA, Article Article 127

(Oct. 2018), 28 pages. DOI: http://dx.doi.org/10.1145/3276497

[32] Cyrus Omar, Ian Voysey, Ravi Chugh, and Matthew A. Hammer. 2019. Live Functional Programming with Typed Holes. *Proc. ACM Program. Lang.* 3, POPL, Article 14 (Jan. 2019), 32 pages. DOI: http://dx.doi.org/10.1145/3290327

[33] 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 (2006), 941–961.

[34] Hila Peleg, Roi Gabay, Shachar Itzhaky, and Eran Yahav. 2020. Programming with a Read-Eval-Synth Loop. (2020). Manuscript submitted for publication.

[35] 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*.

[36] Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming not only by example. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 1114–1124.

[37] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples Into General-purpose Source Code. *The Art, Science, and Engineering of Programming* 3, 3 (2019).

[38] 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*. 1–4.

[39] Christopher Schuster and Cormac Flanagan. 2016. Live programming by example: using direct manipulation for live program synthesis. In *LIVE Workshop*.

[40] Ben Shneiderman. 1983. Direct Manipulation: A Step Beyond Programming Languages. *Computer* 16, 8 (1983), 57–69.

[41] Calvin Smith and Aws Albarghouthi. 2016. MapReduce program synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 326–340.

[42] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. 404–415.

[43] S. L. Tanimoto. 2013. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*. 31–34. DOI: http://dx.doi.org/10.1109/LIVE.2013.6617346

[44] Abhishek Udupa, Arun Raghavan, Jyotirmoy V Deshmukh, Sela Mador-Haim, Milo MK Martin, and Rajeev Alur. 2013. TRANSIT: specifying protocols with concolic snippets. *ACM SIGPLAN Notices* 48, 6 (2013), 287–296.

[45] Priyan Vaithilingam and Philip J Guo. 2019. Bespoke: Interactively Synthesizing Custom GUIs from Command-Line Applications By Demonstration. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 563–576.

[46] Sahil Verma and Subhajit Roy. 2017. Synergistic Debug-Repair of Heap Manipulations. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 163–173.

[47] Bret Victor. 2012. Learnable Programming. (2012). http://worrydream.com/LearnableProgramming/

[48] E. M. Wilcox, J. W. Atwood, M. M. Burnett, J. J. Cadiz, and C. R. Cook. 1997. Does Continuous Visual Feedback Aid Debugging in Direct-manipulation Programming Systems?. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '97)*. ACM, New York, NY, USA, 258–265. DOI: http://dx.doi.org/10.1145/258549.258721

[49] Glynn Winskel. 1993. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, USA.

[50] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. 2017. Precise Condition Synthesis for Program Repair. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, Vol. 00. 416–426. DOI: http://dx.doi.org/10.1109/ICSE.2017.45

[51] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. 2018. Automated Migration of Hierarchical Data to Relational Tables Using Programming-by-example. *Proc. VLDB Endow.* 11, 5 (Jan. 2018), 580–593. DOI: http://dx.doi.org/10.1145/3187009.3177735