Interactive Program Synthesis by Augmented Examples

Tianyi Zhang[†], **London Lowmanstone**[†], **Xinyu Wang**[§], **Elena L. Glassman**[†]
[†]Harvard University, MA, USA
[§]University of Michigan, Ann Arbor, MI, USA
{tianyi, eglassman}@seas.harvard.edu, lowmanstone@college.harvard.edu, xwangsd@umich.edu

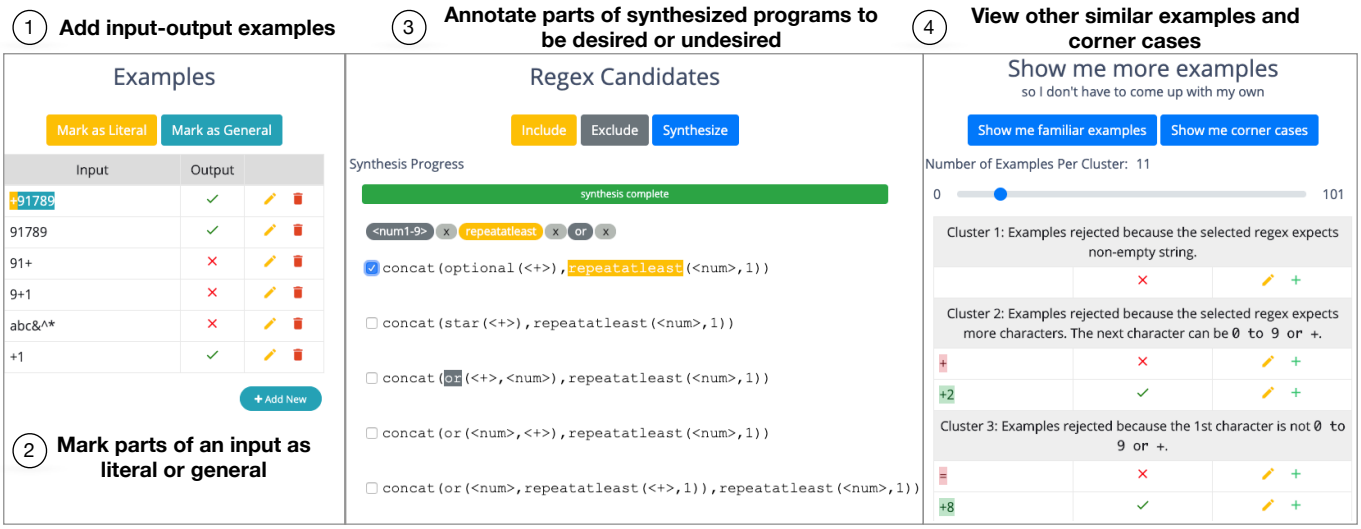


Figure 1: A user interactively synthesizes a regular expression (regex) that accepts phone numbers starting with an optional '+'. After 1) adding some positive and negative examples, the user 2) marks the + sign as literal and the following numbers as a general class of digits to specify how individual characters should be treated by the synthesizer. To refine the synthesis result, the user can either 3) directly mark subexpressions as desired or undesired in the final regex, or 4) ask for additional examples to validate and enhance her understanding of a regex candidate as well as identify and label counterexamples, therefore adding them to the set of user-provided input-output examples.

ABSTRACT

Programming-by-example (PBE) has become an increasingly popular component in software development tools, human-robot interaction, and end-user programming. A long-standing challenge in PBE is the inherent ambiguity in user-provided examples. This paper presents an interaction model to disambiguate user intent and reduce the cognitive load of understanding and validating synthesized programs. Our model provides two types of augmentations to user-given examples: 1) *semantic augmentation* where a user can specify how different aspects of an example should be treated by a synthesizer via light-weight annotations, and 2) *data augmentation* where the synthesizer generates additional examples to help the user understand and validate synthesized programs. We implement and demonstrate this interaction model in the domain of regular expressions, which is a popular mechanism for text processing and data wrangling and is often considered hard to master even for experienced programmers. A within-subjects user study with twelve participants shows that, compared with only inspecting and annotating synthesized programs, interacting with augmented examples significantly increases the success rate of finishing a programming task with less time and increases users' confidence of synthesized programs.

UIST '20, October 20–23, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7514-6/20/10 ...$15.00.
http://dx.doi.org/10.1145/3379337.3415900

Author Keywords

Program synthesis; disambiguation; example augmentation

CCS Concepts

•Human-centered computing → Human computer interaction (HCI); Interactive systems and tools;

INTRODUCTION

Program synthesis has been increasingly applied to many application domains, e.g., software development tools [15, 50, 62, 29], human-robot interaction [47, 35], and end-user programming [32, 22, 36, 6, 38]. Many program synthesis techniques

adopt a programming-by-example (PBE) paradigm, where users provide a set of input-output examples to describe the intended program behavior. Providing examples can be a natural way to express human intent, but examples are also a partial specification: they only describe desired program behavior on a limited number of inputs. There are many possible programs that match the user-provided examples but may diverge from the user-intended behavior on unseen inputs.

Many PBE systems require users to provide additional examples to resolve ambiguities. However, prior work has shown that users are reluctant to provide more than a few examples [31]. Even when they do, inexperienced users are not good at providing representative examples that cover the hypothetical input space [34, 44]. Several interactive approaches have been proposed to enable users to directly refine synthesized programs rather than crafting additional examples [42, 10, 30, 46]. Unfortunately, these approaches assume users are capable of and willing to inspect and edit programs. This fundamental assumption may not hold in many domains, especially for novice programmers and end-users without enough expertise in programming. Some approaches have explored the middle ground by rendering synthesized code in a human-friendly format [42, 10], but users still need to spend substantial time comprehending the program. This is cognitively demanding, especially when multiple candidate programs are synthesized and provided to users.

In this paper, we argue that, for both programmers and non-programmers, it is more natural to clarify their intent on top of the *original* examples they give, rather than the programs synthesized by a *foreign* PBE system. We introduce an interaction model that supports two types of augmentation on user-given examples: (1) *Semantic augmentation* enables users to annotate how input examples should or should not be generalized to other similar contexts by a synthesizer. (2) *Data augmentation* automatically generates many corner cases for users to validate and enhance their understanding of synthesized programs and potentially choose to label as counterexamples. We hypothesize that it would be less mentally demanding for users to interactively augment original input-output examples with richer semantics and select from additional examples automatically generated for them, compared with inspecting synthesized programs and editing them.

We instantiate our interaction model in the domain of regular expressions and develop an interactive synthesizer called REGAE. Regular expressions (regexes henceforth) are a versatile text-processing utility that has found numerous applications ranging from search and replacement to input validation. In addition to being heavily used by programmers, regexes have gained popularity among data scientists and end-users. Despite their popularity, regexes are considered hard to understand and compose, as well as error-prone, even for experienced programmers [5, 4, 55]. As an instance of *semantic augmentation*, REGAE allows users to mark parts of input examples that should be kept verbatim or generalized to a character class such as digits in a regex. As an instance of *data augmentation*, REGAE automatically generates two types of additional inputs, those similar to the user-provided examples and corner cases, and then clusters them based on how synthesized programs behave on them. Hence users do not have to mentally simulate the synthesized regexs on those new inputs. When multiple candidate regexes are selected, REGAE generates input examples that exhibit behavioral discrepancies among them.

We conducted a within-subjects study that involves twelve programmers with various levels of expertise in regular expressions. When interacting with augmented examples, all twelve participants successfully completed regex tasks from Stack Overflow (SO). In contrast, only four participants completed the tasks by manually providing counterexamples and annotating regex candidates as proposed in [46]. Participants felt much more confident about the synthesis result and reported less mental demand and stress, when they were able to see how a regex operates on many other similar examples and corner cases. In addition to the lab study, we demonstrated that our interactive synthesizer can also be used to solve another twenty SO tasks in an average of five synthesis iterations and six minutes, providing quantitative evidence about its effectiveness on a variety of complex regex tasks.

The contributions of this paper are as follows:

- an interaction model based on a combination of *semantic augmentation* and *data augmentation* of user-given examples to resolve intent ambiguity in program synthesis.
- an implementation of the proposed interaction model for regular expression synthesis, as well as a discussion of how to apply this interaction model to other domains.
- a within-subjects lab study and a case study demonstrating how users with various levels of expertise can interact with the synthesizer and guide it to generate desired programs.

RELATED WORK

Program synthesis is the task of automatically creating programs that match the user's intent expressed in some kind of specification. The problem of program synthesis dates back to the 1960s, with a lot of pioneering work on *deductive* program synthesis [21, 41, 40, 57]. Deductive program synthesis takes a complete formal specification as input, which in many cases proves to be as complicated as writing the program itself [24]. More recently, program synthesis techniques have adopted *inductive* specifications (e.g., input-output examples, demonstrations, natural-language descriptions) which are easier to provide by users. In this paper, we focus on augmenting programming-by-example (PBE) techniques that use examples as the specification. Recently, PBE has been pursued for various tasks, e.g., data extraction [33], data filtering [61], data visualization [59], string transformations [22], table transformations [23, 14], automated code repair [12, 50], map-reduce program generation [54], grammar generation [2], and implementing network configurations [51, 65].

Synthesis from ambiguous example-based specifications

A long-standing problem in PBE is the inherent ambiguity in user-provided examples. Since examples only specify the user's intended program behavior for a limited number of inputs, a PBE system may generate many *plausible* programs that are consistent with the user-provided examples but which do not produce intended results on additional inputs

that the user also cares about. Most PBE systems from the programming languages and machine learning communities use human-crafted or machine-learned inductive biases to resolve the ambiguity in example-based specifications. Such inductive biases specify which aspects of the examples to generalize and which programs to prioritize during back-end searches over the program space. Inductive biases can take various forms such as hand-crafted heuristics [22, 23], distance-based objective functions [11], hard-coded distributions [13, 9], and probabilistic models learned from large code corpora [49, 53]. Improving inductive biases in PBE systems can drastically reduce the time to arrive at a user-intended program, but it does not amount to mind-reading: it still needs appropriately chosen examples on which to perform induction and lacks flexibility to address different needs or preferences of users.

Clarifying user intent through interactions

When inductive biases fall short of inferring user-intended programs, many PBE systems fall back to having users actively provide additional examples to disambiguate their intent. This requires users to fully understand the synthesized programs, recognize undesired behavior, and craft counterexamples to refute the undesired behavior. This is cognitively demanding. Prior work has shown that users are reluctant to and not good at providing high-quality examples when using PBE systems [34, 44, 31]. Our work addresses this challenge of eliciting informative examples from the users by generating and clustering many similar examples and corner cases, based on the user's provided examples. Users do not have to stare at a synthesized program and fully understand its semantics. Instead, they can validate and enhance their understanding by glancing over those additional examples.

The idea of automatically generating distinguishing inputs for ambiguity resolution has also been explored by several other work [29, 42, 58]. Jha et al. propose to encode synthesized programs as logic formulae and then use a SMT solver to identify an input that causes those programs to produce different outputs. Scythe [58] is an interactive synthesizer for SQL queries. Similar to Jha et al. [29], it uses a SMT solver to find a distinguishing input when two synthesized SQL queries are not equivalent to each other. However, programs in many domains cannot be easily encoded as logic formulae and fed to SMT solvers. Though FlashProg [42] does not rely on SMT solving, it requires users to provide a large text document as a pool of additional inputs to test on and draw distinguishing inputs from. FlashProg also cannot identify hypothetical corner cases that are not included in the given document. Unlike these techniques, our system converts synthesized regexes to automata and uses *transition coverage* to explore the hypothetical input space and generate unseen data.

Direct program manipulation to guide synthesis

Instead of eliciting additional examples from users, several interactive approaches have been proposed to help users directly inspect and refine synthesized programs [42, 10, 30, 64]. FlashProg [42] paraphrases synthesized programs in English and clusters programs with common structures for ease of navigation. Wrex [10] transforms synthesized programs to readable code and allows users to directly edit them. Wranger [30] and STEPS [64] support task decomposition by allowing users to construct a target program step by step. Wranger [30] recommends a ranked list of primitive operators at each synthesis step and lets users select an operator and edit its parameters. STEPS [64] allows users to specify the latent structure in a text document using nested color blocks. However, users of such systems often find it hard to figure out the meaning of possible primitive operators [64] or the intermediate steps to reach the final solution [25]. Recently, Peleg et al. proposed a new interaction model that enables users to specify which parts of a synthesized program must be included or excluded in the next synthesis iteration [46]. Their key insight is that synthesized programs may have some parts that are completely wrong and some parts that are on the right track, which can be leveraged to prune the search space.

The major difference between our approach and these direct manipulation approaches is that we focus on augmenting interactions with examples rather than synthesized programs; this can reduce the user's need to engage with what might be a complicated synthesized program, even if it is rendered in a relatively human-friendly way. We argue that, compared with inspecting and editing programs, interacting with examples is a more natural way to disambiguate user intent, especially for novice programmers and end-users.

The semantic augmentation feature in our work shares some similarity to *abstract examples* [8]. Drachsler-Cohen et al. proposed an approach to generalize a concrete example to an abstract example by identifying which parts of the concrete example can vary and which parts must remain constant to produce the same output of a synthesized program [8]. The inferred abstract example is presented to users to help them decide whether the synthesized program is correct or not. Compared with abstract examples, semantic augmentation is rather an interactive feature for specification authoring. It provides a flexible interface for users to easily specify how a synthesizer should treat different parts of an input. In other words, semantic augmentation allows users to shape the inductive bias of the synthesizer, by specifying the direction of intended generalization for different parts of an input, from concrete to one of several different possible classes of characters, e.g., capital letters, digits, etc. LAPIS [43] and TOPES [52] introduced more sophisticated semantic patterns on input data, such as structural relationships and numeric constraints, which we wish to support in future work.

PRELIMINARIES

This section briefly reviews a standard PBE technique for synthesizing regular expressions. We choose to focus on the domain of regexes for two main reasons. First, regexes are a versatile text-processing utility with many applications. They are used by expert and novice programmers, as well as end-users with very limited programming knowledge. On the other hand, regexes are found to be hard to understand and compose, even for experienced programmers [5, 4, 55]. Therefore, it is appealing to automatically generate regexes for users. Second, regexes are notoriously hard to synthesize due to high ambiguity in user-given examples [45]. An analysis of 122 regular

expression tasks in Stack Overflow shows that only 18 tasks can be solved using examples only, as detailed in Section 6.2.

The DSL for regular expressions

We use an existing domain-specific language (DSL) to express regular expressions, which is also adopted by other regex generation techniques [39, 7]. Figure 2 describes the DSL grammar. In the simplest case, a regex is a *character class*, denoted by angle brackets. For instance, `<num>` matches any digits from `0` to `9`, `<let>` matches any English letters, `<low>` and `<cap>` match lower and upper case letters respectively, and `<any>` matches any character. We also have character classes that only match one single character such as `<a>`. These character classes can be used to build more complex regexes with different operators such as `startwith` and `star`. For instance, `startwith(<num>)` matches any strings starting with a digit. As another example, `star(<num>)` matches zero or more digits. Operators in this DSL provide high-level primitives that abstract away lower-level details in standard regex. This DSL is also more amendable to program synthesis as well as readable to users. We also note that this DSL has the same expressiveness power as a standard regular language.

$$
\begin{aligned}
e := {} & \texttt{<num>} \mid \texttt{<let>} \mid \texttt{<low>} \mid \texttt{<cap>} \mid \texttt{<any>} \mid \cdots \mid \texttt{<a>} \mid \texttt{<b>} \mid \cdots \\
& \mid \texttt{startwith}(e) \mid \texttt{endwith}(e) \mid \texttt{contain}(e) \mid \texttt{concat}(e_1, e_2) \\
& \mid \texttt{not}(e) \mid \texttt{or}(e_1, e_2) \mid \texttt{and}(e_1, e_2) \\
& \mid \texttt{optional}(e) \mid \texttt{star}(e) \\
& \mid \texttt{repeat}(e, k) \mid \texttt{repeatatleast}(e, k) \mid \texttt{repeatrange}(e, k_1, k_2)
\end{aligned}
$$

Figure 2: The DSL for regular expressions.

The synthesis algorithm

Given this DSL, we have developed a regex synthesizer that performs *top-down enumerative search* over the space of possible regexes defined by the DSL. This is a standard approach adopted by many existing synthesis techniques [16, 15, 63, 62]. Our regex synthesizer takes both positive and negative examples: a positive example is a string that should be accepted by the desired regex whereas a negative example is a string that should be rejected by the regex. Given these examples, our synthesizer generates a regex in the given DSL that accepts all positive strings and rejects all negative strings.

Algorithm 1 describes the basic synthesis process. The synthesizer maintains a worklist of partial regexes and iteratively refines them, until it finds a concrete regex that satisfies the given examples. The worklist is initialized with a partial regex with a symbolic value e (line 1). In each iteration, the synthesizer gets one regex p from the worklist (line 3). If p is concrete with no symbolic values, the synthesizer checks if p satisfies all user-given examples (lines 4-5). Otherwise, p is symbolic, and our algorithm refines p by expanding a symbolic value in p (lines 7-8). For instance, consider a partial regex `concat`(e_1, e_2). e_1 can be expanded to any character class, which yields partial regexes such as `concat(<num>,`e_2`)` and `concat(<let>,`e_2`)`. e_1 can also be expanded to an operator, yielding partial regexes such as `concat(startwith(`e_3`),`e_2`)` and `concat(repeatatleast(`e_3, k_1, k_2`),`e_2`)`. All these partial regexes will be added into the worklist for further refinement (line 12).

Algorithm 1: Top-down enumerative synthesis

Input : a set of string examples E
Output : a regular expression that is consistent with E

```
1  worklist := { e }
2  while worklist is not empty do
3      p := worklist.removeFirst()
4      if p is concrete then
5          if p is consistent with E then return p
6      else
7          s := selectSymbol (p)
8          worklist' := expand (p, s)
9          for p' in worklist' do
10             if p' is infeasible then remove p' from worklist'
11         end
12         worklist := worklist ∪ worklist'
13     end
14 end
```

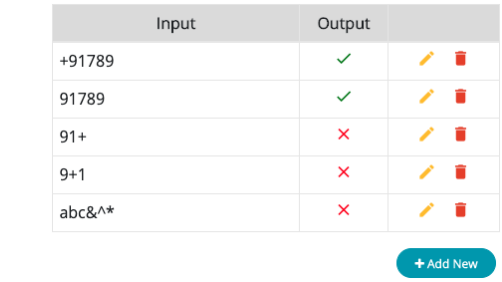
The number of regexes in the worklist will grow exponentially over iterations. To make it scale, different implementations of this algorithm leverage different heuristics to prune the search space (lines 9-11 in Algorithm 1). Despite all these efforts, it is still very challenging to scale existing synthesizers to solve real-world problems where desired regexes are complex, e.g., those in Stack Overflow posts. Furthermore, even if the synthesizer is able to quickly generate a regex that satisfies the user-provided examples, it may still not be the intended one. These factors make program synthesis a long-standing challenge. In the next section, we describe our proposed interaction model to tackle this disambiguation challenge.

SYNTHESIZING WITH AUGMENTED EXAMPLES: A USAGE SCENARIO

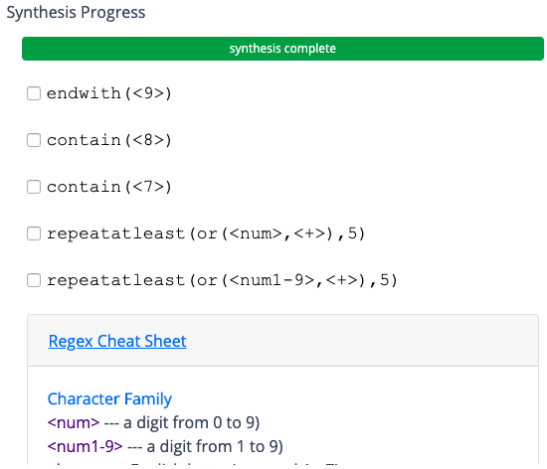
This section illustrates our proposed interaction model based on a realistic task from Stack Overflow [1]. We also compare our proposed method with manually creating counterexamples, which is the defacto way of providing user feedback in many PBE systems, as well as a state-of-the-art method that allows direct and explicit feedback on synthesized programs by annotating parts of a program as desirable and undesirable [46].

Suppose Alex is a data scientist and he wants to extract phone numbers from a text document. He needs to write a regular expression that accepts phone numbers starting with an optional + sign, followed by a sequence of digits. Though he is familiar with R and Python, he has only used regular expressions a few times before. He finds it too time-consuming to ramp up again, so he decides to try an interactive regex synthesizer, REGAE.

Alex starts with providing several input strings that should be accepted or rejected by the desired regex, as shown in Figure 3a. REGAE quickly synthesizes five regex candidates that satisfy those examples (Figure 3b). Though this is the first time Alex sees regexes in a domain-specific language, he finds the function names used in the regexes more readable than standard regexes. For some expressions such as `<num>` and `repeatatleast`, he looks up their definitions in the cheat sheet. Alex glances over those regex candidates and realizes that none of them are correct. The top three regexes only try to match concrete digits, `7`, `8`, `9`, while the last two only expect



(a) The input-output examples



(b) The synthesis result

Figure 3: Alex adds several input-output examples and receives five regexes that satisfy these examples (Iteration 1).

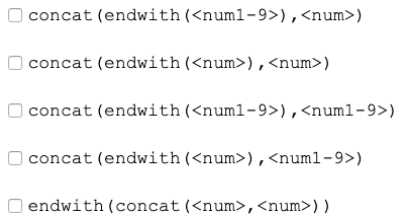
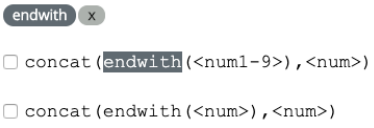
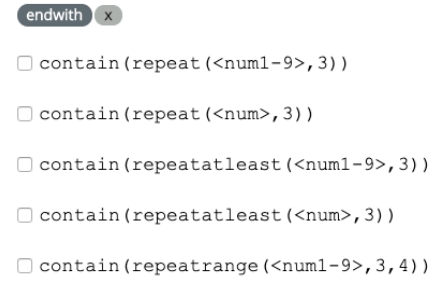


Figure 4: After adding a new positive example 123, Alex receives five new regex candidates (Iteration 2).



(a) Mark endwith as undesired



(b) The synthesis result

Figure 5: Alex marks the endwith operator as undesired to enforce the synthesizer not to consider this operator in the subsequent synthesis iterations (Iteration 3).

input strings to have at least five characters. Apparently, all these candidates overfit the positive examples.

The defacto way: providing counterexamples. To correct this overfitting issue, Alex adds a counterexample 123 to refute regexes in Figure 3b. The synthesizer returns five new regexes (Figure 4). Though Alex does not fully understand the subtle differences between those candidates, all those candidates still look wrong to him since they all use the endwith operator.

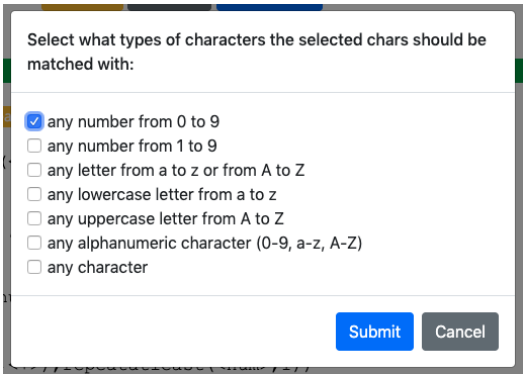
The state-of-the-art: annotating synthesized programs. Alex cannot easily come up with a counterexample that does not end with a digit, so he decides to mark endwith as undesired (Figure 5a). As a result, the synthesizer will not consider endwith in the following synthesis iterations. However, the new synthesis result in Figure 5b does not seem to be improved at all. As the synthesized regexes become more complicated over iterations, Alex starts feeling tired of inspecting those regexes. In particular, he is concerned about going down the rabbit hole since the synthesis process seems off track.

Semantic augmentation: marking parts of input strings as literal or general. Alex decides to step back and change how the synthesizer treats his input examples. First, he marks the + sign as a literal value to specify that it should be treated verbatim during synthesis (Figure 6a). Then, he marks the numbers after + as a general class of digits to specify those numbers should be generalized to any digits (Figure 6b). After adding those semantic annotations to input examples, Alex receives a new set of regex candidates (Figure 6c). Alex finds the synthesis result starts making sense since the + sign finally shows up in the regex candidates.

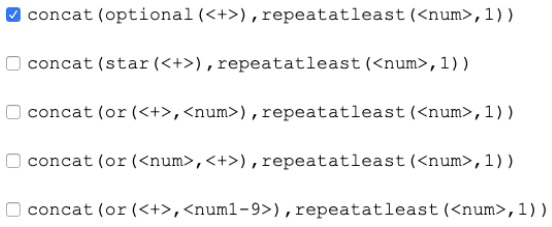
Data augmentation: generating additional input examples and corner cases. The first regex in Figure 6c looks correct to Alex, but he is not sure whether this regex has any unexpected behavior. Alex decides to ask REGAE to generate some additional input examples for him. He selects the first regex and clicks on "Show Me Familiar Examples." REGAE automatically generates many similar inputs by mutating the examples Alex have already given. Negative examples are clustered based on the failure-inducing characters and are also paired with similar positive examples. This design is inspired by previous findings on analogical encoding—*human subjects tend to focus on superficial details when only seeing single examples, while focusing on deeper structural characteristics when comparing and contrasting multiple examples* [18].

By glancing over the descriptive cluster headers and checking the examples in each cluster, Alex realizes that the selected regex allows the first character to be either + or a digit but requires the following characters to be digits only. This seems aligned with his expectations. Alex then clicks on "Show Me Corner Cases" to check if there are any hypothetical corner cases he has not thought of. REGAE performs an deep explo-

(a) Mark the + sign as a literal value.

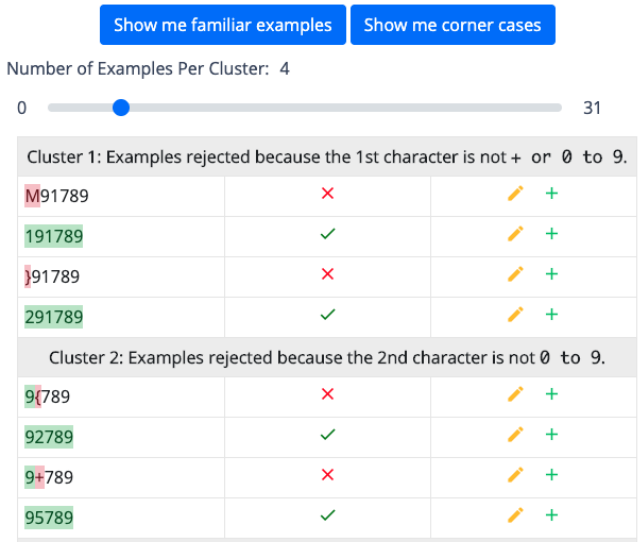


(b) Mark the numbers after + as a general class of digits.

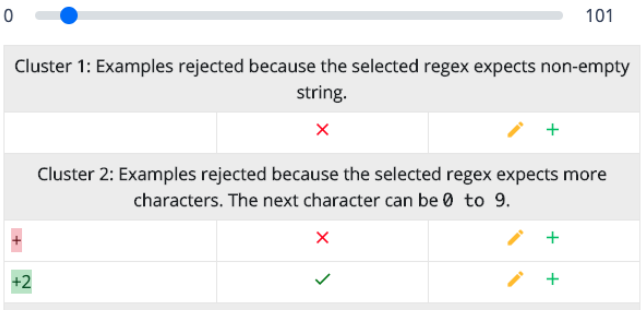


(c) The synthesis result

Figure 6: Alex specifies how individual characters in an input example should be treated by a synthesizer (Iteration 4).



(a) "Familiar examples" that are similar to user-given examples.



(b) "Corner cases" that Alex may never think of.

Figure 7: Alex asks for additional input examples to help him understand the functionality of synthesized regexes.

ration of the underlying automaton of the regex and generates input examples for the uncovered paths in the automaton (Figure 7b). The first corner case is an empty string and the second corner case is a single + sign, both of which are handled as he would want them to be. After inspecting those corner cases, Alex feels more confident about the first regex candidate.

Now Alex is a little indecisive between the first two regex candidates in Figure 6c. Though he vaguely understands the difference between `optional` and `star`, he wants to double check. Alex selects both regexes and clicks on "Show Me Corner Cases" to solicit some examples that distinguish their behavior. By glancing over those distinguishing examples in Figure 8, Alex confirms that the second regex can accept more than one + signs, which is wrong. Alex finally decides the first regex is the correct one, which takes four synthesis iterations without writing a single line of regular expressions.

ALGORITHMS AND IMPLEMENTATION

This section describes the implementation details that support the interaction features described in the previous section.

Incorporating User Annotations into Synthesis

Semantic annotations on input examples are parsed to literal values and general character classes first and then added to an include or exclude set. If a character is marked as literal, it will be added to the include set to enforce the synthesizer to generate a regex that contains this character. Sometimes a user may mark a sequence of characters as literal. Since it is hard to infer whether the user wants the synthesizer to treat the entire sequence verbatim or individual characters in it verbatim regardless of their positions, we decide to add both the entire sequence and individual characters to the include set to give two options for the synthesizer to choose. If a character or a sequence of characters is marked as a general class of characters such as digits or letters, we add the general character class into the include set and add the marked characters into the exclude set.

Given the include set and the exclude set, the `expand` function (line 8 in Algorithm 1) will ensure that none of the elements in the exclude set will be selected to expand a partial regex. It will assign higher priorities to elements in the include set to expand compared with operators and character classes not in

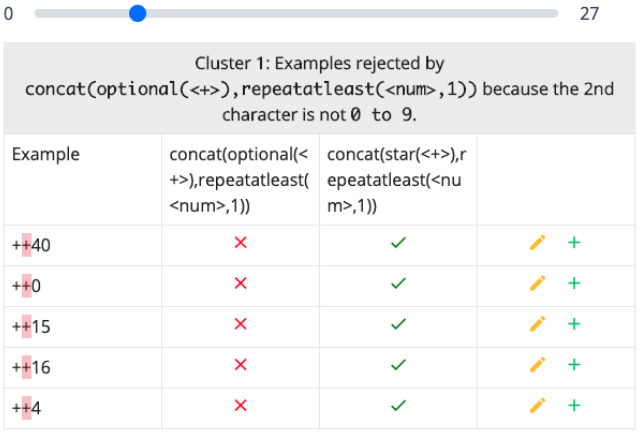


Figure 8: REGAE generates input examples that disambiguate two regular expressions with subtle differences. `optional(<+>)` only accepts strings with zero or one + sign, while `start(<+>)` accepts strings with zero or more + signs.

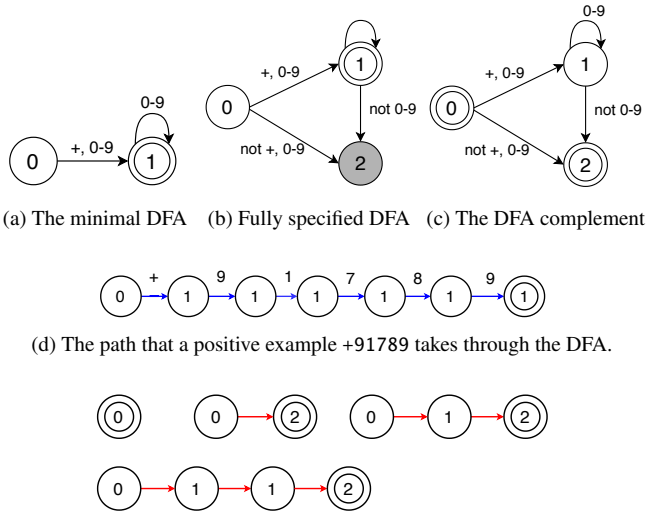
the include set. After each expansion, regexes in the worklist are ranked by the number of elements in the include set they contain. The regex that contains the most elements in the include set will be placed at the beginning of the list and will be evaluated first in the next loop iteration.

Regarding regex annotations, if a character class or an operator is marked as included or excluded, it will be added to the include or exclude set accordingly. If a subexpression is marked as included, it will also be added to the include set to expand a partial regex. If a subexpression is marked as excluded, we add a filter at line 10 in Algorithm 1 to remove any regexes that contain the subexpression from the worklist.

Generating and Clustering Input Examples

REGAE supports two input generation algorithms that complement each other. The first algorithm generates input data similar to user-given examples, while the second generates hypothetical corner cases by traversing the underlying automaton of a regex. Those hypothetical corner cases may look exotic to users but may reveal cases that users may not have thought of.

Example-driven input generation. The first algorithm takes a positive example and a regex as input, and generates additional positive and negative examples similar to the given example (Algorithm 2). To enable systematic examination of the given regex, REGAE first converts the regex to a deterministic finite automaton (DFA) [3] that decides the class of strings represented by the regex (line 3 in Algorithm 2). Figure 9a shows the minimized DFA of the first regex candidate in Figure 6c. Then REGAE examines which transition in the DFA would be taken to accept each character in the given example (line 4). Figure 9d shows the path through which the DFA matches a positive example `+91789` from Alex. Based on the accepted characters of the first transition in the DFA, the first character + in `+91789` can also be replaced with any digits from 0 to 9 to generate other inputs that are still accepted by the DFA (lines 6-7). On the other hand, if + is mutated to a character, such as a letter, that cannot be accepted by any other outgoing



(a) The minimal DFA (b) Fully specified DFA (c) The DFA complement

(d) The path that a positive example `+91789` takes through the DFA.

(e) Four paths that achieve 100% transition coverage of the DFA complement.

Figure 9: The DFA and its complement of a regular expression, `^+?[0-9]+$`. Circled states are accept states. Each transition is labelled with characters that are accepted by the transition.

Algorithm 2: Example-driven input generation

Input : a positive input example *e* and a regular expression *r*
Output : a set of positive and negative examples

```
1  positives := ∅
2  negatives := ∅
3  dfa := convertToDFA(r)
4  states := matchStates(dfa, e)
5  for s_i in states do
6      accept := the accepted chars in the transition from s_i to s_{i+1}
7      p := replace the i-th char in e with a random sample in accept
8      positives := positives ∪ p
9      accept_all := the accepted chars in all transitions from s_i
10     n := replace the i-th char in e with a random sample not in accept_all
11     negatives := negatives ∪ n
12 end
13 return positives ∪ negatives
```

transitions of State 0 in the DFA, the generated input will be rejected by the first transition in the DFA (lines 9-10). In this generation process, we use a random sampling method to select characters accepted or rejected by a transition. This process continues to process each character in the given input example. As a result, we will get many additional inputs that are one-character different from user-given examples.

Coverage-driven input generation. Though the first algorithm generates additional input examples similar to the user-provided examples, which may be easier for users to review, it does not generate farther examples that explore a greater range of the input space, e.g., various lengths, nor does it perform any deeper exploration of the regex's underlying automaton. Algorithm 3 describes a complementary algorithm. It performs breadth-first traversal on a DFA to identify possible paths to all accept states (lines 15-31). Since a DFA may be cyclic, we use transition coverage to guide the traversal process and terminate it after all transitions have been visited, i.e., 100% transition

Algorithm 3: Coverage-driven input generation

```
Input  : a regular expression r
Output : a set of positive and negative examples that cover all
         transitions in the DFA of r and its complement DFA
1  dfa := convertToDFA (r)
2  positives := generate (dfa)
3  complement := compute a DFA that accept all strings rejected by dfa
4  negatives := generate (complement)
5  return positives ∪ negatives
6  function generate (dfa)
7      E := {(∅, "")}
8      positives := ∅
9      visited := ∅
10     s0 := get the initial state of dfa
11     if s is accepted then
12         move an empty string to positives
13     end
14     E := {({s0 }, "")}
15     while isAllTransitionsCovered (dfa, visited) do
16         for (p,e) in E do
17             s := the last state in p
18             D := all destination states transitioned from s
19             for d in D do
20                 c := a random char in the accepted chars from s to d
21                 p' := append d to p
22                 e' := append c to e
23                 if d is accepted then
24                     move e' to positives
25                 end
26                 add (p',e') to E
27             end
28             visited := visited ∪ (s,d)
29             remove (p,e) from E
30         end
31     end
32     return positives
```

coverage. Note that one could also use other coverage metrics such as state coverage and transition-pair coverage [60].

To generate negative examples, REGAE first computes the DFA complement of a given regex by (1) converting the original DFA (Figure 9a) to a fully specified DFA over the entire alphabet (Figure 9b) and (2) converting all accept states in the DFA to non-accept and vice versa (Figure 9c), as described in [28]. The DFA complement represents all input strings that are not accepted by the original DFA. By performing the same traversal on the DFA complement, we can generate inputs that are accepted by the DFA complement, i.e., not accepted by the original DFA. Figure 9e shows four paths that achieve 100% transition coverage of the DFA complement in Figure 9c. Using this coverage-driven algorithm, we can identify paths that have not been covered by user-given examples and generate inputs that users may not have thought of. For each transition in a path, we randomly sample one character from the accepted characters of the transition to build up a new input example.

Distinguishing example generation. REGAE adapts the aforementioned two input generation algorithms to generate examples for distinguishing between multiple regexes. Regarding the example-driven algorithm, REGAE simply generates inputs for each regex and then only retains those inputs that expose different behavior among selected regexes, e.g., accepted by one regex but rejected by other regexes. Regarding the coverage-driven algorithm, REGAE first computes the differences between the DFAs of every two selected regexes, a_1 and a_2, yielding two new distinct automata—one is the intersection of a_1 and the complement of a_2 (i.e., $a_1 - a_2$), while the other is the intersection of a_2 and the complement of a_1 (i.e., $a_2 - a_1$). REGAE then performs the coverage-driven traversal in Algorithm 3 to each of the two automata to generate inputs that are accepted by one regex but not the other.

Clustering and rendering examples. REGAE clusters negative examples based on the automaton state in which these examples are rejected by the regex. The failure-inducing character in a negative example is highlighted in red to indicate where the mismatch occurs. Furthermore, to help users understand why a negative example is rejected, REGAE automatically generates an explanation based on the accepted characters of the transition that rejects the example as the cluster header. Each negative example is also juxtaposed with a positive example to draw a comparison across examples for enhanced learning and behavior recognition, inspired by analogical encoding [18]. By default, input examples in each cluster are sorted by length.

Incremental Computation

Most of the existing synthesis techniques completely restart the search process from scratch after receiving user feedback (e.g., new examples), without leveraging the computation result from previous iterations. As a result, the synthesis process may unnecessarily take a longer time. REGAE adopts an incremental synthesis strategy that resumes the search from the previous iteration. Recall that our synthesis algorithm (Algorithm 1) maintains a worklist of partial regexes. To achieve incremental synthesis, REGAE will memoize the current worklist after each iteration. In the next iteration, instead of initializing the worklist with $\{e\}$ (line 1 in Algorithm 1), REGAE restores the worklist memoized in the previous iteration.

Incremental computation is sound only if a user does not change her synthesis intent. However, in practice, users' mental models are constantly evolving. As a result, a user may revert previous input-output examples or annotations. To ensure the soundness of our synthesis approach, if previous examples or annotations are changed, we decide not to resume the synthesis process from the previous iteration and start a completely new synthesis process from scratch instead.

USER STUDY

We conducted a within-subjects study with twelve participants to evaluate whether they can effectively guide program synthesis using REGAE. We compared two versions of the system: a baseline version with only regex annotation as proposed in [46] and user-generated counterexamples, and a version that added the input annotation (*semantic augmentation*) and auto-generated examples (*data augmentation*). We investigated the following research questions:

- RQ1. Can our interaction model help a user efficiently disambiguate input examples and quickly arrive at a synthesized program with intended behavior?
- RQ2. Can our interaction model reduce the cognitive load of inspecting programs and increase the user's confidence in synthesized programs?

Participants

We recruited twelve Computer Science students (seven female and five male) through the mailing lists of several research groups at Harvard University. Participants received a $25 Amazon gift card as compensation for their time. Four participants were undergraduate students and the other eight were graduate students. Half of the participants had more than five years of programming experience, while the other half had two to five years of programming experience. Participants had diverse prior experiences with regexes. Six participants said they knew regex basics but only used it several times, five said they were familiar with regexes and had used it many times, and one said they had never heard about regexes before. The majority of participants (8/12) said, when writing regexes, they often had to search online to remind themselves of the details of regexes or use websites such as `regex101.com` to test their regexes. Ten out of twelve participants considered writing regexes more difficult than writing other kinds of programs. This is consistent with previous findings that regexes are hard to read, understand, and compose [5, 4].

Programming Tasks

To design realistic programming tasks for regular expressions, we searched on Stack Overflow (SO) using relevant keywords such as "regex", "regular expressions", "text validation". We found 122 real-world string matching tasks that have both English descriptions and illustrative input-output examples. Among those 122 tasks, 18 tasks can be solved using these illustrative input-output examples only. Thus there is no need to solicit user feedback for these 18 tasks. We selected three regex tasks from the remaining 104 unsolved tasks, which require further disambiguation of user intent. The task descriptions and the expected regexes in both our DSL and standard regex grammar are listed below.

Task 1. Write a regular expression that accepts strings that do not contain double hyphens (- -). [Post 2496840]

```
not(contain(concat(<->,<->)))
```

```
^((?!\s\s).)*$
```

Task 2. Write a regular expression that allows only digits, '&', '|', '(', or ')'. [Post 21178627]

```
repeatatleast(or(<num>,or(<&>,or(<|>,or(<(>,<)>)))),1)
```

```
^[\d()&|]+$
```

Task 3. Write a regular expression that accepts phone numbers that start with one optional + symbol and follow with a sequence of digits, e.g., +91, 91, but not 91+. [Post 41487596]

```
concat(optional(<+>),repeatatleast(<num>,1))
```

```
^\+?\d+$
```

Methodology

We conducted a 75-min user study with each participant. Participants completed two of the three regex tasks using REGAE with different interaction features enabled. For one of the two tasks, participants were allowed to use both the semantic augmentation and data augmentation features we propose, i.e., the experiment condition. For the other of the two tasks, they were only allowed to use the regex annotation feature as described in [46], i.e., the control condition. Both the order of task assignment and the order of interaction model assignment were counterbalanced across participants through random assignment.

Before each task, participants were given a 15-min tutorial about the interaction features they would use. In each task, participants first read the task description and then came up with their own positive and negative examples to start the synthesis. They continued to refine the synthesis result using the assigned interaction features, until they found a regex that looked correct to them. They were given 15 minutes for each task. If they did not find a correct regex within 15 minutes, we moved on to the next session. At the end of each task, participants answered a survey to reflect on the usability of the interaction model used in the task. They were also asked to answer five NASA Task Load Index questions [26] to rate the cognitive load of using the interaction model, as shown in Table 1. After finishing both tasks, participants answered a final survey to directly compare the two interaction models. Each survey took about five minutes.

NASA Task Load Index Questions
Q1. How mentally demanding was using this tool?
Q2. How hurried or rushed were you during the task?
Q3. How successful would you rate yourself in accomplishing the task?
Q4. How hard did you have to work to achieve your level of performance?
Q5. How insecure, discouraged, irritated, stressed, and annoyed were you?

Table 1: Participants rated different aspects of the cognitive load of using each interaction model on a 7-point scale.

USER STUDY RESULTS

User Performance. When using the semantic and data augmentation features, all 12 participants successfully found a correct regex with an average of 3.3 synthesis iterations. In contrast, when they can only manually add counterexamples and regex annotations, participants took significantly more synthesis iterations (7.7 on average) and only 4 of them successfully found a correct regex. With timed-out participants' time truncated at 15 minutes, the average task completion time with REGAE is 7.3 minutes, while the average task completion time with the baseline tool is 12.5 minutes. The mean difference of 5.2 minutes is statistically significant (paired t-test: t=4.37, df = 11, p-value=0.0011).

To disentangle the effects of different interaction features, we analyzed the user study screencasts and manually counted how often each feature was utilized. In the control condition, participants had to manually construct counterexamples to refute incorrect regex candidates in more than half of the synthesis iterations (4.5/7.7). Participants in the control condition also heavily utilized the regex annotation feature to prune the search space. On average, they marked 5 subexpressions as included or excluded in 3.1 out of 7.7 iterations. In contrast, this regex annotation feature was significantly under-utilized in the experiment condition—only four of 12 participants used it and they only gave one regex annotation in 0.6 iterations on average. Instead, all participants in the experiment condition used the semantic augmentation feature and gave an average

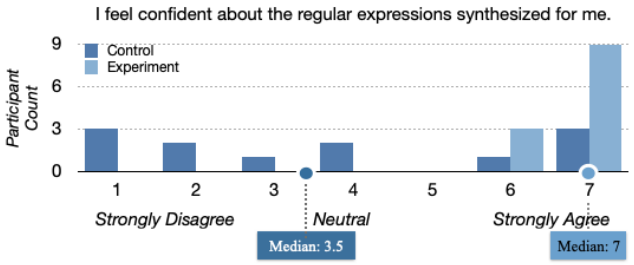


Figure 10: When using REGAE, participants felt twice more confident in the regexes synthesized on behalf of them.

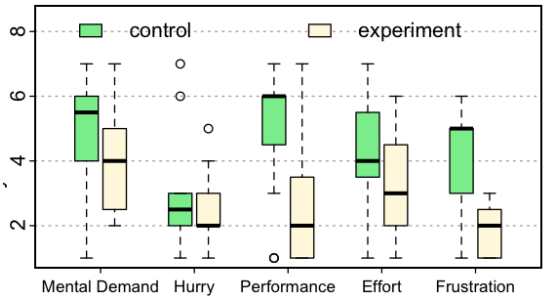


Figure 11: Cognitive load measured by NASA TLX [26]

of 3.2 input annotations in 1.4 out of 3.3 iterations. Besides, all participants heavily utilized the data augmentation feature to generate additional inputs in 1.9 of 3.3 iterations. Seven of 12 participants directly reused automatically generated inputs (3.4 inputs on average) as counterexamples. The comparison of feature utilization between the control and experiment conditions indicates that semantic augmentation and data augmentation contributed significantly to the high success rate of regex tasks in the experiment condition.

We also coded participants' feedback in the post survey to analyze why they failed to synthesize desired regexes in the control condition. Seven of 12 participants found it hard to decide which operators or subexpressions to include or exclude. Because participants were not certain about the final regex, it was difficult to foresee what to include or exclude. Some of them explained that they actually ended up wasting a lot of time since they gave an ineffective or wrong regex annotation due to misconceptions, which sent the synthesizer off track. Six participants complained it was time-consuming to manually craft representative counterexamples to refute incorrect regexes. Five participants found it hard to parse and interpret synthesized regexes within a short amount of time.

User Confidence and Cognitive Load. In the post survey, participants reported that they felt significantly more confident in the synthesized regexes when they were given options to see additional examples and corner cases generated by REGAE. Figure 10 shows the distribution of their responses (7 vs. 3.5 on a 7-point scale). P10 said, *"[REGAE made it] really easy to experimentally verify whether the regex is right or not, also made it way easier to parse the regex after seeing the examples."* Experienced regex programmers also appreciated those automatically generated corner cases. P6 said, *"even though I am decently good at regexes and I am fast at parsing and understanding them, I think many times there are corner cases I simply don't see beforehand. Seeing concrete positive and negative examples is a lot more helpful and I get immediate feedback on correctness of the regex."*

Participants also perceived less mental demand and frustration during the task when interacting with augmented examples (Figure 11). The main reason was that they no longer needed to inspect regexes very carefully or think very hard to find counterexamples. Instead, they could merely glance over the automatically generated examples instead. Though those additional examples were simply rendered in a cluster view, the majority of participants (9/12) found it more helpful than overwhelming. P7 said, *"it was way more intuitive to look at examples and change my input examples based on familiar cases and corner cases."* Specifically, participants appreciated that each cluster had a descriptive header, and individual characters were highlighted in green or red to make failure-inducing characters obvious. Besides, by annotating certain characters in an input example as general or literal, participants did not need to add many input examples and thus saved a lot of manual effort. In the control condition, participants tended to enumerate all possible characters (e.g., all digits from 0 to 9) that should be matched by a desired regex.

Qualitative Analysis. The most appreciated feature (12/12) is to annotate input examples with the notions of generality and specificity. P2 wrote, *"it gives me freedom to specify what characters should be matched verbatim, which is exactly what I wished for."* Eleven participants (92%) preferred to annotate input examples than regexes, because it was more intuitive to directly express their intent through examples. Marking characters as general or literal also significantly reduced the number of examples they needed to provide. In contrast, annotating regexes required careful examination of synthesized regexes, and participants were also concerned with providing wrong regex annotations due to their misconceptions of regex operators. Though semantic augmentation is more preferred than regex annotation, this should not be interpreted as regex annotation no longer being needed. Still, seven participants (58%) found regex annotation helpful in narrowing down the search space. In particular, two participants explained that regex annotation became much easier with automatically generated examples, since those examples helped define some operators that were hard to understand before. Seven participants (58%) really liked the data augmentation feature with no complaints at all. Four participants (33%) liked this feature but also suggested that it was not easy to immediately figure out which examples to focus on.

Participants did point out several areas where the interface could be improved. Five participants suggested to generate regex candidates with greater diversity and variations. Three participants wished to manually add a partial regex in their mind to start the synthesis. Five participants suggested to select a small set of representative examples to focus on rather than showing all generated examples. Two participants wished to have a feature that automatically converts a synthesized

regex to the standard regex format so they can copy and paste it to their own code.

When asked how such an interactive synthesizer would fit into their programming workflow, 10 participants said they would definitely like to use it when they encounter a complex task or when they can not find an online solution. P1 wrote, *"I usually go to regex101.com to test my expression with an example, which takes a while before I arrive at the right expression. With this tool, I can easily come up with the right expression to use in my program."* One participant said she may not use it end-to-end since she generally had a good idea about what to write, but she found the data augmentation feature very helpful to discover unexpected behavior on corner cases. Two participants said they would love this synthesizer to be incorporated into their IDEs as a plugin.

QUANTITATIVE EVALUATION

We conducted a case study with 20 realistic regular expression tasks to investigate the effectiveness of REGAE on a variety of complex tasks. The tasks were randomly selected from the Stack Overflow benchmarks in Section 6.2. The 20 tasks and screenshots of their final solutions synthesized by REGAE are included in the supplementary material. The tasks include standard regex tasks such as validating numbers delimited by comma (Post 5262196), validating US phone numbers (Post 23195619), and validating version numbers (Post 31649251), as well as custom tasks such as accepting strings with only seven or ten digits (Post 2908527) and validating decimals with up to 4 decimal places (Post 30730877).

For each task, the first author read its description on Stack Overflow and solved the task by interacting with REGAE. The author stopped when REGAE synthesized a regex equivalent to the accepted answer in the Stack Overflow post. This simulates an ideal condition where a user is familiar with the tool and has sufficient knowledge of regexes. The purpose of this case study is to investigate to what extent our interactive synthesizer can solve realistic tasks, rather than its learnability or usability.

The experimenter successfully solved all 20 tasks in an average of 5.9 minutes (median=4.9, SD=0.2) and within 5.4 iterations (median=5, SD=3.4). 16 of 20 tasks are solved within 7 minutes and 6 iterations. The power of REGAE was fully unleashed when all three features were used together—*semantic augmentation*, *data augmentation*, and *regex annotation*. Marking individual characters as general or literal provided crucial hints about how to generalize user-given examples during synthesis. Without this feature, the synthesizer often started from a completely wrong direction or times out without identifying any regex candidates in one minute. Though the experimenter was very familiar with the underlying regex DSL, he found it tedious to constantly read many regex candidates over iterations. In particular, regex candidates often became over-complicated after 3 or 4 iterations. In such cases, automatically generating corner cases made it easier to quickly craft counterexamples. The experimenter also found that in many situations, regex annotation could be a convenient way to steer the synthesis direction. Of course, this required a user to be familiar with the regex DSL or have some notion of the final regex. The experimenter often excluded some operators to prune irrelevant search paths through the program space, which significantly sped up the synthesis process.

Four tasks required a divide-and-conquer strategy to solve, which was not expected. For example, one task asks for a regex to validate UK mobile numbers that can be between 10-14 digits and must start with either 07 or 447 (Post 16405187). To solve this task, the experimenter first only added numbers starting with 07 and synthesized a sub-regex, `^07\d{8,12}$` that matches this type of phone numbers. Then he marked it as desired, removed existing examples, and added another type of phone numbers starting with 447. He synthesized another sub-regex, `^447\d{7,11}$` to match phone numbers starting with 447 and also marked it as desired. Finally, he added both types of phone numbers as examples and synthesized the final regex, `^07\d{8,12}|447\d{7,11}$`. Having both sub-regexes marked as desired, it was easier for REGAE to assemble them together to create the final regex. Prior work shows that inexperienced users may find it difficult to decompose a complex task into sub-tasks [34]. It would be interesting to investigate how to teach inexperienced users such task decomposition strategies through training sessions or prompted hints in future work.

DISCUSSION AND FUTURE WORK

In this paper, we propose a new interaction model for interactive program synthesis. Our interaction model is mainly designed for programming-by-example (PBE) systems that take input-output examples as input. There are other kinds of PBE systems that take as input user demonstrations such as video recordings and interaction traces, also known as programming-by-demonstration or PBD [6, 38, 37, 36, 35, 48, 47]. Applying our interaction model to PBD systems requires further instruments, e.g., enabling tagging a video clip or rendering interaction traces in a more readable and editable format for users to operate on. This by itself is an interesting direction for further investigation.

This work demonstrates the feasibility and effectiveness of guiding program synthesis through augmented examples in a specific domain, regular expressions. The proposed interaction model can also be generalized to synthesize other kinds of programs. For example, code transformation synthesis [50] takes program source code before and after transformation as input-output examples and generates transformation scripts that refactor or repair other similar code locations. While the underlying implementation would need to be adapted or extended to fit the underlying synthesizer, the semantic annotation interface is applicable to code transformation synthesis: similar to how users mark characters in an input string as literal or general, users could also mark function calls and variable names that should be kept constant or generalized to match other similar code. While the automaton-based input generation algorithms in Section 5 are specific to regular expressions, the idea of generating additional input-ouput examples to facilitate comprehension and validation of synthesized programs is applicable in other domains as well. There are many other kinds of input generation methods to leverage or adapt. For example, in the domain of grammar synthesis [2], one could leverage grammar-based fuzzing techniques [19, 27, 20] to

generate additional programs as inputs to test a synthesized grammar. As another example, in the domain of generating object-oriented programs [15], one could consider test suite generation techniques such as EvoSuite [17] and Pex [56] to generate additional inputs for synthesized programs.

Our current system has three major limitations that we wish to improve on in future work. First, when users make mistakes or introduce contradictory examples, our system only shows the synthesizer fails to find a satisfying program in the progress bar. It is not able to recognize or pinpoint user mistakes. Second, even without any user mistakes, the synthesizer may still fail to return anything to the user, if the given examples are too complicated to generalize or if a task is too hard to solve within a given time budget. Our current system provides no affordance to debug these synthesis failures nor help users understand which step or input it is stuck on. Finally, the semantic augmentation feature currently only supports specifying characters as literal values or general classes of characters. Hence, it is cumbersome to handle string matching tasks with sophisticated character ordering constraints and length constrains, such as the UK phone number example in Section 8. Though the divide-and-conquer strategy is proven to be a feasible workaround, extending semantic augmentation to express richer semantics such as temporal ordering and input length is probably a more convenient option for users.

Multi-modal synthesis is another active research direction to resolve intent ambiguity in user-given examples. Several systems have been proposed to support multi-modal specifications such as natural language descriptions [7] and verbal instructions [38, 36, 37] in addition to examples. Investigating possible augmentations on these other types of specifications is an interesting direction to pursue in the future.

CONCLUSION

This paper presents a new interaction model that operates on user-provided examples to guide program synthesis. We demonstrate its feasibility and effectiveness in the domain of regular expressions by (1) building an interactive synthesizer for regular expressions, (2) conducting a within-subjects lab study with 12 participants on realistic regex tasks from Stack Overflow, and (3) conducting a case study on another 20 complex regex tasks from Stack Overflow. In the end, we discuss the generalizability of this interaction model and what kinds of adaptations are needed to apply it to other domains such as code transformation synthesis and grammar synthesis.

ACKNOWLEDGMENTS

We would like to thank anonymous participants for the user study and anonymous reviewers for their valuable feedback.

REFERENCES

[1] Regular expression to validate the country code of mobile number in javascript. https://stackoverflow.com/questions/41487596. (????). Accessed: 2020-04-04.

[2] Osbert Bastani, Rahul Sharma, Alex Aiken, and Percy Liang. 2017. Synthesizing program input grammars. *ACM SIGPLAN Notices* 52, 6 (2017), 95–110.

[3] Gerard Berry and Ravi Sethi. 1986. From regular expressions to deterministic automata. *Theoretical computer science* 48 (1986), 117–126.

[4] Carl Chapman and Kathryn T Stolee. 2016. Exploring regular expression usage and context in Python. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 282–293.

[5] Carl Chapman, Peipei Wang, and Kathryn T Stolee. 2017. Exploring regular expression comprehension. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 405–416.

[6] 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.

[7] Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-modal Synthesis of Regular Expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[8] Dana Drachsler-Cohen, Sharon Shoham, and Eran Yahav. 2017. Synthesis with abstract examples. In *International Conference on Computer Aided Verification*. Springer, 254–278.

[9] Samuel Drews, Aws Albarghouthi, and Loris D'Antoni. 2019. Efficient Synthesis with Probabilistic Constraints. In *International Conference on Computer Aided Verification*. Springer, 278–296.

[10] 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*. 6038–6049.

[11] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program repair with quantitative objectives. In *International Conference on Computer Aided Verification*. Springer, 383–401.

[12] Loris D'Antoni, Rishabh Singh, and Michael Vaughn. 2017. NoFAQ: Synthesizing Command Repairs from Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 582–592. DOI: http://dx.doi.org/10.1145/3106237.3106241

[13] Kevin Ellis, Armando Solar-Lezama, and Josh Tenenbaum. 2016. Sampling for bayesian program learning. In *Advances in Neural Information Processing Systems*. 1297–1305.

[14] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017a. Component-based synthesis of table consolidation and transformation tasks from examples. *ACM SIGPLAN Notices* 52, 6 (2017), 422–436.

[15] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W Reps. 2017b. Component-based synthesis for complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 599–612.

[16] John K Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing data structure transformations from input-output examples. *ACM SIGPLAN Notices* 50, 6 (2015), 229–239.

[17] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[18] Dedre Gentner, Jeffrey Loewenstein, and Leigh Thompson. 2003. Learning and transfer: A general role for analogical encoding. *Journal of Educational Psychology* 95, 2 (2003), 393.

[19] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 206–215.

[20] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&fuzz: Machine learning for input fuzzing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 50–59.

[21] Cordell Green. 1981. Application of theorem proving to problem solving. In *Readings in Artificial Intelligence*. Elsevier, 202–222.

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

[23] Sumit Gulwani, William R Harris, and Rishabh Singh. 2012. Spreadsheet data manipulation using examples. *Commun. ACM* 55, 8 (2012), 97–105.

[24] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, and others. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* 4, 1-2 (2017), 1–119.

[25] Philip J Guo, Sean Kandel, Joseph M Hellerstein, and Jeffrey Heer. 2011. Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*. 65–74.

[26] Sandra G Hart and Lowell E Staveland. 1988. Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In *Advances in psychology*. Vol. 52. Elsevier, 139–183.

[27] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*. 445–458.

[28] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News* 32, 1 (2001), 60–65.

[29] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. 2010. Oracle-guided component-based program synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. IEEE, 215–224.

[30] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. 2011. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 3363–3372.

[31] Tessa Lau. 2009. Why programming-by-demonstration systems fail: Lessons learned for usable ai. *AI Magazine* 30, 4 (2009), 65–65.

[32] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. 2003. Programming by demonstration using version space algebra. *Machine Learning* 53, 1-2 (2003), 111–156.

[33] 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.

[34] Tak Yeon Lee, Casey Dugan, and Benjamin B Bederson. 2017. Towards understanding human mistakes of programming by example: an online user study. In *Proceedings of the 22Nd International Conference on Intelligent User Interfaces*. 257–261.

[35] Jiahao Li, Jeeeun Kim, and Xiang Anthony Chen. 2019. Robiot: A Design Tool for Actuating Everyday Objects with Automatically Generated 3D Printable Mechanisms. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 673–685.

[36] Toby Jia-Jun Li, Amos Azaria, and Brad A Myers. 2017. SUGILITE: creating multimodal smartphone automation by demonstration. In *Proceedings of the 2017 CHI conference on human factors in computing systems*. 6038–6049.

[37] Toby Jia-Jun Li, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenze Shi, Wanling Ding, Tom M Mitchell, and Brad A Myers. 2018. APPINITE: A Multi-Modal Interface for Specifying Data Descriptions in Programming by Demonstration Using Natural Language Instructions. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 105–114.

[38] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M Mitchell, and Brad A Myers. 2019. PUMICE: A Multi-Modal Agent that Learns Concepts and Conditionals from Natural Language and Demonstrations. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 577–589.

[39] Nicholas Locascio, Karthik Narasimhan, Eduardo DeLeon, Nate Kushman, and Regina Barzilay. 2016. Neural generation of regular expressions from natural language with minimal domain knowledge. *arXiv preprint arXiv:1608.03000* (2016).

[40] Zohar Manna and Richard Waldinger. 1975. Knowledge and reasoning in program synthesis. *Artificial intelligence* 6, 2 (1975), 175–208.

[41] Zohar Manna and Richard J Waldinger. 1971. Toward automatic program synthesis. *Commun. ACM* 14, 3 (1971), 151–165.

[42] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User interaction models for disambiguation in programming by example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*. 291–301.

[43] Robert C Miller, Brad A Myers, and others. 1999. Lightweight Structured Text Processing.. In *USENIX Annual Technical Conference, General Track*. 131–144.

[44] Brad A Myers and Richard McDaniel. 2001. Demonstrational interfaces: sometimes you need a little intelligence, sometimes you need a lot. In *Your wish is my command*. Morgan Kaufmann Publishers Inc., 45–60.

[45] Rajesh Parekh and Vasant Honavar. 1996. An incremental interactive algorithm for regular grammar inference. In *International Colloquium on Grammatical Inference*. Springer, 238–249.

[46] Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming not only by example. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1114–1124.

[47] David Porfirio, Evan Fisher, Allison Sauppé, Aws Albarghouthi, and Bilge Mutlu. 2019. Bodystorming Human-Robot Interactions. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology*. 479–491.

[48] David Porfirio, Allison Sauppe, Aws Albarghouthi, and Bilge Mutlu. 2020. Transforming Robot Programs Based on Social Context. In *Proceedings of the 2020 CHI conference on human factors in computing systems*.

[49] Veselin Raychev, Pavol Bielik, Martin Vechev, and Andreas Krause. 2016. Learning Programs from Noisy Data. *SIGPLAN Not.* 51, 1 (Jan. 2016), 761–774. DOI: http://dx.doi.org/10.1145/2914770.2837671

[50] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 404–415.

[51] Mark Santolucito, Ennan Zhai, Rahul Dhodapkar, Aaron Shim, and Ruzica Piskac. 2017. Synthesizing Configuration File Specifications with Association Rule Learning. *Proc. ACM Program. Lang.* 1, OOPSLA, Article Article 64 (Oct. 2017), 20 pages. DOI: http://dx.doi.org/10.1145/3133888

[52] Christopher Scaffidi, Brad Myers, and Mary Shaw. 2008. Topes. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 1–10.

[53] Eui Chul Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. 2019. Program synthesis and semantic parsing with learned code idioms. In *Advances in Neural Information Processing Systems*. 10824–10834.

[54] Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 326–340. DOI:http://dx.doi.org/10.1145/2908080.2908102

[55] Eric Spishak, Werner Dietl, and Michael D Ernst. 2012. A type system for regular expressions. In *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*. 20–26.

[56] Nikolai Tillmann and Jonathan De Halleux. 2008. Pex–white box test generation for. net. In *International conference on tests and proofs*. Springer, 134–153.

[57] Richard J Waldinger and Richard CT Lee. 1969. PROW: A step toward automatic program writing. In *Proceedings of the 1st international joint conference on Artificial intelligence*. 241–252.

[58] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Interactive query synthesis from input-output examples. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1631–1634.

[59] Chenglong Wang, Yu Feng, Rastislav Bodik, Alvin Cheung, and Isil Dillig. 2019b. Visualization by Example. *Proc. ACM Program. Lang.* 4, POPL, Article Article 49 (Dec. 2019), 28 pages. DOI: http://dx.doi.org/10.1145/3371117

[60] Peipei Wang and Kathryn T Stolee. 2018. How well are regular expressions tested in the wild?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 668–678.

[61] Xinyu Wang, Sumit Gulwani, and Rishabh Singh. 2016. FIDEX: filtering spreadsheet data using examples. *ACM SIGPLAN Notices* 51, 10 (2016), 195–213.

[62] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019a. Synthesizing database programs for schema refactoring. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 286–300.

[63] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. 2017. SQLizer: query synthesis from natural language. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–26.

[64] Kuat Yessenov, Shubham Tulsiani, Aditya Menon, Robert C Miller, Sumit Gulwani, Butler Lampson, and Adam Kalai. 2013. A colorful approach to text processing by example. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*. 495–504.

[65] Yifei Yuan, Rajeev Alur, and Boon Thau Loo. 2014. NetEgg: Programming Network Policies by Examples. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets-XIII)*. Association for Computing Machinery, New York, NY, USA, 1–7. DOI: http://dx.doi.org/10.1145/2670518.2673879